Consistency Modeling in a Multi-Model Architecture

Integrate and Celebrate Diversity

Doctoral Dissertation

by

Jan Pettersen Nytun

Submitted to the Faculty of Mathematics and Natural Sciences
at the University of Oslo
in partial fulfillment of the requirements for
the degree PhD in Computer Science

September 22, 2010

© Jan Pettersen Nytun, 2010

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 1005

ISSN 1501-7710

All rights reserved. No part of this publication may be
reproduced or transmitted, in any form or by any means, without permission.

Cover: Inger Sandved Anfinsen.
Printed in Norway: AiT e-dit AS.

Produced in co-operation with Unipub.
The thesis is produced by Unipub merely in connection with the
thesis defence. Kindly direct all inquiries regarding the thesis to the copyright
holder or the unit which grants the doctorate.

Contents

Contents i

List of Figures vii

List of Tables xi

Abstract xiii

Acknowledgements xv

1 Introduction 1
- 1.1 Problem Statement and Architecture Sketch 2
- 1.2 Contributions 6
  - 1.2.1 Publication topics 6
  - 1.2.2 Research Method 15
- 1.3 Dissertation Outline 17

2 Introduction to Modeling and Metamodeling 19
- 2.1 Modeling 19
- 2.2 Object-oriented Modeling 21
- 2.3 Instantiation 23
- 2.4 Token and Type Model Roles 23
- 2.5 Metalinguistics and Metamodeling 25
- 2.6 The UML Metamodel Architecture 26
- 2.7 Representing Metamodel Levels 32
- 2.8 Linguistic and Ontological Instantiation 36
- 2.9 Linear and Non-Linear Metamodel Hierarchies 38
- 2.10 Multi-model Architecture 45

3 Related Work **47**
3.1 Instantiation in Some Metamodel Architectures 47
3.1.1 Metalevels in Programming Languages 48
3.1.2 The Type Object Pattern 53
3.1.3 A UML Virtual Machine 54
3.1.4 The Eclipse Modeling Framework (EMF) 56
3.1.5 Java Metadata Interface (JMI) 59
3.1.6 Metamodel for Multiple Metalevels, MoMM 60
3.1.7 Comparing and Relating to Solution (IBe) 65
3.2 Some Consistency Modeling Alternatives 66
3.2.1 A Multiple Representation Schema Language 66
3.2.2 Constraint Satisfaction and Constraint Programming . 67
3.2.3 Triple Graph Grammars (TGG) and QVT 69
3.2.4 ATLAS Model Weaver (AMW) 71

4 Solution **73**
4.1 The Consistency Modeling 74
4.2 Integrating Border environment (IBe) 77
4.2.1 `STAND` 78
4.2.2 `ACT` 85
4.3 The Solution to Consistency Modeling 89
4.3.1 User Roles 90
4.3.2 The Legacy Database Model Stack 91
4.3.3 The Consistency Model Stack 92
4.4 Applications in Addition to Consistency Modeling 99

5 Discussion, Conclusions and Further Work **101**
5.1 Alternative Solutions and Relevant Issues 101
5.1.1 Comparing to Consistency Modeling Alternatives . . . 101
5.1.2 Axiomatic or Recursive Top Level 103
5.1.3 Extending the Multi-model Architecture 104
5.1.4 The Use of Border Sides 105
5.2 Are the Requirements Satisfied? 107

Appendix A - Paper: Modeling of Consistency between Legacy Systems **113**
A.1 Problem Area 114
A.2 Solution Overview 114
A.3 The Consistency Model 117
A.3.1 The Test Environment 117

A.3.2 An Initial Example 117
A.3.3 Consistency Model 118
A.3.4 MDA and the Consistency Model 119
A.4 Modeling Consistency 120
A.4.1 Only Association and Constraints 121
A.4.2 Consistency Modeled with Association Class 122
A.4.3 Ordinary Class as Consistency Class 123
A.5 Selected Solution 125
A.5.1 Which Technique to Choose 125
A.5.2 Consistency Testing Using the Chosen Technique . . . 127
A.5.3 Interpretation of OCL Expressions 130
A.6 Summary and Research Directions 130

Appendix B - Paper: Towards a Data Consistency Modeling and Testing Framework for MOF Defined Languages 131
B.1 Introduction 132
B.2 Data Integration Framework 135
B.2.1 Use of the OMG Meta-Model Architecture 135
B.2.2 Representation of Model and Model Instance 136
B.2.3 What the Framework Should Support 137
B.2.4 Implementation of the Framework 137
B.3 Example Application of the Modeling Framework 138
B.3.1 Consistency Modeling Example 139
B.3.2 Consistency Test Tool 142
B.4 Summary and Conclusions 143

Appendix C - Paper: Representation of Levels and Instantiation in a Metamodelling Environment 144
C.1 Introduction 145
C.2 Metamodeling 146
C.3 Related Work 149
C.4 FORM: Our Basic Representation 151
C.4.1 Definition of FORM 153
C.4.2 The FORM Notation 154
C.4.3 `InstanceSpecification` in MOF 2.0 and UML 2.0 . 155
C.5 XMI 156
C.6 Instantiation 159
C.6.1 Basic Instantiation - Matching Symbols 160
C.6.2 Instantiation of Links 161
C.6.3 Different kinds of Forms 162

C.6.4 Handling Values 163
C.6.5 Derived Consistency with OCL 164
C.7 Conclusions and Research Directions 165

Appendix D - Paper: Accessibility Testing XHTML Documents Using UML 166
D.1 Introduction 167
D.2 Measuring accessibility 168
D.2.1 The WCAG guidelines 168
D.2.2 EARL reporting 169
D.2.3 The EIAO project 169
D.3 Metamodeling using SMILE 169
D.3.1 MATER - Model All Types and Extent Realization . . 170
D.3.2 MATER with set notation 173
D.4 Modeling accessibility for XHTML with UML and OCL . . . 175
D.4.1 The Metamodel 176
D.4.2 The Web Document Model (subset of XHTML) 176
D.4.3 The Model Instance 178
D.5 Conclusion and further work 182

Appendix E - Paper: Modeling Accessibility Constraints 183
E.1 Introduction 184
E.2 The EIAO project 185
E.3 The SMILE Project 185
E.4 The Approach 187
E.4.1 The XHTML-document and the Accessibility Constraints 188
E.4.2 The Three Level Metamodel Architecture 189
E.4.3 The Process of Defining and Evaluating the Constraints 190
E.4.4 Benefits 192
E.5 Conclusion 193

Appendix F - Paper: Automatic Generation of Modeling Tools 194
F.1 Introduction 195
F.2 Meta-modeling and Tool Production 196
F.2.1 Aspects of Meta-models 197
F.2.2 Tools as Meta-model Implementations 199
F.2.3 Tool Production Requirements 200
F.3 Some Meta-modeling Frameworks and Tool Production . . . 201
F.3.1 MDA Meta-modeling 201

F.3.2 XMF-Mosaic 202
F.3.3 Coral 204
F.3.4 Software Factories 205
F.3.5 More Examples 207
F.3.6 The SMILE Framework 209
F.4 Concluding Remarks 211

Appendix G - Paper: A Generic Model for Connecting Models 213
G.1 Introduction 214
G.2 The Siwa Approach 217
G.2.1 Representing One Model 218
G.2.2 Connecting Models 223
G.3 Related Work 227
G.4 Legacy Data Consistency As Example 227
G.4.1 Consistency Modeling And Testing 228
G.4.2 Implementation In Siwa 229
G.5 Summary And Research Directions 231

Appendix H: The Object-oriented Paradigm and Some Basic Philosophy 233
H.1 Seer, Seeing, and the Seen 234
H.2 Form and content 237
H.3 Is Concept the Same as Class? 238

References 245

List of Figures

1.1 Multi-model architecture with three metamodel stacks 4
1.2 Paper found in Appendix A 6
1.3 Generation of consistency data 8
1.4 Paper found in Appendix B 9
1.5 Paper found in Appendix C 9
1.6 Paper found in Appendix D 11
1.7 Paper found in Appendix E 12
1.8 Paper found in Appendix F 13
1.9 Paper found in Appendix G 14

2.1 To ways to view a model 21
2.2 A model (a) and a model instance (b) of this model 22
2.3 Token model examples, "real house" (Miller Hill 2) to the left. 24
2.4 BNF example (a) and the UML metamodel architecture (b) . 27
2.5 Simplified example of the UML metamodel architecture . . . 27
2.6 Instantiation of central UML modeling elements 30
2.7 Part of the instance model of MOF and UML 34
2.8 Representing levels with class (a) and object notation (b) . . 35
2.9 Same as Fig. 2.8 in clabject notation. 36
2.10 Ontological versus linguistic instantiation (partly from [Küh06]) 37
2.11 Smalltalk example [BG01] 39
2.12 Attaching instantiation semantics in a linear hierarchy 41
2.13 Attached instantiation semantics in a none-linear hierarchy . 41
2.14 The intensional and extensional parts of a metamodel 43
2.15 The intentional and extensional parts of a metamodel stack . 43
2.16 Embedding and spanning the UML metamodel stack 44

3.1 Metadata structure for Java objects 49
3.2 Metadata structure for Loops (dotted line is instance of) . . . 50

3.3 Metadata structure for Smalltalk-80 objects 51
3.4 Metadata structure for ObjVLisp objects 51
3.5 Extended version of the Type Object Pattern [HH] 53
3.6 Logical and physical instance of relations as found in [RF-BLO01] 54
3.7 Key Java classes from physical architecture 3.6 55
3.8 Part of the ECore metamodel (a) and generated implementation class (b) 56
3.9 UML object diagram describing the building example 58
3.10 From [AK01], notation for potency, simple and dual field . . . 61
3.11 From [AK01], components and nodes with deep instantiation 61
3.12 From [AK01], the MoMM 62
3.13 Examples of powertypes and use of potency; (c) and (d) are from [KA08] 64
3.14 Map-coloring problem and equivalent constraint-satisfaction problem [Kum92] 68
3.15 Relating UML Class to Relational Table (a) and (b), Building-House grammar rule (c) 70

4.1 Consistency between `Apartment` from model of DB1 and `Building` from model of DB2; other elements are from the consistency model 75
4.2 Some of the models involved when doing consistency modeling and testing 76
4.3 `STAND` embedded in Java (a) and spanning several metalevels (b) 77
4.4 Main part of `STAND` 78
4.5 Model instance connected to model 80
4.6 Example of `STAND` representing a class 81
4.7 Metamodel defining classes, etc. 83
4.8 Part of `ACT` 86
4.9 Naming the border sides and the semantics of the consistency application (a) 90
4.10 Consistency metamodel (`MMCD`) (a), and an instance of it (`MCD`) (c) 93
4.11 Legacy model and data (a), result of consistency testing (b) . 94
4.12 The border sides of `MCD` 95
4.13 The structure and borders sides of `CD` 96
4.14 Dependency graph for example shown in Fig. 4.1 97

5.1 Optimizing reference from model instance to model 105
2 Consistency Modeling Overview 115
3 Integration Model Encompassing Legacy Models and a Consistency Model 118
4 Connection of Constraint to Class (a) and Association (b) . 121
5 Building with Apartments 121
6 Constraints Connected to An Association Class 122
7 The Apartment / Building Problem Solved with An Association Class 123
8 Use of An Ordinary Class Instead of An Association Class . . 123
9 Use of Ordinary Class: The Building / Apartment Example . 124
10 Consistency Classes with Only One Association to Legacy Class 124
11 Has the Suspect Been Elsewhere? 126
12 Metamodel 127
13 Consistency Modeling Overview 134
14 ODM Relative to the Metamodel Architecture 136
15 A UML Component Diagram Showing a General MOF Based Modeling Tool 138
16 Integration Model Encompassing Legacy Models and a Consistency Model 139
17 Three Legacy Models 140
18 Consistency Model: Has the Suspect Been Somewhere Else? . 141
19 UML Component Diagram Showing the Consistency Tester . 142
20 The four-layer metamodel architecture 147
21 Different representations of the same structure. 148
22 Representing the same underlying structure as Fig. 21, but more explicit. 150
23 FORM Context 152
24 FORM - the basic representation 153
25 Viewing the levels as components 154
26 The visual symbols of the FORM-notation. 155
27 FORM representing the same underlying structure as Fig. 21 156
28 `InstanceSpecification` from the UML 2.0 metamodel . . . 157
29 Connecting a `Form` to its description. 161
30 Relating a connection to its description. 162
31 FORM with instantiation information 164
32 The conceptual model of MATER. 171
33 A set-like concrete notion for MATER. 174
34 Two levels with MATER, concrete syntax as shown in Fig.33 174
35 Our metamodeling architecture 175

36 A minimal reflexive metamodel 177
37 The web document model 179
38 A web sample web page 180
39 The sample web page as instance of the web page model . . . 180
40 The instance of the web page model using the notation of MATER 181
41 The Metamodeling Architecture Presented with the help of UML (the model level is only partly shown) 191
42 Structure of a Meta-model 197
43 Decoupling the Structure of a Meta-model 198
44 Tools and Meta-models 199
45 Aspects supported by Coral 204
46 Example of token and type model 216
47 Megamodel 217
48 Part of MATER: the internals of a Siwa model 219
49 The top metamodel of the examples 220
50 Example of how to represent an object in MATER 221
51 A part of the description of class `Building` 222
52 Sketch of connected models 223
53 Part of MATER: connecting Siwa models 224
54 The example (incomplete) of Fig. 46 in MATER 225
55 Architecture of building example (`TMBuilding` is not included) 226
56 Model Object-M in detail 226
57 Consistency between `Apartment` and `Building` 228
58 Consistency modeling metamodel 229
59 Architecture example: consistency modelling and testing . . . 230
60 Subject and object. 236
61 The meaning triangle (a) and a variant of it (b) 239

List of Tables

4.1 Constraining `STAND` 82

Abstract

Central to *Model-Driven Engineering* (MDE) is seeing models as objects that can be handled and organized into metamodel stacks and multi-model architectures. This work contributes with a unique way of doing *consistency modeling* where the involved models are explicitly organized in a multi-model architecture; a general model for creating multi-model architectures that allows semantics to be attached is defined and applied; explicit attachment of semantics is demonstrated by attaching Java classes that implement different *instantiation semantics* in order to realize the consistency modeling and the automatic generation of *consistency data*.

The kind of consistency addressed concerns relations between data residing in legacy databases defined by different schemas. The consistency modeling is meant to solve the problem of exposing inconsistencies by relating the data. The consistency modeling combines in a practical way visual modeling and logic (OCL). The approach is not limited to exposing inconsistencies, but may also be used to derive more general information given one or more data sets.

The consistency is modeled by defining a *consistency model* that relates elements of two given legacy models. The consistency model is expressed in a language specially designed for consistency modeling. The language allows definition of classes, associations and invariants expressed in OCL. The interpretation of the language is special: Given one conforming data set for each of the legacy models, the consistency model may then be automatically instantiated to consistency data that tells if the data sets are consistent or not. The invariants are used to decide what instances to generate when making the consistency data. The amount of consistency data to create is finite and limited by the given data sets. The consistency model is instantiated until no more elements can be added without breaking some invariant or multiplicity. The consistency data is presented as a model which can be investigated by the user.

Acknowledgements

I would like to express my gratitude to the University of Agder for funding my PhD study and the University of Oslo for accepting me as a PhD student.

Thanks goes to my colleagues at the University of Agder, for all the support given through the years. A special thanks goes to co-authors Terje Gjøsæter, Merete S. Tveit, Mikael Snaprud and Andreas Kunert.

I thank my supervisor professor Vladimir A. Oleshchuk at the University of Agder for helping me when being accepted as a student at the University of Oslo, and for our interesting discussions that we had in the beginning of my study.

Thanks goes to, my co-author when writing my first accepted paper, professor Christian S. Jensen for his inspiring cooperation.

I thank supervisor associate professor Arne Maus for generously accepting me as his PhD student, for his constructive and honest critique that helped keeping me on the right track.

I thank supervisor professor Andreas Prinz for believing in the ideas when my own faith was low, for being a coworker and a true helper when using much of his precious time commenting on my work.

I thank supervisor professor Birger Møller Pedersen for sharing his superb knowledge of the field and for all the time he has used to help me on my way, for being the admirable patient supervisor that he his.
For all this and more I am truly grateful.

Again, my sincere thanks go to all of my supervisors – they have really been there for me. However, I am the only person responsible for possible errors that can be found in the thesis.

Last, and not least, thanks go to my family for their support, patience and love.

Jan Pettersen Nytun
Grimstad, Norway, 29th of July 2010

Chapter 1

Introduction

During the past several decades, the Internet and its connected resources have become a huge collective database that is playing a major role for mankind. The database is composed of data sources that together may contain overlapping but inconsistent data. The different data sources are often based on different models (schemas), which complicates the process of revealing inconsistencies. A manual check for inconsistencies can for efficiency reasons only be applied to small data sets, and consequently there is a need to automate this process. The area of *data warehousing* [Inm96] offers techniques that can be used to do this kind of checking. While data warehousing is typically based on SQL, another approach, which is proposed in this work, is based on a combination of visual modeling and first order predicate logic. The approach proposed allows the user to model consistency requirements and then later automatically generate consistency data for selected data sources; this emphasis on modeling (together with generation) coincides with the current research direction called *Model-Driven Engineering*[1] (MDE) [Ken02]. This direction is currently addressing the notion of models and model architecture. Jean Bézivin states [Béz04]:

> ... This consists in giving first-class status to models and model elements, similarly to the first class status that was given to objects and classes in the 80s, at the beginning of the object technology era. The essential change is that models are no longer used only as mere documentation for programmers, but can now be directly used to drive software production tools.

[1] While the main sections of this dissertation uses the term MDE, there are many strongly related terms such as Model-Driven Development (MDD) [AK03], Model-Driven Architecture (MDA) [MM03, KWB03] from the Object Management Group (OMG).

A model-driven approach for defining and testing consistency requirements is presented in this work; it involves several models, e.g., the models of the data sources and the data sets (data sets are seen as *terminal models*, i.e., models that cannot be instantiated further). To allow the definition and utilization of webs of models, like the one being presented, new flexible modeling frameworks need to be developed. A *megamodel* [BJV04,Fav05] is a model whose elements represent and refer to models, metamodels, metametamodels, services, tools, etc. Explicit megamodels are essential to the understanding and evolution of MDE. This work proposes a partial megamodel called IBe for defining multi-model architectures; IBe is also used as the name of the approach as a whole.

Another trend, related to MDE, advocates use of languages tailored to specific problem domains, i.e., Domain Specific Languages (DSLs). This trend is based on the understanding that a language that is not suitable for its application domain will typically inflict limitations on communication and understanding; *(linguistic) metamodeling* meets this challenge as a vehicle for designing suitable languages. Examples of this approach are Language-Driven Development (LDD) [CESW04] and Software Factories (SF) [GSCK04] from Microsoft. The language proposed for doing consistency modeling is a domain specific language; by using IBe, this DSL can be defined and placed as a (meta)model in a metamodel architecture; this model may again be instantiated to statements corresponding to consistency requirements.

1.1 Problem Statement and Architecture Sketch

Comparing and relating different models of the same entity is often of interest since inconsistencies and also a more comprehensive set of properties may be revealed; these operations are complicated when the different representations are stored in different data sources that are defined by different models (schemas). Some inconsistencies are complex and require complex querying for exposure; one example may be to detect whether a criminal suspect has been lying about where he has been, assuming that there are three legacy systems involved. Legacy system one contains information given by the suspect concerning where he has been, legacy system two contains pictures of the suspect, and legacy system three records photos at specific observation posts. In this case, a suspect is exposed as lying if he claims to have stayed in one place, but has been observed at the same time from an observation post located elsewhere. Specifying complex consistency requirements is not

trivial, and despite the widespread use of SQL, this language may not be the best solution when specifying such requirements – "divide and conquer" is a common strategy when solving complex problems and some means to support this strategy also in our case would be beneficial.

Several of the sources available on the Internet offer semantically overlapping data, and it is often of interest to integrate such data sources (e.g., the merging of ontologies [SS04]); however, the lack of consistency among them makes this integration difficult. In certain cases the goal is not to permanently integrate the data, but rather just to expose inconsistency among different data sources with related data. A similar problem is to analyze one single data source to determine whether it matches requirements that have been added to its model; for example, such an approach may be used to model and test accessibility requirements for web documents.

The problems stated above involve data sources, models and also metamodels that are specified more or less explicitly – which leads to the understanding that a modeling environment[2] is needed. Furthermore, in the spirit of MDE, consistency requirements should be modeled explicitly in a *type model*, and the results of testing requirements should appear in the form of a model (some would call this a *terminal model* or a *token model*) that is generated automatically. Continuing along the same line, explicit metamodels defining the languages used is also seen as an advantage.

The contours of metamodel stacks appear when approaching the problem of consistency modeling from an MDE point of view, and it seems natural to define a language (metamodel) to use for the consistency modeling. By so doing, the consistency requirements modeled by the *consistency modeler* become an instance of the metamodel. Finally, the result of testing consistency requirements appears as an instance of the consistency model (i.e., the result appears as a data set where each element is an instance of an element of the consistency model), this instance is called *consistency data*. Fig. 1.1 gives one possible multi-model architecture depicting the situation.

Fig. 1.1 shows three metamodel stacks, one for each of the legacy databases and one for the consistency modeling. In Fig. 1.1 all metamodels conform to the same metametamodel, which is supplied by the environment as an *embedding language*. The embedding language may also be termed *representational language* or *representational model* (also the term *realization*

[2]The terms environment and framework are used interchangeably when describing the proposed solution; in this context the term (modeling) environment means the obvious: An environment for doing modeling. The term framework is also applicable since some (Java) classes are supplied and meant for subclassing, e.g., generic interpreters of class descriptions.

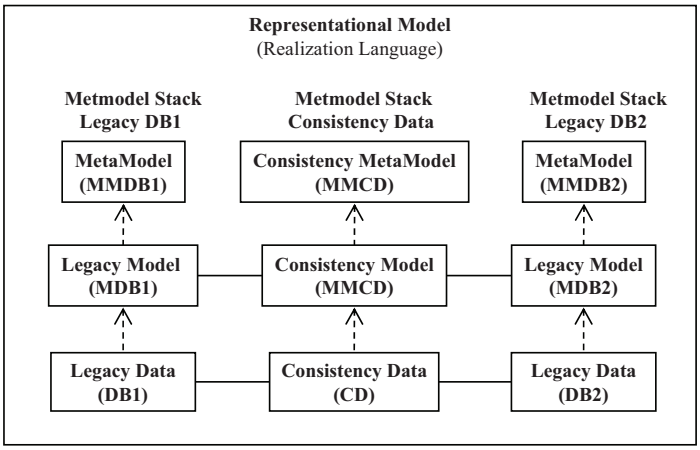

Figure 1.1: Multi-model architecture with three metamodel stacks

language seems adequate since it is already realized and it functions as a tool for realizing the other languages being defined). While a consistency modeler defines only the consistency model, the other models are either supplied or generated automatically.

The metamodel stacks are not hard to see; it is, however, not obvious how to connect the models (the arrows and lines between the models in Fig. 1.1) and determining how the language for doing consistency modeling should look. A possible *consistency metamodel* is supplied by this work together with a partial megamodel that defines multi-model architectures (the megamodel can also be seen as a representational model for multi-model architectures).

The following is a list of requirements and issues involved when defining a suitable environment (framework):

Req. 1 A language (metamodel) is required for consistency modeling.

Req. 2 Consistency data is to be generated automatically (a suitable algorithm is needed) and also the different models must in some way be established in the environment. Consequently, the environment needs to support behavior.

Req. 3 The consistency and legacy models must be connected when making

a consistency model. Consequently simultaneous handling of models in different metamodel stacks is required.
The consistency data is to be generated automatically and connected to legacy data, so simultaneous handling of different terminal models (data sets) is also required.
In short, the environment should support multi-model architectures (e.g., as seen in Fig. 1.1) so that the user is not forced to merge elements from different metamodel stacks, but is allowed to manage the individual models as pluggable modules (objects).

Req. 4 Modeling and generation of consistency data should be tightly integrated and it should not be necessary to leave the modeling environment to generate the consistency data (i.e., the modeling environment is the runtime environment).

Req. 5 Visual modeling is considered beneficial and should be supported.

Req. 2 states that behavior should be supported by the environment. Behavior must include different types of instantiations, e.g., the algorithm for the automatic generation of consistency data is a kind of model instantiation; also the establishing of the other models in the environment may be handled as instantiation.

It is practical to stay in the same environment when doing consistency modeling and generation of consistency data – this is what motivates Req. 4; if the data sets are huge and the consistency model is complex then this argument is weakened; on the other hand, in a situation where the correctness of a consistency model is tested then the argument for a tight integration of modeling and generation of consistency data is strengthened, e.g., having the consistency data generated immediately when the consistency model is changed gives a truly interactive system.

The solution sought is a type of executable modeling where the consistency model is like a statement (declaration) that returns the consistency data when executed (interpreted). Seeing the approach as a form of executable modeling together with Req. 4 gives the understanding that the approach should be more like an interpreter-based approach than a compiler-based approach. (At a later point when the presented technology has matured then also a compiled version may be developed. Compilation will typically give faster execution once the compilation has been performed, but more time will be needed before the modeler gets from modeling to execution. Consequently, offering interpretation and compilation lets the modeler choose the approach that fits best in a given situation.)

The next section will relate the contributions of the work to the requirements.

1.2 Contributions

This section introduces the papers of the dissertation and the research method used. Subsection 1.2.1 explains how the papers relate to each other, forming in turn a coherent research contribution. Subsection 1.2.2 exemplifies contributions when identifying research method used.

1.2.1 Publication topics

It is often the case that the same entity is represented more than once in either identical or different databases. This may lead to inconsistencies, e.g., one data source claims that the number of apartments in a building is 10 and another data source claims it is 11. For data sources with semantically related models, one simple consistency rule may be the following: Two objects (entities) with the same identity must have the same values stored for corresponding attributes; otherwise, they are not consistent with each other.

Modeling and Testing Legacy Data Consistency Requirements:
authors = Jan Pettersen Nytun, Christian S. Jensen year = 2003 type = conference UML

Figure 1.2: Paper found in Appendix A

The paper entitled *Modeling and Testing Legacy Data Consistency Requirements* [NJ03] (Fig. 1.2) focuses on the consistency problems that occur when previously uncoordinated, yet semantically overlapping, data sources are being integrated. The technique proposed is related to *constraint satisfaction* [Kum92] and *constraint programming* [Bar07] where relations between variables may be stated in the form of constraints. The paper is found in Appendix A.

The approach presented allows the *consistency modeler* to introduce new associations and classes with attributes, in this manner a consistency model is defined. The consistency model ties (existing) legacy models together and models consistency requirements (or other types of data that it is possible to derive in this manner). The consistency testing is performed on the consistency model instance level (i.e., the consistency data is a consistency model instance), were the legacy data is found. Initially, the consistency model does not have a model instance due to the fact that this data is automatically derived. Invariants stated in OCL are added to the consistency model and used when establishing the consistency data. The consistency data is derived data as specified by the consistency model.

The invariants are used to decide which instances to generate when making the consistency data – e.g., assigning a Boolean property the value that fulfills an accompanying invariant, an invariant where the property is used as the left hand side in an equality comparison. The approach is declarative: The consistency model declares which instances should be present at its model instance level (given legacy data). The declarations in question provide automatic instantiation.

