Model Driven Construction and Customization of Modeling & Simulation Web Applications

Proefschrift

ter verkrijging van de graad van doctor aan de Technische Universiteit Delft, op gezag van de Rector Magnificus prof.dr.ir. J.T. Fokkema, voorzitter van het College voor Promoties, in het openbaar te verdedigen

op maandag 19 januari 2009 om 12.30 uur

door

Andriy LEVYTSKYY

informatica ingenieur, Yurii Fedkovych Chernivtsi National University, Ukraine geboren te Chernivtsi, Oekraine
Dit proefschrift is goedgekeurd door de promotor:

Prof. dr. H. Koppelaar

Samenstelling promotiecommissie:

Rector Magnificus,
Prof. dr. H. Koppelaar,
Prof. drs. dr. L.J.M. Rothkrantz,
Prof. dr. C.M. Jonker,
Prof. dr. H. Vangheluwe,
Prof. dr. ing. Habil. W. Häckl
Prof. dr. ir. J.L.G. Dietz
Prof. dr. ir. H.J. Sips

voorzitter
Technische Universiteit Delft, promotor
Nederlandse Defensie Academie
Technische Universiteit Delft
McGill University, Montreal, Canada
Technische Universiteit Delft
Technische Universiteit Delft
Technische Universiteit Delft

Keywords:
Modeling and Simulation (M&S); Web-based environment; Conceptual framework;
Efficient customization; Model-Driven Engineering (MDE)

Copyright © 2008 by A. Levytskyy

All rights reserved. No part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without written permission from the author.

ISBN 978-90-8891-081-4
Acknowledgements

This thesis is result of a long research journey. While the journey is the author's own, there have also been many companions that played their role in making this work come to its conclusion.

To start with, I would like to thank my supervisor Prof. Eugene Kerckhoffs for giving me an opportunity to join the NanoComp project, his guidance and allowing me to perform the research described in this thesis. The TU Delft sponsored NanoComp project is acknowledged for providing an interesting and challenging research context.

Special gratitude goes to Prof. Hans Vangheluwe for helping to form the general direction of this research. His assistance in the middle and late stages of the project, as well as inviting me as visiting researcher to the Modeling, Simulation and Design Lab (MSDL) of the School of Computer Science of McGill University in Montréal, Canada is much appreciated. Fellow MSDL students have to be thanked for a friendly and informal environment that provided a very fruitful and creative research atmosphere. Among them, Ernesto Posse and Jean-Sébastien Bolduc, for allowing to reuse parts of their work on generation of DEVS Modeling and Simulation environments. I also would like to thank Juan de Lara of Universidad Autónoma de Madrid, Spain, for his work on AToM³.

I especially would like to thank my promoter Prof. Henk Koppelaar and Prof. Leon Rothkrantz, who remained committed to this project despite the delays and provided much needed encouragements in the final stage of the project. Andrea Cilio, a great friend, whose generous support and constructive comments both improved the quality of and sped up completion of the thesis. Stepan Voevidko, thank you for your tireless support during all these years.

I owe many thanks to my family and especially my mom, who, while being far away, motivated and supported me from the first day of the project. Without her this work would never have started and never have been finished. A big thank you to my little daughter Melody, who strongly motivated me to finish this book as soon as possible. Finally, my wife Marjaneh, who was directly affected by this project more then anybody else. She went together with me through ups and downs of writing the dissertation. For that I thank you.

Andriy Levytskyy
The Hague, December 2008
Summary
Of the thesis:

Model Driven Construction and Customization of Modeling & Simulation Web Applications

by Andriy Levytskyy

The emergence of the WWW and its big popularity in the simulation community gave birth to the concept of Web-Based Modeling & Simulation. The implication of this phenomenon is that traditional single user M&S tools will have to be replaced or supplemented with more powerful tools supporting collaborative practices of users. As complexity of engineered systems continuously increases, the use of M&S in development and demand for powerful Web-Based M&S environments are expected to grow in the future.

The need for new web enabled M&S tools was answered by research communities. These first results often were ad hoc solutions based on merging M&S with Web technologies. A typical application would consist of cementing its ingredients: a modeling paradigm, simulators and collaborative aspects with particular Web technologies. If any of these ingredients would be different in a new application, such hard coded paradigms, collaboration aspects and simulators could not be easily changed or customized mainly due to low efficiency associated with the traditional code-oriented development. Moreover, a key approach in many contributions was the role that web-enabling technologies, such as Java language, played both for specification and implementation of the model. This double role of the web technology invoked a number of disadvantages: First, modeling languages were replaced by a relatively low-level, general purpose programming language, which resulted in decreased productivity of modellers and increased semantic gap between possibly multiple modeling paradigms and the web-enabling technology used for modeling. Furthermore, intertwining of specification and implementation made separation of a model from its simulator harder, thereby reducing portability and reusability of models.

This thesis presents a model-driven approach to construction of web-based environments that could be efficiently customized to modeling and simulation needs of an arbitrary number of M&S application domains. To achieve broad applicability, our approach is based on general concepts in fields of Modeling and Simulation, Distributed Systems, and Collaborative Software. Such stable concepts constitute the collaborative Modeling and online Simulation (cMoS) framework. cMoS provides a general basis for a family of Web-Based M&S applications. Specific M&S applications are supported through
customization of the variation points in cMoS during the operation phase of the environment. To enable efficient customization, the variation points are not implicitly hardcoded as traditionally, but are expressed as models. The use of the resulting models is still limited due to a high barrier of their implementation. This barrier is amended by Model-Driven Engineering (MDE): models of the variation points are computerized and their implementation is automatically generated. The end result is a model-driven and mostly programming free Web-Based environment adaptable to new M&S applications through abstract modeling of the cMoS variation points.
# Contents

## Abbreviations

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xi</td>
<td></td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Research problem ........................................... 2
1.2 Approach .................................................... 4
1.3 Definitions .................................................. 4
1.4 Delimitations ................................................ 5
1.5 Contributions ............................................... 5
1.6 Outline of this thesis ...................................... 6

## 2 Problem domain

2.1 Modeling and Simulation .................................... 9
2.2 Distributed systems ......................................... 10
2.2.1 Key characteristics ...................................... 11
2.2.2 The Internet ............................................... 13
2.3 Collaborative software ...................................... 15
2.4 Summary ...................................................... 16

## 3 Model-Driven Engineering

3.1 What is MDE? .................................................. 17
3.1.1 Software development process ............................ 17
3.1.2 Problems of software development ....................... 20
3.1.3 Definitions of MDE ........................................ 21
3.1.4 Essence of MDE ............................................ 22
3.2 Basic concepts ............................................... 23
3.2.1 System, Model and Meta-step ............................. 23
3.2.2 Language as System ....................................... 26
3.2.3 Transformation ............................................. 28
3.3 MDE approaches ............................................... 31
3.3.1 Model Driven Architecture ................................ 32
3.3.1.1 Modeling space and process ............................. 32
3.3.1.2 Transformation ......................................... 33
3.3.1.3 The 1/2 floor of the MDA building ..................... 34
3.3.1.4 The MDA framework .................................... 35
3.3.1.5 OMG standards and technologies ....................... 36
3.3.1.6 Profiling ............................................... 38
CONTENTS

3.3.1.7 MDA tools .......................................................... 38
3.3.2 Domain Specific Modeling .......................................... 38
3.3.2.1 Modeling space and process ...................................... 39
3.3.2.2 The DSM framework .............................................. 40
3.3.2.3 DSM tools .......................................................... 41
3.4 Summary ................................................................. 41

4 MDE at Work ............................................................. 43
4.1 Data ........................................................................... 43
4.2 Data Flow Diagrams (DFD) ............................................. 44
4.2.1 DFD metamodel ....................................................... 45
4.2.2 POKer code generation ............................................. 46
4.3 Discrete Event System Specification (DEVS) ..................... 49
4.3.1 Metamodel of DEVS with ports .................................... 49
4.3.2 PythonDEVS code generation ..................................... 53
4.4 Process Interaction (PI) ................................................ 54
4.4.1 PI metamodel .......................................................... 54
4.4.2 PI operational semantics ........................................... 58
4.4.2.1 Example rule ...................................................... 58
4.4.2.2 Event execution .................................................. 58
4.5 Dynamic Traffic Routing (DTR) ..................................... 61
4.5.1 Metamodel of a traffic language .................................. 61
4.5.2 ABC Code Generation ............................................... 62
4.6 Summary ................................................................. 64

5 From Models to Implementation ....................................... 65
5.1 From an implementation to models .................................. 65
5.1.1 The NCSE prototype ................................................ 66
5.1.2 Shift to models ....................................................... 67
5.2 A framework for Collaborative Modeling and Online Simulation .......... 69
5.2.1 Separation-related issues ......................................... 71
5.2.2 User ................................................................. 72
5.2.3 User-Simulator relationship ...................................... 73
5.2.4 Scenario ............................................................ 73
5.2.5 Controller ........................................................... 73
5.2.6 Tool ................................................................. 74
5.2.7 Model ................................................................. 74
5.2.7.1 Modeling formalism and languages ....................... 74
5.2.7.2 Simulator independence ..................................... 74
5.2.7.3 Storage independence ........................................ 76
5.2.8 Repository ........................................................ 77
5.2.9 Simulator and Solver ............................................... 77
5.2.10 Simulation and Experiment ..................................... 77
5.3 AcMoS application meta-architecture ................................ 77
5.4 Model-driven engineering processes .............................. 80
5.4.1 General software development ................................... 80
## CONTENTS

7.8 Scenario .................................................. 107
7.9 Experiment data ............................................. 109
7.10 Experiment .................................................. 109
7.11 Controller .................................................. 111
   7.11.1 Domain model ....................................... 111
   7.11.2 Operational semantics .............................. 113
   7.11.3 Server ............................................... 114
   7.11.4 Controller’ .......................................... 115
7.12 Tool ....................................................... 115
   7.12.1 Export functionality ................................. 115
      7.12.1.1 Locations metamodel ......................... 115
      7.12.1.2 Export transformation ....................... 115
      7.12.1.3 Multi-formalism modeling .................... 116
   7.12.2 Code generation guidelines ....................... 116
7.13 Summary .................................................. 117

8 AcMoS Applications ............................................ 119
   8.1 Application domain customization .................... 119
      8.1.1 Application domain definition .................. 119
      8.1.2 Model repository update .......................... 120
      8.1.3 Simulator wrapper ................................ 122
      8.1.4 Scenario .......................................... 122
   8.2 Modeling ............................................... 122
   8.3 Collaboration .......................................... 122
   8.4 Online simulation ...................................... 124
   8.5 Management of simulation resources .................. 125
      8.5.1 The Weighbridge benchmark ...................... 126
      8.5.2 Simulated vs. actual operation ................ 129
   8.6 Summary ............................................... 130

9 Conclusions ................................................... 131
   9.1 Summary ............................................... 131
   9.2 Contributions ........................................ 132
   9.3 Limitations ............................................ 133
   9.4 Possible directions for further research ............ 134

Bibliography .................................................. 135

Curriculum Vitae ............................................... 145
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AcMoS</td>
<td>An instance of the cMoS framework (a cMoS)</td>
</tr>
<tr>
<td>AToM³</td>
<td>A Tool for Multi-formalism and Meta-Modeling</td>
</tr>
<tr>
<td>cMoS</td>
<td>the Collaborative Modeling and Online Simulation framework</td>
</tr>
<tr>
<td>DEVS</td>
<td>Discrete Event System Specification</td>
</tr>
<tr>
<td>DFD</td>
<td>Data Flow Diagrams</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain Specific Language</td>
</tr>
<tr>
<td>DSM</td>
<td>Domain Specific Modeling</td>
</tr>
<tr>
<td>DSME</td>
<td>Domain Specific Modeling Environment</td>
</tr>
<tr>
<td>DTR</td>
<td>Dynamic Traffic Routing</td>
</tr>
<tr>
<td>LHS</td>
<td>Left Hand Side</td>
</tr>
<tr>
<td>MDA</td>
<td>Model Driven Architecture</td>
</tr>
<tr>
<td>MDE</td>
<td>Model Driven Engineering</td>
</tr>
<tr>
<td>MOF</td>
<td>Meta-Object Facility</td>
</tr>
<tr>
<td>MRA</td>
<td>Metadata for Resource Access</td>
</tr>
<tr>
<td>MRD</td>
<td>Metadata for Resource Discovery</td>
</tr>
<tr>
<td>M&amp;S</td>
<td>Modeling and Simulation</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>PI</td>
<td>Process Interaction</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform Independent Model</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform Specific Model</td>
</tr>
<tr>
<td>RHS</td>
<td>Right Hand Side</td>
</tr>
<tr>
<td>SCD</td>
<td>Simplified Class Diagrams</td>
</tr>
<tr>
<td>ZCase</td>
<td>Zope Computer-Aided Software Engineering</td>
</tr>
</tbody>
</table>
Chapter 1
Introduction

The emergence of the World-Wide Web (WWW) and its popularity in the simulation community gave birth to the concept of Web-Based Modeling & Simulation [28]. In [85], Sarjoughian and Zeigler attributed the popularity of this phenomenon to proliferation of organizations that required Modeling and Simulation (M&S) environments in support of their development activities and to the fact that modeling of development artefacts required involvement of many individuals. The implication of the latter is that traditional single user M&S tools will have to be replaced or supplemented with more powerful tools supporting collaborative practices of users.

In [29], Healy observed that traditional M&S tools were often difficult to modify or extend for the web context, thereby stimulating research and development of new tools. This need was answered by research communities. To witness the amount and scope of the produced work, one only needs to browse proceedings of many M&S conferences devoted to the subject, for example [30]. In [75], Page classified the wide range of research and development efforts into five areas: simulation as hypermedia, simulation research methodology, simulation of the WWW, web-based access to simulation programs and distributed modeling and simulation. The application domain of the work presented in this thesis belongs to the last two areas and concerns remote execution of existing simulations, as well as frameworks, tools and environments to support the distributed, collaborative development of simulation models.

Prior to the start of this research, the application domain already counted numerous contributions as documented in [24, 42, 61, 87, 3, 20, 85, 90]. A key approach in many contributions was the role that web-enabling technologies, such as Java language, played both for specification and implementation of the model. The powerful feature set of the Java language allowed to implement special purpose simulation modeling features directly in the Java languages so that modeling and programming language became the same. However, this web-based approach invoked a number of disadvantages: First, modeling languages were replaced by a relatively low-level, general purpose programming language. This neglected the decades long evolution of modeling practices from using general purpose languages to special purpose simulation languages, to higher level model specification languages. The result is decreased productivity of modellers and increased semantic gap between possibly multiple modeling paradigms and the web-enabling technology used for modeling. Furthermore, intertwining of specification and implementation made separation of a model from its simulator harder, thereby reducing portability of
CHAPTER I. INTRODUCTION

models [100, page 30]. The implication was that such models could only be reused with other Java models and executed on one simulator, the Java platform.

The goal of this work is a web-based collaborative environment that could be easily tailored to modeling and simulation needs of an arbitrary number of M&S application domains. The work was started due to inflexibility of available solutions and their weak support for model specification languages required in the NanoComp project [70] that investigated the feasibility of future electronics based on quantum devices. Research and development of complex systems often requires a specific combination of problem solving paradigms. The proposed approach and our original application of Model-Driven Engineering (MDE) enables collaborative users to work with an arbitrary number of preferred modeling paradigms, share models and automatically generate model implementations for simulation on-line or off-line. Direct support for modeling paradigms used by end users results in effective and productive modeling. Perhaps more important than availability of proper paradigms is possibility of combining those paradigms into multi-paradigm solutions. At last, but not least, separation of models from simulators gives users freedom of executing models with simulators of their choice. As complexity of engineered systems continuously increases, the use of M&S in development of such systems and demand for powerful and collaborative M&S environments such as in the focus of this work, are expected to grow.

1.1 Research problem

The problem addressed in this research is:

How to achieve broad applicability of a Web-Based Modeling & Simulation environment and efficiently customize such environment to a possibly arbitrary number of M&S application domains?

Essentially, we argue that broad applicability can be achieved by basing the environment on a general framework that includes fundamental concepts from contributing fields of Modeling & Simulation, Distributed Systems and Collaborative Software. To enable efficient tailoring to specific applications during the operation phase of the environment’s life-cycle, we identify variation points in the framework’s concepts. Customization of the identified points is still limited due to a high barrier of their implementation. Recent advances in model-based development provide a promising approach to this complex problem by lifting the abstraction level of system specifications and enabling automation of development activities. Therefore these points are not implicitly hardcoded as traditionally, but are explicitly expressed as computerized models and their implementation is automatically generated. The end result is a model-driven and mostly programming free eMoS system adaptable to new M&S applications through abstract modeling of the variation points.

The research problem is important for convergence of the fields of Modeling & Simulation and World Wide Web (WWW). The emergence of WWW has stirred academical, governmental and industrial interest in connecting computer modeling and simulation methodology and applications with the Web. Numerous projects have been initiated and contributed toward first results in the Web-Based Modeling and Simulation. These
results however were ad hoc solutions based on merging M&S with Web technologies. A typical application would consist of cementing its ingredients: a modeling paradigm, simulators and collaborative aspects with particular Web technologies. Should have any of these ingredients be different in a new application, such hard coded paradigms, collaboration aspects and simulators could not be easily changed or customized. Given the multitude of modeling paradigms, plethora of modeling languages and simulators, it is neither feasible nor desirable to manually program web-based collaborative environments per each possible M&S application domain or combination thereof. An alternative is a general environment that can be customized to new M&S domains. Such a generally applicable solution needs to be framed around general and stable concepts from related bodies of theories: Modeling & Simulation, Distributed Systems and Collaborative Software. Therefore the first research issue of this thesis is:

What are the frameworks and general concepts in contributing fields of Web-Based Modeling and Simulation?

Information to answer the above question will be gathered in Chapter 2.

General frameworks and concepts from related fields are necessary, but not sufficient for a meaningful use in numerous M&S domains that exist. To give an example, a general concept of model refers to a representation of a system under study. Such a system can be represented under different modeling paradigms. Each paradigm having its own concepts, rules, modeling languages, tools and established base of users. Therefore the second research issue is:

How can a general Web-Based M&S environment be customized to a particular M&S domain?

This research issue is developed in section 5.3.

Finally, the environment needs to be developed and repeatedly customized. Modern software systems are characterised by high complexity, while software engineering methods by low productivity, thus leading to high development costs. Moreover, development of environments such as in the focus of this thesis, suffers from additional complexity associated with the web platforms. For the proposed solution to be effective, it would need to be easily developed and especially customized. Model-Driven Engineering (MDE) is a general term that refers to emerging model-based approaches to engineering. MDE is widely discussed and is currently considered as a promising solution to complexity and productivity problems in engineering disciplines. However, entering MDE is not easy as the subject literature is full of weak definitions, biased statements, conflicting experiences and simply commercial hype. Naturally, questions arise:

What is the essence of Model Driven Engineering (MDE)?

How MDE can be used to develop a solution to the research problem?

The above two issues are researched in Chapter 3 and section 5.4 respectively.
CHAPTER 1. INTRODUCTION

1.2 Approach

To achieve broad applicability of our solution to the research problem, we study general concepts and taxonomies in fields of Modeling and Simulation, Distributed Systems, and Collaborative Software to identify fundamental concepts. Based on the identified and selected findings, this thesis yields an adaptation of the classic Modeling and Simulation (M&S) framework to the web and collaborative context. This framework, called cMoS (collaborative Modeling and online Simulation) is a conceptual model of the domain of Web-Based M&S applications. Models such as this framework, provide software engineers with a proper focus on the problem domain so that resulting software would fit harmoniously with the domain it is being developed for [25]. To deal with complexity of developing a non-trivial web system such as software incarnations of cMoS are, we employ the emerging approach of model-driven engineering.

Specific M&S applications are supported through customization of the variation points in cMoS concepts during the operation phase of the environment. To enable efficient tailoring, the variation points are not implicitly hardcoded as traditionally, but are identified and explicitly expressed as models. To achieve this, we base the generic cMoS system (AcMoS) on the classical four layer metadata architecture [38]. The meta-architecture formulates modeling artefacts and customization activities, which are essential to application domain customization and AcMoS operation. Application of model-driven engineering to this meta-architecture enables efficient and programming-free adaptation of cMoS systems to new Web-Based M&S applications.

To reduce complexity and increase efficiency of developing and customizing a non-trivial web system such as software incarnations of cMoS are, we employ the emerging approach of using models as primary engineering artefacts. We study the general field of Model-Driven Engineering (MDE) to establish the fundamental principles and concepts and then proceed into well established and fixed approaches that realize the MDE ideas. These fundamental definitions are used to uniformly evaluate mainstream approaches in order to select existing or propose new model-driven development processes.

In order to demonstrate the proposed solution to the research problem, we create appropriate MDE assets for the proposed development and customization processes. Consequently, the development infrastructure and model-driven processes are used to develop a software instance of cMoS. Finally we validate our approach in practice: Application domain customization is demonstrated for a number of Modeling and Simulation domains: Data Flow Diagrams (DFD), Discrete Event System Specification (DEVS), Process Interaction (PI) and Dynamic Traffic Routing (DTR). Generic cMoS functionality, such as modeling, collaborative sharing of model specifications and implementations, online simulation and management of shared simulation resources is demonstrated as well.

1.3 Definitions

Definitions adopted by researchers are often not uniform, so key and controversial terms are defined to establish positions taken in this PhD research.
**CHAPTER 1. INTRODUCTION**

**Model-Driven Engineering** refers to systematic use of models as primary engineering artefacts throughout the engineering life cycle [98]. MDE is defined by Kent [44] as a generalization of MDA [45] that includes the notion of software development process. MDE is an open approach that embraces various technological domains in uniform way. In our view, other model-oriented initiatives, e.g. Model Driven Architecture (MDA), Domain Specific Modeling (DSM), Model Integrated Computing (MIC), Model Driven Software Development (MDSD) and Model Driven Development (MDD), are instances of MDE.

**Metamodel** can be defined as conceptual model of a modeling technique [9]. This definition encompasses two types of metamodels [93, page 99]: i) *meta-data* models describe system development representations as in [45, 94, 72] and ii) *meta-process* models that describe system development processes of a technique as in [1]. This research primarily deals with the meta-data type of metamodels.

### 1.4 Delimitations

In this thesis, fields of Distributed Systems and Collaborative Software constitute secondary parent fields and are not in the focus of this research. We do not seek to contribute to these fields, but study them to identify established concepts and taxonomies. We study the field of distributed systems, to identify key characteristics of such systems. This knowledge is applied when designing and implementing a solution to the research problem. In the field of collaborative systems we are interested in classifications of typical business needs that such systems address. When demonstrating practical results of this research, existing tools and systems from the secondary fields are used.

Another delimitation concerns reuse of models. While model reuse is a part of the distributed, collaborative modeling, the thesis aims to achieve portability of models in a heterogeneous environment and does not make provisions to ensure that shared models are reusable.

### 1.5 Contributions

Answering the research issues from section 1.1 yielded contributions, which will be presented in Section 5.2. In summary, from this thesis research came the following primary contributions.

Firstly, an adaptation of the classic modeling and simulation (M&S) framework to the domain of web-based collaborative applications is proposed [57]. In Chapter 5, a framework is proposed that combines general concepts and taxonomies from fields of M&S, Distributed Systems and Collaborative Software. This framework explicitly describes fundamental concepts that are invariant for the broad family of web-based modeling and simulation applications. The framework provides software engineers with a proper focus on the problem domain so that resulting software would fit harmoniously with the domain it is being developed for.
CHAPTER 1. INTRODUCTION

Secondly, a model-driven meta-architecture is proposed [57], which enables efficient customization of application domain independent cMoS instances. Section 5.3 identifies points of variation in the cMoS framework. Through customization of these points, cMoS instances are tailored to new M&S applications. To achieve flexibility and ease of customization, we base organization of such points on the four level meta-architecture. Hence, the essential variation points together with affected cMoS concepts and modeling levels constitute the AcMoS meta-architecture (refer to section 5.3). Application domain customization is demonstrated in chapter 8.

Thirdly, two model-driven development processes are proposed and demonstrated [57]. These processes together with MDE reduce difficulty of developing and more importantly, customizing cMoS instances. Chapter 6 makes possible technology choices and illustrates how a custom MDE development infrastructure is created to support these processes. In Chapter 7, both the infrastructure and processes are applied to create a cMoS instance. Customization of the instance is illustrated in section 8.1.

Fourthly, MDE was successfully applied to model languages for a number of M&S formalisms and implementation technologies [54, 55, 57, 53]. These contributions are described in Chapter 4 and section 6.2 respectively. The produced metamodels and transformation definitions are released in public domain under open source licenses.

Fifthly, control of access to shared simulators was explicitly modelled as abstract, formal and executable transformation definition in the DSL interpretation approach [57]. Hereby, system development and modification occurs at the highest possible level of abstraction, without resorting to software development. Furthermore, applying such transformation to a model, visualizes the behaviour by manipulating and animating model elements. Modeling of behaviour is demonstrated in section 7.11.

In this research, experience showed that MDE tools can be extended with new functionality by means of multi-formalism modeling, meta-modeling and model transformation: meta-modeling allows to create user interface (UI) elements for a new functionality. These elements enable users to provide input parameters if necessary. Multi-formalism modeling allows to mix custom UI elements with a source model in a different formalism. Finally, transformations can be used to specify the behaviour of the new functionality. For example, section 7.12.1 describes how a functionality missing in the cMoS context was added to tool AToM³.

1.6 Outline of this thesis

This thesis is presented following the structure and style guidelines from [77]. There are five parts, each containing one or more chapters. The parts, chapters and relations among them are illustrated in Figure 1.1.
Part 1 lays the foundations for the thesis: Chapter 1 introduces the research problem, research questions and hypotheses. Then the research is justified, the approach is briefly described, definitions and delimitations are given and the thesis is outlined. On these foundations, the thesis can proceed with a detailed description of the research.

This research is about engineering web-based modeling and simulation applications with help of models. Consequently, there are two major topics in Part 2: the problem domain of the M&S applications and model-driven engineering. Chapter 2 describes general concepts in the primary parent field of Modeling and Simulation and the secondary fields of Distributed Systems and Collaborative Software. Chapter 3 introduces the emerging field of Model-Driven Engineering (MDE). Two most used and mature MDE approaches (MDA and DSM) are described in a uniform way that allows to identify their advantages and disadvantages. Concepts and frameworks from the related fields are organised into classifications that will support discussions in further chapters.