Fig. 1.3(d) gives an example of automatically generated consistency data. Fig. 1.3(a) shows the schema (`MDB1`) for a legacy system in the form of a UML class diagram, and it also shows the data (`DB1`) for this legacy system in the form of a UML object diagram; Fig. 1.3(b) shows another legacy system with semantically overlapping data. Fig. 1.3(c) relates the just mentioned models to Fig. 1.1. We can see from Fig. 1.3(a) and Fig. 1.3(b) that `DB1` has two apartments with building id `b1` and that this is consistent with `DB2` that has a building with building id `b1` and an apartment count of 2; `DB1` has an apartment with building id `b2` that is not consistent with `DB2` that has a building with building id `b2` and an apartment count of 0. Fig. 1.3 does not show the consistency model, but Fig. 1.3(d) shows some possibly automatically generated consistency data, e.g., property `cApartmentCount` is `false` for the instance of the consistency class `ConsistencyApartmentBuilding` (this class is defined in the consistency model) which is connected to the `Building` instance with id equal to `b2`. Fig. 1.3(e) relates the models of Fig. 1.3(d) to Fig. 1.1.

The paper makes the following contributions:

- A technique for consistency modeling that includes visual modeling.
- A metamodel to instantiate when the technique is used. For the consistency model to be applicable, it must conform to the specified meta-

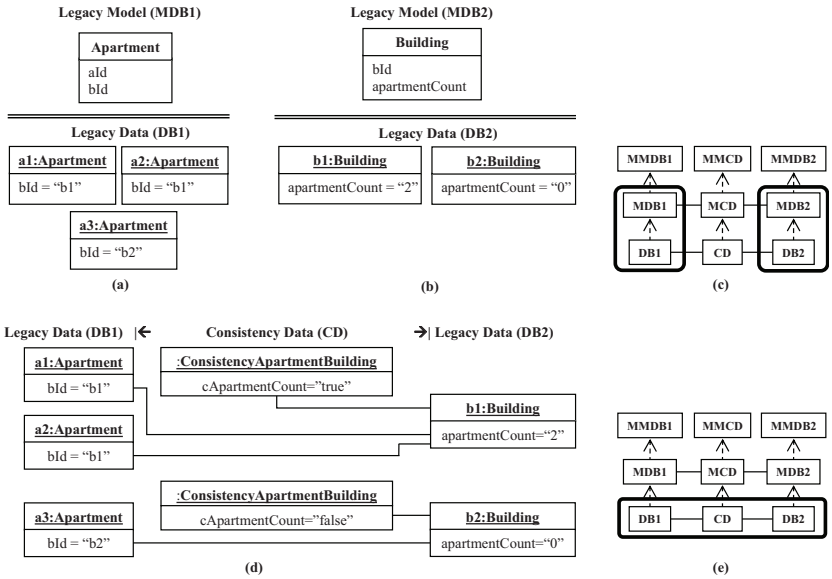

Figure 1.3: Generation of consistency data

model.

- A unique manner of utilizing OCL to produce a model instance.
- An algorithm for automatic testing the consistency of legacy data.

The paper *Modeling and Testing Legacy Data Consistency Requirements* is important to the project's overall objectives, satisfying the following: Req. 1 (a language for achieving consistency modeling) and Req. 2 (an algorithm for the automatic generation of consistency data). It is also demonstrating visual modeling (Req. 5) as it should be experienced by the modeler, but it does not specify how to implement this. Realizing the solution presented in this paper involves the management and organization of models into a multi-model architecture – this issue is the target of some of the later papers.

The focus of the paper *Towards a Data Consistency Modeling and Testing Framework for MOF Defined Languages* [NJO03] (Fig.1.4) is the archi-

Towards a Data Consistency Modeling and Testing Framework for MOF Defined Languages:
authors = Jan Pettersen Nytun, Christian S. Jensen, Vladimir A. Oleshchuk year = 2003 type = conference NIK

Figure 1.4: Paper found in Appendix B

tecture and components for building a modeling framework that supports multi-model architectures. The paper can be found in Appendix B.

The paper considers models to be objects that can be connected to form multi-model architectures and considers Eclipse [Dri01, DFK+03] as a potential implementation platform. Eclipse is designed for building integrated development environments; it has a plug-in architecture that makes it suitable for extensions, and several useful plug-ins are already available.

UML 2.0 introduces the metaclass `InstanceSpecification` that can be used to model an instance of a model element. For example, it may be used to illustrate an object of a class. The paper proposes to use a modified version of `InstanceSpecification` for the representation of model element instances.

Representation of Levels and Instantiation in a Metamodelling Environment:
authors = Jan Pettersen Nytun, Andreas Prinz, Andreas Kunert year = 2004 type = workshop NWUML

Figure 1.5: Paper found in Appendix C

The paper *Representation of Levels and Instantiation in a Metamodeling Environment* [NPK04] (Fig.1.5) continues to elaborate on the modeling

framework (the paper is presented in Appendix C). Its main contribution is the definition of a uniform way of representing metadata and object information in a metamodeling environment. The proposed solution comprises an enhanced instance model in which there are objects, slots, links and references to metadata. The instance of relation is made completely visible, and the representation may be used at all levels; the *metamodel border* between two levels is seen as an interface composed of symbols (e.g., names of classes) which, from the lower level, represent the instantiable elements (types) of the higher level. When combined with its upper and lower interfaces, one metalevel constitutes a manageable module. Such a module may initially be stored together with its interfaces and subsequently retrieved and connected with an adjacent level in the metamodel stack; the two interfaces selected from both the models include one upper and one lower interface. The two interfaces are merged into what is called a (*metalevel*) *border* that comprises the set of symbols from both the interfaces.

The uniform representation of levels has several benefits:

- Tools based on this representation may be used on all levels, e.g., an OCL-evaluator.
- It is possible to compile the classes of the representation in advance and obtain an interpretative solution.
- It is easy to define an XML schema that may be used for all levels (including the data level).

When the paper was written, the instance representation was called Form; a visual syntax for Form is defined. The paper does not give a complete description of instantiation; however, it discusses the problem of deciding whether two given single levels may be seen as neighbor levels. Two levels may be connected by simply fusing the symbols mentioned above; however, this approach may not function well, especially if no namespace information is "stored in" the symbols. The paper considers *basic instantiation patterns* for all the elements of Form, and it proposes to store some information in the instances of Form revealing the basic instantiation patterns that have been used when the instances were established. While this approach may strengthen the process of connecting the levels inside the framework, it does not fully solve the question of instantiation. Nevertheless, the paper does make some major steps towards realizing Req. 3. The paper is found in Appendix C.

The paper *Accessibility testing XHTML documents using UML* [GNPT05] (Fig.1.6) introduces a new application. The paper is presented in Appendix

Accessibility testing XHTML documents using UML:
authors = Terje Gjøsæter, Jan Pettersen Nytun, Andreas Prinz, Merete S. Tveit year = 2005 type = workshop NWUML

Figure 1.6: Paper found in Appendix D

D. The basic instance representation (called Form above) has been modified and it is now being called Model All Types with Extent Realization (MATER). MATER models level borders and levels explicitly, and namespaces are also included in this version.

The paper focuses on the modeling and testing of accessibility requirements for web documents, where accessibility relates to people with disabilities. A web page is created according to certain specifications, e.g., the XHTML specification, and it may then be seen as a model instance of the specification. The model (e.g., the XHTML specification) and the model instance are both represented in MATER. OCL formulas may be attached to the model, thus expressing simple and advanced accessibility requirements. This application may be seen as a subset of the more general application described above (modeling and testing legacy data consistency requirements) – it is a simplification because only one model and model instance are involved; the testing may be completed in the same manner as described above.

The paper *Modeling Accessibility Constraints* [GNP+06] (Fig.1.7), presented in Appendix E, continues to investigate how to do modeling and testing of accessibility constraints, offering a more mature (simpler) and robust approach. The metamodel in question is extremely simple: Only classes with properties and composition are considered. Although the approach specified does not need the metamodel to be present in the framework, the understanding it represents will still pervade a solution. Moreover, the model (e.g., the specification of XHTML) in which the accessibility constraints are attached may be incomplete making this approach more robust. The approach describes how a model is automatically constructed given a model instance (a web document), the created model reflects what the model in-

Modeling Accessibility Constraints:
authors = Terje Gjøsæter, Jan Pettersen Nytun, Andreas Prinz, Mikael Snaprud, Merete S. Tveit year = 2006 type = conference ICCHP

Figure 1.7: Paper found in Appendix E

stance is composed of (elements from the specification not used will not be modeled). The premade model with the attached accessibility constraints is merged with the model that has been created automatically, and at this point the constraints are tested on the model instance. A report describing the accessibility violations is subsequently created. Several benefits of this approach are described in the paper, e.g., the parts of a document that conform to common deviations may be tested for accessibility, which is done by including model elements describing common deviations and attaching accessibility constraint to these same elements.

The paper demonstrates the usefulness of allowing models to be treated as pluggable modules by having a "model instance" first created and then attached to a possible model.

The papers in Appendix E and Appendix D make it clear that "incomplete metamodel stacks" may also be interesting (e.g., starting with a terminal model and then establish a model by doing analysis). This understanding is added to Req. 3 which concerns a model for representing multi-model architectures.

The paper *Automatic Generation of Modeling Tools* [NPT06] (Fig. 1.8), presented in Appendix F, defines a terminology for aspects of metamodels, and it investigates how they are supported by existing metamodeling tools. The term metamodel is used in a general sense: *A metamodel is a model that defines a language completely, including the concrete syntax, abstract syntax and semantics.*[3] The paper places MATER in a broader context in which requirements for metamodeling frameworks are discussed – the paper

[3]Today this is not a common view; often the metamodel is only seen as defining the abstract syntax.

Automatic Generation of Modeling Tools:
authors = Jan Pettersen Nytun, Andreas Prinz, Merete S. Tveit year = 2006 type = conference ECMDA

Figure 1.8: Paper found in Appendix F

functions as a related-work study.

The paper *A Generic Model For Connecting Models In A Multi-level Modeling Framework* [Nyt06], found in Appendix G, (Fig.1.9) presents a generic way of both connecting models placed at adjacent levels and connecting models placed at the "same level". The MATER model is extended to support this weaving of models, where for example two connected models may reside at the same level while conforming to different metamodels at the level above.
The testing of legacy data is used as an example; handling of execution semantics is not a main theme of the paper, but a feasible solution based on *semantic engines* is sketched. A semantic engine is an interpreter that interprets the structure (e.g., the description of a class) where it is attached; attachment in this context correspond to a Java reference to executable Java code. When the Java code is executed, it interprets the structure, and this execution is giving the structure a meaning (e.g., the structure could be the description of a class and the execution could be the creation of an object of the class). By attaching different semantic engines, different interpretations will result. Instead of using the term "semantic engines", the following chapters are simply using the term "*semantics*".

The framework presented has the following properties:

- It is loosely coupled to the instantiation found in its implementation language, this allows a model to have several type models offering different and useful information about the model.
- It is extremely generic, which makes it adaptable to many different modeling needs, e.g., it may allow separate existence of properties.
- It is possible to have incomplete architectures, e.g., XML documents

may be loaded for analysis, and then a model may be produced automatically (Appendix E). This is typically not possible in other frameworks due to their strong coupling to instantiation in the selected implementation language.

The paper proposes a model for representing multi-model architectures, and by this it satisfies Req. 3. A model, in such a multi-model architecture, appears like an "ordinary object" with defined borders (interfaces) to other objects (models), e.g., there will be a border between a model and one that describes it in a type-like fashion (in Chapter 4, which describes the solution, this type of border is called a *descriptor border*). Semantics may be attached to model elements and to borders. Attaching instantiation semantics to a border may, when executed, create a complete new model and place it in a multi-model architecture relating it to several other models. The paper proposes a way of integrating the algorithm (i.e., the algorithm for automatic generation of consistency data) into the multi-model architecture; it proposes to attach it as a semantic engine to a border and by this the paper satisfies Req. 2. The details of this approach are explained in Chapter 4.

A Generic Model for Connecting Models in a multilevel modelling framework:
authors = Jan Pettersen Nytun year = 2006 type = conference ICSOFT

Figure 1.9: Paper found in Appendix G

(Certain "philosophical aspects" of the approach have been discussed in the paper *Metalevel Representation and Philosophical Ontology* [NP04].)

Chapter 4 presents a coherent and a slightly updated description of the solution. After being awarded a series of working titles, the name of the framework (the conceptual model) has now been set to Integrating Border environment (IBe).

1.2.2 Research Method

IT research deals with artificial phenomenon like organizations and information systems. An essential property of artificial phenomenon is that it may be both created and studied [MS95]. The *hypothetic-deductive method* is well suited in natural science which aims at describing and explaining the *reality*. This method can also be used in some areas of computer science, e.g., development of web crawling technologies to detect updates of web pages, which may be done by stating some algorithm together with a hypothesis about its efficiency compared to existing algorithms, followed by real world experiments on the Internet for verification. In other cases traditional scientific methods are difficult to apply, Juris Hartmanis states [Har93]:

> ...an inspection of the experimental work and systems building in computer science reveals different pattern than in physical sciences. Such work deals with...testing feasibility by building systems to do what has never been done before...demos can play the role of experiments. Furthermore, the science and engineering aspects are deeply interwoven in computer science, where the distance from concepts to practical implementation is far shorter than in other disciplines...
> ...we can see that computer science is concerned with the abstract subject of information, which gains reality only when it has a physical representation... The goal of computer science is to endow these information processing devices with as much intelligent behavior as possible.

Alan R. Hevner et al. see *behavioral science* and *design science* as paradigms fundamental to research in the Information Systems discipline [HMPR04]; in this context the behavioral science paradigm is understood to be focused on development and verification of theories that explain or predict human or organizational behavior [HMPR04]. This dissertation is not part of behavioral science – it belongs to design science. Design-science is rooted in engineering, and its purpose is to extend the boundaries of human and organizational capabilities by creating new and innovative artifacts [HMPR04]. The products (i.e., the artifacts) of design science are of four types [MS95, HMPR04]:

Constructs A language for addressing the phenomena, i.e., vocabulary and symbols representing basic concepts.

Models The constructs may be combined to form higher order constructs

called models, used to describe tasks, situations, or artifacts (abstractions and representations).

Methods Ways to perform goal-directed activities, i.e., algorithms and practices.

Implementations (Instantiations) The already mentioned artifacts may be instantiated into specific products, i.e., implemented and prototype systems.

Peter J. Denninger et al. recognize three types of work processes in the discipline of computing: *Theory* (building conceptual frameworks and notations for understanding relationships among objects in a domain and the logical consequences of axioms and laws), *experimentation* (exploring models of systems and architectures within given application domains and testing whether those models can predict new behaviors accurately) and *design* (constructing computer systems that support work in given organizations or application domains) [DCG+89,Den99]. Further, they consider the three processes so intricately intertwined that it is irrational to say that any one is fundamental [DCG+89].

Alan R. Hevner et al. relate the processes to design science and consider there to be two basic activities: *Build* and *evaluate* [MS95]. Build is the process of constructing the artifact and evaluate is the process of determining how well the artifact performs.

Several artifacts are presented in this dissertation, e.g., conceptualizations or constructs like *consistency class* and *consistency association* which are parts of the vocabulary used when the solution is described. The most important artifact is a new unique method to do consistency modeling at a high level of abstraction (Appendix A); this method is considered to have value or utility to people that wishes to do consistency modeling. The following statement given by Salvatore T. March and Gerald F. Smith is claimed to be applicable: *The research contribution lies in the novelty of the artifact and in the persuasiveness of the claims that it is effective* [MS95].

Different variations of the method have been presented, and based on a selected set of criteria, one were chosen (Appendix A.4). The method has also been compared to existing methods that may be extended to solve the problem of doing consistency modeling (e.g., Section 3.2.4, 3.2.2, 3.2.1 and Subsection 5.1.1).

Some additional applications of the method (or more correctly adaptations of the method) have been researched, i.e., modeling of accessibility

constraints (Appendix D and E). These studies have further strengthened the claim that the method is effective.

The proposed method is being partly described by a metamodel (Section 4.3.3) defining the language to use when doing consistency modeling, the metamodel is expressed in UML, and it is by this gaining some rigor.

The concrete syntax of the language, proposed in Appendix A, is a subset of UML class diagrams with OCL. The concrete syntax is by itself supported by common UML diagram tools. However, the interpretation of the language is special and not supported by any existing tool. Consequently, there is no tool that "fully" supports the consistency modeler, e.g., automatic generation of consistency data is missing. Hence, several of the papers addresses the design of a tool to realize the method (Appendix A, B, C, F and G) – this is in conformance with design science, which is characterized by its engineering roots, in which progress is primarily achieved by posing problems and systematically following a design process to specify/construct systems that solve them. The design proposed relates to MDE and is partly given in the form of a model (called IBe) for handling models. The dissertation reports on parts of the work that has been conducted when defining the tool, e.g., instantiation is an important feature that needs to be supported by the tool, a solution to this is described in Chapter 4 and some related work is presented in Chapter 3. One role of the proposed design is to confirm the feasibility of the tool. However, a well functioning tool is needed to evaluate the methods full potential.

The model for handling models, corresponds to an ontology describing model handling – to find the "right concepts" theoretical/philosophical works have been studied (e.g., [Küh06] [Lud03] [Sei03b] [Béz05] [JB06] [Fav04b] [Fav04c] [Fav04a] [Fav05]).

1.3 Dissertation Outline

The remainder of the dissertation is structured as follows: Chapter 2 gives an introduction to the object-oriented way of modeling and metamodeling. It also introduces some concepts used later.

The first part of Chapter 3 explores related work in regard to instantiation semantics and metamodel stacks. The attention towards instantiation and attachment of semantics reflects the current phase of the work. The last part of Chapter 3 presents alternative approaches in regard to revealing consistency among data sources.

Chapter 4 presents the solution, summing up essential points made in

the papers and giving the reader easy access to the work constituting the dissertation.

Chapter 5 discusses the results and offers a few concluding remarks together with some considerations in regard to further work.

Appendixes A-G contain the papers included in this dissertation (presented above).

Appendix H gives an informal introduction to the object-oriented way of thinking and the basis of modeling, it may be omitted by readers familiar with these topics.

Chapter 2

Introduction to Modeling and Metamodeling

Object-oriented metamodeling is rooted in the object-oriented world view, which again aims at mimicking the way a human perceives reality and construct descriptions of "abstract things". Appendix H gives an informal introduction to the object-oriented way of thinking and the basis of modeling. Appendix H fits in before this chapter, but it may be omitted by readers familiar with the object-oriented paradigm and basic language theory. This chapter gives an introduction to modeling and metamodeling; these issues are discussed so that the context of the solution and the vocabulary used to describe the solution is made clear.

2.1 Modeling

The models of MDE are language-based in nature as opposed to physical scale models; the following definition of a model is given by E.D. Falkenberg et al. [FHL+98]:

> A model is a purposely abstracted, clear, precise and unambiguous conception... a model denotation is a precise and unambiguous representation of a model, in some appropriate formal or semi-formal language.

A model is seen as something abstract (conception), communicated by a description (denotation) given in some (modeling) language. Often the distinction between model and model denotation is ignored – the term model

is used when model denotation is meant (this imprecision may also be found in this work).

Thomas Kühne presents the following definition of what a model is [Küh06]:

> A model is an abstraction of a (real or language based) system allowing predictions or inferences to be made.

Jean Bézivin and Olivier Gerbé present another but quite similar definition [BG01]:

> A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system.

Both the previous definitions indicate that some characteristics of the system (subject) are in some way represented by the model – there is a homomorphic relation between model and system. The definitions also indicate that it is possible to use the model to derive information about the system (demonstrate properties). To know that the derived information is valid, it must be possible to map (interpret) the result onto the system. Ed Seidewitz [Sei03a, Sei03b] defines an interpretation of a model as being:

> ...mapping of elements of the model to elements of the system under study such that the truth-value of statements in the model can be determined from the system under study, to some level of accuracy.

In this way a model is given meaning – an example of what it means is explicitly shown.

Herbert Stachowiak [Sta73] is a bit more explicit when it comes to the features of a model and presents the following list (collected from [Küh06]):

Mapping feature A model is based on an original (there is a subject).

Reduction feature A model only reflects a (relevant) selection of an original's properties.

Pragmatic feature A model needs to be usable in place of an original with respect to some purpose.

The model plays a role in relation to what it models and can therefore be understood as a relative concept. Also, Kühne [Küh06] does not accept a copy being a model of what it copies.

Seidewitz outlines two different ways to consider a model [Sei03a,Sei03b]: As *a description* or as *a specification*. Given a model and a system that are not consistent with each other in regard to some aspect, which one of them is *considered bearing the truth*? Natural scientists have *nature* as their system under study; nature is typically seen as *a given system* (Fig.2.1(a)); the mission is to come up with a description (model) that is so good that it can be used to predict and explain natural phenomena. A model can also be a set of statements that specify a system that is to be constructed (Fig.2.1(b)) and in this case the implemented system *is wrong* if it does not follow the specification.

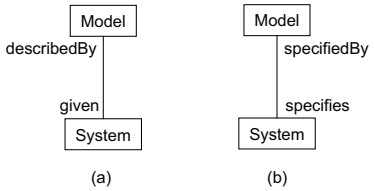

Figure 2.1: To ways to view a model

If you have a system that is fulfilling a specification, then the specification is also a description of the system. Most models found in software engineering are specification models [Sei03b] (aka, "prescriptive models" [Lud03]) and typically the "original" does not exist when the model is created. However, not all software models come about as specification models, e.g., analyzing data may be seen as an attempt to make a model when the data being analyzed are given.

2.2 Object-oriented Modeling

Nigel P. Capper describes the object-oriented world view as [Cap94]:

> Conceptually, object-oriented world view is that of a collection of interacting objects, each with a time-varying status expressed in terms of data attributes and each with behavior expressed as responses to interactions with other objects. Each object is an instance of a particular class (for example, bank account) whose behavior is expressed in terms of methods (that is, function), each triggered by a message (for example, debit account). Classes can inherit data attributes and methods from other, more

general, classes (for example, savings account inherits from bank account).

The great success of object-oriented languages is due to the possibility to map the problem domain more or less directly to a program solution [Ber97]; object-oriented technology minimizes *accidental complexity* [Fre87] otherwise introduced by none object-oriented technology that is used to solve the problem. Of course *essential complexity*, which relates directly to the complexity of the problem itself, is inherent and unavoidable. Essential complexity relates to content and accidental complexity relates to form; form correspond to "manifestation" or the result of a *reification* – in other words object-oriented technology makes the reification process more "straight forward".

The objects of the computer appears as the top of a language stack where one language is defined by the help of the one beneath it; these language levels hide the physical and software particularities of the computer and offer an "advanced medium".

UML [OMG03c] is a general purpose visual modeling language for specifying and visualizing object-oriented systems. UML includes *class diagrams* showing classes. *Object diagrams* are a special case of class diagrams that can show classes and objects that exist at a specific point in time. Fig.2.2(a)

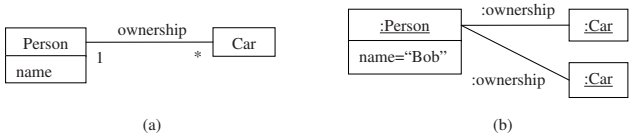

Figure 2.2: A model (a) and a model instance (b) of this model

shows class `Person` associated with class `Car` by the `ownership` association; the `Person` has the property (attribute) `name`. Fig.2.2(b) shows an object that is described by class `Person` and two objects of class `Car`, also two instances (*links*) of the `ownership` association are shown. To summarize:

- An object (or more generally, an instance) is instantiated from a class.
- A *link* is instantiated from an association.
- A *slot*, which can keep a value, is instantiated from a property defined by the class (e.g., `name="Bob"` shown inside the `Person`-object in Fig.2.2(b)).

There are several other diagram types in UML, e.g., sequence diagrams showing the interaction between objects.

We have seen that objects are instances of classes, but what about classes: Are they instances of something? We return to this question later when discussing metamodeling.

2.3 Instantiation

In [RJB05] instantiation is described as *the creation of new instances of model elements*. The instances are considered to be created at run time; they can be modeled by using `InstanceSpecification` [OMG03g] which is a description in a model of an instance or group of instances [RJB05]. The instances are the result of primitive *create action(s)* or *creation operation(s)*. The creation process can be seen as composed of several stages: First an identity is given to the instance; then its data structure is allocated as prescribed by the *descriptor*; and then its property values are initialized as prescribed by the descriptor and the *creation operator*. James Rumbaugh et al. also state [RJB05]:

> Usually, each concrete class has one or more class-scope (static) constructor operations, the purpose of which is to create new objects of the class. Underlying all the constructor operations is an implicit primitive operation that creates a new raw instance that is then initialized by the constructor operation... The exact mechanisms of creating instances are the responsibility of the runtime environment.

A creation operation seems to be the same as a *constructor*. The role of the creation operator is not clearly stated (this may have been done deliberately to have the definition fit several implementations) – is there one creation operator for each class or is there one for all classes? We return to such questions in the next chapter.

2.4 Token and Type Model Roles

Being a model is not an intrinsic property but a role played in relation to what is being modeled. Two basic roles are defined by Thomas Kühne [Küh05]: *token* and *type*. A token model captures the singular aspects, while a type model captures the universal aspects of what it models. A class `Building`

may capture the universal property that buildings have owners. An object that models one specific building is a token model for this building, e.g., it may capture the name of the owner.

Wolfgang Hesse [Hes06] gives the following description of token and type model roles:

> Roughly spoken, the two kinds of projection mentioned above are responsible for the two kinds of models: Feature projection for token models and placeholder[1] projection for type models. Feature projection means that single elements of the original remain distinguishable elements in the model - the "tokens" of a token model. On the other hand, placeholder projection always implies a (non-trivial) classification where class members on the original side are contracted to single classes on the model side to form the "types" of a type model.

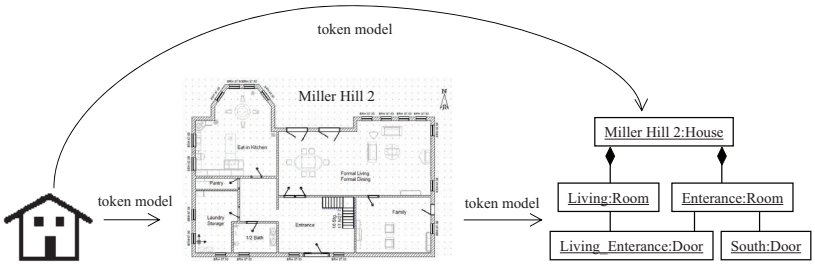

Figure 2.3: Token model examples, "real house" (Miller Hill 2) to the left.

A token model is seen as a projection of the subject expressed in some modeling language, while a type model involves classification in addition to projection and translation. In effect the elements of a token model designates the corresponding elements of the reduced system that appears after the projection. Fig. 2.3 shows an example; the model to the right is token model for both the "real house" and the model in the middle. It is not necessary to indicate explicitly that the model to the right is a token model for the real house, since token model relations are transitive, i.e., since the model in the middle is a token model for the real house and the model to the right

[1] Encyclopedia Britannica defines placeholder as a symbol in a mathematical or logical expression that may be replaced by the name of any element of a set.

is a token model for the model in the middle, then the model to the right must be a token model also for the real house. The model in the middle is not a token model for the model to the right because in this relation there are no reduction feature.

A type model describes the elements of a subject by assigning types to the properties of interest. Since a typed property typically indicates a set of possible values (except for "singleton types"), several elements of the subject may be described by the same type.

The notion of token model and type model gives the understanding that there are at least to relations between models – one may chose to see the type role as a "up/down type of" relation, while the token role may be seen as a "sidewise type of" relation.

2.5 Metalinguistics and Metamodeling

Metamodeling is about using models to define other models, this is closely related to *generative linguistics* [Cho57] where languages are used to define other languages. Generative linguistics has been around for a long time; *Ashtadhyayi* dates back to the 4th century BC and is the earliest known grammar of Sanskrit, and the earliest example of *generative linguistics*; the letters of the Sanskrit are called small mothers.

John Backus and Peter Naur were the first to introduce a formal notation to describe the syntax of a programming language; their notation got the name Backus Naur Form (BNF). BNF is a generative approach to linguistics, a specification is given by a set of production rules that can be used to derive grammatically correct expressions. The production rules may be recursive. Applying the production rules can be seen as building a directed three structure, while doing metamodeling is like building a graph – in other words you have more flexibility when doing metamodeling.

BNF is powerful enough to express BNF itself. We may view BNF as defining a layered architecture (see Fig. 2.4(a)); the top level[2] would be BNF defined in BNF (an instance of BNF); in this case; the next level would be the definition of a language done in BNF (an instance of BNF); the next level would be statements in the language defined by the level above; the lowest level would be runtime instances existing under execution!

In a metamodel architecture one level can be seen as a description where the concepts used to make the description are again described on the level above. In relation to object-oriented programming the lowest level contains

[2]Of historical reasons ("OMG history") this is called the top and not the lowest level!

the objects of a running system.[3] In relation to a running metamodeling tool all elements of the defined architecture would have corresponding run-time instances!

An often used metamodeling example is the legend found on a map, the legend can be seen as a model for the map since it describes the elements of the map (e.g., it describes how a road should be visualized) and again the map is a model for the terrain. Another example: *Personality type* playing the role of being a metamodel – some would call this an *ontological metamodel* and the example above a *linguistic metamodel* – the instances of personality type are specific personality types like the *melancholic* type, which again have persons being melancholic as instances.

The notion of modeling and metamodeling have been discussed in many papers; the work of Thomas Kühne together with Colin Atkinson has already been referenced; and, there are others [Lud03, Sei03b, Fav04b, Fav04c]. [Fav04a, Fav05, Béz05, JB06]. The mentioned papers makes it clear that there are not full agreement on all concepts, and a "stand" will not be taken in this work; however, the term *metamodel* is used in an "old fashion way" as can be seen in Fig. 1.1.

The UML metamodel architecture is an example of the four-layer metamodel architecture of OMG, Fig. 2.4(b) shows this architecture.

2.6 The UML Metamodel Architecture

Fig. 2.5 depicts a simplified example of the UML metamodel architecture; the "ordinary" modeler operates on what is called level `M1`, and this is where class `Person` is placed. The UML metamodel is found on level `M2`; class `Class` is part of the UML metamodel and defines what is meant by a class; class `Person` is instantiated from class `Class`.

Ivan Kurtev et al. [KBJV06] describes the relation between metamodel (`M2`), model (`M1`) and what is being modeled (called *system* in the citation):

> The extraction of elements from system S to build model M is guided by the metamodel MM and the purpose of the model. In other words, the metamodel MM acts as a filter that states which elements of the system can be selected to constitute the model M.

[3] This seems to be a common understanding, but what about for example Java class objects, where are they residing? Questions like that may raise some dispute, however, this is outside the scope of this dissertation.

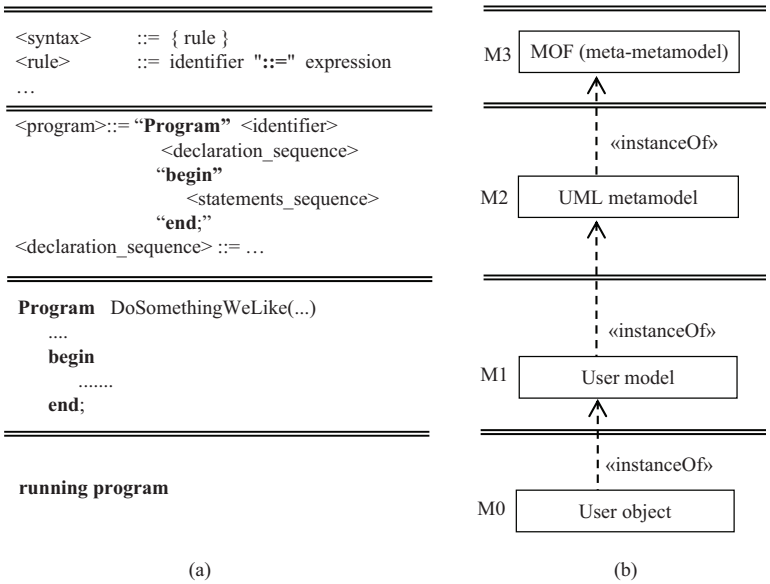

Figure 2.4: BNF example (a) and the UML metamodel architecture (b)

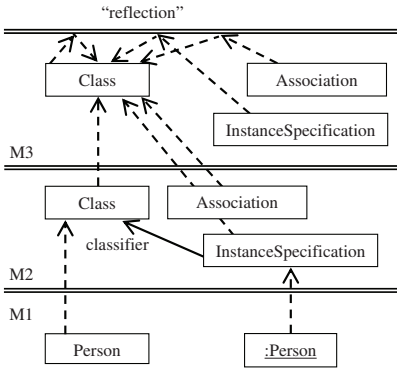

Figure 2.5: Simplified example of the UML metamodel architecture

In isolation each levels of Fig. 2.5 corresponds to proper UML, but UML does not support level borders and it does not support indication of the instance of relations that crosses the level borders between metamodel and model – so Fig. 2.5 is a sort of "pseudo UML".

Fig. 2.5 also shows a part of MOF which resides on top of the UML metamodel architecture (level `M3`). The core of UML defines the object-oriented concepts (e.g., class, association, etc.) and it is structurally equivalent to MOF. MOF is used to define itself – it is *reflexive* (self referencing) – in other words the concepts used at the level are the same concepts that are being defined, e.g., class `Association` is an instance of `Class`; this "self referencing" ends up with class `Class` which is an instance of itself as shown in Fig. 2.5. The top level of a metamodel stack is also typically *minimal reflexive* with regard to what it defines, this means that a minimum number of elements are used – if some elements are removed then some relevant information would be lost (and it may not be self-described any more).

MOF is designed to make metamodels, while UML is a complete software modeling language, e.g., a description of sequence diagrams is also part of the UML metamodel. However, MOF is important when it comes to integration of languages and is meant as a common source for a family of languages.

The four-layer metamodel architecture of OMG is inspired by the CASE Data Interchange Format (CDIF) [Com94] which is a standard (actually several standards) defining a metamodeling architecture for exchanging information between modeling tools. It describes the different modeling languages in terms of a fixed core model, in effect the core model function as a meta-metamodel. A tool that understands the core model can read the description of a specific modeling language, and understand any models written in that language.

The syntax of UML has been described in a notation independent way; this *abstract syntax* defines the elements of UML and how they relate to each other. The abstract syntax is an important part of the UML metamodel, it is typically described with UML class diagrams as can be seen in Fig. 2.5 (`M2`). There is also an agreement on a *visual concrete syntax* (notation) [OMG07b, OMG07c] and a *textual concrete syntax* [OMG04] for UML. The visual concrete syntax of a simple class is a rectangle with the class name inside the top compartment, optional compartments for attributes and operations, etc. Also an XML schema and DTD is defined so that XML can be used as concrete syntax; the serial nature of XML makes this an example of a *serialization syntax*. This mapping to XML is named the XML Metadata Interchange (XMI) [OMG07a].

The *static semantics*, also called *well-formedness rules*, of a language

defines how an instance of a construct should be constrained and connected to other instances to be meaningful. The abstract syntax is a model for an infinite set of graph structures and the well-formedness rules limits this set. Thus, abstract syntax and the static semantics defines the (logical) structure of all possible models. The well-formedness rules of the UML metamodel is given as OCL [OMG03d] constraints related to the abstract syntax. OCL is a language for specifying first order predicate logical statements on graph structures. David Frankel consider the well-formedness rules as being a part of the abstract syntax [Fra03]:

> I, however, consider OCL that specifies constraints on a metamodel's elements to be part of the abstract syntax because it is expressed formally and thus can be used by a generator.

A more powerful visual language (having more "logical power") may reduce or remove the need for using OCL.

Dynamic semantics defines the meaning of a well-formed construct in relation to behavior; when it comes to the UML metamodel the dynamic semantics is given in natural language (English), e.g., some of the semantics for classes ([OMG07b]):

> The purpose of a class is to specify a classification of objects and to specify the features that characterize the structure and behavior of those objects. Objects of a class must contain values for each attribute that is a member of that class, in accordance with the characteristics of the attribute, for example its type and multiplicity.
> ...Operations of a class can be invoked on an object, given a particular set of substitutions for the parameters of the operation. An operation invocation may cause changes to the values of the attributes of that object....

The imperative statements of an object-oriented programming language may be dividend into three categories: The ones concerning creation of new objects which is called *instantiation semantics*, the ones concerning deletion of objects and the ones concerning changes of already existing objects without creating new objects (e.g., change the value of an integer variable). However, no matter what kinds of semantics being involved, when "the structure worked on" is at rest it must conform to the abstract syntax.

The semantics given for UML is "object-oriented semantics", this semantics is not given by MOF but it is attached to the UML metamodel.

Class	**Property**	**Association**	**OCL**
the class concept	the property concept	association concept	OCL as a concept (metamodel)
a specific class	a specific property	a specific association	OCL Formulas
an object of a class	a slot with value	a link between objects	a formula instantiated

Figure 2.6: Instantiation of central UML modeling elements

One may make a MOF metamodel for relational databases and add another semantics that is adequate in this situation. Instantiation of central UML concepts are shown in Fig. 2.6. The class concept (metaclass `Class`) is instantiated to a specific class, which again can be instantiated to an object. The property concept (metaclass `Property`) can be seen as part of the class concept since a specific property will be attached to a specific class and also a specific property get instantiated to a slot that is attached to an object of the class. The association concept (metaclass `Association`) is also interweaved with the class concept; classes of objects gets associated and a specific association can be instantiated to a link between objects.

The UML metamodel approach is "centralized" around the abstract syntax which defines the logical structure of models. Abstract syntax (or more correctly instances of the abstract syntax, also called *abstract graphs*) can be presented (represented) in many ways using different concrete syntaxes. In a modeling framework abstract syntax is in some "concrete way" described – the description is found in the state of the hardware of the machine. Concrete syntax can be understood as abstract syntax plus a mapping to a medium. It is also often the case that a modeler can exploit the medium and add extra information, e.g., one important class may be shown as bigger than the others if a visual medium is used.

The visual concrete syntaxes of UML can be used to describe all abstract syntax. But a typical UML modeling tool does not insists on having all user defined model elements displayed on UML diagrams – typically the

tool offers a tree view/browser of all elements of the model, this is then yet another concrete syntax.

Tony Clark et al. claims [CESW04] that all elements described above – visual concrete syntax, dynamic semantics, well-formedness rules and abstract syntax – are parts of a (linguistic) metamodel. Also James Rumbaugh et seems to adhere to this view in their definition of a metamodel [RJB05]:

> A model that defines the language for expressing other models; an instance of a metametamodel. The UML metamodel defines the structure of UML models. Metamodels usually contain metaclasses.

Others use the term metamodel in a more limited way and consider only abstract syntax together with well-formedness rules as being the metamodel.

The most common way of doing metamodeling is called *strict metamodeling* which means that all elements on one level are instantiated from the level directly above, and also each element of the lower level is an instance of exactly one element of the level above.

Loose meta-modeling allows the instance of relations to be used more freely when it comes to the levels, e.g., an instance on one level may have been instantiated from an element that resides not on the level above but from some level further up. Atkinson states [Atk97]:

> Although loose meta-modeling is appealing ... the boundaries between the levels rapidly break down and one ends up with a single "super" model with all elements logically combined at the same level... In a loose meta-modeling framework, therefore, the model levels essentially end up being nothing more than packages grouping logically related model elements.

The OMG approach for UML 2.x is based on strict metamodeling.

David Frankel asks "How meaningful are metalevels?"[4] and states [Fra03]:

> From a certain point of view, the metalevels are arbitrary...However, despite the fact that this is true in some theoretical sense, there are practical issues that make the designation of absolute metalevels helpful in certain contexts.

[4]The term metalevel is used in a "liberal way", the new level in question may not be a metamodel, it may only be an ordinary class model, e.g., one describing the concept Person.

An example, in regard to the mentioned practical issues, may be to use the same language when defining several DSLs and in this way allow some of the same tools to be applied to the different DSLs.

In the MOF specification [OMG03b] the number of metalevels is discussed:

> One of the sources of confusion in the OMG suite of standards is the perceived rigidness of a "Four layered metamodel architecture" which is referred to in various OMG specifications. Note that key modeling concepts are Classifier and Instance or Class and Object, and the ability to navigate from an instance to its metaobject (its classifier). This fundamental concept can be used to handle any number of layers (sometimes referred to as metalevels). The MOF 1.4 Reflection interfaces allow traversal across any number of metalayers recursively....(The minimum number of layers is two so we can represent and navigate from a class to its instance and vice versa). Suffice it to say MOF 2.0 with its reflection model can be used with as few as 2 levels and as many levels as users define.

2.7 Representing Metamodel Levels

MOFs originally purpose was to provide a standard way of accessing runtime meta-information about objects within a system (CORBA-compliant distributed systems [OMG08]) via standardized interfaces (reflection interfaces) [AK02], and hence there are a number of tools that can automatically produce metadata management software for MOF (meta-)models.

A metadata repository based on MOF is often just called a MOF repository. Given a MOF metamodel, then an XML format for models conforming to it can automatically be produced. It is the OMG standard XMI that specifies how the generation should be performed; the process is applicable for any metadata whose metamodel can be expressed in MOF. Since MOF is an instance of itself this process applies also to MOF metamodels themselves, and even MOF itself! This implies that a MOF repository can export/import – in the form of XML documents – any UML models, the UML metamodel and MOF itself.

A Java interface is defined in the same way as the XML interface; the Java Metadata Interface (JMI) [Jav02] defines the creation, access, lookup and exchange of metadata in the Java programming language for MOF repositories.

MOF is also supporting a generic reflective interface called the MOF Reflective Interface that gives the possibilities to access a MOF repository in a generic way, e.g., by using operation `getMetaClass():Class` to access an objects class (all objects will have this operation). An essential issue when defining a modeling framework is how to represent the modeling elements. However, MOF does not specify the internal representation of metadata, but MOF and also UML defines an *instance model* that can be used for storing model elements in a model repository.

Class `InstanceSpecification` is the core class of the instance model and it is used when showing objects on diagrams (e.g., `:Person` in Fig. 2.5). Fig. 2.7(a) shows a part of the instance model; Fig. 2.7(b) shows a class and a snapshot of an object of the class, the object is shown in the concrete syntax defined for objects. Since also classes are instances (instances of class `Class`) they can also be visualized as objects as shown in Fig. 2.7(c); the same figure is also showing how the object is instantiated from the abstract syntax.

As understood from Fig. 2.7, an instance of the UML metamodel can be shown as an object diagram or as a class diagram; the class diagram being a visual interpretation (i.e., concrete syntax) of the underlying object-graph. In a sense the class diagram notation is using "syntactic sugar", e.g., an attribute is shown inside a compartment and not as an instance of `Property`. When "seeing the level from below" then a class diagram is a natural choice for a human knowing the object-oriented paradigm; the concrete syntax of class is indicating that it is a class and being a class implies instantiation semantics, in other words a class diagram makes it clear what can be instantiated. An object diagram on the other hand shows how a level has been instantiated from the level above. If an object diagram is used when displaying classes then the instantiation semantics is not explicit – we may "find it" by checking the type names, that is if we know the semantics of the types, e.g., if the type is `Class` we may understand that the object is representing a class. Instantiation semantics is in practise some kind of functionality that in some way has to be defined and attached to the structure – neither a class diagram or an object diagram are showing the "details" of this.

We may see from this discussion that the instantiation semantics of metaclass `Class` is different from that of an ordinary class – the instantiation semantics of `Class` gives two levels beneath while an ordinary class gives one. However, structurally it is all about classifier and instance when looking at two neighboring levels, but in the case of metaclass `Class` the objects on the next level get the instantiation semantics of classes – in some way this is

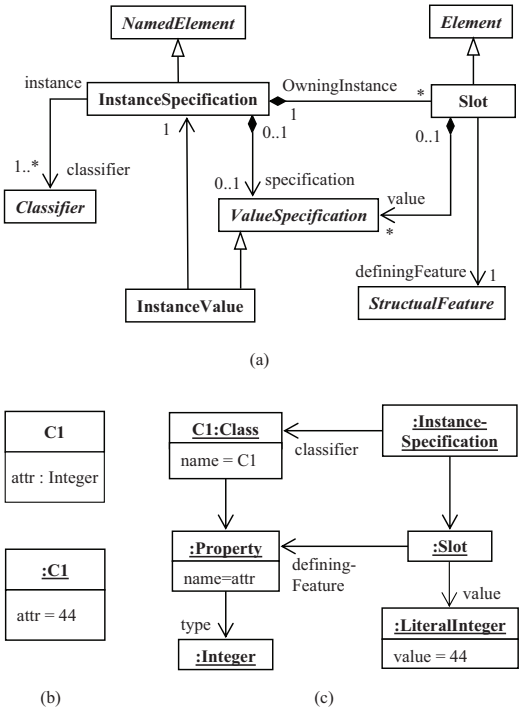

Figure 2.7: Part of the instance model of MOF and UML

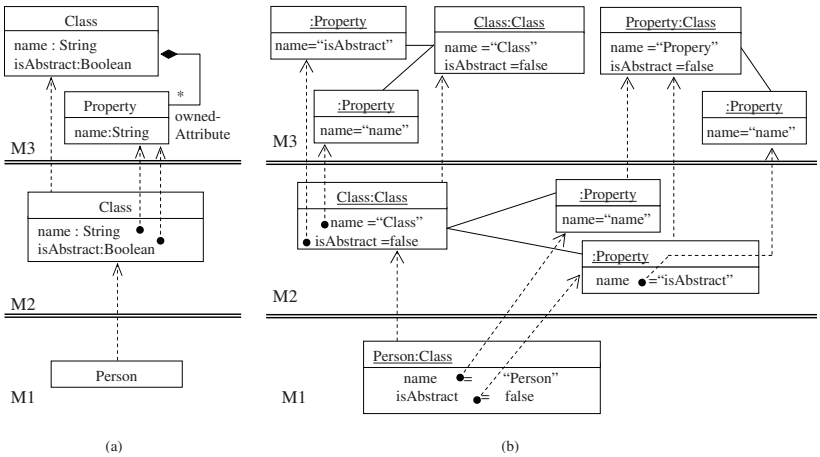

Figure 2.8: Representing levels with class (a) and object notation (b)

attached to these (class) objects (other metaclasses have other instantiation semantics).

A part of MOF (abstract syntax) is shown in Fig. 2.8(a) on level M3, it deals with the structure of classes and states that a class can own attributes (properties). The figure shows how abstract syntax (MOF) has been instantiated to form the (meta-)class `Class` of the UML metamodel, i.e., `Class` on level `M2` is an instance of class `Class` of level `M3` and the attributes are instances of class `Property`. Similarly, the class `Person` on level `M1` is an instance of `Class` on level `M2`.

Fig. 2.8(b) is using the object notation to show the same as Fig. 2.8(a); all attributes are shown as separate `Property` objects being linked to the class they belong to. Fig. 2.8(b) is not complete – to make it complete we would have to insert a definition of the association between `Class` and `Property` on level `M3`[5].

Atkinson and Kühne calls an entity with class and object nature a *clabject* [AK02, AK00b], in fact all classes are clabjects. Fig. 2.9 shows the same as Fig. 2.8 with *clabjects*. The clabject-notation allows you to see the attributes and the slots of a class, these can be shown in separate compart-

[5]The definition would be like: `Class:Class` (already on `M3`) linked to `:Property` linked to `:Association` linked to `:Property` linked to `Property:Class` (already on `M3`). Both the links on `M2` could now have an instance of relation to `:Association` on level `M3`.

ments as done in the figure. This approach [AK02, AK00b, AK00a, AK05] is implying a level-independent model representation which is also proposed in Appendix C.

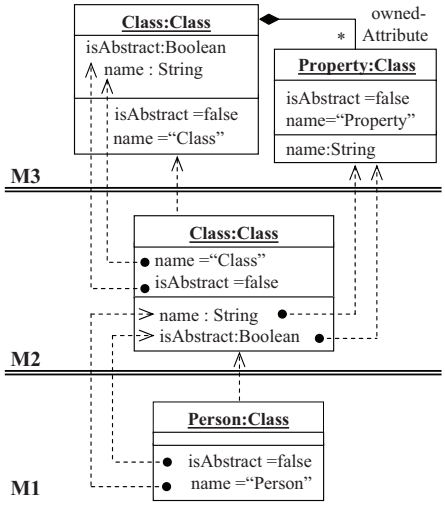

Figure 2.9: Same as Fig. 2.8 in clabject notation.

From the presentation given above we understand that any level can structurally be represented as objects; if there are no instantiation semantics attached, then the objects are terminal objects; if instantiation semantics that only gives one more level is attached, then (some of) the objects are classifiers; if instantiation semantics that gives two more levels is attached, then (some of) the objects are “metaclassifiers”.

2.8 Linguistic and Ontological Instantiation

Kühne [Küh06] identifies two types of instantiation: *Linguistic* and *ontological*, the two types of instantiation can correspondingly raise two different types of metamodel stacks. Hesse [Hes06] approaches the question of ontological versus linguistic classification[6] by using quotation marks to signify if one is talking about the word in itself or its referent. In the example *'Lassie'*

[6]The distinction in question is in metalogic named the *use-mention distinction*.

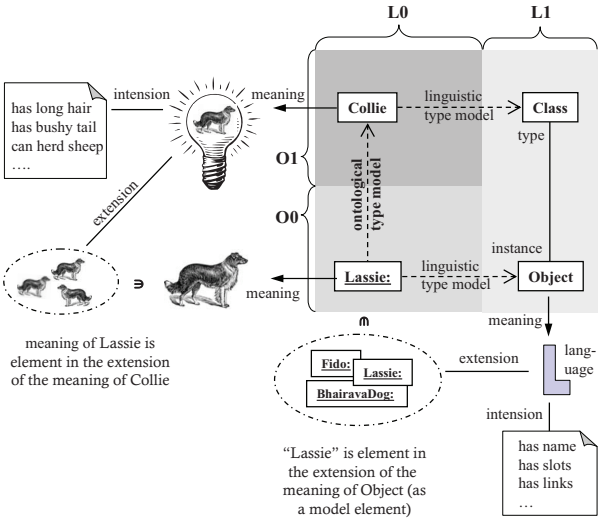

Figure 2.10: Ontological versus linguistic instantiation (partly from [Küh06])

is a name – the word Lassie is linguistically classified as a name; in the next example *Lassie is an animal* – Lassie, the dog, is ontologically described as an animal.

Linguistical models are about the language that is being used, e.g., in Fig. 2.10 `Class` and `Object`, given to the right, define the language (a portion of UML) used to define the token model (`Lassie:`) and the type model (`Collie`). Kühne presents [Küh06] a figure that resembles Fig. 2.10 (a special notation is used [Küh06] by Kühne, this notion is omitted in Fig. 2.10 and instead English is used). Ontological instantiation is demonstrated in Fig. 2.10 where Lassie is seen as an ontological instance of Collie.

The meaning of `Lassie:` is a particular dog called Lassie; the meaning of `Collie`, is the concept of Collie (which refers to various breeds of herding dogs). `Lassie:` and `Collie` relates in the following way: The meaning of `Lassie:` is an element in the extension of the meaning of `Collie`; alternatively this can be expressed as: The intention of the meaning of Collie applies to the meaning of `Lassie:`.

Ontological instantiation relates to the meaning (content) of the involved elements; in general it is not possible to decide automatically what the

meaning of an element is, e.g., to know the meaning of the word Lassie one has to know the dog Lassie and do an interpretation of the word.

Linguistic instantiation is about form and not the content, i.e., it is about how to use the medium (the language) to represent what the modeler wants to model. The language is however representing a world view and one may say that a language possesses an "ontological bias", e.g., in Fig. 2.10 the language foresees that there are classes and objects. In Fig. 2.10 `Lassie:` is seen as an linguistic instance of model element `Object` or alternatively `Lassie:` is seen as an example of what is meant by model element `Object` (entities with name, slots and links).

As we can see from Fig. 2.10, none of the linguistic types are referencing the concepts of the domain and in general (one exception is if the domain is the language used) a linguistic model is not a model of its subject model's subject, e.g., model element `Object` is a model for `Lassie:` but not for Lassie the dog (`Object` is in this context a model element for "object models" and it does not completely correspond to the "general" concept of an object). Relating this to media, `Object` is about how to form the medium (linguistics), while `Lassie:` is about the dog Lassie (content/ontology).

2.9 Linear and Non-Linear Metamodel Hierarchies

With respect to linguistic instantiation two levels are shown in Fig. 2.10: `L0` and `L1` (indications shown at the top). `Class` and `Object` reside at the same linguistic level (`L1`) and both are (linguistically) instantiated to establish the next linguistic level (`L0`). At linguistic level `L0` two ontological levels are presented namely `O0` and `O1`; we understand from this that at one and the same linguistic level there may be many ontological levels; a third level would be established if `Breed` was introduced as an ontological metaclass playing the role as ontological type model for `Collie`; however, linguistic level `L1` does not support this, i.e., `Breed` should be a metaclass and `L1` only support classes and objects.

The linguistic instance of is considered to be an *inter-level* relation, while the ontological instance of is seen as an *intra-level* relation. This is somewhat different from the view that UML 1.x advocates, where a typical presentation of the metamodel architecture (stack) includes: An object at `M0` (e.g., object `b1:Building` representing a specific building), a class at `M1` which the object is an ontological instance of (e.g., class `Building`), class `Class` at `M2` (from the UML metamodel) which the just mentioned class is a linguistic instance of, and may be on top (`M3`), class `Class` from MOF.

Such a stack mixes the linguistic and ontological instance of relations and ignores the differences between the two types of instantiation. In UML 1.x a metaclass called `Instance` (with subclass `Object`) was defined and meant to be used as a descriptor of runtime instances, but this was problematic in relation to the strict metamodeling constrain. The introduction of `InstanceSpecification` in UML 2.x marks a change in the way one sees level `M0` – an interpretation of this change is that `M0` is not considered to be part of the *model stack* any more [BG01, Küh06].

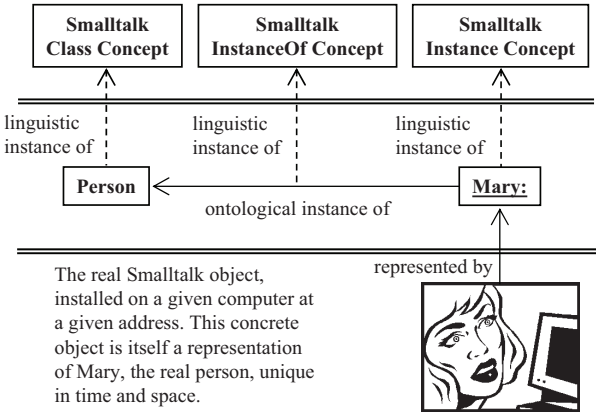

Figure 2.11: Smalltalk example [BG01]

An example demonstrating the new view is given by Jean Bézivin and Olivier Gerbé [BG01], the example is quite similar to what is shown in Fig. 2.11. The figure depicts a metamodel level describing the Smalltalk language, only *class*, *instance* and *instance of* are included in the figure. The next level is the model level showing a model of Smalltalk class `Person` together with a model of a Smalltalk object called `Mary`. The bottom level (`M0`) contains "real" Smalltalk objects installed on a given computer at a given address. The `StkInstOf` (Smalltalk instance Of) relation is analogous to the relation between an instance of `InstanceSpecification` and "its classifier" in UML 2.x, and it correspond to the `ontological instance of` relation. The ontological instance of has been given many different names, e.g., it has been called *logical instance of* [AK05]. The linguistic instance of is usually just called *instance of* when used in relation to UML, but also *conformant to* [Béz04] (or *comforms to*) has been used to name this relation.