Chapter 4 defines a number of Modeling and Simulation languages by means of metamodels and transformation definitions. These practical results are presented so early due to the multiple roles they play in this thesis: Firstly, they serve as practical illustration of the MDE concepts introduced in Chapter 3. Secondly, one of the illustratively defined languages (the Process Interaction language) is used as basis for controlling access to shared simulators as described in Chapter 7. Finally, the presented metamodels and
transformation definitions play role in application domain definition, which is described in Chapter 8.

Part 3 describes the approach of this thesis. Chapter 5 documents how the focus of this research shifted from code-driven to model-driven development. This chapter establishes a framework for collaborative Modeling and online Simulation (cMoS) applications. Another subject is a model-driven meta-architecture that expresses application domain customization via essential variation points of the cMoS framework. Two model-driven development processes are proposed to reduce complexity and increase productivity of developing and customizing cMoS instances. These processes are based on the fundamentals of the Model-Driven Engineering and best practises of the outstanding MDE approaches.

The methodological path from the abstract cMoS framework to a developed environment is illustratively exemplified in Part 4: In Chapter 6, a development infrastructure to support the model-driven development processes is created. Chapter 7 demonstrates how a software instance of the cMoS framework (see section 5.2) is produced with the model driven development processes. Application domain customization of a given cMoS instance and general cMoS functionality like modeling, model sharing, online simulation and management of simulation resources are demonstrated in Chapter 8.

Finally, Chapter 9 of Part 5 concludes the described research, discusses contributions and outlines possible directions for further research.
Chapter 2

Problem domain

The problem domain in focus of this research encompasses the primary parent field of Modeling and Simulation and the secondary fields of Distributed Systems and Collaborative Software. This chapter introduces parent fields, describes related concepts, frameworks and organises this knowledge into classifications that will support discussions in further chapters. The chapter starts with an overview of the multi-disciplinary field of Modeling & Simulation in section 2.1. Next, the secondary fields are presented in sections 2.2 and 2.3 respectively.

2.1 Modeling and Simulation

Modeling and simulation (M&S) strives to solve the problem of understanding and predicting behaviour of systems. Disciplines like mathematics, computer science, cognitive sciences and a variety of application domains have been developing their own specific techniques to solve this problem. In 1976, Zeigler [99] attempted to unify these fragmented research results into a general theory. It provided a framework that isolated and abstracted common concepts in a form applicable to all disciplines and domains. Nowadays, M&S is recognised as a separate inter-disciplinary research area.

This section considers M&S from the viewpoint of the general theory by Zeigler [100]. The basic modeling and simulation concepts and connections between them are presented in Figure 2.1.

![Figure 2.1: The Modeling and Simulation framework](image-url)
CHAPTER 2. PROBLEM DOMAIN

System is a part of the real or virtual environment that is target of modeling. Typically the system is defined under specific conditions in order to meet objectives of a modeling and simulation project. This specification of conditions reflects objectives of the experimenter and is known as experimental frame (not shown in Figure 2.1).

Model is an abstract representation or specification of the system at different levels of knowledge. In the M&S context, a model is typically a higher level specification that includes knowledge about behaviour and internal structure. An important feature of a model is that it is a simplified representation, which allows the modeller to understand and reason about complex systems.

Simulator is a computation system capable of executing model instructions to generate model behaviour. Possible examples are an algorithm, the human mind, a computer. Experimentation (not shown in the framework) concerns setup and execution of a model with a solver. Forms of simulation experimentation:
Interactive experimentation involves running a simulation and making changes to the model to observe the effect. Batch experiment involves setting the experimental factors and letting a simulation to run without interacting with the model. Batch experiment can run for a predefined run-length or until a specific event and for a set number of replications.

Modeling and simulation are the primary relations among entities of the M&S framework:

Modeling is the act of making a model. Other important relationships are validation and verification. Validation is a basic modeling relationship which answers the question if a model accurately represents its system counterpart in the experimental frame of interest [100]. Verification concerns the correctness of transforming an abstract representation of a model (the conceptual model) into an executable representation (the simulation model).

Simulation is the process of executing a model. Robinson [80] defines simulation in its most general sense as imitation of the working a system. The execution produces dynamic output behaviour according to a (simulation) model. Simulation primarily deals with dynamic models.

Models can take a number of forms: A conceptual model is a non-software specific specification of a system. Robinson [80] considers a conceptual model to be equal to the concept of lumped model in [99]. A simulation model is a model implemented in the software in which simulation is to be developed. For a single conceptual model, many simulation models can be created. This allows to select the most appropriate simulation software on basis of understanding the conceptual model and allows to separate the intellectual property of models from technology-specific code, thus fostering portability of models. Some of the model classifications are described in section 5.2.7.

2.2 Distributed systems

A distributed system is a collection of autonomous interconnected computers or devices integrated by software to produce a computing facility.
An example of a distributed system is given in Figure 2.2. The illustrated distributed system provides users with resources: printers, storage media, files, databases. In general, *resource* is hardware, software or data component of a computer system with limited availability. This definition is deliberately kept abstract, as it has to encompass the multitude of possible things that can be shared in a distributed system. Another commodity is *service*, an abstract entity that may be provided by one or more processes running on separate computers and cooperating via a network. Term *process* is used in the sense of operating systems and means a running program. From the object oriented view, resources, processes and even entities in processes can be referred to as *objects*. Objects are given *names* or *identifiers*. The difference between these is that the former can be interpreted by users or programs, while the latter are used by programs only.

Usefulness of distributed systems is often characterised with six key qualities described in section 2.2.1. These often form generic design requirements. The Internet, a particular example of a distributed system is introduced in section 2.2.2.

2.2.1 Key characteristics

One inherent property that follows from the definition of distributed systems is separation of objects. This property underpins key characteristics commonly associated with distributed systems. Couloris [17] identifies six characteristics:

**Resource sharing** allows to make a more efficient use of available resources. Sharing resources raises need for resource management. *Resource manager* is a term that refers to a piece of software that is responsible for management of resources of a certain type. Common management responsibilities include mapping resource names to communication addresses and access synchronization to ensure consistency of
states of shared resources. Moreover, each type of resources may require additional management methods.

Concurrency occurs when \( N \) processes exist at the same time. In a system with 1 processor, concurrency is achieved by interleaving execution of portions of the co-existing processes. In a system with \( M \) processors, \( N \) mutually independent processes can be executed in parallel if \( M \geq N \). This presents an \( N \)-fold performance increase in computation. Distributed systems, by their nature, have potential to achieve benefits of parallel execution of concurrent processes. Opportunities arise from user separation, independence of resources and de-centralized service provision. Concurrency further increases the need for resource management.

Fault tolerance characterises capability to detect system faults and recover from them. There are two major approaches: hardware redundancy and software recovery. Distributed systems allow to reduce the cost of hardware redundancy due to resource sharing.

Openness characterises capability of a computer system to be extended. This characteristic can be applied to both the hardware and software levels. Under the context of distributed systems, openness is defined by the degree of disruption to or duplication of existing services when new resource-sharing services are added. Documentation and publishing of key interfaces, standardization in system integration, availability of general interprocess communication increase openness of systems.

Scalability characterises whether a system or application software needs to change as the scale grows. The scale of distributed systems can range from as few as two computers to thousands computers in internetworks. Every aspect of a distributed system - shared memory, processors, etc. - may affect scalability. Therefore an effective scalable system should be designed with assumption that every such aspect can be extended. The challenges of such design increase as the size and complexity of networks grows.

Transparency is a property of hiding from application users or developers separated components of a distributed system as one whole. The Reference Model for Open Distributed Processing (RM-ODP) [79] distinguishes eight forms of distribution transparency:

- Access transparency hides the difference between accessing local and remote objects.
- Failure transparency hides faults from users and applications, thus allowing them to complete their tasks.
- Location transparency allows to access objects without knowing their location.
- Migration transparency allows moving service providers from one location to another without affecting the provision of the service.
- Persistence transparency provides persistent resources and objects.
CHAPTER 2. PROBLEM DOMAIN

Figure 2.3: The Internet and OSI structures

- *Relocation transparency* hides the relocation of a service from clients using it.
- *Replication transparency* hides existence of multiple copies of a service from users of the service.
- *Transaction transparency* hides the effects of overlapped or concurrent execution, maintaining consistency among the involved objects.

Access and location transparencies, combined known as *network transparency*, most strongly affect usage of distributed resources and are the most important forms of transparency.

The primary 6 characteristics are not given, but rather gained through proper design. These characteristics form generic design requirements for distributed systems.

2.2.2 The Internet

The Internet is a distributed system of interconnected computer networks that use the Internet protocol suite to form a global infrastructure to deliver information and services. The Internet itself stands on three cornerstone infrastructures: *physical, contractual* and *communicational*. The first two are responsible for physical interconnections of existing networks and for agreements that regulate exchange of data traffic among the networks. The latter is a set of protocols that transfers data between computers and applications.

The Internet is a particular example of a *wide area internetwork*. The ideas that lead to the Internet originated from the research and development of national computer networks in the 1970s. The Internet in particular, takes its origin from ARPANET [83], the first large scale computer network developed by ARPA of the U.S. Department of Defense.

The Internet is designed around a layered architecture, which theoretically allows a technology within a layer to be replaced without affecting the other layers. Figure 2.3 illustrates how the Internet architecture compares to the Open Systems Interconnection (OSI) Reference Model, a guideline proposed by the International Organization for Standardization (ISO) to design network systems that are open for communication with other systems. The bottom four layers in the figure provide the basic internetworking communications service - this is the domain of networking engineers. Developers of distributed
Table 2.1: Internet protocol suite

<table>
<thead>
<tr>
<th>Layer</th>
<th>Protocols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>http, ftp, pop3, smtp, ssh, imap, ...</td>
</tr>
<tr>
<td>Transport</td>
<td>TCP, UDP, DCCP, SCTP, ...</td>
</tr>
<tr>
<td>Network</td>
<td>IP, ICMP, IGMP, ARP, ...</td>
</tr>
<tr>
<td>Link</td>
<td>ATM, Ethernet, PPP, SLIP, ...</td>
</tr>
</tbody>
</table>

Table 2.2: The Internet and key characteristics of distributed systems

<table>
<thead>
<tr>
<th></th>
<th>DNS</th>
<th>TCP/IP</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource Sharing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Openness</td>
<td></td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Concurrency</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalability</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Fault Tolerance</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transparency</td>
<td>√</td>
<td></td>
<td>√</td>
</tr>
</tbody>
</table>

systems are primarily concerned with the layers above where communication requirements of specific applications are implemented.

Table 2.1 illustrates the Internet protocol suite and their relation to the layers of the Internet architecture. An important part of the suite is the combined set of TCP (Transmission Control Protocol) and IP (Internet Protocol) protocols. The Internet protocol suite was designed to be independent of the underlying physical medium, which allowed the Internet traffic to be carried across any communication networks and led to wide adoption of the suit.

With the advent of broadband telecommunication networks, the Internet is used as basis for distributed services, applications and systems. Table 2.2 illustrates contributions of the Internet towards usefulness characteristics of distributed systems that are built atop. The same table includes Domain Name System (DNS), a naming solution for the Internet devised by Mockapetris [83, page 171]. DNS is a crucial service that is used across the Internet by applications and services.

Many applications and services exist that are built on top of the Internet transport layer. The two best-known Internet services are electronic mail (e-mail) and the World Wide Web (WWW). Other popular services include file sharing, Usenet newsgroups, Instant Messenger, Gopher, IRC, to name a few. Moreover, new services can be built upon existing services.

WWW, also known as the Web, is worth a special mentioning because it is so ubiquitous to a casual user that the Web became synonymous with the Internet itself. WWW is a global hypermedia information system. It is a collection of documents/pages stored as files on computers distributed around the Internet. Web servers serve these documents to users browsing this space with help of web clients known as browser. The browser uses the HTTP application-level protocol to access and transfer a web page over a TCP connection with a web server. Such pages are rich documents written in HTML and can
CHAPTER 2. PROBLEM DOMAIN

contain multimedia. HTML defines contents, structure and representation of documents and allows to embed links to other pages stored elsewhere on the Web. Each such linking, known as hyperlink, contains a full totality of information (protocol, symbolic Internet name of the server machine, pathname and filename) needed by browser to access the contents of the linked page over the Internet. DNS is used to translate the symbolic Internet name into the address of the server machine.

Today (2006) the Internet includes thousands of smaller commercial, academic, domestic, and government networks and counts over 1 billion of users. It stimulated development of new areas of human activities and had serious impact on how people work and recreate. For an up to date overview of the Internet, we refer the reader to one of the many available guides [14, 40].

2.3 Collaborative software

Collaborative software or groupware is application software that integrates activities of multiple concurrent users within a project.

Research in collaboration has resulted in a large number of tools and environments [97]. A number of classification frameworks exists that help to organize these contributions. Perhaps the most known groupware typology is based on space and time categorization and was proposed by Grudin [39]. In [81], basic distribution architectures and possible subtypes are distinguished to help choosing the best groupware platform for a given collaboration scenario. Malone [60] classified collaborative tools according to their co-ordination capabilities that can be used to manage the most common dependencies in collaboration activities.

The above mentioned classifications focus on the approaches in groupware development or mechanics of collaboration. In the context of this research a more user-oriented classification based on business needs of clients was called for. Sarma [86] developed a hierarchy of collaboration needs according to which existing or to-be-developed groupware can be categorized. In a more abstract framework, Butler and Coleman [10] suggested five empirically derived models of collaboration needs based on interactivity and group size. The latter classification matches better the goals of this research with respect to collaboration (see section 1.4).

Butler and Coleman suggested that the majority of collaborative environments support one or more of the five primary collaboration models:

Library is a common model that is characterised by long-lasting content, many readers and few writers, weak relation between readers and writers, content management.

Solicitation concerns interactions between a small set of requesters and potentially multiple respondents.

Team supports activities of interdependent, controlled, relatively small group of readers and writers.

Community is a relatively large, loose and moderated group that prefers reading over writing.
CHAPTER 2. PROBLEM DOMAIN

Process Support concerns automation of processes and workflows through use of technology.

Figure 2.4 illustrates how these models are related to each other in the typology space. These five collaboration models, pure or combined, provide developers with a framework to uncover how new systems designs and available solutions can contribute to collaborative needs of a client.

2.4 Summary

This chapter introduced fields related to the problem domain of the research. The basic Modeling and Simulation concepts were introduced. The key characteristics of distributed systems provide developers with a framework to see how chosen solutions and designs contribute to properties of the expected system. Finally, a collaboration classification useful for the goals of the research was identified.
Chapter 3
Model-Driven Engineering

This chapter is a guide into the field of Model-Driven Engineering. Before MDE can be introduced, we have to understand its context. Section 3.1.1 briefly describes the software development process and typical problems occurring in software development. Next section number 3.1 studies definitions of MDE available in the literature and identifies the fundamental ideas behind MDE. The following sections discuss concepts that are crucial to realization of the MDE ideas: the modeling architecture, modeling languages as system and transformations. A number of approaches exist that put the MDE ideas into practice. Two most prominent, mature and often contrasted approaches are described in section 3.3.

3.1 What is MDE?

To answer the question, we need to understand the engineering process MDE applies to. This thesis primarily deals with the discipline of software engineering, therefore software engineering process is discussed below. Given the perspective on the process, we can outline the types of problems that software projects deal with. Understanding these problems helps to identify potential benefits of and justify the cornerstone ideas behind MDE. Next, we provide a number of definitions of MDE found in the literature. These are analysed and essential ideas are distilled into an answer to the question posed in the title of this section.

3.1.1 Software development process

There are several models of development process in existence [32, chapter 2]. Instead of describing each individual model, we propose a model of concepts essential to models of software development processes. To avoid a possible confusion with the MDE concept of metamodel, we refer to the proposed model as megamodel. This model is a simplified, yet sufficiently accurate representation of the software development process that captures concepts essential to building a software system. The proposed megamodel i) assumes that the iterative/waterfall dichotomy is the most important division among all software development processes and ii) reduces the scope of the model by excluding activities
that do not directly produce system artefacts, e.g. initial planning, testing, deployment. Figure 3.1 presents the megamodel as UML class diagram.

In the focus of the megamodel is concept Process. Process structures a software development Project. Project is a planned activity that delivers one or more increasingly detailed software artefacts that specify System under development. Code is the typical target of a project as it can be executed, thus instantiating a system and providing a set of Functionalities requested by Stakeholders. System is implemented for a particular Platform, a general framework, either in hardware or software, which allows system to operate. Notice the cardinality of the relationship between System and Platform: in general, software projects have to implement software for multiple platforms. Strongly related to engineering is notion of domain. Analysis of definitions of domain in [18] reveals that domain is "a body of knowledge organized around some focus.,” further identifying that such focus lies in a real world area of expertise and in expertise of building software systems (or parts thereof) for that area. In line with this view, system is a software solution to a problem in a so-called problem domain, e.g. virtual environments, insurance systems, M&S. System itself consists of parts which correspond to solutions domains such as database systems or numerical libraries.

Furthermore, to reduce complexity and to better track development progress, project breaks up the task of software development into smaller chunks. There are two major approaches to how one breaks up a project. These are known as the Waterfall and Iterative process models. The waterfall approach breaks down a project based on activity. Alternatively, the iterative approach breaks down a project on the basis of subsets of
CHAPTER 3. MODEL-DRIVEN ENGINEERING

functionality. Note that to implement a subset of functionality a complete production activity is always required. This is reflected in the cardinality of the activity-project relationship. Given a constant activity, breakdown is achieved by decomposing project into a number of smaller projects or iterations. The resulting sequence is ordered by priorities of functionalities. Examples of the iterative process model are RUP (Rational Unified Process) and Agile processes. It follows that an iterative project with only one iteration is a waterfall project. Both models go through the same activities in order to build a system. Combination of both models is also possible as in the staged delivery process.

Next, we turn to the breakdown of the activity that produces system artefacts. The result of the breakdown is a sequence of sub-activities that are called steps, stages, phases or components depending on the development method. The steps are ordered along the abstraction dimension: each step produces an artefact, which is a refinement of the system specification received from the previous step. Figure 3.2 illustrates how a sequence of common sub-activities relates to different levels of abstraction (the higher, the more abstract). The abstraction pyramid is a metaphor for reduction of complexity with increase of abstraction (upward). Due to the simplification of this megamodel, the illustrated breakdown of concept activity includes only tasks that directly produce system artefacts: Analysis involves activities to transform initial business requirements from the problem domain into formal requirements for software systems. Design is a task of precisely specifying the software to meet the requirements. Coding step involves translating software specifications into executable code. In design or coding, a choice of the platform is made and technological and engineering details that are irrelevant to the fundamental functionality of a software are added to make the system executable.

Finally, each development activity carries costs. Notice how in Figure 3.2 complexity and consequently cost of development are increasing as the system approaches its com-
CHAPTER 3. MODEL-DRIVEN ENGINEERING

Figure 3.3: Relative cost to fix an error over life-cycle (adapted from [6])

Completeness. A further insight can be found in economics of correcting errors during system development. In [6] Barry Boehm produced some compelling evidence showing that the cost of correcting an error increases dramatically the later in the system development it is detected and corrected. Figure 3.3 plots the relative cost of fixing an error in requirements as function of the development step. This evidence illustrates that making changes early in the development process is cheap, but becomes extremely expensive in the later stages.

3.1.2 Problems of software development

Section 3.1.1 described the structure of the software development process, activities and associated levels of complexity and costs. The following is a description of the most typical problems faced in software development.

Semantic gap. Characterizes the difference between two descriptions of an object by different linguistic representations. For example, the gap occurs whenever ordinary human activities, observations, and tasks are transferred into a computational representation. Mapping from real world applications into computer applications cannot be automated and requires technical background knowledge. An example of semantic gap manifestation is the difficulty in understanding the requirements for a software system [32]. Consequences of this problem can be seen in Figure 3.3.

Complexity. Complexity of systems grows faster than abstraction levels provided by development means of engineers, such as platforms and programming languages.
CHAPTER 3. MODEL-DRIVEN ENGINEERING

Productivity problem. Software development is focused on low-level design and coding: writing code is perceived as productive and writing (high-level) documentation is not [45].

Documentation and maintenance. Low- and especially high-level documentation is separate from code. As we have seen in the productivity problem, writing code is being productive and writing documentation is not. Moreover, maintaining documentation in sync with code is a costly and manual process. Consequently, the lack of incentive and required significant effort are the reasons why documentation is often of poor quality and outdated. The documentation problem makes maintenance and change of systems difficult.

Portability. Software industry is characterized by rapidly changing technologies: new technologies and new versions of technologies appear every year [45]. Companies are forced to chase these changes even if solutions do not need to change. To illustrate this point, note the cardinality of the input and output of the design step in Figure 3.2. Migration of existing solutions to new technologies is extremely costly due to the documentation and complexity problems. Moreover, investments in obsolete technologies and systems based on those technologies loose value.

Interoperability. Software systems are often not monolithic: a system may need to interact with other systems, a system may span multiple technologies or be assembled by reusing already existing components. In such cases, (sub-) systems based on different technologies may need to interact with each other, thereby creating need for interoperability.

3.1.3 Definitions of MDE

MDE and model driven approaches, standards and technologies have gathered a lot of attention, numerous articles, books and thousands of pages of documentation. Despite the growing amount of information, Favre [26] observes that the true essence of MDE is not described there and MDE core concepts are not defined precisely. In the following we provide a number of definitions of MDE found in the literature.

The Wikipedia defines MDE as "systematic use of models as primary engineering artefacts throughout the engineering life cycle" [98]. This definition stresses the central role of models, which is in contrast to the traditional development where models are second class artefacts.

Kent [44] defines MDE on the basis of Model Driven Architecture (MDA) concepts by adding notion of software development process and extending model space beyond the abstraction dimension of the PIM-PSM classification of MDA. In [44, page 10], Kent summarises that "a model driven engineering approach must specify the modeling languages, models, translations between models and languages, and the process used to coordinate the construction and evolution of the models". Moreover, powerful tool support is required if the burden of maintaining models in line with code were not to outweigh the benefits of models.
MDE definition would be incomplete without defining MDA. MDA is a particular approach by the Object Management Group (OMG) to using models in software development. In [15], Steve Cook refers to MDA as standardized approach that is based on abstraction of similarities in technological and engineering platforms and relies on the OMG’s modeling technologies, most notably the Unified Modeling Language (UML) and the Meta-Object Facility (MOF). To develop a system, first a model of the system is built that is abstract from technological details. Such model can be ported to multiple technological platforms. Then this model is transformed into one or more models specific to the chosen platforms. In turn, these models are again transformed into the final code. In MDA, transformations are performed by a human or an automated algorithm [74], though the latter is preferred.

Jean-Marie Favre [26] observes that the true essence of MDE concepts is either poorly described in literature or is obscured by the bias towards the particular technological orientation of OMG. The author’s view is that MDE is a more broad and open approach than MDA and accommodates many technological spaces in a uniform way. In this vision, MDA is just a particular incarnation of MDE implemented in the set of technologies defined by OMG.

Noteworthy is the vision by prominent MDA contributors from IBM: Grady Booch, Alan Brown, Sridhar伊genres, Jim Rumbaugh and Bran Selic. In their MDA Manifesto [8], the authors outline three complementary ideas that form the essence of MDA: direct representation, automation and open standards. The first idea recognizes that in order to lessen the semantic gap, the focus of software development has to be elevated from technology domains towards concepts of the problem domain. Automation refers to mechanization of those development tasks that do not require human intelligence. The most important goal is to bridge the gap that exists between abstract models and code as consequence of the direct representation idea. The final idea relates to OMG’s mission to solve tool integration problems through open vendor-neutral interoperability standards.

In yet another publication on MDE, Schmidt [88] sees the subject as approach to solving the semantic gap and platform complexity problems by means of modeling languages that can effectively express domain concepts. Further more, the author stresses that in order to ensure consistency between models and application implementations, tools are needed that can analyse models and synthesize various types of artefacts, including models themselves.

3.1.4 Essence of MDE

We have seen a number of definitions of what Model-Driven Engineering is. Analysis of the definitions reveals a set of essential commonalities.

Models are primary engineering artefacts: All definitions agree that models should be first class artefacts in model driven approaches to engineering. Models can help developers to deal with complexity (refer to page 20) and serve as documentation. Being a primary artefact, places additional requirement on MDE models: model must be a computerized representation. In such a model, each element that can be interpreted by a computer, corresponds to a concept in the modeling domain [15]. This requirement is crucial for distinguishing models that can be primary artefacts from those that cannot.
**CHAPTER 3. MODEL-DRIVEN ENGINEERING**

**Direct representation of domain concepts:** Several MDE definitions emphasize that modeling languages should more directly represent concepts from the problem domain. The primary intention is to reduce the semantic gap (refer to page 20), perhaps the biggest problem in software development. The more directly modeling languages represent concepts of the problem domain, the more value developers get from model driven approaches.

**Automation and tool support:** Another commonality is the need for automation. Automation refers to mechanization of tedious, resource consuming and error prone development activities or tasks that do not require human intelligence. In MDE, automation:

- Enables direct representation by solving the productivity problem (refer to page 21).

- Enables models to be primary artefacts by solving the productivity and documentation problems (refer to page 21).

- Has many useful applications: model refinement, verification, analysis, optimization, model execution, code generation, etc.

**3.2 Basic concepts**

Previous section identified three complimentary ideas that constitute the essence of MDE. So how does one realize these ideas in practice? Experiences with CASE (Computer-Aided Software Engineering) show that it is not feasible to manually build tools for all possible domains and possible automation tasks due to the complexity of tool development. MDE solves this problem by applying MDE to itself: In this approach languages are not coded for a particular tool, but are modeled. Such models are complete language definitions, from which executable specifications for modeling tools can be automatically generated.

Modeling tools that are not hard coded but are capable of understanding and executing language specifications require new kind of modeling infrastructure. The primary building block of these infrastructures is a pattern called meta-step. The pattern and example infrastructures are discussed in section 3.2.1. In contrast to real-world systems, languages as system have a number of specific features that language specifications should properly capture. These features are described in section 3.2.2. Finally, automation of model manipulations is subject of section 3.2.3.

**3.2.1 System, Model and Meta-step**

In computer science models are expressed in a *modeling language*. In the context of MDE, such models are created with the help of computer aided modeling tools and are amendable to automated processing. Traditionally, a fixed number of modeling languages had been hard coded in tools such as CASE, a costly undertaking prohibitive to the idea of directly representing the plethora of business domains. To reduce costs of building
tools for modeling languages, model-driven approach is applied to specify complete and executable definitions of languages that can be executed by a new generation of modeling tools. In this thesis, we refer to such tools as MDE tools.

In the model-driven approach to developing modeling tools, a language is treated as system and is defined by a new kind of model, known as metamodel. In this context, a metamodel is defined as follows (see e.g. [26, 47]):

A metamodel is a model of a modeling language.

Figure 3.4: System, model, metamodel

The act of making a metamodel is known as meta-modeling. Metamodel and language and their relations with Model and System are shown in Figure 3.4. Similar to system and model, there is the RepresentationOf relation between a metamodel and a modeling language. Frequently, language is omitted and metamodel is considered in the context of models only. In this case, relation InstanceOf is often used between the two. In general, InstanceOf denotes membership of an entity to a class that is a set. By the definition, metamodel is not a set, hence usage of InstanceOf is not correct [47]. In [27], Favre denotes the link between a model and a metamodel with relationship ConformsTo. Sometimes in the MDE literature, especially on MDA, a language is not distinguished from its metamodel. Strictly speaking, a language is not a metamodel, but a system modelled by the latter [26]. Furthermore, note the equivalence of the System-Model and Language-Metamodel relationships: it follows that a language is a system, a metamodel is a model and meta-modeling is the same activity as modeling. Indeed, in [26] system and model are considered as roles, which can be assumed by different concepts.

We have seen that a metamodel itself can play the role of a model, be written in a modeling language and conform to another metamodel. This recursive application of modeling concepts to models is known as the meta-step pattern [27]. The meta-step pattern can be applied repeatedly to built a hierarchy of models that span multiple
levels. In this context, a level is merely a convenience concept that assists to organise and reason about models. A level organisation of models is often referred to as \textit{meta-modeling architecture}. A number of meta-modeling architectures are discussed below.

Traditional CASE tools are based on the modeling architecture that is illustrated in Figure 3.5a [66]: The lowest level indicates that the tool is capable of instantiating systems from software specifications or running simulation models. A level higher, models are edited or stored. Metamodels and storage schemas are hard-coded into the topmost level of the CASE tool. This level defines the kinds of models that are supported and how these models can be manipulated by the CASE tool. Change and creation of new modeling languages is severely limited by this architecture, because the metamodel level is fixed in the code of the tool and can be modified only by the tool vendor. In order to remove the limitation of CASE tools, the four layer metadata architecture [38] is frequently used in the meta-modeling community [71, 59]. In this architecture (see Figure 3.5b), the lowest two levels are similar to those of the traditional architecture. The metamodel level is not hard-coded, but contains models of modeling languages. These models are edited and stored in tool’s repository. Metameta-models and storage schemas are hard-coded into the topmost level of the tool. This level defines i) meta-modeling languages, in which modeling languages can be modelled and ii) how models written in these languages can be manipulated. Note that change and creation of new modeling languages is not limited. However, the meta-languages are fixed in the code of the tool and can be modified only by the tool vendor. Both architectures consider meta-steps in absolute way: each of their levels is given a fixed name and plays a certain role as summarised in Table 3.1. An alternative to absolute architectures is the golden braid architecture [43]. As shown in Figure 3.5c, this architecture is not capped and emphasizes the fact that model and metamodel are relative concepts based on the \textit{ConformsTo} rela-
CHAPTER 3. MODEL-DRIVEN ENGINEERING

<table>
<thead>
<tr>
<th>LEVEL</th>
<th>DESCRIPTION</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meta-metamodel</td>
<td>Languages to specify modeling languages</td>
<td>Relation has a source and target Entity</td>
</tr>
<tr>
<td>Meta</td>
<td>Modeling languages to specify systems</td>
<td>Directory can contain Files and other Directories</td>
</tr>
<tr>
<td>Model</td>
<td>System specifications: models, code</td>
<td>Specification of a file structure</td>
</tr>
<tr>
<td>System</td>
<td>Model simulations, software instances</td>
<td>Realised specification as actual files and directories</td>
</tr>
</tbody>
</table>

Table 3.1: Absolute organization of modeling levels

Meta-architectures that are not limited in the capability of changing and creating new modeling languages are crucial to MDE tools.

The described structures are the most common meta-architectures. Their primary goal is to define a flexible and general infrastructure for instantiation, modeling and metamodeling. In principle, other architectures can be designed to address these and other goals: e.g. the MDA meta-architecture discussed in section 3.3.1.3, is similar to the four level architecture, but is extended to provide a framework for exchange of metamodels among different meta-modeling environments. For a comprehensive discussion of meta-architectures created during the human history, the reader is referred to [27].

3.2.2 Language as System

We have seen that a modeling language can be considered as system and (meta-)modelled. In contrast to real-world systems, languages as system have a number of specific features [12, 72, 94]:

abstract syntax defines language concepts, relationships among the concepts and well-formedness rules. The latter, also known as static semantics, defines how individual concepts can be combined into admissible compound structures.

concrete syntax is a notation that is responsible for appearance of language concepts and the way the user interacts with these concepts. This notation can be textual, diagrammatic and combination of both.

semantics describes the meaning of language concepts. Many different approaches to describing semantics exist, e.g. translational, operational, extensional and denotational [12].

Precise semantics is an important asset of a language as it allows to avoid ambiguous interpretation of models written in the language. The abstract syntax is independent from the language presentation and semantics. One language may have one or more different notations, and semantic definitions.
Figure 3.6: Abstract syntax of the HFS language

Figure 3.7: Examples of concrete syntax for the HFS language

As an example, consider a modeling language for a simple hierarchical file system (HFS). Figure 3.6 shows the abstract syntax expressed in the Entity Relationship notation. The model specifies concepts *Directory*, *File*, *Containment* and how the former two are connected by the latter. Cardinality of ends of each connector (line) define how many instances of the related concepts are allowed to be connected. In the above example, one parent directory can contain zero or multiple children of type *File* or *Directory*. On the other hand, any child can be contained by only one parent. Furthermore, *Containment* is required to connect to 1) one parent and 2) one child. Note that the second requirement cannot be indicated by the model structure alone as they also allow zero or two children. To further constrain the relationship, a well-formedness rule can be specified in the form of an invariant that checks if exactly one child is connected via the relationship. Next, a concrete syntax is specified by mapping appearance symbols to concepts of the language. Figure 3.7a and Figure 3.7b illustrate respectively a diagrammatic and a textual notations for the same language. Finally, Figure 3.8 illustrates one possible way to express the meaning of the concepts using Unix shell commands. In plain English, the definition associates the pattern in Figure 3.8a with a sequence of three shell statements: go inside the parent directory, create a child file and go back (see Figure 3.8b). Note that the previous sentence in itself is an informal semantic description of the pattern.

For a given language, multiple models of different levels of completeness can exist. To help assess the quality of metamodels, Clark et al. [12, page 25] proposed five quality
levels: Informal, incomplete and imprecise metamodels that define only a simple abstract syntax belong to the lowest level 1. At level 2, the abstract syntax is relatively complete, semantics is informally defined and correctness of the metamodel is verified. Level 3 requires metamodels with complete and tested abstract syntax and formulated, but not formalized concrete syntax. At level 4, the concrete syntax will have been completed and tested, users will be able to create models that conform to the metamodel and semantics will have been drafted. The language itself will have evolved towards reuse and extensibility. At the topmost level 5, all features of the language will have been defined.

In summary, a model of a language should properly capture the language features. Consequently, quality meta-modeling may require one or more meta-modeling languages that are sufficiently expressive to capture all aspects of a language system.

### 3.2.3 Transformation

Engineering encompasses many steps that gradually lead development of a system from abstract documentation towards executable code. At any of those steps details are added and updates, optimisation, refactoring can take place. A model might be translated into a different worldview, perhaps to better answer questions about the system. Partial specifications may need to be merged to obtain a complete system description. The opposite is also possible, e.g. when the system is decomposed into sub-systems.

Many of the above processes can fit into the transformation pattern shown in Figure 3.9a. Such processes are called transformations. In this pattern [47], the process inputs
one or more source artefacts and outputs one or more target artefacts. The source and target can be i) the same or different models and ii) written in the same or different languages. Note that the cardinality of input and output implies four transformation cases: $1 \rightarrow 1$, $1 \rightarrow N$, $N \rightarrow 1$ and $M \rightarrow N$. A transformation can be one monolithic step or consist of a sequence of smaller intermediate steps.

So what is a transformation in MDE? Again, the answer is not straightforward as many definitions that are found in the literature, are often given in the context of Model Driven Architecture (for a discussion of MDA, refer to section 3.3). Kurtev in [47], notes that the precise definition of transformation and transformation definition is still a subject of investigation. Nevertheless, we will accept as a sufficiently generic definition, the one given by Kleppe et al. for MDA in [45]:

A *transformation* is a process of generating a target model from a source model, according to a *transformation definition*.

A *transformation definition* describes how a model should be transformed.

The rest of this section provides a summary of transformation classifications, characteristics and approaches that can be found in the literature.

A number of classifications can be based on variations in concepts of model and language. An obvious distinction concerns the *kinds of artefacts* being transformed. Suppose a model and a program are two artefact kinds and source and target of a transformation can be of the same or different kind, then the following terms can be justified (see e.g. [65, 19, 91]): model-to-model (or model) transformation, program transformation, model-to-code and code-to-model transformations. In view of many authors, the former term encompasses the others. Next, Kent [44] distinguished transformations among models expressed in the same or different languages. Mens et al. [65] made the same distinction and referred to the former and the latter respectively as endogenous and exogenous transformations. Another system of classification is related to the levels of *abstraction*. In this dimension, Mens et al. [65] divided transformations into horizontal and vertical categories. If abstraction levels of source and target models are the same then a transformation is horizontal, otherwise it is vertical. Typical examples of the former are optimization, refactoring and generating new views. The latter is represented by refinement and reverse engineering. Finally, sources and targets of transformations are technological artefacts and therefore belong to the same or different *technological spaces* [48]. Mens et al. [65] explained the effect of different technological spaces (TS) on a transformation: "transformation tools need to provide exporters and importers to bridge the technological spaces while the actual transformation is executed in the technological space of either the source or target model". In [95], Völter and Stahl distinguished transformations into *generators* and *interpreters*: while functionally equivalent, the former are commonly used for structural aspects of a system and the latter for behaviour. Transformations that are widely used in model-driven development fall into the category of generators. The discussed classifications of transformations are illustrated in Figure 3.10.

Mens et al. [65] identified three general characteristics of a transformation: preservation, level of automation and complexity of the transformation. The former refers to the fact that despite the diversity of transformation types, each transformation preserves
CHAPTER 3. MODEL-DRIVEN ENGINEERING

(a) Model or program transformation

(b) Model-code transformation

(c) Exogenous transformation

(d) Endogenous transformation

(e) Horizontal transformation

(f) Vertical transformation

(g) Export to target TS

(h) Transformation in target TS

Figure 3.10: Examples of transformation classifications
certain aspects in the target model. The nature of such aspects depends on the type of transformation: e.g. refinement adds more concrete details to a model while preserving its correctness and refactoring changes the structure of a model while preserving the behaviour. Further, a distinction is made between transformations that can be automated and those that need a certain amount of intervention. The degree of intervention is not twofold, thus one can speak of levels of automation. For example, translation of system requirements from natural language to a model is inherently manual (see the semantic gap), while code generation has been partially automated for more than a decade. Finally, the authors observed huge differences in complexity of different transformation types, which may call for different sets of tools and techniques.

Turning from classifications and characteristics to practical realizations of transformations, Mens et al. [65] summarised that "to specify and apply a transformation, ideas from any of the major programming paradigms can be used". The authors choose to distinguish all the possible approaches according to the declarative/imp(>rative classification. They concluded that declarative approaches to transformation appeared to be the most promising, at least in theory. In [19], Czarnecki and Holsen viewed a transformation as specialized system and defined explicitly design features of model transformation approaches. In an orthogonal view on the subject, Sendall and Kozaczynski [89] identified three architectural approaches to tools: direct model manipulation, intermediate representation and transformation language support. In direct model manipulation, tools provide users with a set of procedural APIs to access and manipulate an internal model representation. Intermediate representation is a two step process: first a model is exported into a standard form (typically in XML), than the exported model is transformed by the same or external tool. Tools with transformation languages support offer explicit mechanisms to specify and apply transformations. The authors highlighted limitations and advantages of all three approaches, concluding that the former and the latter offer respectively the least and the most potential. Finally, in [74] OMG Architecture Board described different degrees of human involvement in the process of carrying out transformations in MDA. The more humans are involved, the less a process is automated.

We have seen that transformation can have various levels of automation. Striving for complete automation of transformations is one of the fundamental ideas of model-driven engineering, which was discussed in section 3.1.4. Unless otherwise indicated, references to "transformation" in the remainder of this thesis refer to an automated process.

3.3 MDE approaches

A number of approaches exists that put MDE ideas to practical use: Model Driven Architecture (MDA) by OMG is the most known MDE initiative in the software industry. Less known and cardinally different to MDA is Domain Specific Modeling (DSM) that has been promoted by Finnish company MetaCase for more then a decade. In line with the MDA technologies, eXecutable Metamodelling Facility (xMF) aims to solve a number of shortcomings of MDA [12]. At last but not least, a newcomer to MDE is Domain Specific Language Tools (DSL Tools), Microsoft’s own approach that appears to share many DSM ideas [15]. The first two approaches are the best known and are described in sections 3.3.1 and 3.3.2.
3.3.1 Model Driven Architecture

MDA is the Object Management Group's approach to model driven engineering in software development. OMG is an international, open membership, computer industry consortium that has always targeted integration problems through open, vendor neutral interoperability standards [74]. MDA is the next evolution of the OMG solutions that address the integration of both development tools and systems under development through the entire system life cycle.

In the rest of this section the MDE concepts, process, framework and technologies are described and their contributions towards solving the general MDE and MDA-specific problems are shown.

3.3.1.1 Modeling space and process

The MDA approach distinguishes one model classification system based on abstraction of platform similarities. The MDA definition of Platform is given in [74, page 5] as follows:

"In the MDA, the term platform is used to refer to technological and engineering details that are irrelevant to the fundamental functionality of a software component."

Given the above definition, two categories of models can be defined:

- **Platform Independent Model (PIM)** is a formal specification of the structure and function of a system that abstracts away platform details [74].
- **Platform Specific Model (PSM)** specifies a system in terms of implementation constructs belonging to a concrete platform.

One or more PSMs can be generated from a single PIM. The difference between PIM and PSM classifications is not binary: a model is usually measured as being more or less platform specific in relation to another model. Note that both PIM and PSM describe a software system and hence are not computation independent models (CIM).

The PIM/PSM partitioning of the MDA modeling space is reflected in the MDA process as well [45]. Figure 3.11 illustrates both the traditional and the MDA processes. From the figure it is obvious that both processes are strikingly similar. The difference is twofold: 1) the design activity can result in a number of PSMs for a given PIM and 2) the primary artefacts are models instead of informal documentation.

Structuring of the modeling space around the concept of platform enables MDA to address the following problems [45]:

- Portability of systems to new platforms (refer to page 21).
- Interoperability among technologically different parts of the same system (refer to page 21).

Explicit introduction of PIMs in the design step enables separation of business solutions from their implementations obscured by transient technologies. Such technology independent models are inherently portable. The interoperability property requires knowledge of how elements of different platforms are related to each other. Such information can be derived from a PIM and related PSMs because the latter are generated from the former.
3.3.1.2 Transformation

The notion of MDA transformation is identical to that of MDE, in section 3.2.3. This similarity exists due to the close relation between MDE and MDA. This section specializes the MDE transformation concept by adding MDA specific details [45]:

* A transformation is a process of generating a model in a target language from a model in a source language according to a transformation definition.

* A transformation definition is a set of transformation rules that describe how a model in a target language is generated from a model in a source language.

* A transformation rule is a description that maps one or more elements of the source metamodel onto one or more elements of the target metamodel.

Note that according to these definitions, an MDA transformation operates in the layer of metamodels. For further information on transformations in MDA, reader is referred to [45, Chapter 7].

MDA makes no reservations about the source and target languages: both can be the same or different languages and both can describe models at abstraction levels proper for PIM, PSM and Code artefacts. Therefore, one can deduce possible types of MDA transformations, e.g. \( PIM \rightarrow PIM \), \( PIM \rightarrow PSM \), \( PSM \rightarrow Code \) and \( PIM \rightarrow Code \). Detailed descriptions of common transformation types can be found in [74, 45]. The most useful types are those that convert less detailed artefacts into more detailed ones.

Under MDA, it is crucial that transformations are automatic [8]. If transformation activities from PIM to Code are fully automated, the following general problems are addressed:
- Productivity (refer to page 21).
- Documentation and maintenance (refer to page 21).

The above transformations are the most complex development activities, which are traditionally performed manually. Automation of these transformations directly addresses the productivity problem. Moreover, Kleppe et al. [45] note that development resources are released as the result of the automation, and can be re-allocated to enable better PIMs, thus further increasing productivity through better designs. High quality PIMs serve both as documentation and source of information needed to generate a complete product implementation. This effectively eliminates the need for manual synchronization between models and code.

### 3.3.1.3 The 1/2 floor of the MDA building

In MDA, modeling languages have to deal with multiple vertical domains and platforms. The four-layer meta-modeling facility described in section 3.2.1, is capable of to defining modeling languages within a single modeling tool. However, OMG’s goal is to integrate models developed with possibly multiple tools throughout the development cycle. The consortium’s traditional approach to solving the tool integration problem is based on standardisation. Indeed, the standardisation approach manifests itself in the so-called four level self-reflective meta-architecture of MDA.

Figure 3.12 illustrates both the four level and the MDA meta-architectures. In MDA, the four levels from bottom to top are called $M0$, $M1$, $M2$ and $M3$. Both architectures are identical except for the topmost level. The metameta-model level is defined in a non-standard, tool specific and therefore non portable way. The implication from this fact is
that in general, models and metamodels cannot be meaningfully exchanged among tools with different meta-languages (for the sake of simplicity we omit the case when tools have different versions of the same meta-language). A meaningful exchange would require an additional model transformation from the technological space (TS) of the source tool to the TS of the target. In MDA, the $M3$ level is modelled in a standard and tool independent modeling language. Instead of extending the architecture with another modeling level, the $M3$ level contains a self-reflective model. Such model is defined in the language modelled by the model itself. For more details on self-reflective models, the reader is referred to [47]. In contrast to tools based on the four level architecture, all MDA tools belong to the same TS and can meaningfully exchange metamodels and models without intermediate model transformations to bridge different technological spaces.

3.3.1.4 The MDA framework

The framework that lies behind the MDA process is illustrated in Figure 3.13. All the entities and relationships in the diagram can be deduced from the previously introduced MDA concepts and its meta-architecture.

Level $M0$ contains systems, such as Modeling tool (not illustrated) and Transformation tool. Artifacts from the software life-cycle are placed at level $M1$. With respect to
CHAPTER 3. MODEL-DRIVEN ENGINEERING

transformation definition, these can be in relative roles of source and target. Transformation types are defined based on the PIM/PSM/Code categories of source and target models. At level M2, modeling languages of the source and target models are defined. These languages are not fixed in MDA and may evolve as business domains and implementation technologies change. On the top of the MDA framework, there is a standard and tool independent Meta-language. Through this language, new MDA languages for source and target models can be defined if needed. Meta-model transformation can be expressible in the language or an extension of the language in which the models are expressed [44]. Indeed, Transformation Definition Language is defined as extension of the Meta-language. Finally, the Meta-language is modelled in the Meta-language itself (refer to section 3.3.1.3).

In summary, a number of observations can be made. The framework does not reflect the platform-oriented organization of the MDA modeling space. Moreover, no prescriptions about the source and target languages are present in the framework. However, all metamodels must be written in a single meta-language. Due to the self-reflection, this requirement applies to the metamodel of the meta-language as well.

3.3.1.5 OMG standards and technologies

OMG does not own any tools. In order to help the industry to develop MDA tools and ensure their interoperability, OMG developed its own set of standards for MDA. Figure 3.14 shows a selection of current standards associated with MDA and how these standards are related to the MDA meta-architecture. The OMG standards reside in levels M2 and M3 of the MDA meta-architecture. The depicted standards are described below.

**PIM Languages:** COMMON WAREHOUSE METAMODEL (CWM) is a language for modeling data warehousing applications. It is primarily used to describe structural aspects of systems, and therefore lacks constructs to describe behaviour. CWM is targeted for a broad range of applications within data warehousing domain.

UNIFIED MODELING LANGUAGE (UML) is the most wide-spread general modeling language in software industry (OMG document - formal/2006-01-01). In contrast to CWM, UML recognises both structural and behavioural aspects in a software model;
CHAPTER 3. MODEL-DRIVEN ENGINEERING

Aspects are called static and dynamic views respectively. UML provides a number of notations to define these views [32]. The artefacts produced with the help of these notations are called diagrams and constitute parts of the same model. For example, Class diagrams provide the static view on the system, while Use Case, State and Sequence diagrams constitute the dynamic views describing the behavioural aspect. At the time of writing, UML support for the dynamic view is not expressive and powerful enough to sufficiently specify behaviour [45]. A number of amendments exist that partially address but currently do not solve this deficit. Two of these amendments are described below.

**ACTION SEMANTICS (AS)** is an integral part of UML targeted for behaviour. However, it also has a number of shortcomings that prevent it from solving the behaviour deficit of UML. Most noticeably, AS is a too low level language to be used for specifying PIMs; it is better fitted for a PSM language [45]. AS together with UML forms EXECUTABLE UML.

**OBJECT CONSTRAINT LANGUAGE (OCL)** is a pure expression language without side-effects (OMG document - formal/06-05-01). OCL is not a separate standard, but an additional technology. OCL is typically used in OMG standards to make models more precise by imposing well-formedness constraints and more complete by specifying initial values for attributes, pre- and post-conditions of operations and queries [96]. In some cases, comparing pre-conditions (what the subsystem was) and post-conditions (what the subsystem became) of an operation is sufficient to deduce the behaviour of the operation. OCL is highly recommended to be used with UML to increase quality of PIMs.

**PSM Languages:** UML as a general modeling language can be used to describe PSMs, however such an approach would lack precision and well-definedness of more specialised languages. In MDA, a common way to address this is through profiling, an extension mechanism of UML that allows to define new languages with help of profiles. Profiling and profiles are described in section 3.3.1.6.

**Transformation Definition Language:** At the time of writing, OMG finalizes a transformation definition standard called Query, Views and Transformations (QVT). As the name suggests, besides being a language for writing transformation definitions, QVT can create views on metamodels, as well as query models.

**Metalanguage:** META-OBJECT FACILITY (MOF) is OMG’s standard for the MDA Meta-language (OMG document - formal/06-01-01). MOF also bears additional functionality that enables it to build tools for defining modeling languages and interchange models.

Quality of metamodels developed with MOF 2.0 does not exceed the third level of meta-modeling quality (refer to page 27 of this thesis): MOF allows to specify the abstract syntax of a modeling languages, but does not provide direct support for specification of the concrete syntax [13]. Instead, OMG standards describe concrete syntax in terms of informal examples, leaving this feature to be specified by tool vendors. Furthermore, semantics of the language are not modelled, but described in precise, yet informal natural language [13]. As the consequence of the self-reflecting meta-architecture, quality of MOF itself does not exceed level 3.
MOF is primarily used to define UML, CWM and QVT, but can be used to define non-OMG standards that will fit into the MDA framework. Due to the fundamental nature of the Meta-language role in the MDA framework, MOF is the most important OMG standard for MDA.

### 3.3.1.6 Profiling

Profiling is an alternative to meta-modeling with MOF:

*Profiling* is a specialisation mechanism that allows to create a UML variant tailored to a specific domain.

Profiling is based on reuse of the UML metamodel: New language elements are defined by labeling the existing UML elements with new names - *stereotypes*. Such elements can be further re-defined with *constraints*. Finally, *tagged values* are used to add new meta-attributes to stereotyped elements. Semantics can be informally specified in comments. The concrete syntax of UML is typically reused. Language specifications produced with the help of profiling are called *profiles*.

This mechanism is UML-specific and is defined in the UML metamodel. Profiling is very suitable for defining UML dialects specific to implementation platforms, such as CORBA or C++. A list of UML profiles is maintained at the OMG portal [36].

### 3.3.1.7 MDA tools

OMG does not provide implementations, only specifications that enable tools to inter-operate. Tool-vendors are expected to faithfully implement the MDA and its standards. In [15], Cook observes that these standards are semi-formal complex specifications that exist in a number of versions, causing tools to implement different combinations of the standards. Moreover, not all standards are fully or accurately implemented. This variety of standards implementations makes choosing tools hard, as one MDA tool is not equal to another and the distinguishing MDA feature – tool integration through standardization, may not be achievable.

To help tool customers to compare and select tools that optimally fulfill their needs, a number of evaluation criteria has been proposed. Kleppe et al. [45] employs a *functionality* oriented approach. More recently, a selection of MDA tools were compared in [91] on the basis of a detailed list of general and MDA-specific features. In [65], Mens et al. provided a taxonomy of model transformations to help choosing a particular approach that is best suited for customer’s needs.

### 3.3.2 Domain Specific Modeling

DSM is an MDE approach proposed by MetaCase [67]. The approach is primarily focused on direct representation of concepts from the problem domain and full automation of code generation directly from models.

In the rest of this section the DSM concepts, process, framework and tooling are described and their contributions towards solving the software development problems are shown.
3.3.2.1 Modeling space and process

The DSM approach distinguishes two kinds of artefacts: 1) domain models expressed in terms from the problem domain and 2) executable code that can run on the platform typically used in the business domain. The problem domain is not fixed as the approach is capable of adapting to arbitrary domains.