The two types of metamodel hierarchies being discussed are called *linear hierarchy* and *non-linear hierarchy*. Ralf Gitzel et al. states [GOS07]:

> In a naive approach, the elements in the highest layers of a model hierarchy would connect to their instances via physical relationships, defining a modeling language, whereas the lower layers define the logical relationships within the domain. This approach is in effect a linear hierarchy [AK02]...

As we have seen, in a non-linear metamodel hierarchy there will be two linguistic levels; one level (`L2`) defining a language (`L`) and a second level (`L1`) is built by instantiating this language. The language may contain an explicit definition of an instance of relation which may be used to build a complete metamodel hierarchy inside `L1`. Levels nested inside `L1` are considered to be ontological. The following statement is also given by Gitzel et al. [GOS07]:

> One important difference to the linear approach is that the physical instance of between L1 and L2 and the logical `InstanceOf` relationship between O2 and O1 (authors comment: O1 and O2 are nested inside L1) now differ significantly in nature. The physical instance of relation is still implicit... Its instantiation semantics is that of the metamodeling language used (most likely MOF). The logical `InstanceOf` relationship, however, is established by an explicit `Association`... The instantiation semantics associated with `InstanceOf` can be defined suitably by constraints in L2.

The notion of being a linear hierarchy implies strict metamodeling; also a none-linear hierarchy may be based on strict metamodeling in regard to the ontological levels, but if seeing the linguistic level (L2) together with several ontological levels then strict metamodeling is not honored as it is in a linear hierarchy. Another difference between the two approaches, as stated above, is the explicit instantiation semantics for the non-linear approach as opposed to the implicit instantiation semantics for the linear approaches, e.g., in Fig. 2.12(a) the instance of relation between `Building` and `:Building` has not been explicitly modeled in the modeling language.

Instantiation semantics (the code for creating instances), if assuming a joined model and runtime environment, may be attached in different ways; Fig. 2.13 shows how instantiation semantics may be attached in a none-linear hierarchy, and Fig. 2.12(a) and Fig. 2.12(b) shows how instantiation

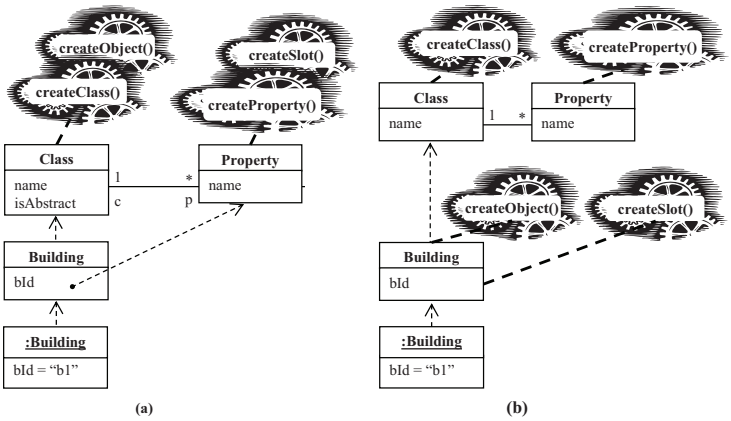

Figure 2.12: Attaching instantiation semantics in a linear hierarchy

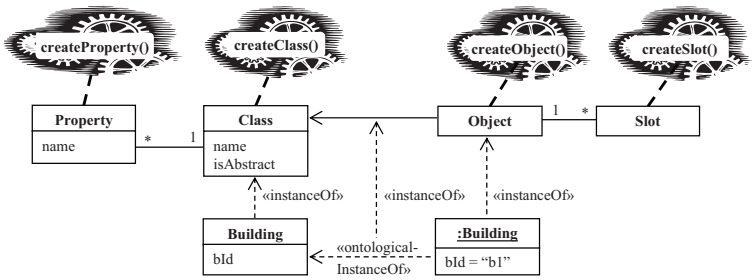

Figure 2.13: Attached instantiation semantics in a none-linear hierarchy

semantics may be attached in a linear hierarchy. The instantiation semantics attached to `Building` in Fig. 2.12(b) may have been attached by the modeling framework when class `Building` was created; alternatively the `createClass()` (i.e., the code for creating a class) may have attached it when it was called, i.e., `createClass()` may have `createObject()` available as depicted in Fig. 2.12(a)).

Fig. 2.14, which is a slightly modified version of a figure found in the dissertation of Kurtev [Kur05], gives a general model for linguistic/ontological instantiation. The metamodel is seen as consisting of two models; the first model describes intentions (Intention Model in Fig. 2.14), e.g., the general structure of classes; the second model (Extension Model in Fig. 2.14) describes elements of extensions of intents that conforms to the first model, e.g., the structure of an instance of a class. The model level consists of two parts constituting two ontological levels: The intentional part describes intent (e.g., a class `Person`) and the other part describes elements of the extension of the intent (e.g., instances of class `Person`). Fig. 2.14 shows how the different parts fit together, e.g., how the intentional part plays the ontological type role for the extensional part and how the metamodel as a whole is an intention with the whole model as an element of the extension.

The dissertation [Kur05] of Kurtev is also containing a figure quite similar to Fig. 2.15 and Kurtev states the following about the figure:

> It should be stated that the outlined meta-modeling architecture in Fig. 2.15 (authors comment: Fig. 2.15 is an adapted version of the original figure) is somehow idealized. First, there is no commonly agreed approach for defining modeling languages. Second, not all the models shown in the figures are explicitly presented in the modeling stack. In many cases only models of intentional parts (e.g. abstract syntax definition) are defined. Third, some models may be reused across levels.

An interesting observation in Fig. 2.15 is relation *linguistic instantiation of** which identifies a structure that defines the basic structure of all elements of all levels. This model, the `Extension Model` of the meta-metamodel, is hardwired and allows tools to manage models in a uniform way, e.g., a MOF tool would store model elements as linked objects with slots according to the instance model of MOF. The model being discussed is often called the representational model (or embedding language) and in effect it function as a link to some underlying medium (language) which allows the models to be represented physically.

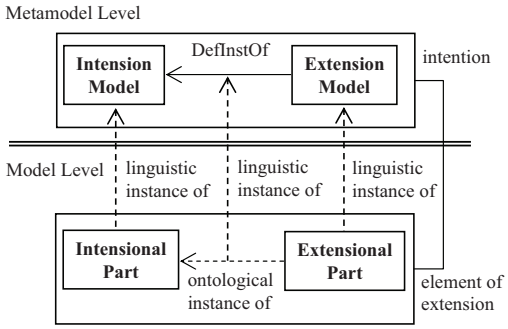

Figure 2.14: The intensional and extensional parts of a metamodel

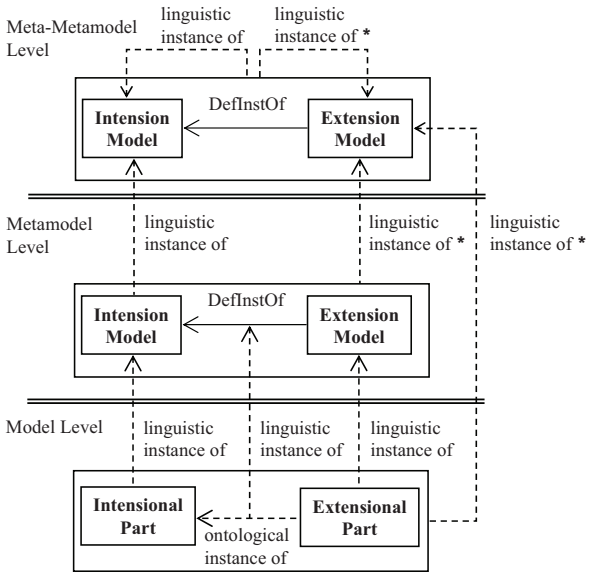

Figure 2.15: The intentional and extensional parts of a metamodel stack

Atkinson and Kühne presents [AK05] Fig. 2.16(a), this figure corresponds to Fig. 2.15 and shows how the full UML metamodel stack is *embedded* in MOF, i.e., MOF is the common representation format of all other levels (all elements are represented as instances of the instance model of MOF).

The ontological instantiation relationship from `M1i` to `M1t` is defined within `M2`. Atkinson et al. state [AK05] that `M2` *spans* both levels `M1t` and `M1i`, all elements from both levels must be well-formed with respect to the rules expressed in `M2`. Fig. 2.16(b) is also given by Atkinson et al. [AK05] and is meant to give a more complete interpretation of the OMG's four-layer architecture. As in Fig. 2.16(a), MOF is a repository format for all the levels and in this way all the levels are embedded by MOF. In Fig. 2.16(b) MOF is also *stacked* on top of the UML metamodel, this captures MOF's role as a logical language definition for the UML metamodel and it shows explicitly MOF's ability to represent itself.

In the next chapter we will see that some programming and modeling languages support not only two ontological levels but many.

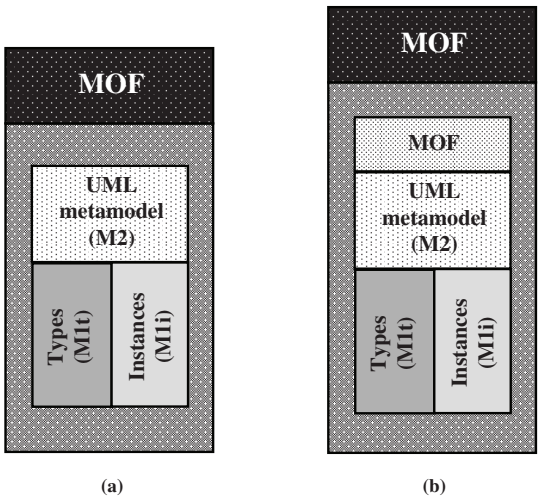

Figure 2.16: Embedding and spanning the UML metamodel stack

2.10 Multi-model Architecture

Gitzel and Hildenbrand defined in [GH05] a *metamodel hierarchy* to be:

> A metamodel hierarchy is a tree of models connected by instance of relations. The term model layer or model level describes all (meta)models with the same distance to the root metamodel in a metamodel hierarchy. Each level is given a unique name, often containing a number.

The term *metamodel stack* has been used in this dissertation, e.g., *a legacy database metamodel stack* which is composed of three models stacked on top of each other (data set, schema and metamodel). If assuming strict metamodeling, then a metamodel stack is defined by:

> A metamodel stack is a set of models that are connected in a linear way by instance of relations, there is exactly one model at each level and there are at least two models involved.

The term *multi-model architecture*, used in the title of this dissertation, is meant to denote two or more metamodel stacks that have at least one model containing "references" to elements of a model which is placed in one of the other metamodel stacks. The references in question are not given by the instance of relation. One may argue that if one model contains references into another model then the models are merged and have become one united model. There is a lot to this argument, but in this work the kind of references are not "into the structure" of the other model, instead identifiers (Ids) are used to achieve the referencing; one may say that the model containing the references does not "assume" anything about the structure of the other model, it only "assumes" that there are elements with so and so names; keeping the models apart works well when it comes to structure, but some behavior may involve knowledge about the structure of models residing in different metamodel stacks (generation of consistency data is an example of this). Chapter 4 will describe how several metamodel stacks are integrated.

Chapter 3

Related Work

Looking back on the list of requirements specified in Section 1.1, Req. 1 states the need for a language for consistency modeling – related work concerning this requirement seems limited, there are not many modeling languages specifically designed to do consistency modeling, but some potential approaches are presented in Section 3.2 and some background material is also included in the papers found in the appendixes (e.g., Section A.3.4, Section B.1 and Section G.3).

Section 3.1 addresses Req. 3, which concerns the need for representing a multi-model architecture. Section 3.1 is also addressing the second part of Req. 2, which states the need for a way to establish the different models in the multi-model architecture.

Req. 4 is more indirectly related to existing solutions, e.g., when a UML virtual machine [RFBLO01] is presented in Section 3.1.3. Req. 5 is considered to be out of scope of this dissertation.

3.1 Instantiation in Some Metamodel Architectures

Several metamodeling frameworks have been described in the papers of this thesis – especially the paper found in Appendix F which presents and compares several approaches. This section is also presenting selected metamodeling approaches, however, these presentations are more focused and detailed on instantiation and on how to organize and represent levels in a metamodel stack. The first part of this section presents how metadata is organized in some programming languages and the following subsections does the same for some modeling approaches. The final subsection discuss how these approaches relates to the solution (IBe) presented in this work.

3.1.1 Metalevels in Programming Languages

The runtime system of Java [LY99] and of similar languages, maintains type identification on all its objects. Three levels are recognized in the Java runtime system (there may be other ways of seing the organization and it is not explicitly stated in the documentation of Java), where the most prominent metaclass of the top level is called `Class`. The object representing this class is called a *class object* and it is seen as an instance of itself (this is established through some bootstrap process). Metaclasses like `Field` and `Method` are also found on the top level. All user defined classes and interfaces will be instances of class `Class` and appear as class objects (usually just called classes) constituting the next level. The top level is frozen, only the two lowest levels can be managed by the user. It is not class `Class` that (solely) creates the class objects – it does not have a public constructor – class objects are created by the Java Virtual Machine when they are loaded. The lowest level is composed of objects instantiated from user classes, e.g., by a call to the `newInstance()` method of the class object representing the right class. All objects on the two upper levels are instances of `Class` and all elements on all levels will be instances of class `Object` (the root of all classes in Java) and consequently the `Object.getClass()` method can be invoked to get metadata on all objects. Fig. 3.1 (a) shows a Java example in the form of a UML class diagram; metaclass `Class` is a *Java generic class*.

The Java virtual machine knows only ordinary classes and there are no objects of generic types; Fig. 3.1 (b) shows runtime objects involved in representing the metadata described in Fig. 3.1 (a) and how they are linked; a specific `Building` object is represented at the bottom, a `getClass`-call on this object returns class object `Building:Class`, a `getClass`-call on `Building:Class` returns `Class:Class` which is an instance of itself.

Fig. 3.1 (c) applies the notion of linguistic and ontological instantiation; the top level defines what is meant by a Java class as it contains class `Class`; this metaclass is seen as an linguistic instance of itself. The top linguistic level spans the next level which contains two ontological levels: User defined classes and instances of these. Class `Object` can be understood as an extension model – the Java concept of being an object – instantiated to create all elements on all levels. The fact that class `Object` is used on both levels (it is the same class shown twice in the figure) may be confusing, but this is how the same extension model applies to all levels.

The Java virtual machine uses the type information when selecting which method to execute on an object. This is so because the instance methods are connected to the class objects and in this way shared among the objects

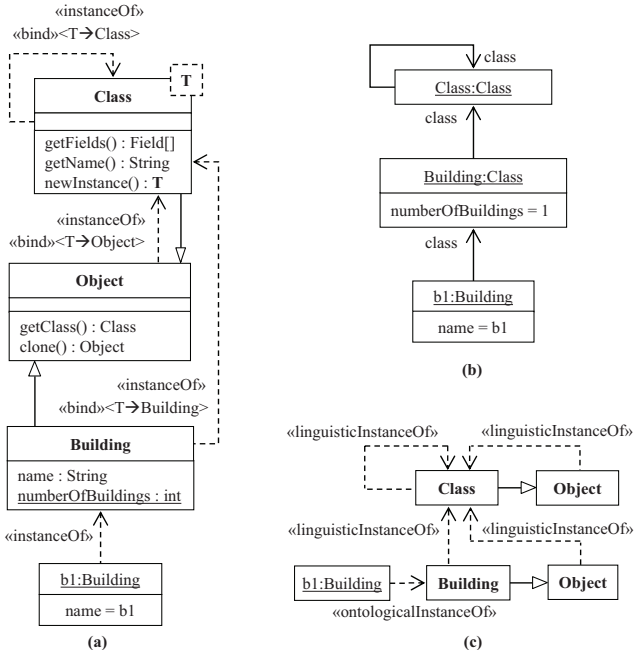

Figure 3.1: Metadata structure for Java objects

of the same class. The metadata of an object can be accessed at runtime through a reflective API (e.g., `getClass()` mentioned above) supplied by the Java language; the API can be used to reflect the structure of classes, dynamically create new objects of a reflected class, access and change the states of objects. It is not possible to dynamically change the structure of old classes, and making new classes dynamically is typically cumbersome (this is more easy in other "more interpretative" languages, e.g., Scheme [ASS96]).

An interesting observation concerns the static declared field `numberOf-Buildings`, as all class variables in Java, it is described and instantiated at the same level. If we look at the UML 2.1 metamodel [OMG07c] we see a similar approach: Class `Feature` is associated to classifiers, superclass to both `Property` and `Operation`, it contains a Boolean attribute called `isStatic` (this variable is not found in the infrastructure specification [OMG07b] only in the superstructure specification [OMG07c]) which specifies whether the

feature characterizes individual instances classified by the classifier (false) or the classifier itself (true).

While the first programming language with classes was SIMULA 67 [DMN70], the first language having classes as objects was Smalltalk-76 [Ing78], here a class was considered an object of a metaclass. The idea of having: metaclasses, classes and objects has been named the *Golden Braid* after being discussed in [Hof80]; it has been used in some Lisp systems like Obj-VLisp [BC87] and CLOS [Kic91]. Fig. 3.2 shows an example of a metamodel

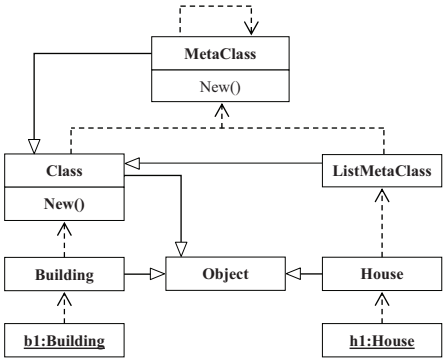

Figure 3.2: Metadata structure for Loops (dotted line is instance of)

architecture in Loops [BC87]. `MetaClass` holds the `New` method which is used to instantiate all metaclasses (logically this includes itself). `Class` is the default metaclass for all classes, its `New` method is used to instantiate classes. Class `Object` is superclass of all types of classes (i.e., all elements in the architecture are objects) and class `Class` is superclass to all metaclasses. The metaclass `ListMetaClass` is user made, e.g., it may contain the description of a property `numberOfInstances` which will be instantiated to a slot in class `House` and used to count number of `House`-instances. A user defined metaclass must be a subclass of `Class`, inheriting the `New` method allowing it to be instantiated to new classes but not to metaclasses – consequently the number of levels is fixed.

Fig. 3.3 shows an example of metadata in Smalltalk-80 [GR83, Ree02], here all classes exists as objects and – unlike Java – each class has a metaclass containing the definitions of its class variables, e.g., `numberOfBuildings` is defined in `Building class` and is instantiated to a slot in `Building` (the

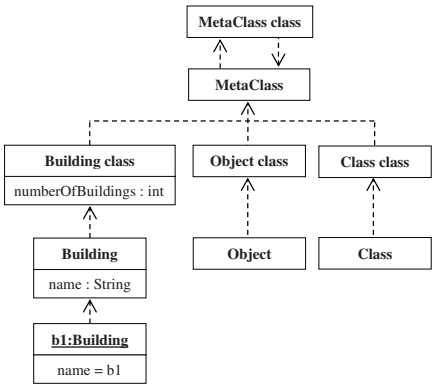

Figure 3.3: Metadata structure for Smalltalk-80 objects

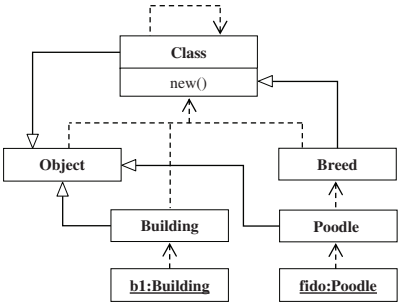

Figure 3.4: Metadata structure for ObjVLisp objects

slot is not shown). We notice that metaclass `Building class` can not exist without its, one and only, object `Building`.

The approaches described above allows only a fixed number of levels; another solution is proposed in [BC87, Coi87] which allows an unlimited number of levels. The model presented in [BC87, Coi87] is implemented in ObjVLisp, here classes and metaclasses are unified giving the user *uniform access and control to all the levels of the language.* A metamodel architecture in ObjVLisp is built by combining instantiation and inheritance, `Object` defines all elements as objects and `Class` makes it possible to erect a multi-level architecture. Both `Object` and `Class` are available for subclassing on several levels.

Fig. 3.4 shows a possible metadata architecture in ObjVLisp; in such an architecture all elements are objects, instantiated from some subclass of `Object` (`Object` is the universal *supertype*); the number of levels is up to the programmer. Metaclass `Class` has the `new()`-method and is the root of the instantiation hierarchy. The `new()`-method is different from "ordinary" methods since it involves three levels; an ordinary method is defined at one level and executed at the next lower level where also the effect of the execution appears; the `new()`-method is defined at one level, executed on the next lower level and the result – which is a new instance – appears at the level below the level where `new()` is executed (it is the same in Java where the `newInstance()`-method is defined in metaclass `Class`). Fig. 3.4 shows class `Building` which is an instance of `Class`; since `Class` is an instance of itself the instance of relation going from `Class` (at the very top of Fig. 3.4) can be imagined as going up one level to a copy of `Class` and it is the definition of `new()` found in this "copy" of `Class` that is used when `Building` is made. Object `b1:Building` is created by using the definition of `new()` found in class `Class`; `Building` do not have a definition of `new()` so `b1:Building` is a terminal instance (it can not be instantiated). `Breed` is created in the same way as `Building`, but `Breed` is a subclass of `Class` inheriting the definition of `new()` which makes it a metaclass and as a metaclass two or more levels may be spanned beneath it (two levels are shown in the figure). Class Object can be seen as the extension model defining what is meant by being an object; the instance of relation to class `Class` is linguistic instantiation, while the others are ontological instantiations.

3.1.2 The Type Object Pattern

The previous section described metamodeling architectures as they come forth through the intrinsic instantiation mechanism of some selected programming languages. This section describes how metadata can be represented on top of the intrinsic instantiation mechanism of the programming language in use by having the application implement its own type system.

The Type Object Pattern is described in [JW97,MO95]. The purpose of the pattern is to: *Decouple instances from their classes so that those classes can be implemented as instances of a class. Type Object allows new "classes" to be created dynamically at runtime.*
The Type Object Pattern is also called the Item Description Pattern [Coa92]

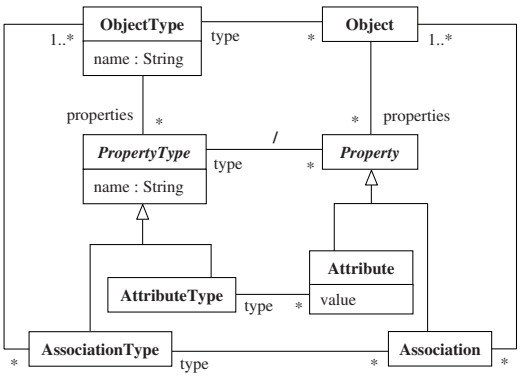

Figure 3.5: Extended version of the Type Object Pattern [HH]

where an item description plays the role as type for some items.

Fig. 3.5 shows an extended version of the Type Object Pattern [HH, YJ07]; the pattern is extended to handle properties and associations.
From Fig. 3.5 we understand that a type is represented as an instance of `ObjectType` and the attributes of the type are described by instances of `AttributeType`. An object is represented as an instance of class `Object` and the ontological instance of is represented as a link to the instance of `ObjectType` that represents its type. The Type Object Pattern is also used in UML when the UML instance model is defined [OMG03g].

3.1.3 A UML Virtual Machine

Dirk Riehle et al. presented in paper [RFBLO01] the architecture of a UML virtual machine. The virtual machine interprets UML models while preserving the *causal connection* between model and model instances. Causal connection is defined as:

> A modeling level is causally connected with the next higher modeling level, if the lower level conforms to the higher level and if changes in the higher level lead to according changes in the lower level.

The approach [RFBLO01] integrates modeling and runtime environment, allowing users to incrementally define and explore models through model execution. Modeling of object behavior is done with UML state charts. The solution can be seen as an implementation of an extended version of the UML 1.X four-level metamodel architecture.

The architecture of the virtual machine is made up of two parts a *logical* and a *physical architecture*. The logical architecture describes how objects logically relate, e.g., the Type Object Pattern is used to connect two adjacent levels. The physical architecture is used to realize the logical architecture. Dirk Riehle et al. state in [RFBLO01]:

> The logical architecture defines how to achieve the causal connection property, and the physical architecture implements how to achieve this property. There can be different physical implementation architectures, driven by different needs.

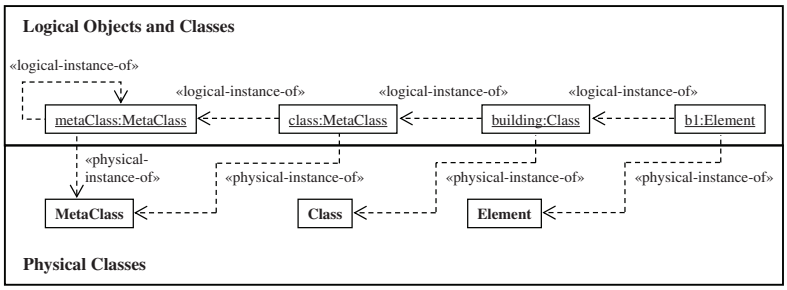

Figure 3.6: Logical and physical instance of relations as found in [RFBLO01]

Fig. 3.6 [RFBLO01] depicts the different types of instance of relations involved: The relation `logical-instance-of` between `building:Class` and `b1:Element` is in this case corresponding to ontological instance of; `logical-instance-of` corresponds to linguistic instance of the other places where it is used in Fig. 3.6. The `physical-instance-of` is given by the representational (implementation) language.

The implementation language for the virtual machine is Java and the classes `Element`, `Class` and `MetaClass` of Fig. 3.6 are Java class objects; these classes are called *physical classes*; the `physical-instance-of` is the Java intrinsic referencing of metadata, it is this relation that is used by the method `getClass()` defined by Java class `Object`. As can be seen in Fig. 3.6 all objects of the logical architecture (see top section of figure) have both a physical and a *logical class*, e.g., `b1` which is a `physical-instance-of` `Element` is also a `logical-instance-of` `building`.

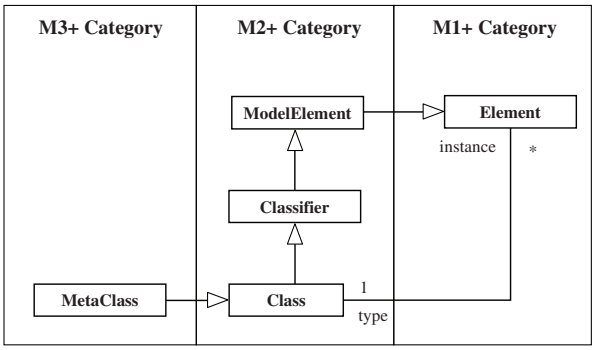

Figure 3.7: Key Java classes from physical architecture 3.6

Being a UML virtual machine, all the classes of the UML 1.X metamodel are found as part of the logical architecture and most of the classes are also found as Java classes being parts of the physical architecture. When a physical class exists, then its instances are used as instances of the corresponding logical class, e.g., logical class `building` is `physical-instance-of` class `Class` and a `logical-instance-of` `class`.