Turning from artefacts to the process, Figure 3.15 illustrates both the traditional and the DSM life-cycles. The traditional process first describes a solution (the topmost step), then designs and implements the solution (the rest of the steps). In contrast, DSM concentrates on designing a solution with a domain model and direct generation of code from the model. Indeed, Figure 3.15 shows that steps "Analysis" and "Design" are not present in the DSM process. After the initial step "Solution Design", the only other development step is "Coding", which fits the transformation pattern and is automated. Since analysis and design steps produce knowledge that is required to realize the solution design in code, this implies that such knowledge must be available in the DSM code generation prior to the solution design step.

Domain specific modeling and full automation of code generation address the following problems:

- Semantic gap (refer to page 20).
- Complexity (refer to page 20).
- Productivity (refer to page 21).
- Portability (refer to page 21).
CHAPTER 3. MODEL-DRIVEN ENGINEERING

In DSM developers and domain experts use constructs that closely correspond to the real-world objects from the problem domain, thereby directly addressing the semantic gap. Moreover, this is the first step of developing any software, therefore the level of abstraction is the highest and the level of complexity is the lowest. Next, automated and full generation of code directly from domain models naturally results in increase of productivity. MetaCase reports productivity gains up to ten times [68]. Finally, models specified in terms of business domain concepts, are intrinsically independent from technological platforms and therefore are portable. Changes in platforms are isolated in code generators.

3.3.2.2 The DSM framework

Figure 3.16: The DSM framework

The framework that lies behind the DSM process is illustrated in Figure 3.16. The elements and relationships of the framework are described in the context of the four level meta-architecture in the bottom-up sequence.

The system level contains systems, such as Modeling tool (not illustrated) and Code Generator. Development artefacts Domain Model and Code reside at the model level. Code generator typically traverses through the models, reads semantic information from them and translates the information into code for the target platform. Code Generation Definition contains a set of rules that describes how instances of domain concepts encountered in models should be represented in a programming language using domain specific
reusable pieces of code called Components. A library of components completes the definition of the Domain Specific Modeling (DSM) Language by extending it with interfaces to the target platform, templates and other code constructs that enable complete code generation. Components are typically created from code developed in previous projects within the given domain. At the metamodel level, DSM languages are defined. The modeling languages are not fixed and may change or evolve together with the business domain. Note that programming languages in which code is written, are not defined in the framework. On the top of the DSM framework, there is the Meta-language and the Transformation Definition Language. These languages are used to model domain specific languages and specify code generation. The framework neither prescribes if the meta-language and transformation definition language should be modelled (or hard-coded), nor specifies relationships between the two.

In summary, two observations can be made. First, the framework represents a limited case of use of transformations: only model-to-code type (code generation) is required. Second, DSM cannot be applied if there is no prior knowledge of mapping domain concepts to executable code.

3.3.2.3 DSM tools

MetaCase developed a DSM tool MetaEdit+[66] and does not focus on standardisation of DSM technologies. In reflection of the DSM framework, a DSM tool must feature a meta-modeling environment, a code generator and a library of components. In general, MDE tools with different technologies can implement the DSM approach. A list of tools with full DSM support is maintained at the DSM Forum [31].

3.4 Summary

This chapter introduced the emerging field of MDE that has a potential to bring a paradigm shift to the traditional engineering. We have seen that MDE is about formulation, formalization and automation of development processes. Moreover, MDE is not a generic approach, but is focused on particular problems and process. Two widely used and mature MDE approaches, MDA and DSM, were described in a uniform way that allows to compare them for applicability to the problem being addressed in this thesis.

MDA is a code oriented MDE with a particular modeling space organization helpful in solving portability and interoperability problems of developed systems. Furthermore, OMG assumes that no single tool can meet all development needs and therefore multiple tools need to be integrated in order to exchange models directly and correctly. OMG achieves this goal through standardization of the technology that defines languages in which models are written. This in fact places all MDA tools into one technological space, thus theoretically removing the integration problem. The standardisation approach manifests itself in the self-reflectiveness of the MDA meta-architecture. The code orientation of MDA manifests itself in the PIM/PSM organization of the modeling space. This is further re-enforced by OMG's modeling standards such as UML, which are centered around software concepts. MDA technologies and tools (not the framework) have proved to be useful in increasing formality of the development process, but has not procured
convincing evidence of significantly raising the level of abstraction for developers and gaining productivity through automation.

Today, leading experts on model-based development [8] recognise the role of domain concepts in successful application of modeling to engineering. DSM is an approach that strongly focuses on direct representation of concepts from the business domain, thus raising abstraction of system development to the level higher than what is typically achieved in MDA. DSM enables experts from the business domain to model designs in familiar domain concepts and automatically generate 100% complete code in the implementation technology of the domain. However, we have seen that DSM cannot be applied first time a system is developed. Furthermore, code generation depends on libraries of reusable software components. Finally, the approach considers the problem and solution domain in the absolute way, that is the former is a business domain and the latter is a platform. Consequently, transformations in DSM are limited to code generation only.
Chapter 4
MDE at Work

Chapter 4 illustrates how MDE is used to define languages for the field of Modeling and Simulation. Each language is treated as a system (see section 3.2.2) and abstract syntax, concrete syntax and semantics features are defined. The chapter starts with a definition of the abstract and concrete syntax of a simple language for the data formalism in section 4.1. Note that the semantics is not defined, as data itself is typically used to express semantics of other formalisms [94, page 57]. Section 4.2 introduces a timeless formalism (DFD) and models the abstract and concrete syntax, as well as the semantics. The latter is defined using the translational approach, that is by translating this language concepts into concepts of another executable language. Section 4.3 continues with an illustration of a DSL for DEVS, a fundamental M&S formalism. In section 4.4, a language for the Process Interaction (PI) formalism is modeled. The PI semantics is defined using the operational approach. Finally, a complete definition of a modeling language specific to the road traffic domain is illustrated in section 4.5. All presented models were developed using MDE tool AToM3, which is described in section 6.1.4.

4.1 Data

The fundamental notion of data is a time function called trajectory or signal, which describes evolution of an entity over time. A time base is a structure:

\[ T = (\text{time}, <) \]

where

- \text{time} is a time set
- \(<\) is an ordering relation on elements of time

The trajectory is a time function:

\[ f : T \to A \]
CHAPTER 4. MDE AT WORK

4.1 Data Segments

A segment is a restriction of \( f \) to a time interval \( (t_1, t_2) \). A restriction of a time function is also a time function. A detailed discussion of time bases, trajectories and segments can be found in [100, pages 99 and 100].

A metamodel for visual modeling of data segments is illustrated in Figure 4.1. The following informally describes each construct:

- **Signal** represents the entity described by the segment. This element contains a signal description and specifies the measurement units for its (signal) values.
- **Time** represents a time base and specifies whether the ordering relationship is total or partial.
- **Signal value** describes the value of signal at certain time.
- **Ordering** relationship on elements of set time (see construct signal value above). The cardinality of this element depends on the ordering property of the time base.

A Data Segment model represents time indexed data collected from a system. The semantics of the data is understood in the context of the source or generative system.

### 4.2 Data Flow Diagrams (DFD)

Data Flow Diagrams (DFD) present the flow of data through a system with the focus on how the data is processed in terms of inputs and outputs. Although simple, data flow diagrams are widely used in practice. The presented DSL is based on DFDs by Gane and Sarson [34].
4.2.1 DFD metamodel

The building constructs of DFD are Data Flow, Data Store, Process and External Entity. Figure 4.2 illustrates how these constructs can be combined together in a metamodel. The following informally describes each construct:

**ExternalEntity** is data object outside the context of the modeled system. External entities are sources and sinks (destinations) of the system’s inputs and outputs. Each is given an alphabetic identifier.

**DataFlow** is a pipeline through which packets of data of known composition flow. The arrowhead indicates the direction of the data flow. Each data flow must have a label describing the data. No alteration of data can take place within a data flow.

**DataStore** is the repository of data inside the system. It is a data queue as opposed to data flow. Each is identified by an unique number prefixed with "D".

**Process** is an entity that transforms an incoming data flow into an outgoing data flow. Each is given a numerical identifier, physical reference (in the lower part of the process box) and is described with an imperative sentence containing an active verb e.g. "convert data".

A valid DFD should also comply with several rules. For example:

- An external entity cannot be connected to another external entity. This rule is implemented as a constraint on element **DataFlow** that is triggered after it is connected to an element. A simplified Python expression for this rule is:
  ```python
  metatype(self.source) != ExternalEntity or metatype(self.destination) != ExternalEntity
  ```
4.2 Data stores receive inputs and outputs only from processes. Another post connect constraint is assigned to \textbf{DATAFLOW}:

\[(\text{metatype(self.source)} == \text{DataStore} \text{ and metatype(self.destination)} == \text{Process}) \text{ or } \]

\[(\text{metatype(self.destination)} == \text{DataStore} \text{ and metatype(self.source)} == \text{Process})\]

In addition, custom rules may be desired in a specific application. The ease and flexibility of the meta-modeling technique allows easy adaptation of the DFD formalism if needed. As an illustration, two new constraints are introduced:

- Do not allow branching. This can be achieved by tuning the cardinality property of the metamodel elements.

- Do not allow loops on the same process. The following post connect condition on DataFlow should hold:

\[(\text{self.source} != \text{self.destination}) \text{ or } (\text{metatype(self.source)} != \text{Process})\]

The concrete syntax of this DFD language (not shown) is modelled after the Gane and Sarson notation.

### 4.2.2 POKer code generation

This section specifies a code generation transformation that generates a textual representation of a DFD model that can be simulated by external solver POKer [51], a Python programmed interpreter of simulation language πDemos [4]. The transformation does not perform any important graph rewriting, hence LHS subgraphs are copied to RHS subgraphs. The code generation is performed by actions of rules.

The \textit{initial action} of the transformation iterates through all the elements of the current model to annotate them with temporary attributes (to be used in the conditions specified below). Annotation \textit{isVisited} helps to distinguish the elements that have been already processed from those that have not yet. Annotation \textit{isCurrent} is used to mark a DataFlow that leads to the element whose code has to be generated next. It also creates the initial data structure of the output:

\{'source': none, 'sink': none, 'body': []\}

The rules are designed to match the pattern shown in Figure 4.3, where the relationship object is a DataFlow, and the entity can be an instance of any other DFD construct. Either the left or right relationship can be omitted. Present objects are labelled with consequent numbers. In the following we briefly describe each rule, in the order of priorities:
Process locates a process according to the LHS shown in Figure 4.4. The rule's action generates a sequence of POKer commands to access, use and release the physical entity implementing the process and annotates objects in RHS. The pre- and post-conditions of this rule are:

pre: \( \text{LHS.obj1.isCurrent and not LHS.obj2.isVisited} \)
post: \( \text{not RHS.obj1.isCurrent and RHS.obj2.isVisited and RHS.obj3.isCurrent} \)

Figure 4.4: LHS subgraph for processes

SourceExternal locates a source external entity matching the LHS shown in Figure 4.5. The action updates the 'source' field of the output data structure with the URL of the data in object 1 and annotates the RHS. The pre- and post-conditions of this rule are:

pre: \( \text{LHS.obj1.isSource and not LHS.obj1.isVisited} \)
post: \( \text{RHS.obj1.isVisited and RHS.obj2.isCurrent} \)

Figure 4.5: LHS subgraph for source externals

DataStore locates a data store matching the pattern shown in Figure 4.6. As semantics of this entity in the POKer context is not defined, no code is generated and only RHS is annotated by the action:

pre: \( \text{LHS.obj1.isCurrent and not LHS.obj2.isVisited} \)
post: \( \text{not RHS.obj1.isCurrent and RHS.obj2.isVisited and RHS.obj3.isCurrent} \)

Figure 4.6: LHS subgraph for data stores
SinkExternal locates a sink external entity according to the LHS shown in Figure 4.7. The action updates the 'sink' field of the output data structure with the URL of the data in object 2 and annotates the RHS. The pre- and post-conditions of this rule are:

pre: \( \text{LHS.obj1.isCurrent \land \text{LHS.obj2.isSink} \land \neg \text{LHS.obj2.isVisited}} \)

post: \( \neg \text{RHS.obj1.isCurrent} \land \text{RHS.obj2.isVisited} \)

The final action writes the generated data structure into an output file. As the last step, it iterates through all the objects on the tool’s canvas removing annotations isVisited and isCurrent.

To give an example of code generation for POKer, consider the model in Figure 4.8: Source external entity a0 contains a reference to a context-in model stored in file repository. Process 0 refers to simulation tool Spectre that can solve the model. Process 1 is a script that converts the output of process 0 into the input suitable for the next process. Process 2 refers to a data visualization application that produces images from the input. Finally, sink external entity a1 refers to the modeler’s file repository.
job = {
    "body": [
        "getR ('Spectre')",
        "hold (50)",
        "putR ('Spectre')",
        "getR ('Spec2TSV')",
        "hold (20)",
        "putR ('script')",
        "getR ('Gnuplot')",
        "hold (10)",
        "putR ('Gnuplot')",
        "close()"
    ],
    'source': 'scheme://host:port/sourcepath',
    'sink': 'scheme://host:port/sinkpath'
}

Table 4.1: Excerpt from the generated code for POKer

Applying transformation DFD2POKER to the above DFD model produces a textual file in the AToM^3 generation directory. This file contains a textual representation of the DFD model that can be executed by POKer. An excerpt from the generated code is pasted into Table 4.1. More details can be found in [54].

4.3 Discrete Event System Specification (DEVS)

Discrete Event System Specification (DEVS) is a well-known formalism that was first introduced by Zeigler in 1976 [99]. DEVS provides a formal basis for a class of discrete event formalisms. It is characteristic for these formalisms to be based on continuous time base bound to a certain time-span, during which only a finite number of events can occur. The state of the systems can be changed only by events and is preserved between arrivals of events.

The presented DEVS application [55] is based on the solver, metamodel and code generation transformation developed at McGill University, Montréal, Canada. These off-the-shelf parts were originally used for generation of DEVS Modeling & Simulation environments with AToM^3[78]. The environment allowed the graphical definition of DEVS models and was capable of generating model representations suitable for simulation by external simulator PythonDEVS [7], an implementation of the standard classic DEVS simulation algorithm.

4.3.1 Metamodel of DEVS with ports

Basic DEVS describes the autonomous behaviour of a discrete event system as a sequence of deterministic transitions between sequential states as well as how it reacts to external input events and how it generates output events. A basic discrete event system specification (DEVS) is a structure:
$DEVS = (X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta)$

where

- $\delta_{int}$ is the internal transition function
- $\delta_{ext}$ is the external transition function
- $\lambda$ is the output function
- $ta$ is the time advance function
- $X$ is the set of input values
- $Y$ is the set of output values
- $S$ is the set of sequential states

The specification above is low-level and adequate for simple systems, but becomes increasingly difficult to apply, the more complex the system is [100, page 125]. Complex systems are often best modeled as collection of interacting DEVS components, in which each component is being modeled independently. Such modular and hierarchical modeling becomes possible with the introduction of ports. In this case events determine the values appearing on ports of components.

Coupled DEVS (DEVN) describes a system at the higher level, as a network of individual components coupled by connecting their input and output interfaces in a modular way. Components can be atomic (basic DEVS) or coupled DEVS models. The connections represent how components influence each other, for example the output event of one component can become via the network an input event of another neighbour component. If the components are defined in DEVS with ports, such coupled system is specified as follows:

$DEVN = (X, Y, D, M_d, EIC, EOC, IC, Select)$

where

- $X$ is set of input ports and values
- $Y$ is set of output ports and values
- $D$ is set of component names
- $M_d$ is a component with name in $D$
- $EIC$ is External Input Coupling specification
- $EOC$ is External Output Coupling specification
- $IC$ is Internal Coupling specification
- $Select$ is tie-breaking function

A possible metamodel of DEVS with ports [78] is illustrated in Figure 4.9. The following is a brief description of the metamodel entities and relationships (constraints are omitted to save space):
Figure 4.9: Metamodel of DEVS with ports
**CHAPTER 4. MDE AT WORK**

atomic devs corresponds to basic DEVS model with ports. It is represented as solid rectangle with its name above the top left corner (see Figure 4.10(a)). Atomic DEVS may contain states connected by transitions.

**state** allows to specify functions \( ta \) and \( \lambda \). A state is represented as solid gray circle with its name in the center (see Figure 4.10(b)). The initial state is distinguished with a marker.

**internal transition** relationship specifies function \( \delta_{int} \). It is represented as solid black connector with the arrow end pointing to the target state.

**external transition** relationship specifies function \( \delta_{ext} \). It is represented as solid dark red connector with the arrow end pointing to the target state.

**contains state** relationship assigns a state to an atomic DEVS. All states assigned to the atomic DEVS form set \( S \) of the component. This relationship is not shown in the graphical representation.

**coupled devs** represents Coupled DEVS with ports and provides function \( Select \). Appearance of Coupled DEVS is the same as that of Atomic DEVS, but instead of states, it contains components \( M_d \).

**contains model** relationship allows to specify components \( M_d \) comprising a coupled model. This relationship is not shown in the graphical representation.

**port** is a component's input or output interface. A port is represented as relatively small square with its name labeled next to it (see Figure 4.10(c)).

**contains port** relationship assigns a port to an atomic or coupled model. This relationship allows to derive sets \( X \) and \( Y \) for the given atomic or coupled model. This relationship is not shown in the graphical representation.

**channel** relationship specifies connections between ports of DEVS components. Note that one outport can be connected to many inports. Channels allow to derive \( IC \), \( EIC \) and \( EOC \) specifications of a hierarchical model. A channel is represented as solid grey connector with an arrow end pointing to the input port of a component.

Note that complete specifications for atomic and coupled DEVS can be derived with the help of the relationships of the metamodel.
4.3.2 PythonDEVS code generation

The rules of the PythonDEVS code generation transformation do not define any important model rewriting aside from annotating as outlined in section 7.12.2. Actions of rules can access the annotations and perform the code generation. For a description of the code generation schemas, reader is referred to the original publication by Posse and Bolduc [78].

To give an example of code generation for PythonDEVS, consider the model in Figure 4.11. In the atomic model A, there is an external transition labelled $evt$ from state $s0$ to state $s1$. This transition has as condition the following Python script:

```python
if e < 1.0:
    if il == 'a' and i2 == 0 or il == 'b' or i2 > 0: return 1
    else: return 0
elif e < 2.0: return i2 >= 1
else: return 0
```

Then the PythonDEVS code generated for A, is as follows:

```python
class A(AtomicDEVS):
    def __init__(self):
        AtomicDEVS.__init__(self)
        self.state = 's0'
        self.elapsed = 0.0
        self.il = self.addInPort()
        self.i2 = self.addInPort()
        self.o1 = self.addOutPort()
        self.o2 = self.addOutPort()
        self.guard1_condition = lambda e, il, i2:
            if e < 1.0:
                if il == 'a' and i2 == 0 or il == 'b' or i2 > 0: return 1
                else: return 0
            elif e < 2.0: return i2 >= 1
            else: return 0

    def extTransition(self):
        s = self.state
        e = self.elapsed
        il = self.peek(self.il)
        i2 = self.peek(self.i2)
        if s == 's0':
            if guard1_condition(e, il, i2):
                return 's1'
        else: return 0
```
CHAPTER 4. MDE AT WORK

4.4 Process Interaction (PI)

This view is one of the four worldvews [2] to represent discrete-event systems. A discrete-event system is described with notions of ENTITY, ATTRIBUTE and EVENT. An entity denotes an object of interest in the system; entities can be static or dynamic. An attribute provides information about a property or an aspect of an entity or a system. An event triggers an activity and denotes anything that causes changes in the state of an entity or a system. Activity is what actually changes the state of an entity over time. An activity begins with an event and ends with an event. A process is a dynamic entity containing a sequence of activities ordered in time. A resource is a static entity that provides services to processes.

One possible metamodel of the PI formalism is described in section 4.4.1. The metamodel specifies abstract and concrete syntax. As we have see in section 3.2.2, meaning of a language can be defined in a number of ways. Next to translating the language concepts to a simulation language that has a precise executable semantics (see sections 4.2.2 and 4.3.2), in section 4.4.2 we illustrate how semantics of a language is defined by means of operational semantics.

4.4.1 PI metamodel

One possible metamodel of the PI formalism is illustrated in Figure 4.12. The following is a brief description of the key PI constructs:

**Machine** is a synonym of the RESOURCE concept. Machines have the following attributes:

- **Name**: is a unique identifier of MACHINE: String
- **State**: denotes availability of MACHINE to serve: Enum{idle, busy}
- **Tproc**: is typical duration of service: Time
- **Capacity**: denotes capability of MACHINE to serve: N
- **Load**: denotes MACHINE’s capacity occupied with serving: N

Visual presentation of MACHINE is in Figure 4.13

**Piece** is a synonym of the PROCESS concept. Pieces have the following attributes:

- **Name**: is a unique identifier of PIECE: String
- **Tcreation**: is event time: Time
- **Tinitproc**: is the start time of being served by a machine: Time
- **Tendproc**: is the end time of being served by a machine: Time
- **Body**: is a sequence of events: sequence{event}
- **EVnext**: is the iterator for events in body: N
- **ResourceID**: contains names of acquired machines: sequence{Name}
Timer
- Time type: Float init. value
- FinalTime type: Float init.
- Name type: String init. val

ProcrntGenerator
- IAT type: Float init. value
- Desp type: Float init. valu
- IntTime type: Float init.
- MaxTransactions type: Inte
- Tnext type: Float init. val

Machine
- Name type: String init. val
- IAT type: Float init. value
- Tproc type: Float init. val
- Tend type: Float init. valu
- Capacity type: Integer

MachineMapping
- Name type: String init. val

MachineQueue
- Name type: String init. val
- Tcreation type: Float init.
- Tintproc type: Float init
- Tendproc type: Float init.
- Name type: String init. val
- Body type: List init. value

Process
- Tcreation type: Float init.
- Tintproc type: Float init
- Tendproc type: Float init.
- Name type: String init. val
- Body type: List init. value

Piece
- Tcreation type: Float init.
- Tintproc type: Float init
- Tendproc type: Float init.
- Name type: String init. val
- Body type: List init. value

PieceDeclarations
- Name type: String init. val

Figure 4.12: PI metamodel
Visual presentation of PIECE is in Figure 4.14.

The presented model is based on the PI metamodel developed by Juan de Lara [21]. The original is adjusted and extended for modeling the behaviour of the Controller (see section 7.11.2). The following additions are Controller-specific:

**MachineMapping** entity denotes a space of machines. It corresponds to set of resources $\mathcal{F}$ in $\pi$Demos.

**ProcessClass** entity declares a class of process as in $\pi$Demos.

**PieceDeclarations** entity denotes a namespace of process classes. In contrast, the corresponding $\pi$Demos’ set of names $\Sigma$ contains defined names of all entities in the system.

Furthermore, three new relationships that are not directly related to $\pi$Demos nor to the PI worldview, are added:

- **manageElement** relationship indicates an operation on a sequence element:
  \[ \text{Enum}\{\text{append, insert, remove}\} \]

- **manageItem** relationship indicates an operation on an item of a mapping:
  \[ \text{Enum}\{\text{update, delete}\} \]

- **has** relationship indicates association between a mapping and an item.

The relationships above allow explicit expression of the operational semantics of operations on mapping and sequence data structures (see Table 4.3).
### Table 4.2: Graph grammar rules of Controller activities

<table>
<thead>
<tr>
<th>Event</th>
<th>GG Rule</th>
<th>Priority</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A</td>
<td>IMPORTPROCESS</td>
<td>50</td>
<td>Adds an external process to EL.</td>
</tr>
<tr>
<td>NEWR</td>
<td>NEWR</td>
<td>40</td>
<td>Creates a new resource.</td>
</tr>
<tr>
<td>DECP</td>
<td>DECP</td>
<td>40</td>
<td>Creates a new process class.</td>
</tr>
<tr>
<td>NEWP</td>
<td>NEWP</td>
<td>40</td>
<td>Creates a process from a class.</td>
</tr>
<tr>
<td>GETR</td>
<td>GETR</td>
<td>40</td>
<td>Acquires a non-busy resource.</td>
</tr>
<tr>
<td></td>
<td>BLOCKPROCESS</td>
<td>40</td>
<td>Blocks the process acquiring a busy resource.</td>
</tr>
<tr>
<td>HOLD</td>
<td>HOLD</td>
<td>41</td>
<td>Simulates using resource for known time by scheduling the process into the future.</td>
</tr>
<tr>
<td>USER</td>
<td>USER</td>
<td>40</td>
<td>Moves the process to the resource for unknown time until service is complete.</td>
</tr>
<tr>
<td>N/A</td>
<td>RELEASERESOURCE</td>
<td>30</td>
<td>Moves a served process from the serving resource to EL. This rule complements USER.</td>
</tr>
<tr>
<td>PUTR</td>
<td>PUTR</td>
<td>41</td>
<td>Releases an occupied resource.</td>
</tr>
<tr>
<td>CLOSE</td>
<td>CLOSE</td>
<td>40</td>
<td>Unblocks a delayed process.</td>
</tr>
<tr>
<td>N/A</td>
<td>REMOVEPROCESS</td>
<td>25</td>
<td>Deletes a valid process that has exhausted its events.</td>
</tr>
<tr>
<td>N/A</td>
<td>PROCESSERROR</td>
<td>25</td>
<td>Aborts execution if unhandled process error occurs.</td>
</tr>
<tr>
<td>MAIN</td>
<td>ADVANCE TIME</td>
<td>100</td>
<td>Advances logical time.</td>
</tr>
</tbody>
</table>

### Table 4.3: Graph grammar rules for auxiliary operations on data structures

<table>
<thead>
<tr>
<th>Priority</th>
<th>Rule</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DELETEITEM</td>
<td>Removes an item from a mapping.</td>
</tr>
<tr>
<td>2</td>
<td>UPDATEFIRSTITEM</td>
<td>Adds an item to an empty mapping.</td>
</tr>
<tr>
<td>3</td>
<td>UPDATEITEM</td>
<td>Adds an item to a mapping.</td>
</tr>
<tr>
<td>11</td>
<td>POPLASTELEMENT</td>
<td>Pops the first and last queue element.</td>
</tr>
<tr>
<td>12</td>
<td>POPELEMENT</td>
<td>Pops the first queue element.</td>
</tr>
<tr>
<td>14</td>
<td>SORTELEMENTS</td>
<td>Sorts two neighboring elements in ascending order by a given property.</td>
</tr>
<tr>
<td>15</td>
<td>APPEND1STELEMENT</td>
<td>Append an element to an empty queue.</td>
</tr>
<tr>
<td>16</td>
<td>APPENDELEMENT</td>
<td>Append an element to a queue.</td>
</tr>
</tbody>
</table>
4.4.2 PI operational semantics

The PI operational semantics is based on πDemos [4], a small process-oriented discrete event simulation language that provides simple and consistent formulation of the synchronization mechanism for process-oriented systems.