Fig. 3.7 shows some of the important classes of the physical architecture, other important classes would for instance be `Attribute` and `Association`. Class `Element` is the root of the hierarchy shown in Fig. 3.7 and it specifies the capabilities needed to be an object with slots and links. Class `Class`

adds capabilities needed to be a class, e.g., method `addFeature`() to add an attribute. Class `MetaClass` adds capabilities that are convenient for a metaclass to possess.

The UML Virtual Machine approach is using an extended version of the Type Object Pattern and all the four levels of the UML metamodel architecture are supported.

3.1.4 The Eclipse Modeling Framework (EMF)

The Eclipse platform [Dri01] is offering some basic components that allow additional software plug-ins to be configured into software solutions. Eclipse has with success been used to develop Integrated Development Environments (IDEs); the Java Development Toolkit is a prominent example of this. Eclipse is open source and its plug-in architecture has allowed several interesting plug-ins to be integrated.

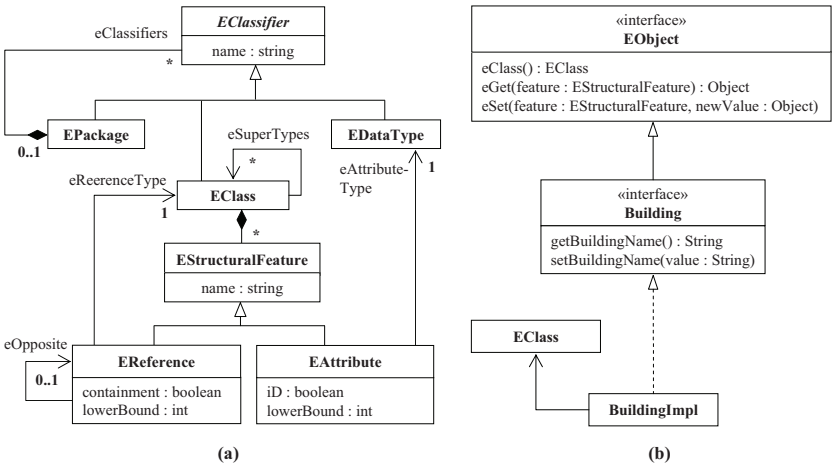

Figure 3.8: Part of the ECore metamodel (a) and generated implementation class (b)

An important part of this rich software ecosystem [Fra05b] is the Eclipse Modeling Framework (EMF) [BSM+04, Ecl04, BHJ+05] which is a metadata management framework that makes it possible to integrate disparate tools within the ecosystem. EMF is one of the most important platforms when

it comes to MDD and there is a large community of people involved. EMF is based on a metamodel called ECore which is compatible with Essential MOF (EMOF) [OMG03h]; an XMI file with a MOF model can be read and represented.

EMF lets you define a model as an instance of ECore; this can be done in different ways, e.g., XML Schema, annotated Java, UML model. When you have an ECore instance, then EMF can support you in different ways; below is a non-exhaustive list describing what can be produced automatically:

- Java interfaces for the classes defined by your model, Java implementation classes and a factory.
  The code produced supports change notification which allows integration with other EMF-based tools. The code also supports a reflective API for accessing metadata.

- XMI serialization and support for object persistence.

- Adapter classes that can be used for viewing and command-based editing of data of the ECore instance; based on the adapter classes, a working editor that allows you to create and manipulate data of an ECore instance.

There are many components that in some way works together with EMF, e.g., the Eclipse Graphical Modeling Framework (GMF) [GMF] allows you to define visual syntax for your EMF model and then automatically generate a visual environment with toolbar options and the possibility to draw models composed of graphical shapes. There is also an EMF-based implementation of the UML 2.1 [Ecl07] metamodel for the Eclipse platform meant to support the development of modeling tools.

Fig. 3.8(a) shows a part of the ECore metamodel; `EClass` is used to model a class, `EAttribute` is used to model class attributes and `EReference` instances are used to model associations. All generated classes will implement the `EObject` interface (Fig. 3.8(b)), which provides the reflective API.

An example will demonstrate how EMF can be used to represent models. Assume an ECore instance (a model) containing class `Building`; the automatic code generation produces a Java implementation class called `BuildingImpl` which implements the generated interface `Building` and the interface `EObject` (Fig. 3.8(b)).

Fig. 3.9 describes the situation where a `Building` object with name `b1` has been created. The object, `Class:Class`, at the upper right corner is the Java metaclass `Class` which is an instance of itself. Object

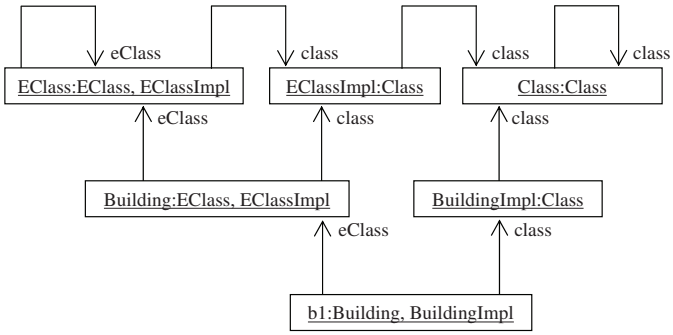

Figure 3.9: UML object diagram describing the building example

`BuildingImpl:Class` is the automatically generated Java implementation class which in Fig. 3.9 is represented as a Java class object. `BuildingImpl:Class` can be instantiated to get an object that will represent a specific building, e.g., `b1:Building,BuildingImpl`. Object `b1` contains a link to object `Building:EClass,EClassImpl` which represent the EMF meta-information about class `Building`; this is why `Building` is included as classifier for `b1` (additionally object `b1` have the automatically produced interface `Building` as classifier since class `BuildingImpl` implements this interface). Fig. 3.9 is a simplification, e.g., the `eClass`-link from `b1:Building,BuildingImpl` is actually represented once for all `Building` objects by the package that contains the model, logically this could be approximated as a link from `BuildingImpl:Class` to `Building:EClass,EClassImpl`.

EMF is embedded in Java since Java is the platform that EMF is built on. One may be confused investigating EMF since two languages are used simultaneously, e.g., there are two objects both representing a building class. The "basic representation format" (extension model) is Java – on top of this, ECore defines `EObject` which can be seen as yet another extension model. The Java metadata is on Fig. 3.9 found by following the `class`-Links; by repeatedly following the `class`-Links the Java metaclass `Class` will be found on top of the stack. The EMF metadata is found by following the `eClass`-Links and here `EClass` is found on top of the stack.

Object `b1:Building,BuildingImpl` is an ontological instance of class `Building:EClass,EClassImpl` and of class `BuildingImpl:Class`. `b1:Building,BuildingImpl` is a linguistic instance of `EObject` and of the "Java instance concept".

Class `Building:EClass,EClassImpl` is not an ontological instance of any class. However, linguistically it can be seen as an instance of `EClass`, `EClassImpl` and of the "Java instance concept".
Class `EClassImpl:Class` is not an ontological instance of any class; linguistically it can be seen as an instance of `Class` (the "Java class concept").
Class `EClass:EClass,EClassImpl` is a linguistic instance of the "Java instance concept".

The example we have seen in this section involves two (ontological) user defined levels: a model level consisting of class `Building` and a model instance level consisting of a specific `Building`. We could define a model level which represents a metamodel (e.g., the UML metamodel), the model instance level would then be a model conforming to the metamodel (e.g., a UML model); still only two user defined levels are supported. We understand from this discussion that EMF is an example of what [AK05] calls *a two-level approach*. First an instance of ECore is defined (ECore and the ECore instance constitutes the first level pair); then Java code is generated (Java source code corresponding to the ECore instance) and this code can then be instantiated (Java generated classes and their instances constitute the next level pair). The model may in some way be "uplifted" so that it becomes an ECore instance and hence, a model instance may be instantiated and in this way a new level is manifested (this would in effect give three user levels).

3.1.5 Java Metadata Interface (JMI)

JMI is a specification [Jav02] resulting from a Java Community Process; it defines how to manage MOF 1.4 metadata in the Java programming language; management includes creation, access, storage, lookup and exchange of metadata.

There are several metamodeling repositories based on JMI, including a reference implementation from Unisys [JMI] and Sun's open-source implementation called Metadata Repository (MDR) [Mat].

JMI builds on the same principles as EMF – it represent an alternative Java language based approach to metadata management based on MOF 1.4. The reflective programming capability of JMI is an implementation of the MOF 1.4, while EMF has its own.

A tool like MDR can import a model represented in XMI and automatically produce JMI interfaces for accessing the metadata and also automatically provide implementations of the JMI interfaces (the generated interfaces are implemented automatically as needed during the MDR runtime).

There is support for transforming a UML model to a MOF model, which again can be used as basis for code generation.

JMI does also have a generic reflection mechanism that enables browsing and discovery of metadata without specific knowledge of the generated interfaces and implementations.

3.1.6 Metamodel for Multiple Metalevels, MoMM

Deep instantiation is described by Atkinson and Kühne [AK01] as an alternative to the old *two-levels only modeling philosophy* which is classified as *shallow instantiation*. Colin Atkinson et al. state [AK01]:

> ...the traditional instantiation model is "shallow" precisely because the class facet of a model element always has to be explicitly documented for each model element that represents a type. In other words, a class can never receive attributes and associations from its classifier, only slots and links.

The approach [AK01] does not avoid explicit descriptions of class facets, on the contrary: The approach formalizes a way of attaching instantiation information at one level, such that this information can potentially dictate instantiation of lower levels.

Atkinson and Kühne introduce [AK01] the concept of *model element potency*; the potency is an integer which is attached to each model element at every level in a modeling framework and it defines the depth to which a model element can be instantiated. Potency 0 corresponds to a concept that is not intended for further instantiation, e.g., an object, a slot or an abstract class. Potency 1 corresponds to instantiation one time, e.g., class `Person` would have potency 1. Potency 2 corresponds to instantiation twice, e.g., a metaclass or a metaassociation. Potency 3 corresponds to instantiation three times and so on. Colin Atkinson and Kühne state [AK01]:

> Finally, the semantics of shallow instantiation could be captured by the constraint that the potency of elements cannot be greater than one.

UML defines three levels and two instantiation semantics; one instantiation semantics gives user models from the UML metamodel; and one gives model instances from a user model. This understanding is given by the UML specification [OMG10]: An instance of metaclass `Class` is a class, and

a class is a type that has objects as its instances...The instances of a class are objects. In this interpretation of the UML specification the potency of class `Class` is two.

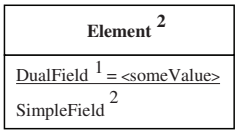

Figure 3.10: From [AK01], notation for potency, simple and dual field

Atkinson and Kühne also introduce [AK01] the notion of *simple field* and of *dual field*. A simple field with potency 0 has a value and corresponds to what traditionally is called a slot; a simple field with potency 1 is an attribute (description of a slot) and consequently it does not have a value. Instantiating a simple field with higher potency than 0 makes it a simple field with potency one less.

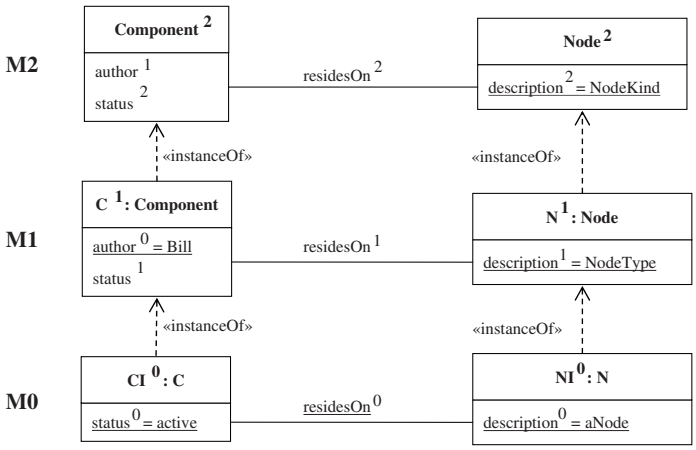

Figure 3.11: From [AK01], components and nodes with deep instantiation

Dual fields are special and not found in ordinary modeling since they have a value even when the potency is higher than one, Atkinson and Kühne state [AK01]:

> Basically, a dual field of potency `n` corresponds to a set of `n` simple fields, all identical except that they all have different potencies (`n`, `n-1`,..., `0`), and the field of potency `0` has a value.

Fig. 3.10 shows the notation introduced (Fig. 3.10 is a bit simplified, level indication is not shown); potency of a model element is given as a superscript; a simple field of potency 0 has a value and is then underlined[1]; a dual field is always underlined since it always has a value.

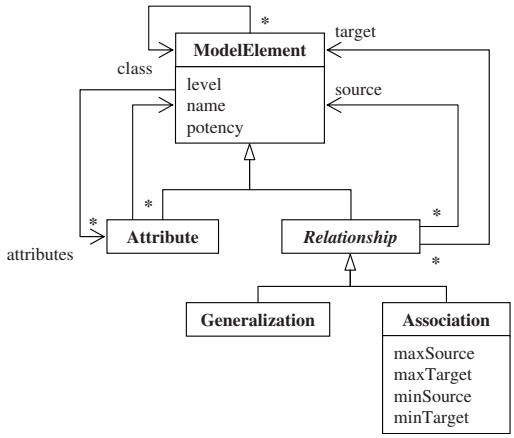

Figure 3.12: From [AK01], the MoMM

Fig. 3.11, which is found in [AK01], gives a demonstration of the concepts. `Component` and `Node` are defined at `M2`, all properties associated with `Component` and `Node` instances are defined at this level. The `author` field of `Component` has potency 1 and is instantiated on level `M1` where it keeps the name of the person that has defined component type `C`, in the example this is `Bill`. Field `description`, belonging to `Node`, is a dual field with potency 2, it is instantiated on the next two levels and it keeps a value on all three levels. It describes `Node` on level `M2`, `N` on level `M1` and `NI` on level `M0`.

The `residesOn` at `M2` is a "meta-association" since it has potency 2, it is an association at `M1` stating that components of type `C` can reside on nodes of type `N`; at `M0` `residesOn` is a link that states that component `CI` is placed at node `NI`.

[1]This notation is in conflict with static attributes of classes which also are underlined

[AK01] presents a preliminary meta-metamodel which is shown in Fig. 3.12, since this is meant as a meta-metamodel then maximum potency of 2 is the only one being meaningful.

The approach presented in this section has been further elaborated by Thomas Kühne and Daniel Schreiber: A prototype implementation called DeepJava has been presented [KS07]; DeepJava is extending Java and it allows true multi-level modeling.

Since object-oriented approaches typically only support two levels (class and object) they will inevitably introduce accidental complexity, and some sort of workaround, when several levels are needed to do domain classification. Several workarounds have been identified [KA08]:

> Consequently, if one needs to create a model of a domain involving more than two levels using a two-level language like the UML, one is forced to use artificial workaround mechanisms or modeling patterns that allow the properties of multi-level scenarios to be mimicked using only two levels. Typical examples addressing this need include static variables, tagged values, stereotypes, powertypes, reflection, and a number of variations of the "Item Description" pattern. The problem with such workarounds is that they complicate and obscure the meaning of a domain model.

One of the mentioned workaround mechanism, powertypes, is defined by Odell [Ode94]: *A powertype is an object type whose instances are subtypes of another object type.*
Also UML has the notion of powertype, this is demonstrated in Fig. 3.13(a) where `PersonalityType` is the powertype. Often a class can be specialized into different sets of subclasses; such a set (called a *generalization set* in UML 2.X and *discriminator* in UML1.x) is given by a powertype. Looking more into the example of Fig. 3.13(a) we realize that `PersonalityType` may have several subtypes; one subtype could be called *The four humors* – which is the one presented in Fig. 3.13(a) under the more general term `PersonalityType` – another subtype would be *Keirsey Temperament Sorter*. Both the personality types mentioned gives a different set of subclasses of `Personality`.

Fig. 3.13(b) shows the same as Fig. 3.13(a) but in a layered fashion; in UML1.x one may have a dependency relation stereotyped `powertype` from `Personality` to `PersonalityType`. We understand from Fig. 3.13(b) that this is a kind of metamodeling, but since class `Personality` plays a role in this setup it seems not possible to use powertypes in a strait forward

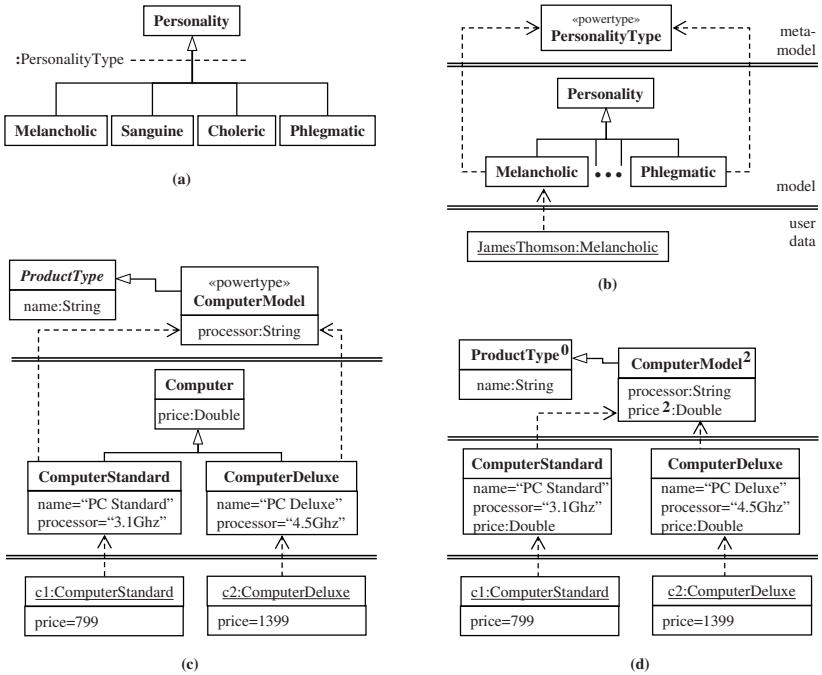

Figure 3.13: Examples of powertypes and use of potency; (c) and (d) are from [KA08]

way in all metamodeling situations; however, powertypes have been used to do advanced metamodeling [GPHS06]. Fig. 3.13(c) uses powertypes while Fig. 3.13(d) uses potency, and we notice that use of potency makes it possible to omit class `Computer` (or class `Personality` in our first example).

3.1.7 Comparing and Relating to Solution (IBe)

Subsection 3.1.1 shows that programming languages organizes their metamodel stacks differently and allow different numbers of levels; the most flexible ones allow an arbitrary number of levels. We have also seen that instantiation semantics are attached and "executed at different levels", so the programming languages are in this respect exhibiting a lack of uniformity.

Subsection 3.1.2 presented the Type Object Pattern. This pattern is limited since it only allows two ontological levels, i.e., it is not possible to assign a type to an instance of type `ObjectType`. The same limitation is also seen when it comes to the `InstanceSpecification` of UML. However, the Type Object Pattern can easily be extended to more levels as is done in the UML virtual machine approach presented in Subsection 3.1.3. The UML virtual machine approach is strongly coupled to one specific language, i.e., the UML 1.x.

The representation of metalevels in JMI implementations, presented in Subsection 3.1.5, is in principle the same as for EMF. EMF, presented in Subsection 3.1.4, is typically used with a metamodel stack of depth 3. There are ways to come around this – the partial prototype implementation [PNCW06] of IBe demonstrates this. The prototype can be seen as an instance of ECore that defines a modeling language that supports non-linear metamodel hierarchies (as described in Section 2.9). However, IBe is an "independent" language and EMF is only used as an implementation language when making the prototype. Similar prototypes can be defined by help of the other modeling frameworks being presented. The choice of EMF, when making the prototype, was based on the widespread use of EMF (Eclipse is probably the most used environment today when it comes to MDD) and on its maturity.

The MoMM approach, presented in Subsection 3.1.6 allows an unlimited number of levels and this makes it interesting. The principle of potency is interesting since it allows information to be attached high up in the metamodel stack so that its applicability is at its optimum – in a sense this resembles attaching general information as high up in a class-subclass hierarchy as possible. IBe allows semantics to be attached to model elements; the semantics attached are Java code that are meant to be executed. When

execution is performed new model elements may be created on lower levels, and some semantics may be attached to the new elements in this process. The MoMM approach may be supported by IBe by attaching the right semantics, i.e., IBe offers some basic constructs for representing levels and for connecting levels, and these may be used, while the dynamic part is defined to support the MoMM approach.
If seeing the following solution chapter as a feasibility study, then ease of understanding becomes an argument. IBe used for consistency modeling only demands three levels and implementing the MoMM approach seems like adding unnecessary complexity (assuming the reader is familiar with three level metamodel stacks not based on MoMM).

The lack of uniformity seen for programming languages, when it comes to organization of levels and instantiation semantics, can also be seen for the modeling frameworks. A framework that could easily be tailored for different handling of instantiation semantics would be beneficial when doing further research on this topic. Such a framework would not have "fixed instantiation semantics" as the approaches just presented, but would allow different instantiation schemes to be (easily) defined. IBe is meant to function as such a framework. This requirement is, however, not significant in the context of this thesis, but nevertheless it explains some of the rationales behind the later presented solution.

The approaches presented in this section do not fully support multi-model architectures and in this respect they run short in relation to Req. 3. Also Req. 2 is not fully meet since none of the approaches offer an algorithm as specified by this requirement. The next section presents approaches that are more complete when it comes to supporting modeling of consistency requirements.

3.2 Some Consistency Modeling Alternatives

The approaches presented in this section offer different ways of revealing consistency among data sources.

3.2.1 A Multiple Representation Schema Language

The same entity may be represented more than once in either identical or different databases, which often results in the occurrence of inconsistencies among the different representations of the same entity. For data sources with semantically related models, one simple consistency rule may be the following: Two objects (entities) with the same identity must have the same

values stored for corresponding attributes; otherwise, they are not consistent with each other (e.g., one data source claims that the floorage of an apartment is 125 square meters and another lists 100). This problem was the target of paper *A Conceptual Schema Language for Managing Multiply Represented Geographic Entities* [FCJNS05]. In this paper Anders Friis-Christensen, Christian S. Jensen, Jan P. Nytun[2] and David Skogan describes a declarative language that can be used to specify rules that match objects representing the same entity, maintaining consistency among these representations, and restoring consistency if necessary. The following is a quote from [FCJNS05]:

> We propose an approach to the modeling of multiple represented entities, which is based on the relationships among the entities and their representations. Central to our approach is the Multiple Representation Schema Language that, by intuitive and declarative means, is used to specify rules that match objects representing the same entity, maintain consistency among these representations, and restore consistency if necessary. The rules configure a Multiple Representation Management System, the aim of which is to manage multiple representations over a number of autonomous federated databases.

The paper *Modeling and Testing Legacy Data Consistency Requirements* (Appendix A) is like [FCJNS05] focused on the consistency problems that occur when previously uncoordinated, yet semantically overlapping, data sources are being integrated. The language presented in Appendix A is not limited to testing of consistency among multiply represented entities – it is a more powerful (visual/logical) language for deriving data, were derivation of consistency data is merely one application. However, it does not propose any way of handling inconsistencies. If ignoring syntactical differences, then the approach presented in this section corresponds to a subset of the MOF 2.0 Query, View, and Transformation (QVT) specification [OMG09].

3.2.2 Constraint Satisfaction and Constraint Programming

Vipin Kumar focus on those *constraint-satisfaction problems* (CSP) that can be stated as follows [Kum92]:

[2]The paper was published in 2005, but the work started some years earlier as an important part of the Ph.D. work of Anders Friis-Christensen. Among other things, I contributed to the development of a metamodel for the defined language and a formal description of the UML stereotypes that were introduced.

> We are given a set of variables, a finite and discrete domain for each variable, and a set of constraints. Each constraint is defined over some subset of the original set of variables and limits the combinations of values that the variables in this subset can take. The goal is to find one assignment to the variables such that the assignment satisfies all the constraints. In some problems, the goal is to find all such assignments.

Kumar gives [Kum92] an example based on the map-coloring problem depicted in Fig. 3.14; the regions that are to be colored are given in Fig. 3.14(a) and Fig. 3.14(b) shows a corresponding constraint graph where each node is a variable, and each arc represents a constraint between variables; in Fig. 3.14(b) the variables represent regions and a constraint between two variables disallows identical color assignments to these two variables (i.e., the two variables represent adjacent regions).

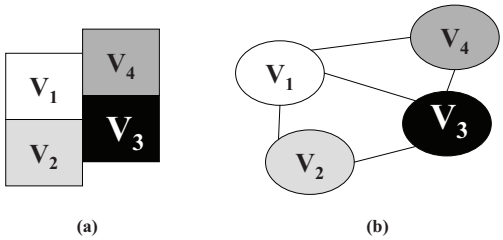

Figure 3.14: Map-coloring problem and equivalent constraint-satisfaction problem [Kum92]

The constraint-satisfaction approach can be seen as a part of *constraint programming* [Bar07]; Roman Barták defines constraint programming [Bar99]:

> Constraint programming is the study of computational systems based on constraints. The idea of constraint programming is to solve problems by stating constraints(requirements) about the problem area and, consequently, finding solution satisfying all the constraints.

The solution to the consistency modeling and the automatic generation of consistency data presented in Chapter 4 can be classified as a form of constraint programming; Boolean variables are constrained by OCL constraints, and model elements like links and objects are generated automatically if specified OCL constraints are fulfilled. One way of solving a CSP is

to systematically generate all possible combination of the variables and test each to see if all constraints are satisfied, this is called the *generate-and-test paradigm*; a variation of this technique is also proposed in this work.

Some work has been done when it comes to UML and constraint programming, e.g., in [CCR08] Jordi Cabot et al. describe how to translate both class diagrams and OCL constraints into a CSP, the CSP is then giving the possibility for checking compliance of the diagram with respect to several *correctness properties*. However, no work has been found that corresponds to the way UML and OCL are utilized in IBe.

3.2.3 Triple Graph Grammars (TGG) and QVT

Andy Schürr introduced [Sch94] Triple Graph Grammars (TGG) as *a new formalism for the specification of complex interdependencies between separate and, in general, quite different graph-like data structures.* TGG can be seen as a declarative formalism that allows specification of bidirectional translations between graph structures. The correspondences are modeled explicitly and are not restricted to the case of one-to-one relationships [Sch94]. Schürr presents translation between a programs syntax tree and its corresponding control flow diagram as an example [Sch94]; the syntax tree plays the role of being the left graph (LG) and the control flow diagram is playing the role of being the right graph (RG); additionally there is a *correspondence graph* (CG) with *correspondence rules* which links and defines the "intergraph" relationships between LG and RG. Given a syntax tree, then a corresponding control flow diagram may be automatically generated; in this case the syntax tree function as a *source graph* and the control flow diagram function as a *target graph*, the translation is bidirectional so it is also possible to translate from the control flow diagram to a corresponding syntax tree.

Metamodeling and also model transformations are central to MDD, and the ideas of TGGs have been adopted by the OMG in their MOF 2.0 Query, View, and Transformation (QVT) specification [OMG09]. QVT requires that source and target models conform to MOF metamodels. Fig. 3.15(a) and (b) shows an example that is often used when presenting TGG and QVT (see [Kn05] or [GGL05] for more details). Fig. 3.15(a) shows a *triple graph grammar schema* relating `Class` and `Table`; Fig. 3.15(b) shows an accompanying *triple graph grammar rule* that may be used to make a corresponding table when a class is given. Fig. 3.15(b) is demonstrating the use of constraints (written in the comment symbol); the constraints states that the name of the class must match the name of the table, and additionally

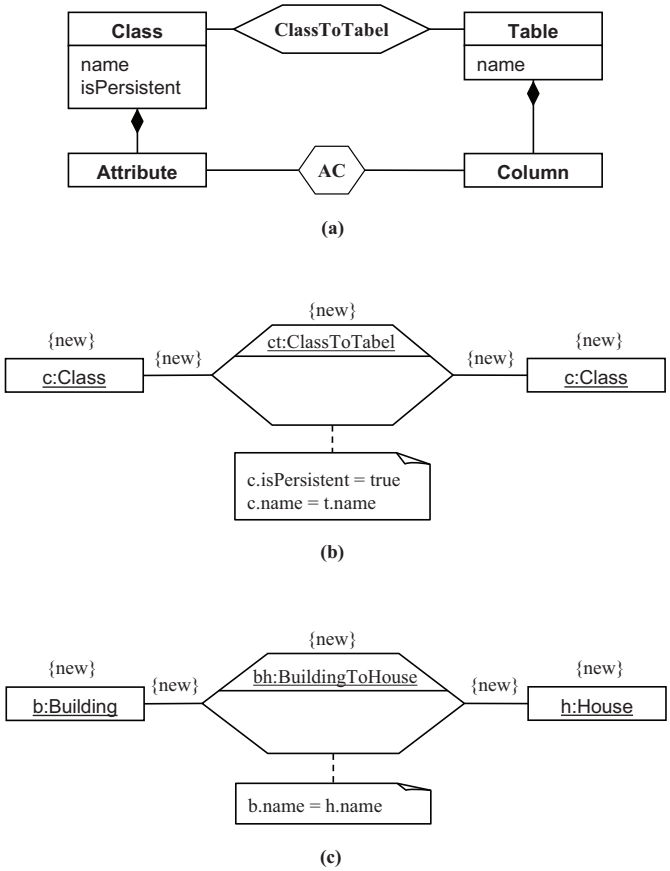

Figure 3.15: Relating UML Class to Relational Table (a) and (b), Building-House grammar rule (c)

the class must be persistent (i.e., attribute `isPersistent` must be true for the class).

MOF supports the principal object-oriented concepts (in the same way as UML), and this allows class models in general to appear as MOF instances (e.g., a model containing "non-metaclasses" like `Building` and `House`). Grammar schemas can consequently be attach to such models and transformations can be defined for instances of the class model. Fig. 3.15(c) demonstrates this by defining a grammar rule on instances of two classes found in different models.

IBe has several similarities with TGG/QVT, e.g., in both approaches there is a left graph, a right graph and a graph in the middle relating the other two graphs. The Multi-model architecture shown in Fig. 1.1 is close to the architecture pattern seen for QVT. QVT constrains the architecture by requiring that:

- The two metamodels of the legacy systems are MOF.
- The consistency metamodel is the QVT metamodel.

In TGG/QVT *correspondence links* are parts of the correspondence graph and they link nodes from the left and the right graph. The same concept is found in IBe where it is called *consistency association*. It is possible to attach constraints to both correspondence links (e.g., Fig. 3.15(c)) and consistency associations (e.g., Fig. 4.1).

Despite the similarities there are some important differences between the two approaches and we return to these in the discussion chapter (Chapter 5).

3.2.4 ATLAS Model Weaver (AMW)

Marcos Didonet Del Fabro et al. describe [FBJV05,FBJ$^+$05] how two models are woven together with yet another model; such a model is called a *weaving model* and it is used when transformations are performed.

Rondo [MRB03] is a programming platform for generic model management and it includes high-level operators used to manipulate models and mappings between models. AMW [FBJ$^+$05] goes further and allows extensible mappings (AMW is related to QVT, but different). AMW is a generic model weaver that allows the specification of correspondences between model elements from different models – models are in this way connected. For example, if two models have model elements that models the same entities and uses the same "namespace" for the Ids, then the two models can be weaved

together based on this information, i.e., entities that have same IDs will be linked. AMW is based on a metamodel called KM3 [JB06] that allows a total of four levels. This approach seems applicable to our problem since it do target multi-model architectures and it allows up to four metalevels. However, the type of consistency requirements we need to specify can be complex and a combination of visual modeling and logic is more appealing then specification of complex mappings as is needed in AMW. Section 5.1.1 is further investigating AMW in relation to IBe.

Chapter 4

Solution

The paper in Appendix A introduced an approach for testing consistency between data residing in different legacy databases. The approach included a metamodel describing the language to use when defining a consistency model, and the approach required a metamodeling framework that allowed different types of instantiation. The definition of and also the application of such a framework is discussed in the appendixes. This chapter sums up the parts of the papers that are essential to this thesis. This chapter gives a coherent and slightly updated presentation of the work; it is updated in the sense that some names have been changed and a few more details are supplied (especially when it comes to instantiation). Different working titles have been used for the framework – the name is now *Integrating Border environment* (IBe). The purpose of this chapter is to present the proposed consistency modeling and explain how IBe may be used to realize this type of modeling. A multi-model architecture is meant to be defined by instantiating IBe (IBe seen as a model).

When fully developed, IBe is meant to support the definition of static structure and also of behavior (e.g., execution of state machines). A static structure is what appears when the running system is at rest and the dynamics (behavior) is about change. IBe is composed of two interwoven parts (two models); one part is called `STAND` and it defines the static aspect; the second part is called `ACT` and it gives the dynamics. `STAND` is the most developed part, while `ACT` is not that mature and rests heavily on the embedding language, which is Java. Attaching semantics amounts to attaching interpreters to a structure given by `STAND`; an interpreter is capable of interpreting the structure it is attached to. At the current state the interpreters

are written in Java and there is no additional language defined for behavior. `STAND` structures may of course be used to define behavior; in this thesis the `STAND` structures presented are more like declarations than prescriptions (imperatives) since they do not state how to "do something" – this is left to the interpreters.

In the following, the term "metamodel" [Küh06] is meant to mean a model that is a model (in a "type-like-fashion") of another model which again is a model (in a "type-like-fashion") of a third model [Küh06].
The term "instance" is in the following being used in a general sense: It is used to denote the model elements of a model (since they have been instantiated from some model element from the level above) and it is also used on "complete" models, i.e., a model is playing the role of being a *model instance* in relation to another model – if the last model describes, in a "type-like" fashion, the former model (this corresponds to two levels in a metamodel stack).

James Rumbaugh et al. defines instantiation as [RJB05]: *The creation of new instances of model elements.* Attaching *instantiation semantics* is then to attach interpreters that create new instances based on the `STAND` structures where they are attached.

The proposed approach to consistency testing is declarative: The user defines a consistency model which functions as a declaration in the sense that an instance (i.e., the consistency data) of the consistency model is automatically produced – the consistency data connects the data of the two involved databases and contains consistency data. Once the consistency model instance (i.e., the consistency data) is produced it does not change, and there is only one possible consistency model instance that is possible given two databases and the consistency model.

In the following discussion the databases are considered to be snapshots of databases as opposed to operational databases.

Section 4.1 describes the proposed consistency modeling approach. Section 4.2 presents the basics of IBe. Section 4.3 describes a solution to the problem of consistency modeling in the form of an IBe application.

4.1 The Consistency Modeling

We start this section with an example; the same example that can be found in Appendix A. Fig. 4.1 shows an integration of two legacy models, where one is a description of apartments (class `Apartment` found in model `MDB1`) and the other a description of buildings (class `Building` found in model

`MDB2`). The consistency requirements are as follows:

1. The number of apartments that is given as a property in class `Building` should be equal to the number of apartments (i.e., instances of class `Apartment`) with the same building id (attribute `bId`).

2. A building should have at least one apartment, and an apartment should belong to exactly one building.

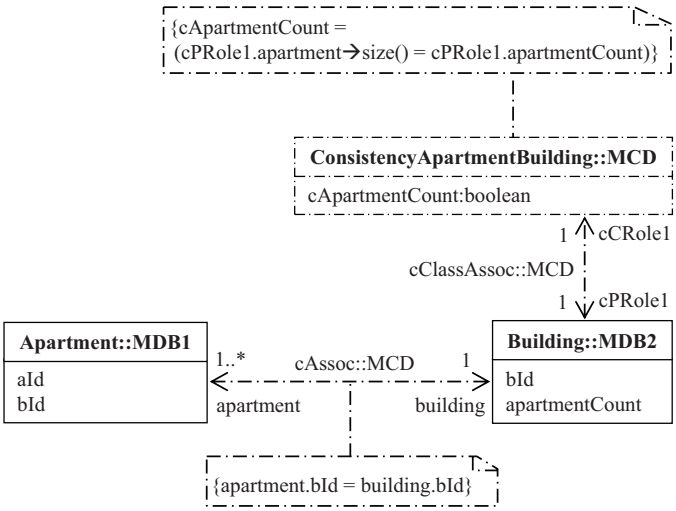

Figure 4.1: Consistency between `Apartment` from model of DB1 and `Building` from model of DB2; other elements are from the consistency model

The elements with dash-dotted line style in Fig. 4.1 constitute a *consistency model*; this model is (manually) made by the user. When the consistency model is made, consistency testing of legacy data may be performed automatically. In our case there is one legacy data source with information about apartments (DB1) and one about buildings (DB2). Consistency testing results in a report revealing legacy data that do not fulfill the consistency requirements.

The association (Fig. 4.1) between `Apartment` and `Building`, including the attached invariant expressed in OCL, models consistency requirement 2. When testing is performed on the legacy data, a link is created between

an `Apartment` and a `Building` instance if the invariant is fulfilled; if the multiplicity on the association is broken, this is reported in a consistency report which is information in addition to the consistency data.

Consistency requirement number 1 is specified with the help of class `ConsistencyApartmentBuilding`, property `cApartmentCount` and its attached invariant. During testing, instances of type `ConsistencyApartmentBuilding` are created and linked to `Building` instances; slot `cApartmentCount` will be set to the value that fulfils the invariant; if the value is `false` then a consistency violation has occurred. Note that links between `Building` and `Apartment` instances are traversed when the values of `cApartmentCount` slots are set. We see from this example that standard OCL-statements are used to decide what instances to produce when the consistency model is being automatically instantiated.

Fig. 4.2 gives an overview of the kind of models involved in doing the proposed consistency modeling and testing: There are two databases involved, data of the first database is represented by `DB1` and the data of the second database is represented by `DB2`; the arrows with dotted line (Fig. 4.2) represent the instance of relation; `MDB1` and `MDB2` represent the models (schemas) of the two databases while `MMDB1` and `MMDB2` represent the metamodels of the two databases. If we relate this to the example above, then `DB1` is the database describing apartments and `DB2` is the database describing buildings. The models (including `DB1` and `DB2`) constituting the two metamodel stacks of the databases are just read, and they are not changed during the consistency modeling and the automatic generation of the consistency data.

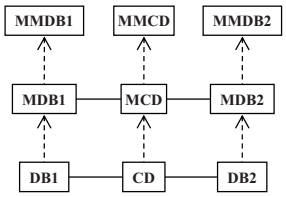

Figure 4.2: Some of the models involved when doing consistency modeling and testing

`MMCD` represents the metamodel of the consistency model `MCD`. The automatically produced consistency data is represented by `CD`. If we relate this to the example above then the elements with dash-dotted line style in Fig. 4.1

belong to MCD.

Several of the figures in Section 4.3 will present the different elements of Fig. 4.2 in more details and Fig. 4.2 is then added to indicate which part of the multi-model architecture that is being described.

4.2 Integrating Border environment (IBe)

The parts of IBe that have been implemented have been implemented in Java, and as such Java functions as an embedding language (depicted in Fig. 4.3(a)). STAND consists of a set of Java classes described in Fig. 4.4. At runtime the STAND classes are represented as Java class objects which are instantiated to form model structures. Hence, instantiation in IBe is achieved with the help of instantiation in Java.

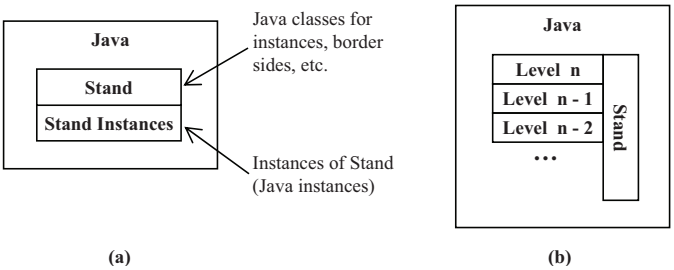

Figure 4.3: STAND embedded in Java (a) and spanning several metalevels (b)

STAND defines a (mega-)model for defining multi-model architectures and ACT defines where semantics may be attached. The actual definition of dynamic semantics is done by programming Java classes that may be connected at certain places to structures defined by STAND.

The consistency modeling and automatically production of consistency data (introduced in the previous section) are to be handled in the same tool, and consequently modeling environment and runtime environment are meant to be the same environment; and further, there will be run-time instances behind every model element that appears.

In this view the data sets (the data in a database) are considered to be models, they are considered to be *terminal models* since they can not be instantiated further into model instances in the multi-model architecture. The approach presented in this work is not a generative one (however, IBe

may be adapted so that it supports compilation, etc.) – the modeling environment "is the software system" and this makes it natural to see a data set as a separate model.

In the following sections STAND and ACT will be presented in more detail.

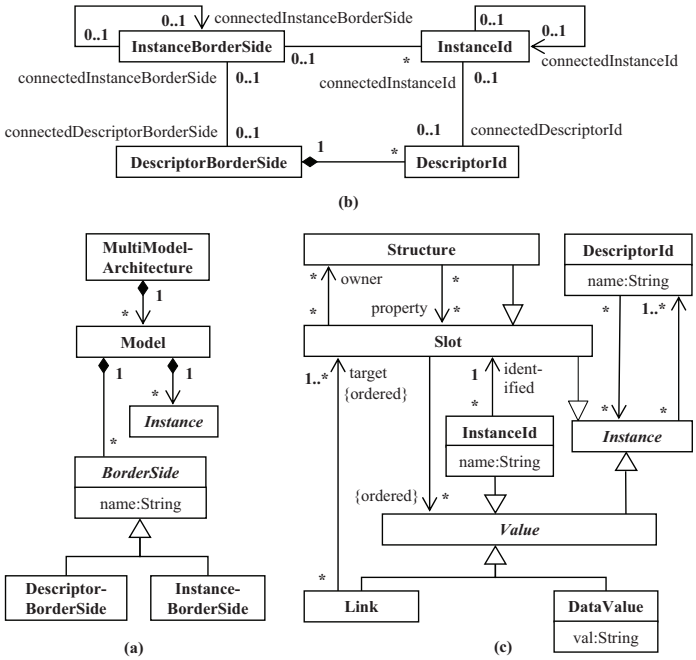

Figure 4.4: Main part of STAND

4.2.1 STAND

STAND is an extended instance model where the instance model is, if ignoring name differences, not very different from what is described by the Instances package of UML and MOF (this part of STAND is found in Fig. 4.4(c)).
In STAND the instance model is extended to include representation of models which may be connected to form model and metamodel architectures; Fig. 4.3(b) shows how STAND may span several metalevels (only one metamodel stack is shown). To be more precise STAND gives two linguistic levels,

STAND itself and an instance of STAND, the instance of STAND can contain several ontological levels (and when STAND is used to define a language, then also linguistic levels[1]).

A model is built up of instances of subclasses of *Instance*. The term instance is used instead of model element to indicate that in this environment, where the model and the runtime environments are joined, the elements are all instances. The instances (instances of Instance) are identified with instances of InstanceId and descriptions of the instances are referenced by *descriptor identifiers* which are instances of DescriptorId. *Instance identifiers* refer to instances inside the model. Descriptor identifiers refer (indirectly through a connected instance identifier) instances typically residing inside another model, a model that describes in a "type like" fashion the instances of the model in question. A descriptor identifier function as a name for an instance that again function as a description; the description is placed at the metalevel above what is being described. Such a description corresponds to what James Rumbaugh et al. calls a descriptor [RJB05].

An instance identifier is an id or a key as you find it in a relational database (or like ID attributes in XML documents); a descriptor identifier is somewhat similar to a foreign key referencing metadata (or like element names in XML documents). Several instances may share the same descriptor identifier, while an instance identifier identifies only one instance in the model.

All instances have at least one instance descriptor. Instances of DataValue represent values of primitive types. For simplicity all data values may be thought to be of type String; the supplied figures do not show the descriptor identifiers for instances of DataValue (how to treat primitive types and values is discussed in Appendix D).

There are two ways to relate models: One correspond to the instance of relation (e.g., an object is an instance of its class) and the other is a way of relating models where none of the models plays the instance-of-role. The last mentioned relation may be used to connect models residing at any metamodel level, e.g., the consistency model is connected to each one of the legacy models through such relations.

A model is related to another model by connecting one border side from each model; two connected border sides establish a border. There are two types of border sides, one called *instance border side* (called InstanceBorderSide in STAND, see Fig. 4.4(a) and (b)) meant to reference the in-

[1]Ontological or linguistic levels? From the "IBe framework perspective" there is no difference when looking at the levels found in the instance of STAND.

stances in a model and one called *descriptor border side* (called `DescriptorBorderSide` in `STAND`) meant for establishing the instance of relation. An instance border side is composed of instance identifiers and a descriptor border side is composed of descriptor identifiers. Two instance border sides may be connected to establish the model relation which is not the instance of relation; with respect to consistency modeling this relation is simply used to reference elements of database models (e.g., `MDB1`) from the consistency model.

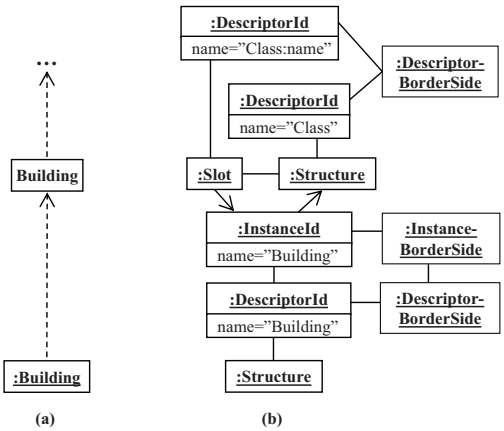

Figure 4.5: Model instance connected to model

A model is set to be a description of another model by connecting a descriptor border side to an instance border side; the descriptor border side is part of the model instance, while the instance border side is part of the describing model. The descriptor identifiers of the descriptor border side are connected to the instance identifiers of the instance border side. Fig. 4.5 presents a simple example: A class called `Building` and an object of this class. Fig. 4.5(b) shows, in the form of an UML object diagram, how this can be represented using `STAND`: The object is found at the bottom; it is connected to a descriptor identifier with name `Building` which is part of a descriptor border side; the descriptor identifier is also linked to an instance identifier with same name; the instance identifier is connected to an instance border side which is connected to the mentioned descriptor border

side; the descriptor and instance border sides constitute the border (called a *descriptor border*) between the model and the model instance; the class is represented as a structure that has a slot with the mentioned instance identifier (`Building`); both the structure and the slot have descriptor identifiers found at the top – these two descriptor identifiers are connected to yet another descriptor border side at the top.

There are some constraints on how to instantiate `STAND`, these are given as OCL constraints in Table 4.1.

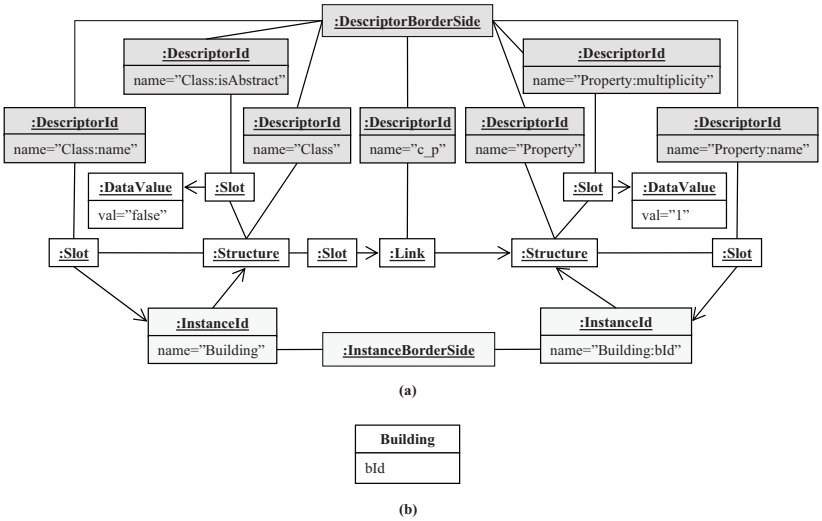

Figure 4.6: Example of `STAND` representing a class

Fig. 4.6(a) shows one way to represent the class shown in Fig. 4.6(b) with `STAND`. The objects in Fig. 4.6(a) with gray fill color are parts of border sides. (The link between slot and containing structure is actually two links instantiated from the `property` and the `owner` association; the same goes for the link between instances and descriptor identifiers.) Several representations are possible, e.g., linking of a property to its "class structure" could have been done so that it also would be possible to navigate from the property to the "class structure". When building tool support (e.g., a visual editor) it seems natural to select one way of doing it and then stick to it – but `STAND` is flexible.

Context `InstanceBorderSide` **inv**:
– – A border side can only be connected to one other border side.
`connectedInstanceBorderSide.size()=1` **implies**
`connectedDescriptorBorderSide.size()=0`

Context `InstanceBorderSide` **inv**:
– – A instance border side can not be connected to it self.
`connectedInstanceBorderSide <> self`

Context `InstanceId` **inv**:
– – An instance Id can be connect to either an instance Id or a
– – descriptor Id.
connectedDescriptorId.size()=1 implies
connectedInstanceId.size()=0

Context `InstanceId` **inv**:
– – An instance Id can not be connected to another Id unless it is
– – connected to an instance border side.
`instanceBorderSide.size()=0 implies`
`connectedInstanceId.size()=0 and`
`connectedDescriptorId.size()=0`

Context `InstanceBorderSide` **inv**:
– – If an instance border side is connected to another instance
– – border side, then its instance Ids can only be connected to
– – instance Ids belonging to the other instance border side.
`connectedInstanceBorderSide.size()=1 implies`
`instanceId.forAll(id | id.connectedDescriptorId.size()=0`
`and (id.connectedInstanceId.size()=1 implies`
`id.connectedInstanceId.instanceBorderSide=`
`connectedInstanceBorderSide))`

Context `DescriptorBorderSide` **inv**:
– – If a descriptor border side is connected to an instance border
– – side, then its descriptor Ids can only be connected to
– – instance Ids belonging to the instance border side.
`connectedInstanceBorderSide.size()=1 implies`
`descriptorId.forAll(id | id.connectedInstanceId.size()=1`
`implies id.connectedInstanceId.instanceBorderSide=`
`connectedInstanceBorderSide))`

Table 4.1: Constraining `STAND`

The Metamodel of the example can be seen in Fig. 4.7, e.g., descriptor `c_p` is the association between class `Class` and class `Property` of the metamodel (property `isId` of metaclass `Property` is ignored in Fig. 4.6(a), but it may be used to find instance identifiers). An IBe model (Fig. 4.4) may

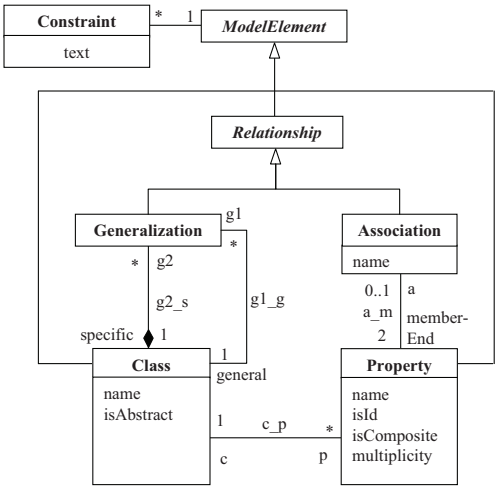

Figure 4.7: Metamodel defining classes, etc.

be treated as a pluggable module that may be connected to other models through border sides as described above; in this view a descriptor border side is like a special type of required interface – it is an interface to a model that is meant as a description of it. However, even if two models may be connected with a descriptor border (i.e., the descriptor identifiers and the instance identifiers match) it may not be the case that the "model instance" actually is a model instance of the other – some "checking" functionality is needed to verify that the instances of one model could have been instantiated from the instances of the other.

As a basis, `STAND` can give rise to many different types of applications, e.g., it may be used as a basis for an analytical tool as described in Appendix E and F. Often the following procedure is used when doing MDD: First a metamodel is created, then models are built according to this metamodel and then model instances. However, in an analytical application a model

instance may appear first and then the problem is to find its model, e.g., is the model instance a standard XHTML document or is it a "none standard XHTML" document? (More about this in Section 4.4.)

The use of borders allows models to be connected without having them merged into one model. But what is meant with *merge* in this context? With a "coarse-grained view" a model may be seen as composed of identifiers and "structure tieing" the identifiers together; if another model have a reference into the structure of another model, then at the level above (where the type models of the two models are placed) this reference must have been specified by help of elements from both type models (e.g., by an association); this implies that at least on type model "knows about the other" and the type models are then considered to be merge, i.e., they are seen as one and the same model, consequently the models on the lower level are also together defining one model. One may differentiate between a *full merge* (references from "each model" into the other) and *one-way merge* (reference from only "one model" into the other). The situation is different if the identifers are used to connect the models, the identifiers may appear in both models and knowledge about the structure of the other model is not needed. On the other hand, just using the identifiers when connecting models may lead to a situation where the models are not "really" fitting together (e.g., that a type model is not really a type model for the connected model). In some cases it is also necessary to code into the identifier names some structural information for the process of connecting the models to succussed, e.g., the name "name" is often appearing many places, but "Class:name" and "Property:name" may be unique. The use of identifiers to connect models is a *encapsulation technique*.

One an the same model might play different roles in relation to other models and this is made explicit with the use of borders. A border might "look different from its two border sides" and this is captured by having two border sides represent one border, e.g., when two models are to be connected by the instance of relation (i.e., an instance border side is to be connected to a descriptor border side) the instance border side may contain more instance ids than what is found on the descriptor border side (in this case it may be that the model that plays the model instance role can be connected to a less comprehensive model than the one playing the model role in this case). The use of border sides allows the use of different names in two models that are connected, e.g., an instance border side may contain the instance id `Class` and still be connected to a descriptor border side containing descriptor id

`Klasse` (the Norwegian word for class), for this to work there must be some special scheme for matching ids with different names.

It is also possible to attach semantics to border sides, semantics that may generate a model instance (comprised of a huge number of objects of different types) – an example of this is explained later.

Typically a class will not contain references to all its instances, but an object will typically "know" its class. However, in `STAND` (Fig. 4.4) `DescriptorId` has an association with navigation towards ***`Instance`*** and this may be used to go from an instantiated element to its instances (if the association is "used/mantained" by the application). The solution to the consistency modeling presented below queries descriptor identifiers to get the instances (i.e., finding the instances of a given class) and it seems practical, at least in that application, to use the association (even if the instances may be found by using the links given by the association that goes from `Model` to ***`Instance`***).

It should also be noted that the finer granularity given by using the term "border side" contra just "border" is often not necessary and the term "border" is therefore often used. When the term "*descriptor border*" is used, a descriptor border side and an instance border side are connected to form the border; the term "*instance border*" has the obvious meaning.

IBe `STAND` does not have any limits on the number of levels, and it supplies – based on the instance of relation – a generic way of stacking models on top of each other to form metamodel stacks; IBe is in this respect a non-linear approach since it offers an explicit way of handling the instance of relation; the instantiation semantics is however not fixed, i.e., there are no class, or metaclass concept defined in `STAND`; the instantiation semantics is given by what is chosen and attached as instantiation semantics.

4.2.2 `ACT`

`ACT` is the part of IBe that is meant to give the dynamic semantics and as Fig. 4.8 shows, semantics may be connected to instances and also to border sides. Allowing semantics to be attached to instances opens up for an "ordinary object-oriented approach", e.g., defining a method may be done by defining some `STAND` structure describing the class and the method, and then attach semantics to that structure. Border side semantics gives several possibilities, one example would be to define semantics that implements SQL, attaching this semantics to a descriptor border side would allow querying the instances found in the model by using the descriptor identifiers (also OCL querying could be supported in this way).