The operational semantics of activities is defined as graph grammar rules. Table 4.2 lists all original πDemos events and their corresponding rules. In addition, operational semantics of operations on queues and mapping data structures were factored out into separate rules. This factorisation helped to remove variability solely attributable to incidental conditions due to our realization of the data structures in the visual language defined by the PI metamodel. The result was reduction of the total number of rules and better focus on the essence of the activity behaviour. Rules implementing operational semantics of data operations are summarised in Table 4.3.

Both activity and data operation related rules form the graph grammar model of the operational semantics. Currently, there are some twenty rules in the model. For the sake of brevity, we present a detailed description of an example rule, followed by description of the event execution model.

4.4.2.1 Example rule

Rule PROMOTEPROCESS releases a busy resource that delays at least one process. The state of such resource remains busy and the blocked process is moved from the head of waiting queue DELAY to EL. The rule is executed if:

1. The LHS shown in Figure 4.15a is matched in the host graph (note that in the figure, a machine corresponds to a resource and a piece to a process).

2. Associated condition is true: the machine in LHS is the one referred to in the imminent event putR.

If the above holds, the matched part of the host graph is substituted with the subgraph shown in Figure 4.15b. Note new objects labelled 10, 11, 13. The entities and relationships in RHS are initialised as follows:

1. Objects copied from LHS retain their properties.

2. Event pointer EVnext of the current process Piece21 is incremented.

3. Properties of blocked process Piece5 are copied to Piece10.

Finally, the action of the rule specifies positions of the RHS objects in order to enhance the visual representation of this rule’s effect.

4.4.2.2 Event execution

The event execution is based on the next-event approach:

Next event to take place is always the imminent event in the body of the current process.
CHAPTER 4. MDE AT WORK

Figure 4.15: Subgraphs of the PROMOTEProcess rule
Informally, the operational semantics of event execution is as follows: if \( EL \) is empty, \textsc{controller} idles until at least one event notice \( (en) \) appears on \( EL \). Otherwise the first \( en \) becomes current. If the body of the process referred by the current \( en \) is empty then this \( en \) is removed from \( EL \) and \( EL \) is examined again. Otherwise, \textsc{controller} idles until time event time \( (evt) \) of current \( en \) and triggers the imminent event from the body. Unless there is shutdown, execution starts again. Note that whenever \textsc{controller} is idle, \( EL \) is being updated with new \( en \)s that might have arrived since the previous update of \( EL \).

The operational semantics of the execution is realised by organising rules of the controller model into groups, each group having its own base priority \( P_{\text{base}} \). These groups, in the order of decreasing priority are: data operations, process removal, activities, process import and time advancement. Within the same group, a rule is assigned relative priority \( P_{\text{rel}} \). If pattern matching of two and more rules within a group is deterministic on the basis of \textit{LHS}s and conditions, then these rules can share the same priority level. The following illustrates the priority assignments for the groups of rules:

1. Data operations: \( P_{\text{base}} = 0 \)
   
   (a) Operations for mapping structures: \( P_{\text{rel}}^{\text{move}} > P_{\text{rel}}^{\text{update}} \)
   
   (b) Operations for queue structures: In order to minimize the size of queue and increase efficiency of visual sorting, \( P_{\text{rel}}^{\text{delete}} > P_{\text{rel}}^{\text{sort}} > P_{\text{rel}}^{\text{insert}} \)

2. Process removal: \( P_{\text{base}} = 20 \). In order to minimize the size of queue \( EL \), \( P_{\text{base}} < P_{\text{rel}}^{\text{PI}} \)
   
   (a) Removing processes: \( P_{\text{rel}}^{\text{remove}} = 0 \).
   
   (b) Rules for error handling: \( P_{\text{rel}}^{\text{error}} > P_{\text{rel}}^{\text{move}} \)

3. Activities: \( P_{\text{base}}^{\text{PI}} = 30 \)
   
   (a) Releasing resources: \( P_{\text{rel}} = 0 \). In order to maximize availability time of resources and minimize delay of blocked processes, \( P_{\text{rel}} < P_{\text{rel}}^{\text{PI}} \)
   
   (b) Rules for \( \pi \)-Demos commands: \( P_{\text{rel}}^{\text{PI}} = 10 \)

4. Process import: \( P_{\text{base}} = 50 \)
   
   In order to minimize the size of queue \( EL \) and increase efficiency of visual operations, \( P_{\text{base}} > P_{\text{base}}^{\text{PI}} \)

5. Time advancement: \( P_{\text{base}} = 100 \)
   
   Logical time may pass after all other rules have been tried.

Concrete priorities for individual rules are derived from formula: \( P_{\text{base}} + P_{\text{rel}} \). Example rule priorities are given in Tables 4.2 and 4.3.
4.5 Dynamic Traffic Routing (DTR)

Dynamic Traffic Routing is one of the research topics at Man-Machine Interaction Group, TU Delft. More specifically, the research is focused on use of Ant-Based Control (ABC) algorithms for dynamic vehicle routing in cities. A typical project would produce improvements to the algorithm or its use. Before an improved routing system can be implemented in the real world, it has to be thoroughly tested. For this purpose a simulation environment would be created and complemented with some kind of facility to define traffic situations in the simulator-specific format.

For the above mentioned domain we briefly introduce basic concepts of road networks and propose a possible definition of the modeling language. A transformation definition is illustrated for a dynamic vehicle routing simulator developed at MMI.

4.5.1 Metamodel of a traffic language

In general, traffic is a part of a geographical map with locations and roads. A road is a unidirectional named way between two locations, its starting and ending points. A location has specific coordinates and a name. In the context of a road, two kinds of locations can be distinguished: “from locations” and “to locations”. An intersection is a location that represents the starting or ending point of two or more roads.

A possible metamodel of the traffic language is illustrated in Figure 4.16. The following is a brief description of the metamodel:

Road is a named entity characterized by the number of lanes, its length, maximum allowed speed and priority.
Intersection is a named entity characterized by a type and position on map. Position is defined by a non-negative x- and y- coordinate. Intersection type can be normal, with traffic lights or roundabout. An intersection with traffic lights is characterised by a cycle, which is the time in seconds before the light cycle starts again.

to_location relationship specifies an intersection where a road is going to.

from_location relationship specifies an intersection where a road is coming from.

Light relationship specifies traffic light directions for a road going to an intersection of the traffic light type. Per road, there are three possible directions: left, ahead and right. Each direction can have a different duration of the green light.

Any road must be connected to one 'from_location' relationship and one 'to_location' relationship. The same holds for any intersection. Moreover, an intersection must be connected to at least two 'from_location' relationships or at least two 'to_location' relationships. Finally, all three relationships are allowed to be connected to one intersection and to one road exactly.

4.5.2 ABC Code Generation

ABC (Ant-Based Control) simulator is a dynamic router planner developed by Ronald Kroon [46] at MMI, TU Delft. To clarify the simulator-specific construction rules, we briefly introduce the format of target models. An ABC simulation model at minimum requires the following text files (see Figure 4.17): a file with the extension '.int', which defines type of each intersection. A file with the extension '.road' specifies whether roads are main, their number of lanes, the length and the maximum speed. A file with the extension '.map' defines which intersections are connected by which roads and in what direction. A file with the extension '.light' assigns traffic lights to possible directions at proper intersections. Finally, a '.city' file is the main file that states all files that together form a traffic model. All model information is structured and written in a C-like style.

The demonstrated transformation is defined according to the code generation guidelines described in section 7.12.2. The generator is designed so that each rule addresses a particular aspect, e.g. relations among concept instances or instance properties of a certain concept, and writes extracted information into one or more files of the target simulation model. Furthermore, each rule has an associated condition that prevents the rule from matching to the same subgraph of the host graph more than once. Main rules
CHAPTER 4. MDE AT WORK

<table>
<thead>
<tr>
<th>Priority</th>
<th>Action</th>
<th>Rule</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>defineIntersections</td>
<td>Finds an intersection and adds its definition to the &quot;.int&quot; file.</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>defineRoads</td>
<td>Finds a road and adds its definition to the &quot;.road&quot; file.</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>mapIntersections</td>
<td>Finds an intersection and adds its id and position to the &quot;.map&quot; file.</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>mapRoads</td>
<td>Finds a road, its from and to intersections and adds ids of the three to the &quot;.map&quot; file.</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>selectLight</td>
<td>Finds and select an intersection with a traffic light.</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>defineLight</td>
<td>Given a selected intersection, finds an incoming road and defines lights for the road.</td>
</tr>
<tr>
<td>last</td>
<td>FINAL</td>
<td>-</td>
<td>Finalizes &quot;.int&quot;, &quot;.road&quot;, &quot;.map&quot; and &quot;.light&quot; files.</td>
</tr>
</tbody>
</table>

Table 4.4: Main graph grammar rules of the ABC code generator

of this definition are summarised in Table 4.4. To save space, we detail just one sample rule.

![Left-hand side (LHS)](image1)

![Right-hand side (RHS)](image2)

Figure 4.18: Subgraphs of the MAPROADS rule

Rule MAPROADS is executed if:

1. The LHS shown in Figure 4.18a is matched in the host graph.

2. Associated condition is true: the road in LHS has not been processed yet by this rule, i.e. the road has no 'visited' annotation (see the code generation guidelines).

If the above holds, the matched part of the host graph is substituted with the subgraph shown in Figure 4.18b. Following the above code generation guidelines, nothing is changed, nor created or deleted in the host graph. Code is generated in the action of the rule according to the following steps:
1. open the `.map` file, append road id and ids of the start and destination intersec­tions and close the file.

2. finally, annotate the road in RHS as visited by this rule.

Execution of this transformation definition starts with the initial action which creates files of an empty ABC model. After that, the LHS of some rule is matched against some subgraph of the host graph, checking if the associated condition is true, and if so, replacing that subgraph by the corresponding RHS of the rule, subsequently performing any additional actions associated with the rule. For example, rule mapRoads will be executed once for each road without the 'visited' annotation. If more than one rule matches the host graph, priorities act as tie-breakers. When no match is found among all rules, the final action is executed, finalizing model files and removing temporary annotations from the host graph.

4.6 Summary

This chapter illustrated how MDE concepts are used to model domain specific languages for a number of M&S formalisms. Each language was treated as system and modeling all language aspects, including the semantcs, was exhaustively illustrated. Please note that demonstrated MDE concepts are applicable to languages outside the M&S domain as well. Indeed, chapter 6 will show how modeling languages for engineering domains are defined using the same concepts. With the help of an MDE tool, such as AToM³, these abstract models of languages are automatically transformed into language implementa­tions that can be executed by the original or another target modeling tool. Most of the models in chapters 7 and 8 are created with the modeling languages realised from the models of languages defined in this chapter and chapter 6.
Chapter 5

From Models to Implementation

Chapters 2 and 3 introduced parent theories and provided information necessary to solve the research problem. This chapter describes the methodology that addresses the research problem. Section 5.2 establishes a generic and domain independent framework for the family of M&S applications. This framework is expressed in terms and concepts from the related parent fields (see Chapter 2). An important characteristic of a cMoS system is repeated application domain customization during the operational phase of life-cycle. In section 5.3, we formulate activities and artefacts specific to customization and operation of a cMoS system as a meta-architecture. Application of model-driven engineering (MDE) to this meta-architecture enables efficient and programming-free adaptation of cMoS systems to new Web-Based M&S applications. In section 5.4, two model-driven development processes are proposed in order to create and more importantly, customize software instances of the framework. The processes are based on the fundamentals of the model-driven engineering (MDE) and employ the outstanding MDE approaches that were introduced in section 3.3. The chapter, however, starts with a brief overview of how focus of this research shifted from coding to modeling.

5.1 From an implementation to models

The research presented in this thesis started with the development of a web-based environment for a collaborative M&S application in the NanoComp project (also referred as stakeholder) at Delft University of Technology (TU Delft). The stakeholder requirements were:

The environment should enable concurrent users to share project models and solve models with shared simulators accessible through the Web. Models would be developed off-line by modellers using familiar modeling tools. Selected models can be published in the environment, so that other users can find, simulate and download published models through the web. The downloaded models can be reused off-line using proper modeling and simulation applications.

The beginning of the research coincided with the birth of Web-Based Modeling and Simulation (see section 1). Participation in modeling and simulation conferences and literature study revealed many similar ongoing projects, yet none of the available solutions
were easy to reuse within the NanoComp context for a number of reasons: the Java-orientated environments were inappropriate due their limitation of modeling languages to Java. Other environments that integrated users modeling and simulation tools, naturally supported the way of working specific to their original stakeholders. Tailoring a third party environment to the NanoComp needs required a development effort using technologies chosen by the third party. Hence it was chosen to develop a new environment.

To avoid following the suit of other solutions, this research postulated an additional requirement:

**Tailorability:** application domain customization of the environment should be so efficient that it becomes feasible to realise an arbitrary number of M&S applications.

This requirement was originally sought to be achieved only through goodness of traditional development.

Due to the affiliation with the NanoComp project, the environment was named NanoComp Simulation Environment (NCSE).

### 5.1.1 The NCSE prototype

This section describes a prototype of NanoComp Simulation Environment. For reasons of simplicity and space, only architecture is described. The three-layer architecture of the NCSE prototype is shown in Figure 5.1 and the layers themselves are described below in a bottom up sequence.

The bottom layer contained the environment’s models and off-the-shelf simulators, distributed over a network. Simulators were individually integrated using the wrapping technique. Various applications running on heterogeneous software and hardware platforms were successfully integrated in NCSE [52, 56]. Models were registered by modellers and uploaded to shared repositories. Simulation outputs were stored in temporary repositories until they were registered as models or purged. Models were available only as files
in formats native to their modeling and simulation tools. Repositories were provided by file servers.

The middle layer consisted of a web-based environment with software agents and a centralised resource manager called Controller [51]. In general, both shared models and shared simulators could be resources. In the NCSE context, there were no reasons to consider models as limited resources, hence authenticated users could download and request online simulations unrestricted. Simulators, however, were considered resources due to a number of issues, e.g. licenses restricting concurrent operation of commercial products and numerous simulation instances exceeding computational capabilities of deployment nodes.

The other part of the middle layer consisted of agents that helped web users to register new models and find, simulate and download registered models. The activities performed by the agents and data flows among them are illustrated in Figure 5.2 using the Gane and Sarson's DFD notation [34]. The Trader Agent assisted modellers to (un)register files stored in model repositories and search among registration records. The Mediator Agent helped to download model files or remotely execute them on one of the associated simulators. Simulation results in a temporary file repository were treated like models too: the Trader could create a registration record and associate it with an available simulator. The Representation Agent was hard coded to interpret data produced by certain simulators and meaningfully present supported data models to the experimenter. The latter could decide to discard or keep the results by registering them.

Finally, the front layer provided interface to the environment. For example, users could interact with the agents by means of web clients.

5.1.2 Shift to models

From the user point of view, the NCSE prototype was a kind of network file storage with remote simulation capabilities. Moreover, the application required users to make additional manual efforts to maintain a database of shared models. At a closer look, the NanoComp Simulation Environment suffered from a number of major shortcomings:

- Freedom of choosing traditional modeling and simulation tools (refer to the tailorability requirement on page 66) resulted in an environment with models in multiple tool specific formats. These often closed formats limited portability and reusability of models. Moreover, such models are black boxes that could not be interpreted and processed by agents, thus limiting functionality of the environment.

- Very dynamic behaviour of the controller was complex to code, thus making this component difficult to adapt to environments other than the NCSE prototype. This was limiting applicability of the proposed solution.

- Adding support for a new family of models or file format required manual programming of software agents.

Further development was needed to include new tools, families of models and agent functionality, but already at that time development went with difficulty because:
CHAPTER 5. FROM MODELS TO IMPLEMENTATION

- Documentation was lagging behind the actual implementation, thus making NCSE hard to maintain and extend.

- Manual programming required increasing development resources to keep up with the evolving and growing character of the environment.

Because of the development problems, the NCSE prototype never reached a stable release.

In summary, environment was hard to tailor to new modeling paradigms and NCSE models were not portable. It was clear that the requirements of this project could not be met with the traditional development approach and traditional tools. Gradually, our initial code-oriented approach was discarded and a new solution direction was found in model-driven engineering.

In the light of the new research direction, the project requirement on page 66 was redefined and extended as follows:

**Tailorability:** the environment should be easily tailorable to modeling and simulation needs specific to target application domains.

**Reduced programming:** the environment should be generated from models as much as possible.

Note that the new tailorability requirement was oriented towards client's needs rather than client's tools. The rationale behind this was that tools are mere means to address needs and for a given need multiple tools can be used. This relaxation of the original requirement allowed to replace traditional user tools with equivalent MDE tools.

The NCSE prototype was analysed and generalised into the cMoS framework (see section 5.2), model driven engineering was applied to the development process (section 5.4) and domain specific applications were defined through a programming-less meta-architecture (see section 5.3). In record for this project times, above-mentioned issues were solved, new functionality was added, hard coded parts of NCSE were modelled and auto-generated and domain specific applications were created with models (see Chapter 6).

5.2 A framework for Collaborative Modeling and Online Simulation

The COLLABORATIVE MODELING AND ONLINE SIMULATION (cMoS) framework specifies a family of M&S applications with collaborative and distributed properties. The framework embodies the general concepts in fields of Modeling and Simulation, Distributed Systems and Collaborative Software. The framework is illustrated in Figure 5.3.

cMoS is primarily oriented towards modeling and simulation, which is evident from the basic M&S concepts (see section 2.1) forming its foundation. Note that the proposed framework does not affect modeling, therefore entity SYSTEM and relationship MODELING are omitted for the sake of simplicity. To explicitly indicate that modeling languages
are not fixed, the framework includes the concept METAMODEL [27]. Metamodel specifies LANGUAGE and TOOL with which MODEL is developed. TRANSFORMATIONS are used to convert models among various model dimensions, which are discussed in section 5.2.7. Coupling of and transformation between models specified in different paradigms leads to Multi-Paradigm modeling [69].

A general collaborative requirement (see page 65) is sharing project models and simulators. Figure 5.4 illustrates this requirement using the group size and interactivity based classification of collaborative software (see section 2.3). Concept REPOSITORY is essential to the Library collaborative model. USER is a part of the Community or Team model. The Solicitation model is not applicable to and Process Support is not required in the problem domain of this research. Therefore these two collaborative models are grayed out in the figure.

The word 'online' in the framework’s name indicates that simulations are accessible through the Internet. One of the key characteristics of distributed systems is resource sharing (see section 2.2.1). In cMoS, simulators are limited and shared resources, which is reflected in the relationship between SIMULATOR and USER: more than one user can interact with a (shared) simulator. Furthermore, users are separated from both models
and simulators, which creates an opportunity for concurrency. Moreover, multiple independent instances of a simulator are possible, which allows multiple users to carry out simulations concurrently. Concurrent simulations and sharing of simulators raises the need for resource management. CONTROLLER is a resource manager that synchronizes access to shared simulators. The way CONTROLLER is connected to relationship SIMULATOR-USER, indicates that its operation is transparent for the ends of the relationship. Table 5.1 summarises possible contributions of the involved cMoS concepts towards usefulness of a Web-based distributed system. The cMoS contributions are not given, but are gained through design described in consequent sub-sections.

POLICIES are technical, administrative, procedural, organizational, and other global rules that can constrain operation of the environment. Some of the examples are: intellectual property rights, license restrictions and load balancing. Concrete rules are defined within context of a framework instance. Policies are implemented by collaborative software and the controller.

In this dissertation, any instance of the cMoS framework is generally referred to as AcMoS (a cMoS). An optional prefix indicates the application domain of the instance, for example: DEVS-AcMoS means a cMoS instance for the DEVS modeling and simulation.

5.2.1 Separation-related issues

Separation of objects is a natural property of distributed systems, therefore cMoS needs to address a number of additional separation-related issues.

Access issue arises when there is a difference between accessing local and remote objects. Hiding this difference constitutes access transparency, an important characteristic of distributed systems. This issue can be solved with remote proxy, a particular type of software design patterns [33]. In the cMoS context:

Remote proxy provides a reference to a remote object and acts as local interface to the latter.
CHAPTER 5. FROM MODELS TO IMPLEMENTATION

In this thesis, \( O' \) denotes a remote proxy of object \( O \), where the type of \( O \) can be any concept from the cMoS framework.

**Location** issue arises when in order to access a remote component, additional information is needed about its location and access methods. The totality of this information is referred to as *Metadata for Resource Access* (MRA). MRA is a set structure:

\[
MRA = \text{Set}(\text{attribute})
\]

where

\( \text{attribute} \) is a metadata term.

MRA is essential to *location transparency* of cMoS.

**Discovery** issue exists because distributed resources may lack detailed description, thus making it hard for an external user to find contents matching his or her search criteria. This issue can be addressed by enhancing such resources with additional searchable information [84]. In this thesis, the totality of this information is referred to as *Metadata for Resource Discovery* (MRD). MRD is a set structure:

\[
MRD = \text{Set}(\text{attribute})
\]

where

\( \text{attribute} \) is a metadata term.

MRD and MRA can be used with both remote objects and their proxies. Concrete sets of terms for MRD and MRA are best determined in the context of a specific cMoS instance. Standards for interoperable online metadata terms can be found in [5].

### 5.2.2 User

**User** is a member of a collaborative team or community.

From the definitions of the Team and Community models in section 2.3, the user may act in one of the following roles:

- **Contributor** is a creator of models (modeller). Contributors are typically members of a collaboration team.
- **Consumer** is a user that reads models. Consumers are typically members of a collaboration community.
- **Experimenter** is a user that uses the simulators of the environment.

A user can be a human or a computational system.
5.2.3 User-Simulator relationship

This relationship is included due to the fact that simulation always requires an interaction in the beginning of and possibly one or more interactions in the course of execution. Such interaction has to overcome two obstacles:

1. Due to the separation of cMoS elements, a user is not able to interact directly with a simulator.
2. Experimenters may not know how to interact with the simulator. This is especially the case among the community users.

The first obstacle can be circumvented with the help of a remote proxy for SIMULATOR. The second obstacle is solved if interaction knowledge from simulator experts can be reused by experimenters.

5.2.4 Scenario

Scenario is knowledge of a goal-driven interaction steps between a user and a specific simulator. Scenario is a structure:

\[ \text{Scenario} = (\text{MRD}, \text{Body}, \text{Params}, \text{MRA}) \]

where

- \text{MRD} is discovery metadata for simulation output.
- \text{Body} contains template of interaction instructions expressed in simulator specific syntax: \text{Sequence}: \{string\}.
- \text{Params} is a set of parameter values for \text{Body}.
- \text{MRA} is metadata for access to simulator.

Useful scenarios are created in advance and can be re-used by experimenters during the lifetime of the corresponding simulator. Scenarios are not always needed. Scenarios must be executable by simulators.

5.2.5 Controller

Controller is a resource manager that synchronizes access to shared simulators.

The way CONTROLLER is connected to relationship USER-SIMULATOR, indicates that its operation is transparent for the ends of the relationship.

In the cMoS framework, the Controller is the most complex element because it has to deal with dynamic scenarios. Normally, elements would be modelled with software concepts (refer to section 5.4.1). Had the Controller been completely defined in terms of domain-specific concepts instead of more detailed software concepts, it would have made this component easier to adapt or change. An approach to such conceptual development is described in section 5.4.2 and the controller itself is modelled in section 7.11.
### 5.2.6 Tool

**Tool** is an MDE-tool with support for modeling, meta-modeling and transformation.

The role of the Tool in the context of customization of a cMoS system is described in section 5.3.

### 5.2.7 Model

This concept embraces the broad range of models in the M&S discipline. Many classifications exist that provide an organized view on the variety of models and highlights their most important classes. The rest of this section describes categorizations of models related to relationships of MODEL with the other elements of the framework. The categorization dimensions and related cMoS elements are summarized in Table 5.2.

#### 5.2.7.1 Modeling formalism and languages

Language with the related concept of Metamodell, is the most important point of variation in the cMoS framework. Numerous formalisms can be identified for example on the basis of common types of segments from the system specification by Zeigler [100]. In [94, page 51], Vangheluwe summarises a number of formalisms and semantic relationships among them in the Formalism Transformation Graph (FTG) as illustrated in Figure 5.5. Each node of the graph represents a formalism. Solid arrows indicate behaviour-preserving formalism transformations. The horizontal line at the bottom represents the observation frame at which input and output data trajectories can be described. Vertical dotted arrows indicate availability of a simulator that maps models of a formalism onto a behaviour at the trajectory level. Note that for the sake of readability the availability information is incomplete and was deliberately reduced in the figure.

For each formalism, one or more languages can be in use. For the sake of simplicity we assume a one-to-one relationship between a formalism and a language. The arrows in FTG indicate possibilities to express semantics of a language. In chapter 4 of this thesis a number of modeling formalisms are introduced and their corresponding languages are defined using MDE concepts of meta modeling and transformation.

#### 5.2.7.2 Simulator independence

The M&S framework separates model from simulator [100]. This separation provides a number of benefits, among which are portability and reusability as the same model may

<table>
<thead>
<tr>
<th><strong>DIMENSION</strong></th>
<th><strong>RELATED cMoS ELEMENT</strong></th>
<th><strong>REMARK</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Formalism</td>
<td>Language, Metamodel</td>
<td>See e.g. section 8.1</td>
</tr>
<tr>
<td>Abstraction</td>
<td>Simulator</td>
<td>See e.g. section 8.1</td>
</tr>
<tr>
<td>Storage</td>
<td>Repository, Tool</td>
<td>See Figure 8.7 and 8.4</td>
</tr>
</tbody>
</table>

Table 5.2: Dimensions of M&S models in the cMoS context
Round node - continuous formalism
Square node - discrete formalism

Figure 5.5: Formalisms (adapted from [94])

- PDE
- KTG
- System Dynamics
- Bond Graph (a-causal)
- Bond Graph (causal)
- DAE
  - non-causal set
  - causal set
  - causal sequence (sorted)
- Transfer Function
- Scheduling hybrid DAE
- Difference Equations
- DEVS
- DEVS & DESS
- Petri Nets
- Statecharts
- Cellular Automata
- Process Interaction
- 3 Phase Approach
- Activity Scanning
- Event Scheduling
- Data
be executed by different simulators and reused in another model. Parallels can be drawn
with model-driven engineering in which separation of designs from technological and
engineering platforms enables portability of engineering models to multiple technological
platforms (refer to section 3.3.1.1).

Consequently, two kinds of models are distinguished: conceptual models and simulation models. Robinson [80] provides a definition for the conceptual model:

"The conceptual model is a non-software specific description of the simulation
model that is to be developed, describing the objectives, inputs, outputs,
content, assumptions and simplifications of the model."

The key feature of such models is independence from the simulation software in which
the simulation is to be developed. Following this feature, the simulation model can be
defined as follows:

Simulation model is a conceptual model, whose instructions are explicitly
expressed in statements of the language of a simulation software.

From these definitions, it follows that simulation model is a refinement of a conceptual
model along the abstraction dimension.

It is worth noting that in this research both conceptual and simulation models are
amendable to interpretation by computer (refer to section 3.2.1). This is in contrast to the
traditional view, in which simulation models are computerized, while conceptual models
received less attention and implied to be non-executable "documentation" for simulation
model coding. A similar situation exists in the traditional software development, where
executable code is the primary focus, while system designs containing system descriptions
in diagrams and natural languages play the secondary role.

5.2.7.3 Storage independence

In general, to share models, common repositories should store models from different
modeling and simulation environments. In [3], Bernardi and Santucci motivated how
dissociating the contents of models from the format specific to concrete M&S environ-
ments is crucial to exchanging models for reuse in environments other than the original
ones. Reflecting this, the authors introduced a storage-related concept, which can be
paraphrased as follows:

Context refers to technological details irrelevant to the contents of a stored
model.

Consequently, the authors proposed to classify models with respect to their relation to
context:

Context-out model is an abstraction of model expressed in a context-independent
format.

Context-in model is a context-out model formatted for a direct use in a given
environment.

Note that the storage dimension is orthogonal to the abstraction dimension. The original
works by Bernardi and Santucci mostly concerned simulation models, while this research
focuses on conceptual models.
5.2.8 Repository

Repository is a place where models are stored, maintained and collaboratively shared. Two types of repositories are distinguished: Model type contains context-in and context-out models, and File type contains files of context-in models. Furthermore, a repository is temporary if its contents is purged on a regular basis.

5.2.9 Simulator and Solver

Solver is a limited resource provided by a simulator integrated into a cMoS instance. Functionally, solver executes scenario, which causes the integrated simulator to map an input simulation model onto a behaviour at the data level. Next, solver saves the generated data to a repository and registers the output as context-in or data model. Section 7.6 discusses development of solvers for an example AcMoS.

5.2.10 Simulation and Experiment

Simulation is a process that can be described with items of information such as start time, end time, output, output locations and so on. This information can be useful for experimenters. One way to capture such information is with the following concept:

Experiment is a persistent representation of a simulation process.

An example of experiment is modelled in section 7.10.

5.3 AcMoS application meta-architecture

Specific M&S applications are built on top of a cMoS instance (AcMoS). An AcMoS consists of physical and software instances of the cMoS elements TOOL, REPOSITORY, SOLVER, CONTROLLER and User. These instances are based on multiple meta levels (refer to section 3.2.1). The instances, their levels, and entities and relationships within those levels are specified by the AcMoS application meta-architecture as illustrated in Figure 5.6. Horizontal dashed swim lines indicate borders of modeling levels. An entity represents an artefact. A diamond shape entity represents a transformation. Entities marked with double border are not generic and have to be defined for each specific application domain. Such entities are variation points in application domain customization. Entities with no filling are "white box" systems. An entity, whose internal structure may be ill-defined is considered "black box" and shown filled with gray color. Finally, relationships are represented by arrows. Dotted connectors indicate relationships that are transformations. Transformations are defined according to one or more transformation definitions (this information is not shown in the figure).

At the system level, there are running instances of software. Solvers are considered "black box" systems whose internal meta-architecture does not go beyond the system layer. Solver can read input data from and write generated data to Repository. The
A special case of models in the repository are Data models in MDA behaviour of models of various formalisms is mapped onto data trajectories through simulation [100, 94, page 78].

A level higher, conceptual models exist in either TOOL or REPOSITORY. In TOOL, models conform to metamodels and therefore are well-defined open systems. Because a relation between TOOL to toolbox can be used recursively and meta-levels are imaginary, artefacts from all levels of the architecture can exist at this level: transformation models, meta-models, and models. To generate simulation models from conceptual models, a transformation is required. In REPOSITORY, models conform to schemas. A special case of models in the repository are Data models in MDA. Behaviour of models of various formalisms is mapped onto data trajectories through simulation [100, 94, page 78].

A level higher, conceptual models exist in either TOOL or REPOSITORY. In TOOL, models conform to metamodels and therefore are well-defined open systems. Because a relation between TOOL to toolbox can be used recursively and meta-levels are imaginary, artefacts from all levels of the architecture can exist at this level: transformation models, meta-models, and models. To generate simulation models from conceptual models, a transformation is required. In REPOSITORY, models conform to schemas. A special case of models in the repository are Data models in MDA. Behaviour of models of various formalisms is mapped onto data trajectories through simulation [100, 94, page 78].
CHAPTER 5. FROM MODELS TO IMPLEMENTATION

51], therefore models of the data formalism are expressly stated at this level. Given a mapping between a metamodel and a schema, conceptual models can be exchanged between TOOL and REPOSITORY. Through REPOSITORY, users can view and share stored models at the same level of detail as in TOOL. Note that context-out models may be stripped of well-formedness rules and concrete semantics as these are defined in their metamodels. Also, stored models should not be edited because repositories normally do not interpret and apply well-formedness rules. Storing and retrieving models can occur often, therefore it is convenient for users to have TOOL and REPOSITORY interoperating tightly at this level.

At the meta-level, all variety of application domains is defined by means of metamodels. Within TOOL, metamodels define textual and visual or combined languages that can express models in terms of specific domains, thus addressing the tailorbility requirement of cMoS (refer to page 69). Such metamodels are transformed (see "generate DSME" in the figure) into executable tool specifications, which are interpreted by TOOL to configure itself into a domain specific modeling environment (DSME). The "generate DSME" transformation is a way to specify metamodel’s semantics and is usually, but not necessarily, hard coded in TOOL itself. Moreover, from the same metamodels, schemas for database repositories are generated by means of automatic transformations. This enables all instances of any source metamodel to be ported to REPOSITORY in the model level and vice versa. The Data schema allows storing data generated by solvers as data models in the repository. Note that given the schemas of both conceptual and data models, relations can be established between data signals and corresponding variables in models. At the meta level, interactions between TOOL and REPOSITORY happen at infrequent times of adding a new M&S domain, therefore both do not need tight interoperation and generated schemas can be manually installed in repositories.

Finally, at the top level METALANGUAGES exist in TOOL. These languages are either hard coded or modelled. The latter normally requires another meta-level or self-reflection and is needed for special purposes, e.g. standardisation. Note that standard metalanguages and bridges between non-standard metalanguages play important role in tool interoperability and model sharing. cMoS requirements can be met by tools with hard coded metalanguages, therefore the architecture does not need to extend further. TRANSFORMATION DEFINITION LANGUAGE is considered a metalanguage as well. Potential quality of languages (refer to section 3.2.2) and therefore usefulness of cMoS, depends on language modeling capabilities of the metalanguages.

The described meta-architecture formulates activities and variation points, which are essential to application domain customization and AcMoS operation. Concrete M&S applications are defined by filling in the variation points of the architecture (entities with double border in Figure 5.6) with models of domain-specific parts. Application of model-driven engineering (MDE) to this meta-architecture enables efficient and programming-free adaptation of cMoS systems to new Web-Based M&S applications, thus contributing to the "Reduced manual programming" requirement on page 69.
CHAPTER 5. FROM MODELS TO IMPLEMENTATION

5.4 Model-driven engineering processes

This section applies MDE approaches to the development process in order to reduce system complexity and increase engineering efficiency. Two processes are proposed.

5.4.1 General software development

This approach is used when it is impossible to map concepts from the business domain to concepts in the technology domain. For example, when solution should be portable to multiple technology domains or such domains are not known in advance, as it is the case with the cMoS framework. In this context, a step-wise refinement approach like MDA is more appropriate.

Benefits of MDE approaches like MDA depend on increased levels of abstraction at which software is developed. At the time of this research, the focus of the software development process was on writing code [45]. Currently there is shift to writing PSMs, but benefits are still limited because there is virtually no abstraction gain with PSMs. In general, the most difficult step, creating PSM from PIM, has yet to benefit from model driven development.

PSM developers are traditionally equipped with UML profiles, relatively low-level extensions of UML close to code of target implementation technologies. In contrast to this tradition, the MDA framework is neutral towards modeling languages and domains. This independence from the prescribed OMG’s technologies forms the basis of this engineering process:

A high level and high quality DSM language specific to a particular technology is used instead of an UML profile.

In other words, the DSM approach is applied to the design stage of the MDA process. It becomes possible to apply DSM because domain knowledge of the target implementation technology is available at this stage.

While problem domain of such a DSM language is the technology domain, produced models are not straight visualization of code. Developers are shielded from detailed implementations of cross-cutting concerns, contracts and boilerplate coding typical for the chosen platform. Moreover, DSM languages are known for being rich, well constrained and highly communicative, features that enable quality models. Using a DSM language for a technology domain allows to raise development abstraction to levels higher then those typical for today’s PSM technologies of OMG.

The effect of introducing DSM on the MDA process is illustrated in Figure 5.7. The only difference is the additional step “High-level design” and its output artefact. At the new stage, developers create detailed domain models. Alternatively, they can transform PIMs into domain models and add missing details if needed. The output of this stage is validated and 100% complete domain models, from which complete PSM or Code can be automatically generated. The proposed approach is compatible with the MDA framework despite that it introduces a new step in the development process. The biggest change is application of DSM technologies for PSM languages.
5.4.2 DSL interpretation

In this approach a system is implemented with the help of (i) a domain specific modeling language, (ii) event generator and (iii) an event interpreter as follows:

At the development time, a domain expert uses a DSM language to model a system. At the run-time, the event generator provides the system with events and the event interpreter interprets these events and changes the system state in the model.

In [73], a similar approach was used to execute domain-oriented system requirements specifications. Due to the strong direct representation of domain concepts, this approach is close to DSM, but requires direct model-to-model transformations.

Figure 5.8 illustrates a conceptual model of a possible realization of the approach by means of meta-modeling and model transformations. In the figure, roles of the entities are shown in italic. A system and its events are described in a domain specific language. The interpreter is a transformation process executed by Transformation Tool that reads the source model and rewrites the same model according to the semantics of system events. Such transformation may change the state of the system and create internal events. Note that in order to communicate with external systems, the transformation should be able to generate external events as well. The definition of this transformation constitutes an executable specification of the system’s behaviour. The sequence of events is normally provided by the external context of the system. Alternatively, these events can be generated by a transformation. In that case, system behaviour is simulated.

This approach is characterised by a very high level of abstraction, direct involvement of domain experts and absence of software development. All these factors contribute to
fast development times. One disadvantage of systems based on DSL interpretation is a possible performance penalty due to the overhead introduced by the event interpreter. In the case of the described approach realization, there is an additional overhead associated with the transformation tool that interprets the transformation definition of the event interpreter.

5.5 Transformation definitions are models too

Transformation definitions are sometimes implied to be models, but as evident from tool reviews [91], mostly are programs. Such definitions lack formal and abstract properties of models described in section 2.1. The following is a list of some of the benefits that can be achieved if transformation definitions are abstract, formal and processable models:

- Transformations definitions are complex because they involve knowledge of multiple domains and techniques. In case of an atomic transformation between two domains, expertise is required for meta-modeling and transformation techniques, metalanguage, transformation definition language and source and target domains. The abstraction property of models can reduce this complexity.

- Transformations can be used to extend tools with new functionality (see section 7.12.1). Modeling languages can reduce complexity of transformation definitions and make such extensions feasible.

- Abstract transformation definition languages are easier to metamodel than detailed programming languages. Transformation definitions that conform to such metamodels can be ported among different technological spaces (see section 5.3).

- If transformation definitions are modelled in a well-defined and formal language, they themselves can be subject to transformation. A possible application of this
scenario is transforming interpreter model specified by a transformation definition into a more efficient executable form (refer to section 5.4.2).

• Finally, formal definitions may be subject to formal methods of analysis.

These desired properties of transformation definitions constitute an additional and optional requirement to the cMoS element Tool.

5.6 Summary

This chapter described the methodology to solve the research problem: it introduced the cMoS framework for a family of M&S applications that are characterised by model sharing and online simulations. Next, a meta-architecture for cMoS instances was described. This meta-architecture formulates activities and variation points, which are essential to application domain customization and AcMoS operation. Application of model-driven engineering (MDE) to this meta-architecture enables efficient and programming-free adaptation of cMoS systems to new Web-Based M&S applications. To this end, two model-driven engineering processes were proposed: for the general development process, a generative approach based on the MDA framework but using DSM technologies instead of UML profiles is used. The approach allows to raise abstraction at which development happens to a higher level than it is typical for MDA today. In the alternative process, a system and its behaviour are modelled in an executable language with business concepts from the problem domain. Hereby, system development and modification occurs at the highest possible level of abstraction, without resorting to programming.
CHAPTER 5: TRANSFORMATION DETERMINANTS AND MODELS
Chapter 6
Development Infrastructure

One major topic of Chapter 5 is devoted to definition of model-driven development processes. This chapter demonstrates how MDE assets are created to support these processes. For this demonstration, arbitrary software implementation technologies and MDE technologies are selected in section 6.1. The development infrastructure consists of a number of model-driven development methods. Each method provides a modeling language and one or more transformations that automate development tasks. In sections 6.2 and 6.3, these development methods are defined for each model-driven development process.

6.1 Selected technologies

This section selects a number of implementation technologies. These are not prescribed, but incidentally chosen to illustrate the support for the demonstrated processes.

6.1.1 General programming

Python (www.python.org) is a modern interpreted, high-level language that features one of the most powerful and easy to use implementations of the object-oriented paradigm coupled with clear and readable syntax. Compared to other high-level languages, Python programs are faster to develop, significantly shorter and easier to maintain. Python is extremely portable, neutral to standards and compatible with other languages. All these qualities combined with its open-source nature, have made Python extremely popular [92].

6.1.2 Communication infrastructure

The infrastructure is provided by XML-RPC, a Remote Procedure Call protocol that works over the Internet. Whenever applicable, established client- and server-side frameworks, python libraries and code for XML-RPC are reused and adapted from [35].
6.1.3 Web development

Custom web-based parts of AcMoS are implemented as Zope applications. Zope (www.zope.org) is a web application server that is typically used as intranet and extranet server, document publishing system, portal server and platform for collaboration [16]. Zope features a strong *through-the-web* interaction model, allowing users to update their Zope-based web sites from anywhere in the world. Zope’s extensive list of predefined content types can be easily extended through both custom-made and numerous off-the-shelf products [49, Chapter 12]. Zope carefully follows standards for usability and accessibility. Zope is technology-neutral, can interoperate with most relational database systems and runs on a vast array of platforms. Zope natively supports XML-RPC, therefore all Zope objects can communicate via XML-RPC.

6.1.4 MDE technologies

Historically, AToM³ [22] was used in this project to fulfill the roles of (i) the Tool concept in the cMoS framework and (ii) the MDE tool in the development methods. In both roles, AToM³ was used for the following functionality: modeling, meta-modeling and model transformation. This choice of MDE technology limits the DSLs developed in this research to visual languages.

6.1.4.1 Modeling

In AToM³, modeling is performed by means of a visual editor: one selects a modeling concept from a possibly multiple language TOOLBAR, places it on the canvas and modifies its properties with the Edit tool. Further on, such created instances of modeling concepts can be coupled using the Connect tool. These tools, together with other generic modeling tools form the general modeling toolbar. Models, metamodels and source and target models in transformation rules (see below) are handled in the same editor.

6.1.4.2 Meta-modeling

AToM³’s metalanguage is the Entity Relationship (ER) formalism extended with constraints and appearance properties. The ER formalism [11] is used to model reality in terms of entities and relationships among them. Concept ENTITY denotes anything that can be distinctly identified (e.g. a model, a simulator). Entities are described by their attributes. Concept RELATIONSHIP is an association between entities, e.g. the phrase “a model conforms to a metamodel” states an association between a model and a metamodel.

Figure 6.1 (a) illustrates the abstract syntax of AToM³’s metalanguage. Each construct, whether an entity or relationship, is defined as follows:

\[
\text{Construct} = (\text{name, attributes, cardinality, constraints, appearance})
\]

All constructs are uniquely identified by property *name*. Property *constraints* is a set of rules that controls how a construct can be connected to another construct to form a meaningful composition. In the used version of AToM³, the constraint language is
Python. Possible number of incoming and outgoing connections of a construct is defined in its cardinality rules. Both constraints and cardinality define the static semantics of the language (see section 3.2.2). Property attributes defines a set of structural and behavioural properties of the construct and supports semantics of the construct. Finally, visual presentation or concrete syntax of a construct is defined in the appearance property.

Figure 6.1 (b) summarises the concrete syntax of the metalanguage. Note that cardinality and constraints are not displayed in the concrete syntax of the metalanguage. In order to enhance expressiveness of illustrations, models presented in this thesis can be decorated with labels providing such hidden and other additional information.

Along with the properties defined for each construct, the modeller can define additional global properties in the metamodel itself. These properties can be used to document and globally constrain conforming models.

Given a metamodel, AToM$^3$ can generate a meta-specification, which, when loaded into the meta-level of AToM$^3$, turns the latter into a modeling environment for the designed language. A part of this meta-specification is a specification of User Interface. This specification is a model in its own right and belongs to the so-called "Buttons" formalism. By default, this specification creates a button for every construct of the language. At any time, the modeller is free to edit this specification to e.g. customise the default interface or specify additional buttons that can launch transformations frequently used with the given metamodel.

AToM$^3$ provides a very strong support for modeling languages as systems. Using the quality criteria described on page 27, produced metamodels can reach level four out of possible five levels. This quality can be further increased by complementing metamodels with semantics specified in transformation definitions. The meta-modeling capabilities of AToM$^3$ match well the tailorability requirement on page 69.

6.1.4.3 Model transformation\(^1\)

One approach to manipulate graphical structures, such as models in AToM$^3$, is graph transformation. Graph transformation extends the idea of term rewriting to arbitrary graphs. The theory behind graph transformation has been thoroughly studied (see for example [82]). The central notion in graph transformation is that of a graph grammar [23], which is used as AToM$^3$'s transformation definition language. A graph grammar is

---

\(^1\)Material in this section is adapted from [55] with permission of all authors.
Figure 6.2: Conceptual model of AToM³'s transformation definition rule

a collection of productions or rules each specifying how a (sub)graph of a so-called host graph can be replaced by another (sub)graph. Some graph grammars are enriched with additional conditions and actions per rule. These can be used to model side-effects, such as code generation.

Informally, the operational semantics of graph grammars, as implemented in the transformation tool of AToM³, is as follows: We start from a host graph and a graph grammar. A direct derivation is the result of matching some subgraph of the host graph to the left-hand side of some rule in the grammar, checking if the additional condition is true, and if so, replacing that subgraph by the corresponding right-hand side of the rule, subsequently performing any additional actions associated with the rule. Some graph rewriting systems associate priorities to the rules, so that if more than one rule matches the host graph, the priorities act as tie-breakers. An execution or trace is a sequence of direct derivations. This informal definition is most closely related to the so-called single-pushout approach (SPO) to graph transformation [82, 23].

Figure 6.2 illustrates the conceptual model of AToM³'s graph grammar rule. Each rule contains two (sub)graphs: the source and its replacement. These graphs are respectively called the left-hand side (LHS) and the right-hand side (RHS) of the rule. In addition, a condition and an action are associated with each rule. Such pieces of behaviour are programmed in Python. Finally, rules are assigned priorities.

AToM³ features a very promising approach to practical realization of transformations (refer to the overview of transformation approaches on page 31). The tool provides the transformation language support, as well as more traditional direct model manipulation through application programming interface (API). Moreover, the transformation language (GG) allows to specify transformations in abstract and declarative way using modeling concepts directly from the source and target domains. In summary, the transformation capabilities of AToM³ are very adequate for the transformation challenges of this project (see section 5.5).

6.2 Software development process

Turning from technologies to development processes, Figure 6.3 illustrates how the selected technologies are related to MDE artefacts in the general software development process. Languages are shown as boxes. Directed edges between boxes order languages by their level of abstraction. In development, artefacts of more abstract types are nor-
nally converted into artefacts of more concrete types by means of transformations. Note that transformations are not shown in the figure. The languages and their roles are described below in a left to right sequence.

The MDA choice for a PIM language is UML, a complex general language with multiple views on the software system. Experience shows however that the majority of UML modeling is done with few widely used views [32]. UML Class diagrams is one such view. Moreover, a recent review of MDA tools [91] indicated that class diagrams is a common choice for platform independent modeling. In line with these findings, the PIM language of the given process was based on the class diagrams. An AToM³ implementation of class diagrams is described section 6.2.1. Another PIM language, Entity Relationship (ER), is AToM³'s metalanguage and is used for development of AcMoS applications at the meta layer, as described on page 79. ZProduct is a domain-specific language for modeling web applications for Zope. Custom web-based parts of AcMoS are to be implemented as Zope applications. ZProduct and a suit of related transformations form the so-called ZCase framework, which is described in section 6.2.3. Finally, Python is this project's choice for a general programming language (refer to section 6.1.1). Python Class Diagrams, a PSM representation of Python code, is described in section 6.2.2.

6.2.1 Simplified Class Diagrams (SCD)

Simplified Class Diagrams (SCD) is a possible and simplified realization of UML class diagrams. The constructs of SCD are based on the metamodel of UML 1.4 [37]. Since the metalanguage of AToM³ does not feature the inheritance mechanism used in structuring the UML metamodel, full descriptor of each SCD construct was obtained by combining the segment descriptors of the corresponding UML element and that element’s ancestors. The concrete syntax of the language is defined after the standard UML notation.

The metamodel of SCD is illustrated in Figure 6.4. For reasons of brevity, full descriptors and well-formedness rules are omitted. The SCD constructs are:

SCDAssociation is a semantic link between instances of associated classifiers. Association consists of at least two association ends, each specifying a connected classifier, and properties defining valid participation of those classifiers in the relationship.

![Diagram of selected languages and their relation to development artefacts](image-url)
SCDClass defines structure and behaviour of a set of objects by declaring attributes, operations and methods. All objects instantiated from a class, share the structure (attributes and their initial values) and behaviour as defined by the class.

SCDGeneralization is a taxonomic relationship indicating that one element (child) is a specialization of a more general element (parent). Ancestors are parents of an element together with their ancestors without duplicates. Generalization is used by the inheritance mechanism to incrementally build the full descriptor of an object from the segment descriptors of the element itself and those of the element’s ancestors.

SCDObject represents an instance of a class. Such an instance has identity and attribute values.

We stress that SCD is fewer in quantity and simpler in features than the original UML Class view and this fact is reflected in the name of the language. Another important difference with respect to the UML is that SCD is not MOF-based, hence SCD is not directly compliant with the OMG’s approach to solving the tool integration problem.

6.2.2 Python SCD

Python SCD provides sufficient means to specify a Python module. This PSM language is defined by reusing the SCD metamodel and extending the stereotype property of element SCDCLASS with Python related values: "module", "function" and "variable".

6.2.2.1 Python code generation

Given a Python SCD diagram, transformation SCD2PYTHON generates a Python module. The transformation definition is straightforward: visual Python syntax is converted into a textual Python syntax.
6.2.3 ZCase - Zope Product framework

ZCase is a custom framework for creating web applications for Zope. There are a few ways to develop such applications. The easiest and fastest way is to build an application in instance space through the web [49, Chapter 12]. The other way is to create the so-called product [63, page 35]. In comparison to instance-space applications, products are more powerful and flexible, easier to maintain, extend and distribute, but are typically more complex to build. ZCase separates the core functionality of a web application from all aspects of Zope product development. These tedious and error-prone aspects are not programmed but automatically generated from abstract models. ZCase drastically reduces complexity and increases efficiency of product development.

The framework consists of a domain specific language ZPRODUCT to create complete product models and a suit of validation and code generation transformations.

6.2.3.1 ZProduct modeling language

Figure 6.5 illustrates the metamodel of ZProduct. The following is an informal description of each Product development concept defined in the metamodel:

- **ProductClass** is the main class of the product that is being developed.

- **BaseClass** is a complete reference to a Zope specific class that fulfils one or more contracts (or requirements) imposed on web applications [63, page 35]. A product class fulfils a contract by subclassing (see acquisition on the current page) from a proper base class. Note that in ZProduct, this construct can be repurposed to inherit non-contract features from user defined classes.

- **Acquisition** relationship provides means to indicate which base classes and in which order, the product class inherits from. In case of multiple inheritance, the source class takes precedence over the target.

- **MetaType** indicates a Zope product. It can be the given or any other product.

- **Aggregation** relationship indicates meta types whose instances are allowed to be contained by the given product.

- **View** defines a screen in the Zope Management Interface (ZMI) of the product. Views, also known as management tabs, allow managers to control different aspects of the product instance through the web.

- **nextView** relationship provides means to connect views into a sequence. The target view follows the source view. The order of connected View elements defines the order of the management tabs in ZMI of the product.

- **Property** declares an instance attribute of the product class as property manageable by users through the web.

- **nextProperty** relationship provides means to compose a list of properties. The order of connecting the Property elements defines the order of properties in the Properties
Figure 6.5: The ZProduct metamodel
management screen of the final product. The target property follows the source property.