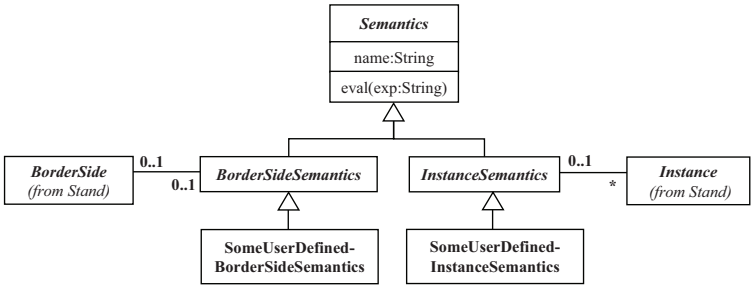

Figure 4.8: Part of ACT

Fig. 4.8 shows that semantics is defined in the form of Java classes that either extends BorderSideSemantics or InstanceSemantics; the semantics will be executed by calling the eval() method with an appropriate argument. IBe provides means to attach such Java classes and ways to trigger their execution. This approach opens up for changing the semantics as needed since the semantics are explicitly attached.

The IBe demonstrator [PNCW06][2] that has been developed is based on EMF, this means that STAND is implemented as an ECore model and instances of this ECore model represent multi-model architectures; this may look like a trick since we end up with having all the STAND model levels at one EMF level. However, it has merely given a jump-start by supplying a lot of useful code; EMF provides *setter* and *getter* methods for the classes of STAND; these methods may be seen as parts of the dynamic semantics. EMF also provides an editor so that STAND structures may be built.

Attaching semantics corresponds to attaching interpreters; an interpreter is capable of interpreting the structure it is attached to. In a computer system the interpreter will ultimately be a computer processor (we have discussed some of these aspects in another work [MNPW10]), but in our object-oriented context the interpreter is considered to be a software object composed of data and executable code. Most of the interpreters involved in the consistency application are concerned with instantiation, which in our context is the same as instantiating STAND (this means instantiating the Java classes that constitute STAND). The term *instantiator* is introduced to

[2]It was Andreas Prinz that first proposed to do the implementation in EMF.

denote interpreters concerned with instantiation. Instantiation of a complex model element typically leads to a cascade of "smaller" instantiations and creating a complete model instance may involve a huge number of smaller instantiations.

There are many ways to organize instantiation, e.g., creating a model instance may be done by one single interpreter or it may be done by an interpreter that triggers the execution of several other interpreters. Instantiation may also be organized differently with regard to levels; one possible conceptual mapping (somewhat simplified) to IBe of the solution seen in Java, is to consider there to be three levels:

Metaclass Level This level contains metaclass `Class`, metaclass `Property`, etc. These elements are instances (class objects) of elements found on this same level. Metaclass `Class` contains the description of the `newInstance()`-method which may be used to create new objects.

Class Level This level contains user classes and classes from supplied packages. These elements are instances (class objects) of the elements found on the metaclass level. All classes are instances of metaclass `Class` and this allows method `newInstance()` to be invoked on all concrete classes.

Instance Level This level contains the objects of the user classes which are created with help of the mentioned `newInstance()`-method.

As we understand from the text above, metaclass `Class` defines the method `newInstance()` which is general enough to create instances of all classes; `newInstance()` uses the description of the class (e.g., the description of the fields) when creating an object of that class; the object created is placed at the lowest level. The same way of organizing instantiation may be done in IBe, assume the following metamodel stack:

Metamodel This model contains structure corresponding to metaclass `Class`, metaclass `Property`, metaclass `Association`, etc.
This model is seen as an instance of it self, i.e., its descriptor border side is connected to an instance border side supplied by the same model.

Model This model contains user classes. These elements are instances (class objects) of the elements found in the metamodel. All classes are instances of metaclass `Class`, properties are instances of metaclass `Property`, associations are instances of `Association`, etc.

Terminal model This model contains the objects of the user classes. A user class is instantiated to an object of that class, a property is instantiated to a slot, an association is instantiated to a slot, etc.

The kind of instantiation described above coincides with instantiation as described in UML, but in UML the terminal model is considered to be composed of runtime instances and it is not seen as a model. Often the UML models (user models) are used as the source for code generation and consequently the runtime instances may be the runtime instances of a programming language like Java or C++.

Returning to the description of the terminal model above, this model does not contain classes or entities that can be further instantiated – it is composed of *terminal entities* like slots, objects of user classes, etc. The model level is composed of: Entities (e.g., user classes) that may be instantiated to instances that can not be further instantiated and also of terminal entities (e.g., name of user classes are given as values stored in slots). The metamodel is composed of: Terminal entities (e.g., the name "`Property`" of the `Property` metaclass is stored in a slot), entities that may be instantiated to instances that may not be further instantiated (e.g., an instance of metaclass `Property` representing the `name` property of metaclass `Class`) and finally of metaclasses like `Class` and `Property` which may be instantiated to instances that again can be instantiated. One may say that the metamodel in question is a metamodel and also a model for itself.

The metamodel is considered to be an instance of itself; and further, if we consider an *"instance of"-relation chain* to stop at class `Class` when going up the chain, then some elements have an instance of relation chain of length one and others of length two, e.g., metaclass `Property` has length one since it is a direct instance of metaclass `Class`, while a slot containing the name of a metaclass has length two since it is an instance of metaclass `Property` which again is an instance of metaclass Class. For every entity in all the three models the following is true: The number of entities in the "instance of"-relation chain going up added to the number of repeated instantiations that are possible downwards is always two in this setup.

"Metaclass Property" above is categorized as a metaclass, but when instantiated twice down the metamodel stack, then slots appear; a slot is not normally considered to be an object so the term *metaentity* may be more appropriate than metaclass, however, in the following text the term metaclass is still used.

The kind of instantiation just described is "the ordinary one", but it does not answer where the instantiators should be attached – only what

their results should be. One possibility (which is extreme in respect to how much the instantiator must "know") is to have one main instantiator that is attached to metaclass `Class`, this is possible since all entities have an "instance of"-relation chain that ends here; every other entity that may be instantiated has a very basic instantiator that simply delegates the instantiation job one level up the chain by sending an "do-instantiation" message to its descriptor (together with relevant parameters) – finally it ends up at the main instantiator which checks what kind of entity that is to be instantiated and then performs the instantiation accordingly, e.g., the main instantiator will create a slot if an instance of `Property` is instantiated and a class if metaclass `Class` is instantiated. The basic instantiators are attached to the entities by the main instantiator when it creates the entities – in this way instantiation semantics is propagated down the "instance of"-relation chain.

As already mentioned, it is also possible to attach semantics to border sides; when executing this type of instantiation a complete model instance may be created in one operation. This type of instantiation may be performed in a context where complete models with borders are included and play roles. An instantiator attached to a border side may do the job alone or it may utilize other instantiators like the one described above. We return to an example of this kind of instantiator when describing how the consistency data is automatically produced.

4.3 The Solution to Consistency Modeling

This section presents a solution to the consistency problem using the concepts of IBe. The different parts of Fig. 4.9(a) are addressed and discussed (Fig. 4.9(a) is a detailed version of Fig. 4.2). The two legacy database model stacks are treated in a similar way and conceptually the complexity is not reduced by only showing one of them (adding one more legacy database is done in the same way as the first one), and hence, Fig. 4.9(a) is only showing one legacy database model stack (when needed, a number is attached to indicate which legacy database being considered). Fig. 4.9(b) and (c) gives an overview picture of what Fig. 4.9(a) shows, i.e., it shows either database stack 1 or 2. Fig. 4.9(a) exposes where semantics may be attached, e.g., semantics may be attached to the "internal instances" of `MCD` (this semantics is named `SMCD`) and at the descriptor border side of `MCD` (this semantics is named `SDMCD`).

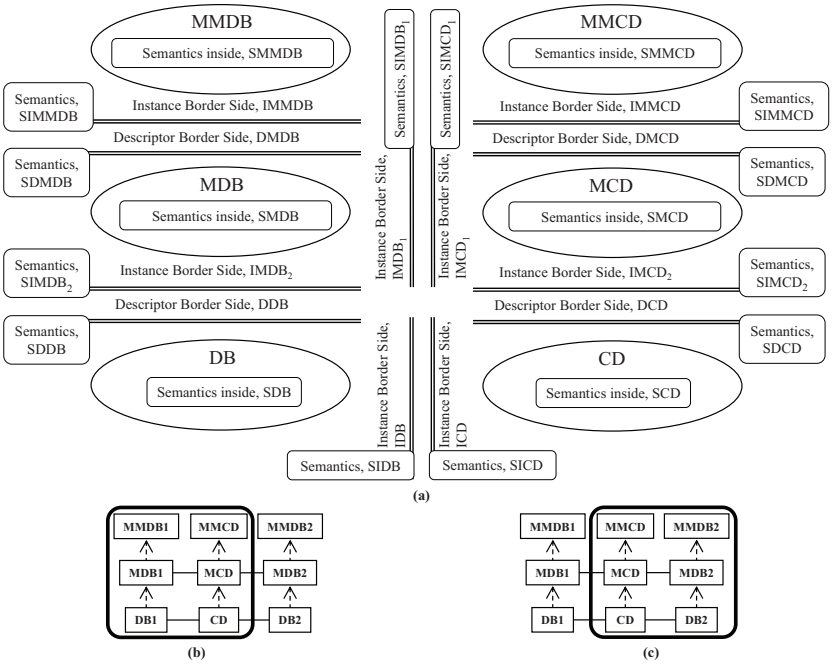

Figure 4.9: Naming the border sides and the semantics of the consistency application (a)

4.3.1 User Roles

It is possible to identify at least three types of roles when working with IBe: IBe framework developer, metamodeler (kind of user) and modeler (kind of user). A framework developer is one that develops the framework itself, i.e., designing and implementing the very basics of the framework. A metamodeler is one that establishes metamodels needed when doing modeling and organizes them into premade multi-model architectures (model configurations) – in our case these metamodels are the ones that must be in place when doing the consistency modeling and testing (i.e., automatic production of consistency data).

For IBe the difference between the role of being a framework developer and a metamodeler is not so distinct since also a metamodeler typically

needs to do some programming when defining semantics using Java. At time of writing, IBe is at an experimental and very early stage and it is by far not ready for an "ordinary modeler".

As already mentioned IBe is meant to be applicable to several different types of problems, and hence, one may envision IBe as a general tool – not tailored specifically for doing consistency modeling, but being general enough to cater for consistency modeling. The version of IBe presented below is however meant for doing consistency modeling – it should be seen as a setup meant for consistency modeling and testing.

While the task of the metamodeler is to offer a model configuration, the main tasks of a (consistency) modeler are:

- Select the two legacy data model files (MDB1 and MDB2); the tool will then load the two data models and make them available to the modeler.
- Do the consistency modeling.
- Start the automatic production of the consistency data after having selected the two legacy data sets (DB1 and DB2).
- Query the automatically produced consistency data to check for inconsistencies.

Actually, the number of legacy databases may be one or more; if only one, then the consistency checking is applied to data residing in the same database.

4.3.2 The Legacy Database Model Stack

If we assume that the legacy database is a relational database, then the schema, `MDB` (correspond to `MDB1` or `MDB2` in Fig. 4.2), defines the tables, the fields in each table, and the relationships between fields and tables. If some object-relational mapping is used, then we may see `MDB` as an instance of the UML metamodel and instead of tables one sees classes etc. Accordingly `MMDB` is understood as corresponding to the kernel of the UML metamodel or MOF.

We further assume that `MDB` appears in the form of an XMI-file; what is needed to load `MDB` is then a special type of XMI-reader. One type of readers may use an explicit representation of `MMDB` when interpreting the XMI-file; the instantiation semantics may have been distributed to the instances of `MMDB` (`SMMDB`) and trigged when creating the instances of `MDB` (as described

in Section 4.2.2). A reader that does not use an explicitly represented MMDB, will in some way have an implicit representation of MMDB since MDB when loaded will conform to MMDB – a reader of this kind may be seen as an "implementation of MMDB".

A descriptor border side (DMDB in Fig. 4.9(a)) may be produced by the reader since the XMI-file contains type information (e.g., the type of an XML element may be metaclass Class), a type name is then added to the descriptor border side when read. An instance border side may also be produced by extracting the names/ids given in the XMI-file and then add them to the instance border side. The XMI-file will reflect the namespace information defined by the UML metamodel and this may be used to avoid name conflicts on the border sides.

Looking at Fig. 4.2 we see that MMDB is not connected to other models, and hence by using a reader for MDB that does not use an explicit representation of MMDB the need for having MMDB represented in the stack is elevated.

Loading DB may be done in much the same way as MDB, we may again assume that the data appears in the form of an XML-file containing the data structured in an "XMI kind of way". The reader may be made so that MDB is not explicitly needed when the data are loaded into IBe, even so, MDB is explicitly needed since it is connected to MCD and accessed when the consistency model is made.

The reader used to establish MDB may be attached to a premade descriptor border side as semantics (premade by the metamodeler); when the semantics is triggered it lets the user select an XMI-file of the correct type and then it establish MDB; the same is also possible when it comes to DB.

4.3.3 The Consistency Model Stack

Fig. 4.10(a) shows MMCD which is the same consistency metamodel as presented in Appendix H [Nyt06] except for a few name changes. Fig. 4.10(c) shows how to instantiate the consistency model example given in Fig. 4.1 (MCD). Fig. 4.10(b) and Fig. 4.10(d) shows the position of MMCD and MCD in the multi-model architecture. The marking around MCD (Fig. 4.10(d)) is including the instance of relation to MMCD since this is supplied as type information, e.g., Building is of type CProxyClass. (The place of OCL in this architecture is discussed in Section 5.1.3.)

Fig. 4.10(c) shows that legacy class Apartment is represented in the consistency model by a *proxy class* (Apartment:CProxyClass), the same goes for class Building. Using proxy classes makes it possible to avoid having references into the legacy models and instead have the border sides integrating

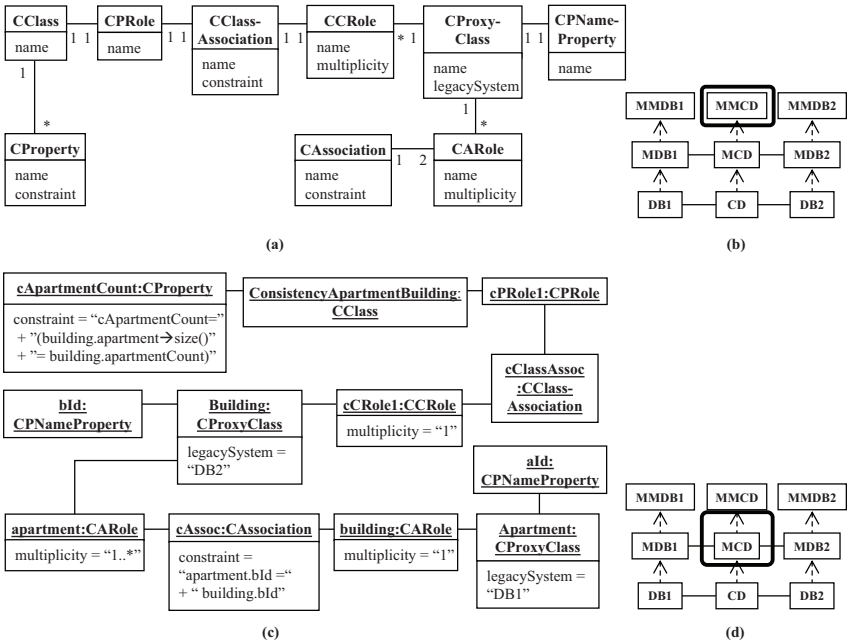

Figure 4.10: Consistency metamodel (MMCD) (a), and an instance of it (MCD) (c)

the models. Fig. 4.1 shows `Building` from `MDB2` and the corresponding proxy class in `MCD` as one class; `bId` is represented in `MDB2` and also in `MCD` where it is represented as an instance of `CPNameProperty`; property `apartmentCount` is not part of `MCD` only `MDB2`. A similar approach is used for `Apartment`.

Class `ConsistencyApartmentBuilding` is a *consistency class* (an instance of `CClass`), its *consistency property* `cApartmentCount` is an instance of `CProperty` which also stores the accompanying constraint.

`CClassAssociation` is instantiated when defining an association between a consistency class and a proxy class. `CAssociation` is instantiated when a *consistency association* is defined between two legacy classes, i.e., it goes between two proxy classes that represent the two legacy classes.

Fig. 4.11(a) shows the two lowest levels of the legacy database model stack example. We can see from the figure that `DB1` has two apartments

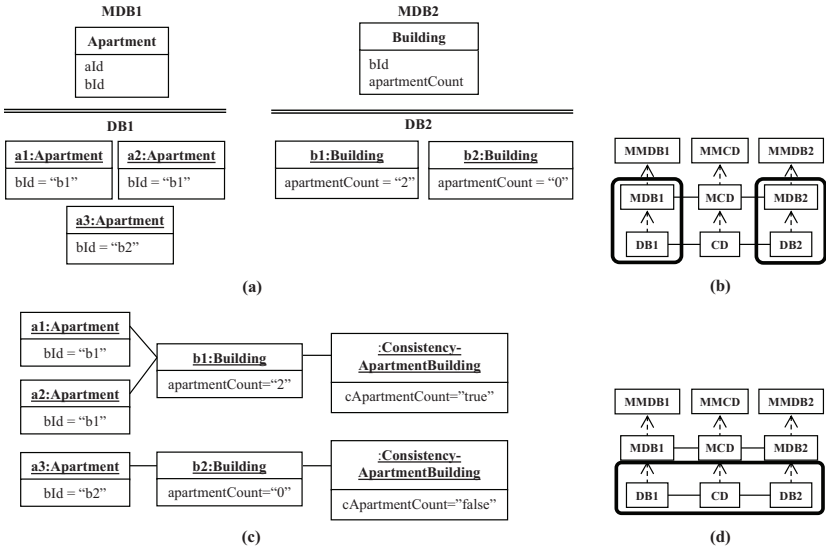

Figure 4.11: Legacy model and data (a), result of consistency testing (b)

with building id `b1` and this is consistent with `DB2` which has a building with building id `b1` and apartment count with value two; `DB1` has an apartment with building id `b2` which is not consistent with `DB2` which has a building with building id `b2` and apartment count with value zero. Fig. 4.11(c) shows the automatically produced consistency data, e.g., property `cApartmentCount` is `false` for the instance of the consistency class `ConsistencyApartmentBuilding` which is connected to the `Building` instance with id equal `b2`.

Fig. 4.10 together with parts of Fig. 4.11 show the consistency model stack for an example, they represent an interpretation of the underlying `STAND` structure.
Fig. 4.12(a) presents the border sides of `MCD` as they are represented in `STAND`. Fig. 4.13(a) depicts the structure and borders sides of `CD`; the object with gray fill color constitutes the border sides, while the objects in the middle shows the structure; the stippled lines represent the instance of relation; only the consistency structure that relates `a3:Apartment` and `b2:Building` is shown.

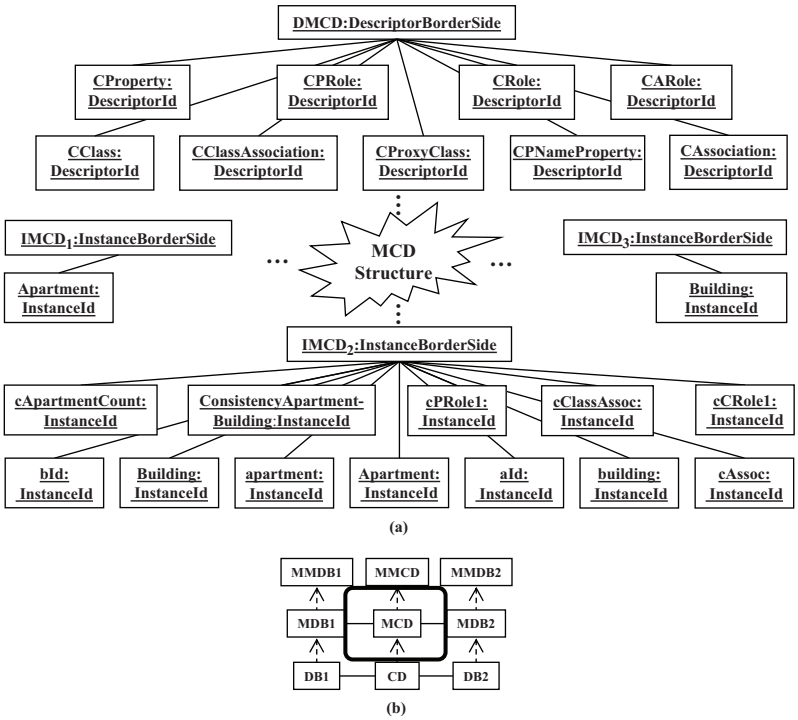

Figure 4.12: The border sides of MCD

So far the different levels of the consistency modeling stack have been described, but how these levels have been established has not been explained. The solution proposed in Appendix H [Nyt06] is to have an instantiator attached to DMCD which allows "MMCD" to be instantiated into MCD – this semantics is called SDMCD in Fig. 4.9(a) (Appendix H [Nyt06] uses the term *border engines* for attached semantics); SDMCD is preprogrammed by the metamodeler and is part of the model configuration needed when the modeler makes MCD.

The kind of instantiation performed when MCD is created is actually similar to the ordinary one (e.g., instantiating CClass resembles instantiating metaclass Class), and it may be solved by attaching instantiators to the

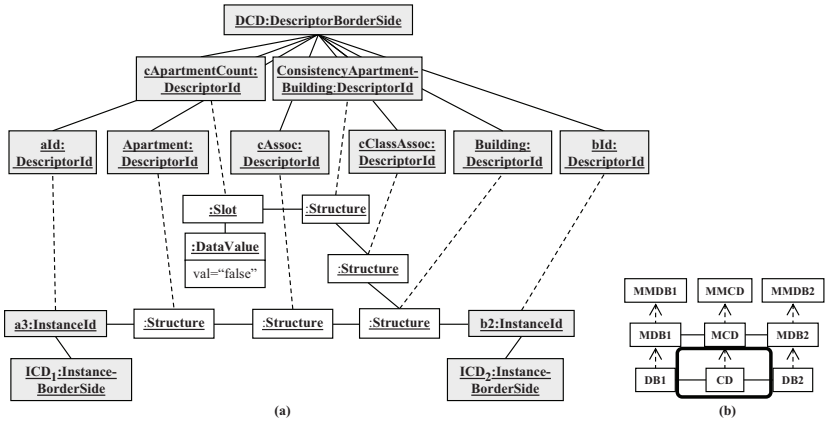

Figure 4.13: The structure and borders sides of CD

entities of MMCD. The consistency modeling should in some way be done visually by the modeler, and the instantiation will then be integrated into this visual support – however, how to give this support is not explained in this work.

$\mathtt{IMCD}_1$ (seen as instance border side towards $\mathtt{MDB}_1$) contains instance identifiers corresponding to the names of all classes in MDB1 that have a corresponding proxy class in MCD - in our example this is only Apartment. $\mathtt{IMCD}_1$ is connected to $\mathtt{IMDB}_1$ which contains the instance identifiers of all classes that may have proxy classes in MCD (the identifiers of $\mathtt{IMCD}_1$ is equal to the set of identifiers of $\mathtt{IMDB}_1$ or a subset of it); $\mathtt{IMDB}_1$ may be automatically produced by querying the descriptor border of MDB1 when it is loaded.

The most complex instantiation is performed when the consistency data (CD) is automatically produced. CD is a model instance of MCD and the instantiation requires knowledge about not only MCD but also about $\mathtt{DB}_1$ and $\mathtt{DB}_2$. In Appendix D I propose to have an instantiator attached to $\mathtt{IMCD}_2$; this instantiator is implementing the algorithm presented in Appendix A; a slightly modified version is presented below.

If the consistency modeler does not pay attention he may introduce cyclic dependencies among the elements of CD that are meant to be made – if this is the case then it is not possible to make CD. Cyclic dependencies can be exposed by inspecting the constraints specified in MCD. The inspection is

done by building a dependency graph where there are three kinds of nodes:

CAssociation Nodes One node of this kind will represent the set of links found in `CD` that have been instantiated from one particular `CAssociation` (Fig. 4.10(a) shows the consistency metamodel). `cAssoc` in Fig. 4.1 is an example of a particular `CAssociation`.

CClass Nodes One node of this kind will represent the set of objects that have been instantiated from one particular `CClass`. `ConsistencyApartmentBuilding` in Fig. 4.1 is an example of a particular `CClass`.

CProperty Nodes One node of this kind will represent the set of slots that have been instantiated from one particular `CProperty`. `cApartmentCount` in Fig. 4.1 is an example of a particular `CProperty`.

There is a constraint attached to each node: The constraint attached to a `CAssociation` node is the one attached to the corresponding `CAssociation`, the constraint attached to a `CClass` node is the one that is attached to the `CClassAssociation` going to the corresponding `CClass` and finally the constraint connected to a `CProperty` node is the one attached to the corresponding `CProperty`. If no constraint is specified then `true` is assumed. The edges (i.e., the dependencies) of the dependency graph are found in the following way: There is an edge from each `CProperty` node to the `CClass` node which represents the `CClass` containing the `CProperty`; if the constraint of a node has a reference to another node (i.e., a reference to the corresponding consistency model element of the node) then there is an edge to this node.

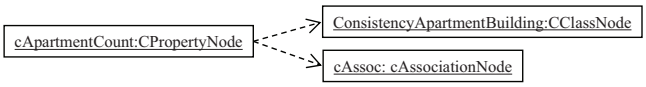

Figure 4.14: Dependency graph for example shown in Fig. 4.1

Fig. 4.14 shows the dependency graph for the example shown in Fig. 4.1. The graph is also showing the possible orders that the elements of `CD` may be produced in: The instances of `ConsistencyApartmentBuilding` and `cAssoc` must be created before the instances of `cApartmentCount`.

How to automatically produce `CD` will now be described, but first a list of needed assumptions:

- $\mathtt{DB}_1$ and $\mathtt{DB}_2$ are loaded; $\mathtt{DDB}_1$ and $\mathtt{DDB}_2$ have been extracted and are available for querying.
- `MCD` is loaded and is available.
- The instantiator is attached to the descriptor border (i.e., to $\mathtt{IMCD}_2$ or `DCD`) between `MCD` and `CD`.
- There are no cycles in the dependency graph for the elements of `CD`.

The consistency data may be created by following the steps below:

1. Extract $\mathtt{IDB}_1$ and $\mathtt{IDB}_2$. $\mathtt{IDB}_1$ is to contain all the instance identifiers of the objects that are instances of the classes in $\mathtt{MDB}_1$ that have proxy classes in `MCD`; the proxy classes are given by `MCD` and the instance identifiers are found by querying $\mathtt{DDB}_1$; given the situation shown in Fig. 4.11(a) then the only proxy class is `Apartment` and `a1`, `a2` and `a3` will be placed at $\mathtt{IDB}_1$; $\mathtt{IDB}_2$ is extracted in the same way.
2. Create $\mathtt{ICD}_1$ and $\mathtt{ICD}_2$. The instance identifiers of $\mathtt{ICD}_1$ will be the same as the ones constituting $\mathtt{IDB}_1$ and the instance identifiers of $\mathtt{ICD}_2$ will be the same as the ones constituting $\mathtt{IDB}_2$.
3. Connect $\mathtt{ICD}_1$ with $\mathtt{IDB}_1$ and connect $\mathtt{ICD}_2$ with $\mathtt{IDB}_2$.
4. For every instance identifier of $\mathtt{ICD}_1$ and $\mathtt{ICD}_2$ create and connect *proxy objects*, e.g., create a proxy object of proxy class `Apartment` and connect it to the identifier instance `a1` that is part of $\mathtt{ICD}_1$; the proxy objects are parts of `CD` and should of course also be connected to the correct descriptor identifiers.
5. Create all possible instances of `MCD` that fulfil the specified constraints; the order of creation is given by the dependency graph for the elements of `CD`, e.g., `cAssoc` instances (Fig. 4.1) will be tried created for each possible pair made by one instance from proxy class `Apartment` and one instance from `Building`; if the constraint attached to `cAssoc` is satisfied then the instance is kept.