**File** allows to specify an external artefact in the file distribution form of the product. Files assume different roles. These roles are defined by the relationships File participates in.

**help** relationship binds a help topic to a management view. Contents of the help topic is provided by a File instance.

**isSelection** relationship associates a Property instance of selection or multiple selection type with another property or a method. The target entity should provide a list of strings from which the selection(s) can be chosen.

**constructor** relationship specifies a File instance that contains an implementation of the product constructor.

**method** relationship binds either a View action or an operation of the ProductClass with a File containing the implementation of such an action or operation.

**Information** provides mandatory and optional pieces of information to be included in a valid product distribution.

**Layout** specifies names for the conventional sub-directories in the file structure of product distribution.

**Icon** specifies an image that will identify instances of the product in Zope management interface.

Under Zope there are two fundamental categories of objects: items and containers. Figure 6.6 shows possible models of item and container products. For the sake of simplicity, the presented models specify minimum products. These two models form the basis for all ZProduct models presented in Chapter 7.

### 6.2.3.2 Product validation

Transformation **VALIDATEZPRODUCT** evaluates if a given ZProduct model defines a valid Zope product. The validation rules are based on [63, Chapter 3, page 33] and numerous online resources in the Zope community. The current transformation definition contains about 50 rules, too many to present here.

Each transformation rule tests for an individual invalid or questionable pattern. If a rule is executed then its action marks the matched element or a combination of matched elements with ERROR or NOTE. The final rule marks the model as valid if there are no errors. Figure 6.7 illustrates the concrete syntax of the validation constructs.
Figure 6.6: Fundamental Zope products
6.2.3.3 Product PSM generation

Given a valid ZProduct instance, transformation ZPRODUCTToSCD generates a complete class diagram of the Python module implementing the source model. The transformation definition is based on [62, 63], as well as many online resources in the Zope community. A brief summary of the transformation definition follows.

In four steps, a Python SCD diagram is added to the ZProduct model: Firstly, the source model is traversed and annotated (on annotations refer to section 7.12.2). Secondly, external modules are specified to be imported according to dependencies present in the ZProduct model. Thirdly, an SCD class is created to represent the product class. Annotated ZProduct properties, views and methods are used to model attributes and generate code for methods of the class. Finally, the class’s ancestors are created and additional modules are imported if needed.

The resulting SCD diagram is an intermediate PSM representation of a Python module, which can be transformed into code according to transformation definition SCD2PYTHON (refer to section 6.2.2.1).

6.2.3.4 Product distribution generation

Transformation ZPRODUCT2FILES generates a product’s directory structure and files from a valid source ZProduct model. The rules of this transformation are straightforward: the LHS graph contains a LAYOUT element and various file related patterns involving elements INFORMATION and FILE. The condition ensures that the rule is not applied to already processed subgraphs. Finally, the action of the rule creates a file at the location specified by the LAYOUT element. Empty files and directories are not created.

6.2.3.5 AToM³ metatype to Zope product

Transformation ERToZPRODUCT is applied to an instance of either ER entity or ER relationship, or a model conforming to the ER metamodel. In three steps, a ZProduct equivalent of the source node is created: Firstly, a proper ZProduct model is loaded: the generic multigraph (section 7.4.1) for a model or the generic node (section 7.4.2) for an entity or a relationship. Secondly, attributes of the source are copied into properties of the target ZProduct model. Thirdly, the target product is updated with the name of the source instance. The first step is performed manually. The result of the transformation is a model of a Zope product that represents the source, thus enabling instances of the source type to be stored in Zope.

6.3 DSL interpretation

This approach does not require a software development process and therefore does not need any special infrastructure for software technologies. The modeling, meta-modeling
and transformation technologies provided by the selected MDE tool are sufficient for the approach.

6.4 Summary

This chapter developed MDE assets in order to support the development processes defined in Chapter 5. These assets enable automated generation of executable code from abstract models of concepts that constitute the eMoS framework (see section 5.2). Moreover, ZCase and transformation ERToZPRODUCT realize code generation of model repository from variation point METAMODEL (see section 5.3) at the time of application domain customization. Thereby the infrastructure developed in this chapter is a realization of our approach to addressing both the tailorability and reduced programming requirements (see page 69).

This infrastructure cannot be realized independently from concrete technologies to be used at each step of the development processes. Therefore, a number of implementation and model-driven engineering technologies were selected and described. For each modeling and implementation technology a development method was realized using the selected MDE technologies. The metamodels and transformation definitions of the realized methods and relations among them are illustrated in Figure 6.8. In the figure, boxes are metamodels and diamond shapes are transformation definitions. Note that languages Python and Distribution are not metamodel based. This is indicated by displaying their names in italic face. Dashed arrow indicates that the target entity is a part of the source of the arrow.
Chapter 7

cMoS Instance

AcMoS (a cMoS) is an application domain independent instance of the cMoS framework. Chapter 5 introduced the cMoS framework, the AcMoS meta-architecture and two MDE processes. For these processes, Chapter 6 developed the model-driven development infrastructure. This chapter employs this infrastructure to model AcMoS software system. Section 7.1 provides the view on the AcMoS that focuses on decomposition of the system into independent parts, called components. Parts of the AcMoS that can be provided by existing off-the-shelf products are described in section 7.2. Remaining sections present models of custom software parts that have to be developed to complete the AcMoS. All such parts, with the exception of the cMoS element Controller, are engineered using the model-based development process described in section 5.4.1. Due to the behavioral complexity of the Controller, this part is realized using the DSL Interpretation approach (see section 5.4.2).

7.1 AcMoS components

AcMoS is a software incarnation of the cMoS framework that contains but does not define parts specific to application domains. In Figure 5.6, such parts are recognized by double borders. To simplify the system and increase opportunity for building the AcMoS by matching and combining existing software pieces from various manufacturers, the system is divided into independent components as illustrated in Figure 7.1. The components and their primary interrelationships are described below in the bottom-up sequence.

FILE REPOSITORY stores files of context-in models. The repository is accessible locally and remotely. If stored contents is periodically purged, then the repository is temporary.

SOLVER wraps a (simulation) application to turn the latter into a computational resource. Each solver has access to a file repository to read files of context-in models for an experiment execution and write data generated by the experiment into files. Translation of the data produced by the simulation application into the form conforming to the Data schema is responsibility of the solver. If such translation is not possible, then the written data is not associated with the Data schema. When scenario execution is completed,
Figure 7.1: A breakdown of AcMeS into components.
CHAPTER 7. CMOS INSTANCE

solver calls back the requester of the experiment to notify and pass information about the output files.

**CONTROLLER** manages computational resources in a manner transparent to involved components. It delays requests to solvers to execute scenarios until the addressed solvers are free. When controller is notified that a scenario execution is completed by a solver, it delegates the notification and output information to the originator of the experiment and frees the next delayed execution request if any.

**GROUPWARE** provides a collaborative web platform for components realising the AcMoS application. Through this component, 1) content is managed (create, change, publish, share, find, delete object, etc.) and 2) teams and communities are supported. Note that solvers and context-in models are external to this component and to achieve location transparency, they are represented in GROUPWARE by proxy components SOLVER’ and CONTEXT-IN MODEL’.

**EXPERIMENT** is automatically created when a user executes a context-in model’. It searches Groupware for instances of components SOLVER’ and SCENARIO and presents found results to the user to define an experiment. When experiment definition is complete, EXPERIMENT allows the user to execute the experiment, monitor execution status, view output, register output as persistent context-in model or delete it.

**CONTEXT-OUT MODELS** are stored in the groupware component, which implies that Model repository is a part of the groupware component. In general, this repository can be factored out into a separate component. The Model repository provides functionality to export its contents in one or more model interchange formats that can be loaded by TOOL.

**TOOL** can transform conceptual models into a context-out format which GROUPWARE can import. This allows high level model specifications to be shared for reuse in modeling. To share a model for simulation, Tool transforms the model into the context-in format of the target solver (code generation). Such generated file(s) are created by Tool in a persistent File repository. An instance of CONTEXT-IN MODEL’ is manually created and associated with the generated file(s) through GROUPWARE. Import of any models into the tool is achieved by loading models from the File repository. Such models must be in the tool’s context-in format.

**DATA MODELS** are realised as extension of the component for context-in models. The files associated with the data model, must contain data conforming to the Data segment metamodel (see section 4.1). This component links data segments to variables of the model from which the data was generated. In this AcMoS, such linking is established through name equivalence of the signal and the variable and is realised as a hyperlink from the former to the latter.

**CLIENT** provides a member of a team or community with a means to interact with the groupware component. Through Client, 1) content is managed and 2) experiments are executed.
7.2 Selected off-the-shelf parts

Parts with generic functionality are often available as off-the-shelf products. Development of a CMOS instance can benefit from reusing available parts that can fulfil the roles of AcMoS components. Such parts are described below.

7.2.1 Groupware

The collaborative needs as defined in Figure 5.4 on page 71 are provided by Zope (also see section 6.1.3), a web application server that is typically used as intranet and extranet server, document publishing system, portal server and platform for collaboration [16]. Zope’s security architecture and membership support [49, Chapter 6] provide a system to implement community and team collaboration models. Zope Object Database (ZODB) [76] is a part of the web application server. Taking advantage of the object database features [49, Chapter 9] requires few, if any, changes in logic of Zope applications, thus facilitating the library model of collaboration. Moreover, there is a number of Zope applications, for example Plone [64], that implement contents management frameworks and provide advanced collaborative functionalities.

7.2.2 Client

Zope-based groupware is fully functional through any web browser. However, a standards-compliant browser can provide users with additional benefits and ease of use. These benefits increase with higher levels of the so-called Plone support by browsers [58]. These levels are also valid for Zope. At the time of research, the highest level defined by Plone developers is 4, and browsers capable of this level are Internet Explorer 5.5 and 6.0, Mozilla-based browsers: 1 and up, Opera 8 and up.

7.2.3 Model repository

This component is a part of Groupware: context-out models are stored in Zope’s own transactional object database (ZODB). The transactional model of ZODB ensures strong integrity between connected data sources and clients.

7.2.4 File repository

This type of repository stores model files and is realised as one or more FTP servers. The demonstrated AcMoS uses a free FTP server software.

7.2.5 Simulator

Off-the-shelf simulation tools, applications and packages are used as basis for solvers (see for example [52, 56]).
CHAPTER 7. CMOS INSTANCE

7.2.6 Tool
ATOm\(^3\) is used to model, specify new languages and transform models in the context of AcMoS applications. ATOm\(^3\) also fulfilled the role of MDE tool in the development methods (refer to 6.1.4).

Off-the-shelf, ATOm\(^3\) is nearly, but not completely capable of realizing component Tool as illustrated in Figure 7.1. Namely:

- The tool does not provide direct support for export of ATOm\(^3\) models into standard model interchange formats or into the context-out format of the model repository.
- The tool also functions as code generator, which is not a typical application for graph transformation, ATOm\(^3\)’s approach to manipulate graphical models.

Section 7.12 describes how the export functionality is bootstrapped with meta-modeling and model transformation functions and presents guidelines used in this research for code generation with graph grammars.

7.3 Model’, context-in

Context-in model’ is a Zope application that implements a proxy for a context-in model. A context-in model can manifest itself as single or multiple files. Accordingly, there are atomic and compound kinds of proxies.

7.3.1 Atomic
Context-in model is a searchable persistent item-like Zope product illustrated in Figure 7.2. The product’s MRD and MRA attributes are declared as user-manageable web properties. The SOURCE property associates the context-in model with the corresponding simulation model file.

7.3.2 Compound
This kind extends the atomic context-in model with a single feature: object containment. The design of the product assumes that among multiple files of an executable model, there is one master file. This file is referenced by the instance of the compound model product itself. The contents of the instance holds one or more atomic context-in models’, each referencing the remaining files.

The model of this product is not shown as it is the same as that of the atomic product with the addition of standard base class OBJECTMANAGER, which provides containment-related features. As a final touch, the management interface of the product is updated to use methods inherited from this base class.
Figure 7.2: Model of the Atomic context-in model' product
7.4 Model, context-out

Context-out model is an instance of a Zope data type in ZODB. This data type should accommodate instances of any metamodel specified in Tool. At minimum, abstract syntax information must be preserved. In this AcMoS, models are graphical structures and therefore design of the context-out model is based on the multigraph concept as defined by Harary in [41]. The sequel briefly describes a family of products MULTIGRAPH, NODE and ARC that form the basis, from which numerous products of language-specific context-out models are made.

7.4.1 Multigraph

Informally, a multigraph is a directed graph $G$, which consists of a set $V$ of objects called nodes joined by links called arcs (set $A$):

$$G = (V, A)$$

$V$ is independent of $A$. $G$ is an unlabelled graph. Loops and multiple edges, edgeless graphs (no edges) and empty graphs (no nodes, no edges) are possible in this product.

The model of the product is illustrated in Figure 7.3. It is a searchable persistent folderish Zope product. Within the Zope context, a node can be an instance of any product, including the MULTIGRAPH product itself. On the other hand, an arc must be an instance of the ARC product (that can be additionally installed to allow graphs other than empty and edgeless ones). Note that this product has view Export through which users can export a multigraph instance.

7.4.2 Node

The Node product is not meaningful by itself; instead, it is used as basis for nodes specialized for a given metamodel. On specialization of nodes, refer to section 6.2.3.5.

The model of the NODE product is illustrated in Figure 7.4. It is a non-searchable persistent item-like Zope product. It does not have any user-manageable properties, as these will be specified in specializations of the product.

7.4.3 Arc

Arc is a directed edge. An edge is a pair of distinct nodes in a graph. If an arc is directed from $x$ to $y$, then $y$ is called the head and $x$ is called the tail of the arc:

$$arc = (\text{tail}, \text{head})$$

The model of the ARC product is illustrated in Figure 7.5. It is a non-searchable persistent item-like Zope product. User-manageable properties tail and head hold references to nodes. Halfedges (edges with only one end) and loose edges (no ends) are possible in this product.
Figure 7.3: Model of the Multigraph product
7.5 Data model

Figure 7.4: Model of the Node product
Figure 7.5: Model of the Arc product
7.4.4 Export

The export function returns a context-out model in a model interchange format. Currently, the only supported format is the context-in format of AToM³. AToM³ models are graphical structures based on the graph concept, which made implementation of the export function straightforward.

7.5 Data model

Data model is an extension of product Atomic Context-in Model' (see section 7.3). The extension is two-fold: First, the product should parse a file with data conforming to the Data Segment metamodel and present the data as segment in a tabular form. Second, if the corresponding model contains a variable with the name equal to that of the signal, the latter is presented as hyperlink to the variable.

7.6 Solver

Solver is a simulator integrated into AcMoS (see section 5.2.9). The integration is implemented using the adapter design pattern [33]. An adapter or wrapper is based on server-side framework SIMPLEXMLRPCSERVER [35, page 356], which adds server capabilities to the simulator. The solver is also responsible for translation of the data produced by the simulator into the form conforming to the Data schema. If such translation is possible, then it is indicated so by the scenarios associated with the solver (see the Format property of scenario in section 7.8). The public interface and behaviour of the wrapper class is illustrated in Figure 7.8. The adapter has to be developed for each simulator.

7.7 Solver'

This is a Zope application that implements a proxy of a solver. Solver' is a searchable persistent folderish Zope product that can contain instances of the Scenario product (see section 7.8). This product extends the generic container product (see section 6.2.3.1). Solver has a very simple custom behaviour (delegation of calls to Solver); its structure is illustrated in Figure 7.6.

7.8 Scenario

Scenario is a very simple, behaviourless concept (see section 5.2.3). It is only meaningful in the context of the owning solver' instance. MRD is used to describe scenario instances in a uniform way. No validation of input parameters is performed.

Scenario can be created by repurposing DTMLDOCUMENT, a standard Zope data type [49, page 419] and extending it with the scenario-related properties:

\[ \text{scenario} = \text{DTMLDocument} \cup (\text{Title, Description, Format, Body}) \]
Figure 7.6: Model of the Solver' product
where

- *DTMLDocument* contains the scenario parameters input form
- *Title* is a metadata term defined by Dublin Core Metadata Initiative (DCMI) [5]
- *Description* is a DCMI metadata term
- *Format* is a DCMI metadata term describing the scenario output
- *Body* is a parametrized template of interaction instructions

The easiest way to implement the scenario component is to implement it as *instance space* application [49, page 233] rather than a product as in section 6.2.3. Such implementation is not covered here as describing it and the instance space development much overweight the simplicity of the component itself.

### 7.9 Experiment data

This is an agreed data structure that is used by the experiment and solver components to exchange information related to experiment execution. Experiment data is a structure:

\[(input, output, scenario, params, target, source)\]

where

- *input* is a sequence of input MRAs
- *output* is a sequence of output MRAs
- *scenario* is Body of scenario
- *params* is a set of parameters for the scenario body
- *target* is MRA of a solver instance
- *source* is MRA of an experiment (see on the current page)

Experiment data is implemented as structure of the Python dictionary type.

### 7.10 Experiment

Experiment is a searchable persistent folderish Zope product that can contain context-in or data models' created as output of an experiment. The model of the Experiment product extends the generic container product (see section 6.2.3.1) and is illustrated in figure 7.7. Python class PSEXPERIMENT implements the ProductClass and provides the following primary responsibilities:

- Collect information about the input model, solver, scenario and scenario parameters and create an experiment object.
Figure 7.7: Model of the Experiment product
- Initiate a real experiment by passing experiment data to the target solver.
- Maintain and display status of experiment ("running", "finished").
- Provide the experimenter with access to experiment outputs.

This behaviour is quite complex and is illustrated with a sequence diagram in Figure 7.8.

7.11 Controller

In order to easily change or completely redesign Controller, this component is developed following the DSL Interpretation approach (refer to section 5.4.2). The domain model of the controller is described in section 7.11.1. Section 7.11.2 describes how Controller’s operational semantics is modelled using the graph grammar formalism: an example of a graph grammar rule is given and event execution is explained. How the controller model is turned into an operating server and how this server is discovered and accessed is explained in sections 7.11.3 and 7.11.4, respectively.

7.11.1 Domain model

Design of the Controller was historically based on the process oriented view (refer to section 4.4) and πDemos [4], a small process-oriented discrete event simulation language that provides simple and consistent formulation of the synchronization mechanism for process-oriented systems.

Controller is a structure:

$$Controller = (State, Clock, Activities)$$

where

- $State$ represents all processes and resources in AcMoS at a given time
- $Clock$ provides time services
- $Activities$ change the state of the controller and system

Clock is an independent system that provides local time according to logical or physical base [100, page 34]. Behaviour of activities is defined in section 7.11.2.

$$State = (EL, R, D)$$

where

- $EL$ is a sequence of event notices $en$ ordered by their event times: $sequence : \{en\}$
- $R$ is a set of resources: $set\{Resource\}$
CHAPTER 7. CMOS INSTANCE

112

Figure 7.8: Typical interactions during an experiment
CHAPTER 7. CMOS INSTANCE

\[ D \] is a set of processes: \( \text{set}\{\text{Process}\} \)

*Event notice* represents a process in \( D \) and contains time of the process’s next event \( \text{evt} \). Time value for \( \text{evt} \) is provided by clock. The notice located at the head of \( EL \) is called *current*. If two or more notices have the same event time then they are placed on \( EL \) in the order of their arrival.

\[
\text{Resource} = (id, C, Q, \text{busy})
\]

where

- \( id \) is a resource identifier unique in \( R: \text{Name} \)
- \( C \) defines the capacity value of a resource: \( \mathbb{N} \)
- \( Q \) is a waiting queue for event notices: \( \text{sequence}\{\text{en}\} \)
- \( \text{busy} \) indicates if resource is busy: \( \{\text{True}, \text{False}\} \)

Capacity \( C \) indicates the maximum number of processes that a resource can serve at a given time. The capacity value decreases every time a process is granted access to the resource and increases again as the process releases the resource. When \( C \) becomes 0, the resource stops granting requests to processes and becomes busy. If a process attempts to acquire a busy resource, the process is *blocked* by moving its event notice from \( EL \) to the end of the resource’s \( Q \).

\[
\text{Process} = (id, \text{body}, r)
\]

where

- \( id \) is a process identifier unique in \( D: \text{Name} \)
- \( \text{body} \) is sequence of events: \( \text{sequence}\{\text{event}\} \)
- \( r \) contains \( ids \) of acquired resources: \( \text{sequence}\{\text{Name}\} \)

The next event in the \( \text{body} \) of the current process is called *imminent*. All possible events correspond to known activities. Event execution is described in section 7.11.2.

The presented domain model is realised as Process Interaction (PI) metamodel described in section 4.4.1.

### 7.11.2 Operational semantics

Controller’s behaviour is based on the operational semantics of \( \pi \)-Demos [4]. Each activity of the controller corresponds to a command of \( \pi \)-Demos; event execution also follows the approach used in \( \pi \)-Demos. An implementation of the \( \pi \)-Demos operational semantics was described in section 4.4.2.
7.11.3 Server

Server is a Windows application consisting of two threads: the main AToM³ thread and an instance of the SimpleXMLRPCServer framework [35, page 356], which (the instance) adds server capabilities to AToM³. AToM³ must have a Process Interaction (PI) model pre-loaded. This model is a specification of AcMoS resources in the PI formalism (refer to section 4.4). The secondary thread listens for requests to resources and whenever one arrives, creates an instance of meta-type PIECE in the resource model by calling AToM³ API. Each such piece has a body with the following sequence of PI events:

\[
\text{body} = (\text{getR}(R), \text{useR}(R), \text{putR}(R))
\]

where

\( R \)

is name of the requested resource

The main thread processes requests by continuously rewriting the model according to the graph grammar rules of the controller’s behaviour model. In terms of the DSL Interpretation approach (see Figure 5.8), external system is an instance of the SimpleXMLRPCServer framework, transformation is the main AToM³ thread and system model is a model in the PI formalism.

An example of the AcMoS resource model is shown in Figure 7.9. Pieces that are created by the secondary thread are labeled with keyword "new". Note that these pieces are not participating in any relationships with the other elements of the model. Presence of such unconnected pieces causes AToM³ to execute rule IMPORTPROCESS (see Table 4.2) for each new piece. This rule specifies that an unconnected piece should be appended to \( EL \), an action that will be consequently carried out by rule APPENDELEMENT or
7.11.4 Controller

A proxy of Controller is realized by an instance of the SOLVER product, whose properties describe location and access to the controller. This instance has reserved id Controller so that it can be easily referred to in the groupware system.

7.12 Tool

This section describes how the missing export function (see section 7.2.6) is added to AToM³ and presents guidelines used in this research for code generation with the AToM³ transformation technology.

7.12.1 Export functionality

AcMoS requires that models are exchanged between modeling tools and model repositories. To import a context-out model from a model repository, one simply has to export the model in the AToM³ format, download it and consequently load it in AToM³. Unfortunately, there is no simple workaround to exporting AToM³ models into a model repository. In the following this missing functionality is bootstrapped by means of metamodeling, model transformation and multi-formalism.

7.12.1.1 Locations metamodel

The purpose of custom language Locations is to provide the end-user with a (graphical) interface to specify location, access and authentication details of remote systems.

The Locations metamodel contains one entity Repository, whose structure is illustrated in Figure 7.10. Property type = Enum(Zope, FTP) specifies the type of the repository, based on which the proper export routine can be selected. Currently, the communication protocol is implied by the repository type. All other properties are self-explanatory. A valid model in this formalism contains only one location definition. Figure 7.11 illustrates a Locations model that describes a Zope repository.

7.12.1.2 Export transformation

Every rule of this transformation implements an independent and complete export routine. The following holds for all rules:

The LHS graph of a rule contains a pattern instance of Repository as illustrated in Figure 7.12. This pattern specifies the kind of repository and communication protocol
that the rule is designed for. The \textit{condition} prevents that the rule is executed multiple times in the course of the transformation. Finally, the \textit{action} of the rule provides i) a client for the communication protocol [35] and ii) an export routine for the target system.

In the case of a Zope repository and the XML-RPC communication, the final action of a proper rule will execute as follows: an XML-RPC client calls the repository to create a context-out model (see section 7.4). Then the rule iterates twice over all AToM$^3$ entities or relationships of any formalism other than Locations. In the first run: for each iteration item, the client calls the created model to create a node (see section 7.4.2) of the type corresponding to that of the item and to update the node's properties with the semantic information of the item. The second run saves connection information of each item as arc instance (see section 7.4.3) within the created context-out model.

\subsection{Multi-formalism modeling}

This feature of AToM$^3$ allows combining models of different formalisms into one multi-formalism model. To export any model (e.g. the one shown in Figure 7.9), one has to add a model of the LOCATIONS formalism (e.g. from Figure 7.11) to the former model and execute the export transformation on the resulting multi-formalism model.

\subsection{Code generation guidelines}

Graph transformation is the approach employed by AToM$^3$ to manipulate graph structures and transform models. In the context of AToM$^3$, code generation means producing textual structures from graph structures. One way of doing this is via a transformation where the source and the target are the same. Rules of such transformation do not perform any important rewriting, but use the graphical nature of the source language to traverse and annotate the source model with temporary information that is needed for code generation. Code itself is generated by side-effects encoded in actions of rules,
Figure 7.13: An application domain independent AcMoS

which can access the annotations. This approach is primarily used for code generation in domain specific AcMoS applications described in Chapter 8.

A common application of annotations in the above approach is to mark which nodes of the model have been already processed. Often a code generation scheme for a given node may need to apply for an arbitrary number of the node’s neighbours as in the case of parent/child pattern. Such situations cannot be handled by a single rewriting rule, since the LHS of a rule always has a fixed number of nodes. A solution is to have an auxiliary rule that adds an unvisited neighbour to a collecting annotation. Such rule should have a priority higher than that of the actual code generation rule, since code generation should happen only after all the relevant neighbour’s information has been collected. Annotations are usually removed in the final action of the transformation.

7.13 Summary

This chapter demonstrated a model-based development of a domain-independent AcMoS. To enable reuse of third-party parts, the AcMoS was decomposed into independent components. Next, off-the-shelf parts that could realise components with general functionality, were selected for reuse in the AcMoS. Finally, the remaining cMoS-specific AcMoS components were developed using the model-driven processes described in section 5.4: by default, a generative model-driven approach using the MDE assets from chapter 6 was used. Due to the behavioural complexity of the cMoS Controller, this component was realised using the interpretation approach by explicitly modeling controller’s
operational semantics as abstract, formal and executable transformation definition. The
meta-architecture of the developed software is shown in Figure 7.13. In contrast to the
AcMoS meta-architecture illustrated on page 78, all AcMoS parts that are generic or
cannot be automatically generated from other parts, have been realised in this chapter.
Parts outlined with double border and labelled with the question mark are application
domain-specific variation points and are defined at customization times during the oper­
ational phase of the system. Such parts will be illustratively realised by means of models
in chapter 8.
Chapter 8

AcMoS Applications

Chapter 6 demonstrated how an instance of the cMoS framework is developed with the model-driven engineering processes. The resulting software is a domain-independent AcMoS that can be efficiently tailored to an arbitrary number of M&S application domains. This chapter briefly illustrates the key features of the cMoS framework using the given instance as example: tailoring of AcMoS to specific M&S domains is described in section 8.1. Modeling is illustratively exemplified for a traffic system in section 8.2. Collaboration is outlined in section 8.3. In section 8.4, online simulation is demonstrated by executing the above-mentioned traffic model. Finally, section 8.5 illustrates how the AcMoS environment manages access to and use of shared simulators by multiple concurrent users.

8.1 Application domain customization

This section illustrates how the developed domain-independent AcMoS is efficiently tailored to a new application domain. The customization process takes four steps:

1. Define a new application domain.
2. Update Repository to store models of the new domain.
3. Create a wrapper for the simulator.
4. Define an execution scenario for the solver (see section 5.2.3).

Out of the four steps, step 3 involves traditional programming. The efficiency and flexibility benefits of the proposed framework lie in steps 1, 2 and 4 (last being the least contributor to the benefits). Each step of the customization process is demonstrated in the consequent subsections.

8.1.1 Application domain definition

According to the AcMoS meta-architecture (see the entities marked with double border in Figure 5.6), tailoring to a new application domain requires a domain-specific metamodel and a transformation definition for at least one simulator. The latter should specify
8.1.2 Model repository update

The purpose of this step is to enable repository storage of conceptual models of a defined application domain in a tool-independent format and at the same level of detail as in Tool. The AcMoS meta-architecture (see Figure 5.6) defines repository update as transformation from a metamodel (e.g. that of the DTR domain) into a storage structure definition for the model repository. The important property of this transformation is that the abstract syntax of the language has to be preserved in the resulting structure of the model storage.

The model repository update is implemented as a chain of more basic transformation steps as illustrated in Figure 8.1. Each such step is a sequence of one or more transformations from the custom MDE methods described in section 6.2. The update starts with two models loaded into AToM³: any metamodel and a basic context-out model written in the ZProduct DSL (see section 7.4). The latter will be used to define storage structure for a single language concept from the former. Furthermore, this ZProduct model predefines structure, behavior, user interface, security options, etc. common to all repository updates. Figure 8.2a shows a thumbnail view of the initial canvas. The first step in the transformation chain is to update the ZProduct model with information specific to the concept manually selected by a developer. This step is carried out with a sequence of transformations: ERToZProducts and ZProduct2SCD (see section 6.2.3). The result how code executable by a simulator is generated from models conforming to the domain-specific metamodel.

This thesis has numerous examples of adding domain support as specified above. In chapter 4, a number of DSLs in the field of Modeling and Simulation were modeled and implemented using the MDE approach. Specifically, section 4.5 defined the Dynamic Traffic Routing (DTR) domain, which will be used as running example in the rest of this chapter. Moreover, section 6.2 presented a number of metamodels and transformation definitions for software technology domains (for a summary see Figure 6.8). AToM³ itself is distributed with models that provide support for Deterministic Finite Automaton (DFA), Nondeterministic Finite Automaton (NFA), Petri Nets and General Purpose Simulation System (GPSS) to name a few. Even more metamodels and transformations are available in the AToM³ community.

Figure 8.1: Transformation chain of the repository update
Figure 8.2: Artefacts in the transformation chain
(see Figure 8.2b) is a concept-specific intermediate ZProduct model and a Python SCD model. The latter contains concept-specific behaviour derived from the structure of the former model. The second step is to generate code from the Python SCD model and integrate it into the intermediate ZProduct model. This is fulfilled with transformation SCD2Python. Figure 8.2c shows the result, a complete ZProduct model of a language concept. The final step is generation of an executable distribution from the complete ZProduct model. This step is fulfilled with transformation ZProduct2Files. The result is a complete directory structure with nearly 20 files and can be directly installed in a Zope repository.

Steps 1, 2 and 3 of the transformation chain have to be repeated for each element of the metamodel and the metamodel itself. In the DTR case, the whole process takes less than one hour and generates more than 100 artefacts.

### 8.1.3 Simulator wrapper

Wrapper is an adaptor for a specific simulator. A wrapper and a simulator form an AcMoS solver. Wrappers are developed so that resulting solvers can realize the solver concept as described in section 5.2. Out of the four application domain definition steps, wrapper development involves traditional programming and currently does not contribute to the efficiency and flexibility benefits of the proposed framework. For more information on using the wrapper technique to integrate simulators into AcMoS, we refer the reader to [50, 52, 56].

### 8.1.4 Scenario

Simulation with the ABC simulator (see section 4.5.2) does not require any special interactions in addition to those provided by the default simulator wrapper. Therefore the execution scenario is empty in the case of the DTR domain.

### 8.2 Modeling

Modeling is performed off-line in AToM³. For a description of the tool's modeling capabilities, refer to section 6.1.4.1. All models presented in this thesis (some 40+) were created with AToM³. To continue with the demonstration, Figure 8.3 shows a model of a traffic system specified in the modeling language developed for the DTR domain. This model specifies five intersections, of which the central one (labeled 'i4') is a roundabout. Intersections are connected by two roads of opposite directions, thus forming a two way connection. Each road is a single lane, low priority, with speed limit of 50 km/h.

### 8.3 Collaboration

Collaboration functionality is orthogonal to the cMoS framework and concerns model sharing. In the presented AcMoS, the collaboration aspect is completely provided by off-the-shelf component Zope. The following is a brief description of collaboration in the context of Zope.
A Roundabout Test

This model specifies 5 intersections, of which the central one (id '14') is a round-about. Intersections are connected by 5 roads of opposite directions, thus forming a 2-way connection. Each road is a single lane, low priority, with maximum speed of 50 km/h.

Simulation
There is at least one solver for this model.

Execute
You can download the model and solve it elsewhere.

Download

Figure 8.3: A traffic model of a roundabout

Figure 8.4: Repository models of the roundabout traffic system
CHAPTER 8. ACMOS APPLICATIONS

The export function described in section 7.12.1 enables import of AToM³ conceptual models into Zope. Such imported models are context-out models that retain their level of detail: every AToM³ model and its elements, every element and its attributes correspond to native Zope objects. Figure 8.4a shows a default textual view on the conceptual roundabout model imported into Zope model repository. Note that each model element is further hyperlinked to a detailed view on the element and its attributes. Consequently, such models can be exported from Zope and imported into AToM³ for modeling. On the other hand, simulation models are added by users to Zope as context-in type. These models are proxy objects for executable models stored in file repositories. Figure 8.4b shows the default view on a context-in model of the roundabout traffic system. These models can be executed online (see section 8.4) or downloaded for off-line simulation outside the AcMoS environment.

Conceptual and simulation models, being native Zope objects, can benefit from all document management and collaborative features of Zope. For example, a model can be indexed by Zope search engine, managed by its authors, follow a workflow among contributors, shared among team members and published for broader community. As proof of concept, more than 30 models of both types have been uploaded to the online library.

8.4 Online simulation

Online simulation involves a sequence of interactions among AcMoS components. The central role in these interactions is played by component EXPERIMENT, as illustrated with a sequence diagram in Figure 7.8. Simulation execution starts by clicking on the execute button in the public view of a context-in model. This event creates an instance of
the EXPERIMENT product, which allows the user to select a solver and then an execution scenario. Figure 8.5a shows a complete experiment setup for a simulation model of the traffic system modelled in section 8.2. An experimenter can initiate simulation by clicking on the Start button. This event submits the experiment data via Controller to the ABC solver (more on the controlling aspect in section 8.5) and changes the status of the experiment to "running". While in this status, experiment instances cannot be changed. Concurrently, the solver downloads the model files from a repository, executes the simulation, saves output files to a file repository, updates the experiment data with MRAs for the output files and calls back the experiment instance with the experiment data. In its turn, the experiment creates a context-in model' associated with the output files and updates the status to finished. Figure 8.5b shows the experiment in the finished status. Each item in the output section of the experiment web-page is a hyperlink to a file in the file repository. At this point, the experimenter can either view the output files (see two examples in Figure 8.6) by following the hyperlinks or delete the experiment together with the output context-in model'.

8.5 Management of simulation resources

AcMoS is an online multi-user environment with limited computational resources, solvers. There can be multiple concurrent internal processes competing for, acquiring and releas-
Figure 8.7: Model of the initial state of the Weighbridge example

Table 8.1: Sequence of events in the Weighbridge example

<table>
<thead>
<tr>
<th>LINE #</th>
<th>EVENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>decP(van, [getR(W), hold(3), putR(W)])</td>
</tr>
<tr>
<td>2</td>
<td>newR(W)</td>
</tr>
<tr>
<td>3</td>
<td>newP(V1, van, 0)</td>
</tr>
<tr>
<td>4</td>
<td>newP(V2, van, 2)</td>
</tr>
<tr>
<td>5</td>
<td>hold(6)</td>
</tr>
<tr>
<td>6</td>
<td>hold(0)</td>
</tr>
<tr>
<td>7</td>
<td>close()</td>
</tr>
</tbody>
</table>

8.5.1 The Weighbridge benchmark

Figure 8.7 shows the initial state of the Weighbridge environment modelled in the PROCESS INTERACTION AcMoS (a definition for this application domain was given in section 4.4). There is empty set of names $D$, empty set of resources $R$, event list $EL$ and timer $Time$ showing logical time 0. A single process named $MAIN$ is scheduled on $EL$ as current. This process defines all events that will happen in the system. These events are summarised in Table 8.1 and their description can be found in Table 4.2. According to
The system consists of a single weighbridge $W$ (line 2) with service time of 3 time units per vehicle (line 1). There are two vehicles scheduled to arrive at the weighbridge at clock times 0 and 2 (lines 3 and 4). Operation stops at clock time 6 (line 5).

The benchmark starts with the Controller having the weighbridge model loaded. After running for 6 time units, the controller has executed events from process $Main$ that declared a new process class of vans under name $VAN$, created and initialised the weighbridge $W$, scheduled two new $VAN$ processes $V1$ and $V2$ at times 0 and 2 respectively, and rescheduled $Main$ to become current at time 6. Figure 8.8 illustrates the state of the weighbridge system at this point of execution (step #6 in [4]).

After the last event of $Main$, $V1$ becomes current and proceeds to acquire the weighbridge. $V1$ is served by $W$ for 3 time units, which is simulated by re-entering $V1$ in $EL$ at time 3, that is after $V2$. $V2$ is current now and attempts to acquire the weighbridge as soon as system time reaches the scheduled time of $V2$: $TIME.Ttime = V2.Tcreation = 2$. At this step (#9 in [4]), $V2$ becomes blocked because $W$ is busy and is moved to the delay queue of the weighbridge (see Figure 8.9).

In the next step, simulation clock moves to 3. $V1$ becomes current again and its imminent action releases $W$. This promotes blocked process $V2$ by granting it ownership of $W$ and re-entering it into $EL$ at the current time, after $V1$. The weighbridge remains busy. Figure 8.10 illustrates the state of the system at this point of execution (step #10 in [4]).

By now, $V1$ has exhausted its events and is removed from the system. It is the turn of $V2$ to use the weighbridge for 3 time units, which schedules $V2$ in $EL$ at time 6, after $Main$. $Main$ is current again, but before terminating the run at time 6, it reschedules...
CHAPTER 8. ACMOS APPLICATIONS

Figure 8.9: Process V2 becomes blocked

Figure 8.10: Process V2 is promoted and scheduled after V1
itself 0 time units into the future with event hold(0) (see Figure 8.11). This moves Main behind any processes scheduled for the current time, thus allowing those processes to finish their execution (step #13 in [4]). V2 becomes current. It releases the weighbridge and is deleted. Finally, Main triggers final close event and is deleted too.

8.5.2 Simulated vs. actual operation

The presented benchmark is a simulation because all system events are predefined in the weighbridge model (see Figure 8.7). The simulated events (see Table 8.1) strictly follow the original weighbridge scenario by Birtwistle and Tofts [4, page 10]. There are three differences between a simulated benchmark and a real life scenario: Firstly, the time is logical for the former (see rule ADVANCETIME in Table 4.2) and physical for the

<table>
<thead>
<tr>
<th>EVENTS</th>
<th>SOURCES</th>
</tr>
</thead>
<tbody>
<tr>
<td>decP(...)</td>
<td>Main</td>
</tr>
<tr>
<td>newR(W)</td>
<td>Main</td>
</tr>
<tr>
<td>newP(V1, van, 0)</td>
<td>Main</td>
</tr>
<tr>
<td>newP(V2, van, 2)</td>
<td>Main</td>
</tr>
<tr>
<td>hold(6)</td>
<td>Main</td>
</tr>
<tr>
<td>hold(0)</td>
<td>Main</td>
</tr>
<tr>
<td>close</td>
<td>Main</td>
</tr>
</tbody>
</table>

Table 8.2: Weighbridge events and their possible sources
latter. The second difference lies in the source of the benchmark events. In a simulation, process Main is the only source. In the case of reality, events \( \text{newP}(V1, \text{van}, 0) \) and \( \text{newP}(V2, \text{van}, 2) \) are generated by the controller server (see section 7.11.3) in response to web-users executing Weighbridge models. The events and their sources are summarised in table 8.2. The final difference lies in the fact that service time \( T_{\text{proc}} \) cannot be predefined in actual scenarios. Therefore, event \( \text{useR}(\mathbb{W}) \) is used instead of \( \text{hold}(T_{\text{proc}}) \). For more details on \( \text{useR} \), refer to section 7.11.2. This difference is reflected in the definition of class VAN:

\[
decP(\text{van}, [\text{getR}(\mathbb{W}), \text{useR}(\mathbb{W}), \text{putR}(\mathbb{W})])
\]

Despite all these differences, the resource management behaviour remains equivalent for both the simulated and actual event scenarios.

### 8.6 Summary

The goal of this research is a Web-based environment that could be efficiently customized to modeling and simulation needs of an arbitrary number of M&S application domains. This chapter illustrated how an AcMoS can be tailored, given an abstract definition of a M&S application domain. Multiple examples of such definitions were described in chapter 4. As a matter of fact, all models presented in this thesis (some 40+) were created with tools generated from these definitions. Moreover, AToM³ itself is distributed with models that provide definitions for numerous modeling formalisms and simulators. Even more metamodels and transformations are available in the AToM³ community. These third-party definitions can be used with the demonstrated AcMoS as well. Application of MDE to the AcMoS meta-architecture enabled automated (where it matters) transformations of such abstract domain definitions into executable customizations of the AcMoS. High quality MDE tools, such as AToM³, played crucial role in automation of such transformations. Given a ready application domain definition, customization of the AcMoS takes less then one hour and more then 100 implementation artefacts are automatically generated. This presents such significant productivity increase that an arbitrary number of M&S application domains can be added to the AcMoS. Other essential functionality of cMoS, such as offline modeling, collaborative sharing of models, online simulation and management of simulation resources was illustrated as well. Verification of the practical results demonstrated that the research goal was achieved.
Chapter 9

Conclusions

This dissertation treats model-driven construction of Web-Based Modeling and Simulation applications. In particular, attention is paid to how models can be applied to achieve generality and tailorability of such applications across multiple M&S application domains and to simplify development and customization activities. This chapter concludes the dissertation and is divided into four sections. Section 9.1 summarises the investigation described in this thesis and its achievements. The contributions that stem from the work presented are listed in section 9.2. Limitations are presented in section 9.3. Finally, section 9.4 concludes this chapter with an outline of possible directions for future research.

9.1 Summary

The goal of this work is a Web-based environment that could be easily tailored to modeling and simulation needs of an arbitrary number of modeling and simulation (M&S) application domains. The main barriers to this goal were insufficient productivity associated with the traditional code-oriented development especially during application domain customization in the operational phase of the web-environment.

The investigation described in this dissertation can be divided into four themes: First, the paper establishes a framework for collaborative Modeling and online Simulation (cMoS). Second, it describes a model-driven meta-architecture that formulates activities and variation points, which are essential to customization of cMoS instances to specific M&S applications during the operation phase. Third, two model-driven development processes are proposed in order to reduce AcMoS (a cMoS instance) complexity and increase engineering efficiency at development and customization times. The processes are based on the fundamentals of the model-driven engineering (MDE) and apply outstanding MDE approaches in a novel way. Finally, the proposed framework, meta-architecture, engineering processes, an AcMoS and application domains are demonstrated in practice: First of all, arbitrary implementation technologies and MDE technologies (Python, Zope, ZODB, AToM³, etc.) are selected and a development infrastructure for the model-driven processes is created. Using this infrastructure, custom parts of AcMoS are developed. These, together with reusable generic off-the-shelf parts, realize a domain independent AcMoS. Customization of this AcMoS was verified for a number of domains: Data Flow
Diagrams (DFD), Discrete Event System Specification (DEVS), Process Interaction (PI) and Dynamic Traffic Routing (DTR). MDE allowed to significantly elevate abstraction of this repeated activity and automate code generation. Generic cMoS functionality, such as modeling, collaborative sharing of conceptual and simulation models, online execution of simulation models and management of shared simulation resources is demonstrated as well.

Verification of the practical results demonstrated that the original barriers were overcome and the research goal was achieved. In the context of the proposed solution, previously less noticeable bottlenecks came into sight: transfer of knowledge from M&S domain experts to AcMoS developers and integration of simulators into AcMoS.

9.2 Contributions

The research presented in this thesis produced the following main contributions:

An adaptation of the classic modeling and simulation (M&S) framework to the domain of web-based collaborative applications. In Chapter 5, we proposed a framework that combines general concepts and taxonomies from fields of M&S, Distributed Systems and Collaborative Software. This framework explicitly describes fundamental concepts that are invariant for the broad family of web-based modeling and simulation applications. The framework provides software engineers with a proper focus on the problem domain so that resulting software would fit harmoniously with the domain it is being developed for. The proposed framework (cMoS) was defined in section 5.2 and a software incarnation of it was developed in chapter 7.

A model-driven meta-architecture to enable efficient customization to multiple application domains. Differences among all possible application domains can be isolated to variation points in the cMoS concepts. Through customization of these points, an application domain independent cMoS instance can be tailored to a concrete M&S application. To achieve flexibility and ease of customization, we base organization of such points on the four level meta-architecture. Hence, the essential variation points together with affected cMoS concepts and modeling levels constitute the AcMoS meta-architecture (refer to section 5.3). Application domain customization was demonstrated in section 8.1.

We studied the general field of Model-Driven Engineering (MDE) in order to establish the fundamental principles and concepts (refer to chapter 3). Two widely used and mature MDE approaches, Model Driven Architecture (MDA) and Domain Specific Modeling (DSM), were described in a uniform way that allowed to compare them for applicability to the problem being addressed in this thesis.

This work is an example of a successful application of MDE to a non-trivial engineering problem. Two model-driven development processes were proposed and demonstrated. These are based on the fundamentals of the model-driven engineering and outstanding realizations of MDE (refer to section 5.4). MDE was applied in order to efficiently 1) build an application-independent instance of cMoS (AcMoS) and 2) customize the instance to
a concrete M&S application domain in the operation phase. The primary success lies in the fact that manual development of an instance customization is raised to the PIM (Platform Independent Model) level of abstraction and consequent code generation is completely automated (except for one limitation described in section 9.3).

Operational semantics of the access control to shared simulators was explicitly modelled as abstract, formal and executable transformation definition in the DSL (Domain Specific Language) interpretation approach. Hereby, system development and modification occurs at the highest possible level of abstraction, without resorting to software development. Furthermore, applying such transformation to a model, visualizes the behaviour by manipulating and animating model elements. Modeling of behaviour was demonstrated in section 7.11.

Direct practical implications, as this research applied MDE to model a number of languages for popular M&S formalisms (e.g. Data, Discrete Event System Specification, Process Interaction) and implementation technologies (Python, Zope). These were presented in Chapter 4 and section 6.2. All models are available as open-source software under the MIT License. An open source project for Zope Product framework ZCase is hosted on SourceForge (http://sourceforge.net/).

In this research, experience showed that MDE tools can be extended with new functionality by means of multi-formalism modeling, meta-modeling and model transformation: meta-modeling allows to create user interface (UI) elements for a new functionality. These elements enable users to provide input parameters if necessary. Multi-formalism modeling allows to mix custom UI elements with a source model in a different formalism. Finally, transformations can be used to specify the behaviour of the new functionality. For example, section 7.12.1 described how the export functionality was added to tool AToM³.

9.3 Limitations

This section discusses limitations that became apparent during the progress of the research.

1. Solver integration is accomplished in the traditional way. It is the only manual code development activity that occurs during application domain customization of AcMoS (see section 8.1).

2. The model-driven process described in section 5.4.1 does not raise abstraction of AcMoS development up to the level of PIM models.

3. Communication protocols among AcMoS components were not modelled and auto-generated, but coded in the traditional way.
9.4 Possible directions for further research

This section is written to help students and other researchers in selection and design of future research. The implications of the thesis are (in no particular order):

Firstly, metrics need to be developed to measure and select MDE tools on basis of quality of their meta-modeling and transformation functionality. Quality of MDE tools plays crucial role in achieving high levels of abstraction and efficiency when defining a modeling language and code generator for a new M&S domain. At the time of this research such metrics were incomplete or missing.

Secondly, since middleware as Controller is common in distributed environments, it may be worth to investigate modeling business logic as demonstrated with the Controller (see section 7.11).

Thirdly, it is interesting to research possibilities of automated code generation of efficient implementations of transformation definitions. Section 7.11 illustrated how the interpretation-based MDE approach was used to abstractly model complex behaviour as transformation definition. Efficient realizations of such transformations would allow to enhance performance parameters of their execution.

Fourthly, this work did not address efficiency of solver integration. In the context of AcMoS customization, wrapper development is the only manual coding activity of application domain definition.

This research did not address modeling relations between entities of the framework and consequently modeling communication between software components. A Language-driven approach can be employed to conceptually model and built bridges between source and target metamodels (as proposed in MDA) to implement the communication.

Finally, this research stressed the need for explicit models of major paradigm and technologies, and especially mapping among them. Libraries of such models and off-the-shelf parts would have made this and other similar projects more approachable.
Bibliography


[23] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars (a survey). In *Proceedings of the International Workshop on Graph-Grammars and Their


BIBLIOGRAPHY


Curriculum Vitae

Andriy Levytskyy was born in Chernivtsi, Ukraine on January 2, 1974. In 1991 he enrolled as a student of the Computer Science Department in Yuriy Fedkovych Chernivtsi National University, where he received his engineer diploma with distinction in 1996. Since 1995 he worked as teaching and research assistant at the same department. From June 1999 till January 2004, Andriy Levytskyy was a PhD student at Delft University of Technology, Faculty of Information Technology and Systems, Mediamatica Department. In this function he performed research to support exchange and simulation of models of different formalisms, in different languages and on different computers within multidisciplinary project NanoComp. In 2003, the author worked for three months as visiting researcher at Modeling, Simulation and Design Lab at McGill University, Canada, where he was involved in modeling formal languages and construction of domain-specific modeling and simulation environments with AToM³. The above research lead to this dissertation.
Propositions belonging to the thesis:

**MODEL DRIVEN CONSTRUCTION AND CUSTOMIZATION OF MODELING & SIMULATION WEB APPLICATIONS**

by Andriy Levytsky

1. The essence of MDE is Formulation, Formalization and Automation of a development process.

2. Productivity improvement, reduction of human error and knowledge reuse are the most direct and basic benefits of MDE.

3. MDE does not solve any problems of software development, but shifts them from a primary to a secondary development process.

4. Transformation definitions are models too.

5. MDE is often coined a revolution. However, pragmatic application of MDE for an organization is a question of evolution, not revolution.

6. A fool with a modeling tool is still a fool.

7. The code-oriented Open Source Definition has become outdated.

8. Good direct support of problem domains by MDA tools is like a good steak cooked by a vegetarian: it is possible but unlikely.

9. Despite being a simplification, abstraction remains a complex matter.

10. It is advisable to add modeling as mandatory in computer science education.

These propositions are considered defendable and as such have been approved by the promotor Prof. dr. H. Koppelaar.
Stellingen behorende bij het proefschrift:

MODEL DRIVEN CONSTRUCTION AND CUSTOMIZATION OF MODELING & SIMULATION WEB APPLICATIONS

door Andriy Levytskyy

1. De essentie van MDE is Formulering, Formalisering en Automatisering van een ontwikkelingsproces.
   Dit proefschrift

2. Productiviteitsverbetering, vermindering van menselijke fouten en het hergebruik van kennis zijn de meest directe en fundamentele voordelen van MDE.
   Dit proefschrift

3. MDE lost problemen van software ontwikkeling niet op, maar verschuift hen van een primair naar een secundair ontwikkelingsproces.
   Dit proefschrift

4. Transformatiedefinities zijn ook modellen.

5. MDE wordt wel een revolutie genoemd. Echter, pragmatische toepassing van MDE voor een organisatie is een kwestie van evolutie, geen revolutie.

6. Een dwaas met modeleergereedschap is nog steeds een dwaas.

7. De code-georiënteerde definitie van Open Source is gedateerd.

8. Goede rechtstreekse ondersteuning van probleemdomen door MDA tools is als een goede steak gemaakt door een vegetariër: het is mogelijk maar onwaarschijnlijk.

9. Ondanks het feit dat abstractie een vereenvoudiging is, blijft het toch een ingewikkelde materie.


Deze stellingen worden verdedigbaar geacht en zijn als zodanig goedgekeurd door de promotor Prof. dr. H. Koppelaar