An OCL interpreter, not described in this work, is needed to see if the constraints are satisfied.

The `CD` made can be examined by the consistency modeler to see how $\mathtt{DB}_1$ and $\mathtt{DB}_2$ relate.

4.4 Applications in Addition to Consistency Modeling

The applications presented in Appendix D and E are based on an early version of IBe, and they are highly related to the consistency modeling already described. While the basic principals of these papers were briefly presented in the introduction the main purpose of this section is to make it more clear how they relate to IBe and to the consistency modeling presented above.

Web pages (in our case XHTML documents) should be accessible to all users, independent of disabilities or choice of web browser; the aim of the projects presented in Appendix D and E is to measure the accessibility of web pages by relating them to modeled accessibility requirements.

The multi-model architecture in question is a metamodel stack containing 3 models:

Metamodel The top level is a metamodel that describes concepts needed to define the XHTML-standard as a model. A metamodel defining a subset of the UML metamodel has been chosen (the subset corresponds to the metamodel presented in Fig. 4.7); this choice makes it possible to use OCL to specify accessibility requirements.

Model A subset of the XHTML-standard is represented as an instance of the metamodel found at the top level.
This level is where the accessibility modeling is performed; accessibility requirements are specified in OCL and attached to the modeling elements.

Model Instance An XHTML-document represented as an instance of the model level above. The evaluation of the accessibility requirements (i.e., the OCL constraints) are performed on this model instance; a report is generated that states to which extent the web document fulfils the accessibility requirements.

If we extend the metamodel with a selected part of the consistency metamodel, then we can model the accessibility requirements in the same way as done when modeling the consistency requirements (i.e., visual modeling together with OCL). The situation is less complex when implementing the modeling of the accessibility requirements: Only 1 model is involved as opposed to 3 when implementing the consistency modeling (the same is also the case at the model instance level).

Appendix E is a continuation of Appendix D; it investigates how to use "incomplete models" – this fits well with IBe, e.g., not all descriptor Ids referenced in a "model instance" might be described in the attached "incomplete model".

Appendix E is also demonstrating the usefulness of allowing models to be treated as pluggable modules by having a "model instance" first created and then attached to a possible model – again this fits well with IBe.

Chapter 5

Discussion, Conclusions and Further Work

In this chapter the different findings are discussed, some conclusions are made and some further work proposed. The first section concerns some relevant issues and some alternative solutions. The last section compares the solution to the requirements stated in the introduction.

5.1 Alternative Solutions and Relevant Issues

Many conceptual issues have come up through this work and a few of them are discussed in the following, but first a comparison of the solution to some consistency modeling alternatives.

5.1.1 Comparing to Consistency Modeling Alternatives

TGG and QVT, introduced in Subsection 3.2.3, are based on an architectural pattern that resembles the one described in the solution chapter (Chapter 4). The main concern to these approaches are synchronization of models (e.g., synchronization of two legacy data sets where one is source model and one is a target model that is updated) and the generation of new models based on some structure-preserving mappings from source models. The focus of these approaches are not the generation of the consistency model instance as it is for the IBe solution – this model will therefore typically not be a part of the architecture.

QVT contains an imperative language that might be used to define an interpreter of consistency models (as they are described in the solution chap-

ter); the interpreter could in some way be attached to the correspondence graph; this interpreter could then generate the consistency data. However, this would require some programming and it would be a new application of QVT. Again, the focus of TGG and QVT are transformations, while deriving data is the focus of the solution – in this respect the solution seems closer to constraint programming[1] (presented in Subsection 3.2.2).

In regard to consistency modeling, the presented solution is simpler and will typically be easier to learn than QVT. The solution requires knowledge of UML and OCL; in addition the user must understand the special use of "invariants"; the concrete syntax is UML and some way of indicating which model the elements belong to (e.g., by using dash-dotted line style as demonstrated in Fig. 4.1). QVT on the other hand, includes: UML, OCL, some new concrete syntax and a new language to describe correspondences.

A user of the solution needs to understand that there must be no circular dependencies between the data to generate (a tool may assist the user in this regard) – otherwise the consistency modeling is very much like ordinary UML modeling as opposed to using QVT which would imply more than ordinary UML modeling.

The strengths of QVT and the proposed solution are different, and QVT can not easily replace the proposed solution in regard to the proposed consistency modeling.

An approach that have several similarities with the proposed one is the Atlas Model Weaver (AMW) [FBJ+05, FV09] (see Section 3.2.4).
In AMW a metamodel for model weaving is proposed. The metamodel includes elements that resemble the proxy class (see Fig. 4.10(a)) and the association between proxy classes. Consistency classes are not defined.

Marcos Didonet Del Fabro et al. state [FBJ+05]:

> ...model weaving...Its primary objective is to handle fine-grained relationships between elements of distinct models, establishing links between them. These links are captured by a weaving model. It conforms to a metamodel that specifies the link semantics. Typical application domains of model weaving are database metadata integration...Obviously more abstract constraints information, for example expressed in OCL, may be attached to a link...we conclude that there is no standard weaving metamodel

[1] If not already done, a formal comparison of TGG to constraint programming seems interesting; these approaches seems to be closely related!

> WMM capable of capturing all weaving semantics...Each application domain has different needs that must be considered by the developers. The design of a base metamodel of which most weaving metamodels may be seen as extension is a delicate compromise between expression power and minimality.

As stated above, different types of semantics are handled by extending the base WMM, one extension may be *equality* between elements so that equal elements may be linked (e.g., class `House` in one model may be linked through an *equality link* to class `Building` in another model). A weaving model (WM) may be used to automatically generate transformations, e.g., one that takes elements from one model and map them to elements in another (the two models in question are instances of the two models that have been weaved by the WM).

The metamodel presented in Fig. 4.10(a) (with the consistency class excluded) is like a basic weaving metamodel and it can be used to relate elements from different models. Also the metamodel may be extended with different semantics. However, the semantics attached to the consistency model is unique. The purpose of the consistency metamodel is to facilitate definition of models that can be used to derive (consistency) data given two data sets; the consistency class together with properties are essential when deriving the data – anything similar has not been found for AMW. When it comes to the consistency modeling, only one mapping is considered and this is the one that takes two data sets and a consistency model and generates the consistency data. The conclusion is: Despite similarities the approaches are different in an essential way.

5.1.2 Axiomatic or Recursive Top Level

The topmost lever can be describe recursively by a self-describing model as demonstrated by MOF or it can be described axiomatically by another language which is not part of the metamodel stack [GOS07]. The presented multi-model architecture (see Fig. 4.2) has a recursive top level for the legacy database metamodel stacks (also MOF could have been used as a top level); the practical value of this top model may vary, e.g., a reader may load a model (i.e., the model corresponding to the database schema) into the architecture by utilizing this top level model or not. However, at least conceptually there is a value to this model since it explicitly describes the loaded model.

The practical value of having the top model recursively defined is also

questionable, however, conceptually it is satisfying to have a situation where as much as possible is explicitly described; using an axiomatic approach is in a sense hiding more than a recursive approach since a description of the top model is not explicitly given for the axiomatic approach. In Fig. 4.2 the consistency metamodel is forming an axiomatic top level, and again MOF may be added on top giving a recursive top level; for this to function instantiation semantics must be handled in some way that allows all the levels beneath MOF to be established.

The following statement is given by Ralf Gitzel et al. [GOS07]:

> ...The core aspect of these arguments is that a recursive metamodel allows those who already understand the language to look up details of its syntax without needing to be proficient in another language at the same level, whereas an axiomatic top-level model requires less overhead in the hierarchy.

This statement indicates that there are arguments for or against each of the two approaches; IBe `STAND` is open for both of them, and thus allowing the most appropriate approach to be selected in a given situation.

5.1.3 Extending the Multi-model Architecture

If we look closer at the consistency metamodel stack we see that OCL statements are to be added as "pure text" and stored as property values. Another approach is to define also OCL by a metamodel (e.g., as specified by the OMG Editor [OMG05b]); this metamodel is then to be placed at the level above the consistency model. The consistency metamodel and the OCL metamodel must in some way be integrated (i.e., making a bind to OCL) otherwise it is not possible to connect an OCL statement to the consistency model – one solution is then to connect the OCL metamodel to the consistency metamodel at the same places as the textual OCL statements were attached (or more correctly were they were specified as properties) in the already presented solution.

Connecting the OCL metamodel and the consistency metamodel can be done by a full merge of the two models into one coherent model. IBe allows also another approach: The models may be connected by an instance border which allows them to be kept as individual models. The possibility of integrating models without doing a full merge is justifying the term multi-model architecture (this principle is already demonstrated by having the consistency model connected to the models of the legacy databases). The arguments for using the term multi-model architecture is further strengthen

by adding a metametamodel on top of the consistency metamodel (the consistency metamodel is not self-describing), in this case we have metamodel stacks that have different depths in one and the same architecture (the legacy metamodel stacks are only three levels deep).

5.1.4 The Use of Border Sides

Modeling a border with two border sides comes at a price: The metadata for an instance is reached by following three links as can be seen in Fig. 5.1(b). If the identifiers of the instance border side have the exact same names as the identifiers of the descriptor border side and the number of identifiers on each border side is the same, then the number of links to follow may be reduced to two; this optimization is demonstrated in Fig. 5.1(c) where one border (of type `DescriptorBorder`) has replaced the two border sides and matching identifiers have been replaced by objects of type `Symbol`. Fig. 5.1(d) shows

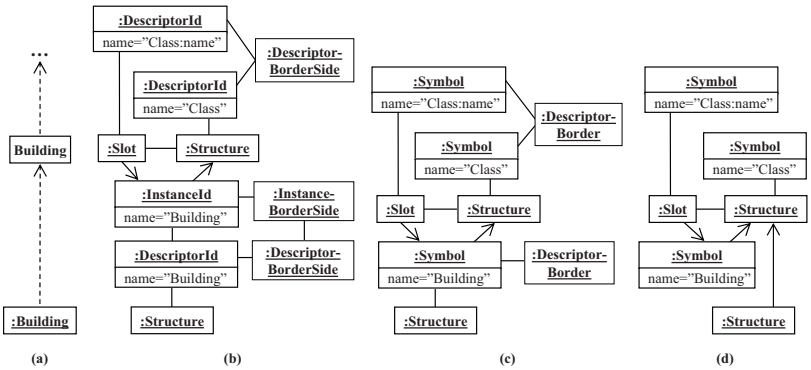

Figure 5.1: Optimizing reference from model instance to model

yet another optimization where there is a direct link from an instance to its describing instance – such an optimization may be temporarily established or it may be permanent which means that the two models in question have been merged. The same type of arguments in regard to optimalization may also be valid for instance border sides.

The notion of being a model is extremely general (e.g., a Java source file is a model), consequently the number of possible relationships between models

are overwhelming. Kühne discusses [Küh05] two fundamentally different model relations: The type model role and the token model role.

The functionality attached to the type model role can be instantiation or functionality for checking if a model can be seen as an instance of another model. A descriptor border side is like a special type of required interface – it is meant to be an interface to a model whose instances (accessed through an instance border side) can function as a description of it. In this respect a type model can be seen as a component which offers a description.

The token model role fits well when defining a system by repeated refinements of models, e.g., a PIM is a token model for a PSM when they both model the same system.

The solution to the consistency modeling problem includes instance border sides; this is a model-to-model relationship which does not relate to levels; in Appendix G instance borders where presented as a way of connecting models placed at same level, and it was meant to establish a token model role; the constraint that the models must be on same level has later been removed since it may be used between models placed in different metamodel stacks (the number of levels may vary from metamodel stack to metamodel stack). The instance border sides have been used to establish a "shallow" token model role between the consistency model and the legacy models – the role is limited to having proxy classes in the consistency model that corresponds to classes in the legacy models. The consistency model is however containing other model elements that does not fit the token model role. However, the notion of token and type model roles coincide with the notion of descriptor and instance identifiers.

The notion of token model as described by Kühne [Küh05] is constrained: Only one of the models plays the role of being a token model and this model should in some way be "reduced" compared to the other due to the reduction feature of models (i.e., a token model only reflects a selection of an original's properties). Often two models have overlapping information, and at the same time both models may be more detailed than the other one in different respects – these situations can not be captured by a token model role.

The use of instance borders presented in this work is extremely generic: Their main purpose is to reference elements in models without doing a merge, and in this respect enforce encapsulation of the (inner) structure of a model; it is left as further work to decide if `InstanceBorderSide` should have subclasses (e.g., `TokenModelBorderSide`), and what kind of semantics to attach in those cases.

5.2 Are the Requirements Satisfied?

Chapter 1 outlined six requirements for a solution of the consistency modeling problem, each of these requirements are addressed below.

Req. 1 A language (metamodel) is required for consistency modeling.

A unique language has been proposed and its syntax is defined by the metamodel presented in Fig. 4.10(a). The combination of visual modeling (UML) and logic (OCL) gives the language its strength. The language is simple to understand if OCL and UML are known in advance. Both UML and OCL are playing important roles in software modeling; OCL is continuously being reused in a growing number of areas and UML is the de facto standard for doing software modeling, and consequently many potential users will quickly learn how to use the language.

The language proposed in this dissertation may be extended to allow more complex visual modeling, e.g., by having associations between consistency classes. This will work as long as it is possible to establish an order of instantiation when it comes to generation of the consistency data (`CD`), i.e., there must be no cyclic dependencies between type of elements being automatically generated.

Req. 2 Consistency data is to be generated automatically (a suitable algorithm is needed).

Section 4.3.3 describes an algorithm that will give automatically generation of consistency data. The main problem to solve is the order to use when the consistency data is being produced and this is only possible if there are no cyclic dependencies among the elements to create, and a helpful feature in a tool offering consistency modeling would be to continuously indicate if there are any cyclic dependencies or not (the dependencies are checked at the consistency model level and consequently this seems to be manageable).

The previous chapter gives a solution – a solution that is flexible enough to support instantiation in "the small" (e.g., ordinary class instantiation) and in "the large" (creating the whole consistency model instance (i.e., the consistency data) in one coherent instantiation process).

The current solution is based on programming in Java; the semantics of Java code is assumed to be know and attaching semantics is then attaching Java code to instances of IBe `STAND`; the Java code can be inspected by using

the Eclipse Java editor, but from the "IBe perspective" the code appears as a black box solution. Another solution is to develop `ACT` into a complete language together with `STAND` for defining behavior; behavior in this context would typically mean manipulation of instances of IBe `STAND`; since also behavior is described by structure, `STAND` may also play a major role in this respect. Development of `ACT` into a more complete separate language is left for future work, but such a solution would be more autonomous and more controllable by the framework. However, the described solution with use of Java confirms the feasibility of the approach.

Req. 3 The consistency and legacy models must be connected when making a consistency model. Consequently simultaneous handling of models in different metamodel stacks is required.
The consistency data is to be generated automatically and connected to legacy data, so simultaneous handling of different terminal models (data sets) is also required.
In short, the environment should support multi-model architectures (e.g., as seen in Fig. 1.1) so that the user is not forced to merge elements from different metamodel stacks, but is allowed to manage the individual models as pluggable modules (objects).

The previous chapter presented a model for representing multi-model architectures (Fig. 4.4); the model can be seen as a megamodel or at least as a part of a megamodel. However, the model is not "mega" in the sense that all kinds of models can be directly placed in such an architecture; a model must be converted into the defined representation language, i.e., it must be represented as specified in Fig. 4.4(c) which is a part of the megamodel. Further work may add the possibility of actually integrating models represented in a rich diversity of languages into the same multi-model architecture (e.g., by introducing some sort of adapters).

The conversion of a model so that it fits into an IBe architecture is based on extracting two sets of identifiers, these two sets are typically available: Instance identifiers are identifiers that identify singular instances in the model, while descriptor identifiers are used to do classification, i.e., a descriptor identifier may be used on several instances.

The identifiers of a border side belongs to a namespace, and `STAND` may be extended to allow relations between the identifiers of a border side and in this way support an explicit representation of a namespace. The relation needed seems to be composition as defined in UML; Subsection D.3.1 presents a first attempt to solve this.

IBe as an ontology for MDE is obviously not complete, but still, as explained in the previous chapter, IBe fulfills Req. 3.

Req. 4 Modeling and generation of consistency data should be tightly integrated and it should not be necessary to leave the modeling environment to generate the consistency data (i.e., the modeling environment is the runtime environment).

In the proposed solution, the modeling and the runtime environment will be one and the same environment. As opposed to a generative approach where the model is used to generate "the software system" – the modeling environment "is the software system".

There is however a limitation when it comes to implementing new semantics in Java: Loading an updated class is not possible if the earlier version is already loaded[2]. Currently, defining the semantics is done by writing Java code in Eclipse, and a strict interpretations of what constitutes the IBe environment may see this as an activity outside the environment, anyhow, this is a weak point in the solution; defining a new language for dynamic semantics that is managed by IBe will resolve this.

The generated consistency data does not change and consequently a snapshot is "permanently true". This is correct for one consistency model given two legacy data sets (in a sense the data sets are like parameters when generating the consistency data), but a user friendly solution may allow the consistency modeler to change the consistency model and at the same time see the change this inflicts on the consistency data (this is only possible if the legacy data sets are manageable in size). One may even allow the metamodel (or any model in the architecture) to be changed and handle the consequences of this. We have discussed some of the challenges this would raise [MNPW10]:

> ...due to the separation between modeling environment and runtime environment, the impact of model changes on existing data is difficult to support. However, these data are an essential part of the system, especially in enterprise systems. This paper describes an integrated modeling-runtime environment that forms the basis for enabling fine-grained evolution of large-scale enterprise systems using generative techniques. This integration

[2]This limitation may be removed in the future, e.g., JRebel is a plugin that enables the Java Virtual Machine to reload changes made to Java class files on-the-fly [Kab07].

> means that the runtime system maintains its own models and can, therefore, be locally adapted, without missing impact information. The integration between modeling and runtime, moreover, ensures that all dimensions of possible impacts of a change can be traced: (1) between models and dependent models; (2) between models and implementation; and (3) between models and instances, the data...it does not explicitly support the removal of model classes. However, it does allow for replacing groups of model classes with corresponding new versions and, on top of this, a mechanism may yet be offered to restructure the data correspondingly. This will also be the subject of further study.

Extending IBe with such features as described above will take IBe closer to the vision of having a fully united runtime and modeling environment.

Req. 5 Visual modeling is considered beneficial and should be supported.

This work does not fully describe how to achieve visual support, it propose a visual syntax and a way of doing visual consistency modeling (see Fig. 4.1 for an example of the visual syntax). The visual modeling involves the two legacy models and the consistency model, the software giving the visual modeling must therefore be aware of several parts of the multi-model architecture – how to formalize this in a good way is a challenge that is left as future work.

To conclude: A solution for doing MDE based consistency modeling has been presented; Req. 1-4 have been meet; even if Req. 5 has not been fully solved, it should be clear that it is solvable.
Treating models as modules combined with the possibility of attaching semantics in various places introduces several interesting possibilities – one of them being the proposed consistency modeling, others could be different types of weaving and integration of models.

IBe as a platform is far from being fully researched; however, it seems to be flexible and well suited for experimentation in regard to metamodeling. The way of attaching semantics and "where" it is proposed to be attached have in this work been chosen so that it is easy to understand, but it seems feasible to have IBe "simulate" the approaches presented in Chapter 3 when it comes to instantiation and handling of metalevels. The consistency metamodel is placed in a linear hierarchy (even if IBe, at least partially, is a

non-linear approach), but what about changing the consistency metamodel so that it gives a non-linear approach? Such questions may be answered by further work.

Appendix A

Modeling of Consistency between Legacy Systems

Jan Pettersen Nytun[1,2] and Christian S. Jensen[1,3]

[1]Faculty of Engineering, Agder University College
Grooseveien 36, N-4876 Grimstad, Norway
[2]Department of Informatics, University of Oslo
P.O.Box 1080 Blindern, N-0316 Oslo, Norway
[3]Department of Computer Science, Aalborg University
Fredrik Bajers Vej 7E, DK-9220 Aalborg Øst, Denmark

Proceedings of the ≪UML≫ 2003 - The Unified Modeling Language,
6th International Conference San Francisco, CA, USA, October 20-24, 2003
Lecture Notes in Computer Science 2863, pages 341-355

This appendix presents the paper: *Modeling and Testing Legacy Data Consistency Requirements* [NJ03], with coauthor Christian S. Jensen; the paper was presented at UML 2003; the paper is important since it unveils research issues targeted in following chapters. The theme of the paper is consistency between legacy data - which includes modeling and testing of consistency requirements. Appendix D and E presents papers that discuss different types of applications, e.g. modeling of accessibility constraints; other papers (Appendix B, C, G) discuss issues related to the design of a tool for specifying and testing consistency requirements; especially Appendix G is important, the theme of that paper is the specification of a generic model for connecting models in a multilevel modeling environment.

A slightly modified version of [NJ03] is presented in the following sections.

A.1 Problem Area

An increasing number of data sources are available on the Internet, many of which offer semantically overlapping data, but based on different schemas, or models. While it is often of interest to integrate such data sources, the lack of consistency among them makes this integration difficult.

The same problem arise when an enterprize adopts a new software system, that system must typically work in a setting with several existing legacy systems. For example, public administrations, in their strive to create one single, public IT infrastructure, may build a new system that integrates previously separate databases that concern different aspects of physical properties (land parcels, buildings, etc.). It is important to the new system that the different representations of the same physical properties are consistent. And the introduction of the new system provides an opportunity to improve the quality of the existing databases.The paper focus on data integration, or more specifically on the consistency problems that occur when previously uncoordinated, but semantically overlapping data sources are being integrated.

A.2 Solution Overview

In our approach UML (e.g., class diagrams) and its accompanying Object Constraint Language (OCL) are used to model and test for consistency. The paper explores different possible modeling techniques and forms a rec-

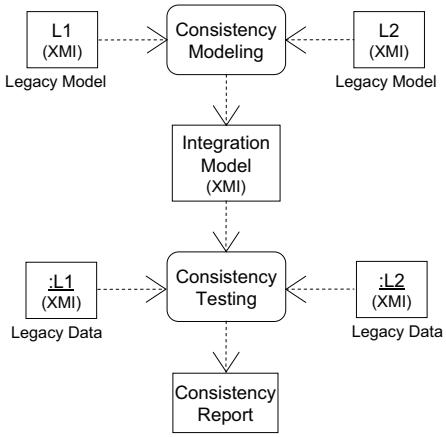

Figure 2: Consistency Modeling Overview

ommendation for how to accomplish consistency modeling and testing. The notion of *something being consistent with something else* can be applied in many contexts. For example, an implementation can be consistent with a model, meaning it is a correct implementation of the model. Our specific context are legacy systems: we consider how to model consistency among data managed by different legacy systems.

Most of today's legacy systems use relational technology for persistent data storage. Having two databases with overlapping models (i.e., parts of their schemas describe the same reality, or miniworld) one consistency rule could be: *two objects with the same identity, modeling the same real-world entity, must have the same values stored for corresponding attributes; otherwise, they are not consistent with each other*. We investigate how to exploit UML in this type of modeling situations and how to perform the actual consistency testing.

Fig. 2 offers an overview of the approach. The models of the legacy systems are UML models represented in the XMI [OMG02] format (meta-model level M1 [OMG03c]). The output of the *consistency modeling* is an *integration model* where the two legacy models have been integrated and the desired consistency has been expressed explicitly. We assumed that the modeling activity was manual.

The paper explores the use of various subsets of UML for the consistency

modeling, and it recommends the use of a particular subset of UML notation together with guidelines for this task. A tool or plug-in may be designed based on this, to give extra support for this modeling approach. However, an ordinary UML tool will do since the recommended notation is a subset of UML. Next, *consistency testing* is done automatically. The paper describes *consistency testing* tailored to one selected consistency modeling technique. The consistency model and legacy data are inputs to the *consistency testing* activity. The legacy data are instances of the legacy models that were integrated in the integration model. The data are represented according to the XMI format (metamodel level M0) and can, e.g., be snapshots of legacy databases. The output of the consistency testing activity is a report describing the consistency violations that were revealed.

Our approach is related to constraint programming [Bar99]: the modeler declares constraints, and the test environment will later find a solution that is consistent with the constraints. We assign values to so-called consistency attributes by evaluating declared constraints. Our proposal also involves so-called consistency associations between legacy classes. For these, the test environment will, in a sense, try to break multiplicity constraints; and if it succeeds, a *wrong cardinality* consistency violation occurs.

Maintaining consistency among different representations of the same entity stored in different databases has been studied before [CW93, FCJ03, RSK91]. We consider a notion of consistency modeling that seems more general than most related work. In comparison with the most related work [FCJ03], we do not rely on an extension of UML (we stay within UML), and our testing is quite different.

An extension to OCL has been proposed [CWD00] with the objective of describing quality-ensuring constraints on geographic data. We do not extend OCL, but instead propose to introduce special associations and classes to support the specification of complex consistency constraints. We believe that this aids in obtaining a very practical approach.

This paper is structured as follows. Sect. A.3 defines concepts, e.g., consistency model and integration model, that are used throughout the paper; a first example of consistency modeling is also introduced. In Sect. A.4, different consistency modeling techniques are described. Sect. A.5 proceeds to discuss the modeling techniques, and one technique is selected as the most useful. This section also covers the automatic testing of such a model. Finally, Sect. A.6 offers a short discussion, conclusions, and directions for further work.

A.3 The Consistency Model

We proceed to define what we mean by consistency model, how it relates to Model-Driven Architecture (MDA) [OMG03a], and the role of the test environment.

A.3.1 The Test Environment

Consistency checking may be challenging in a highly dynamic context: OCL operators such as **forALL** and **allInstances** are hard to implement when objects come and go. The operators will function well if the object structure is static. In particular, a snapshot of a system is static and can easily be used (see [TvS02] and [CL85] regarding the recording of distributed global state).

On the other hand, the restriction to a static context may impose limitations on the use of operation calls (query operations) in OCL expressions. If the operations are completely described in the model (e.g., by the use of action semantics) they may be interpreted at test time.

The test environment may offer some support for the consistency modeling. For example, in relation to geographic information, spatial functions may be part of the test environment. Such functions can then be used freely in OCL expressions. At consistency modeling time, the classes supported by the test environment can be seen as part of a special, dynamic legacy model (thus query operations will work properly). Such support can greatly strengthen the consistency testing.

A.3.2 An Initial Example

Consider a simple case that relates to object replication. The package IM shown in Fig. 3 contains legacy-model elements and additional modeling elements for describing consistency: IM is an example of an integration model, and two legacy models are shown that are stored in package L1 and L2, respectively. The package stereotyped ⟪consistencymodel⟫ is explained in the next section.

The dashed-dotted line between class L1::C1 and L2::C2 indicate that objects of type L1::C1 may be tested for consistency against objects of type L2::C2. The package notation is cumbersome to read, so we omit it in the remainder of the paper, even if this in a strict sense makes some expressions syntactically incorrect.

We should be able to capture the following consistency requirement in

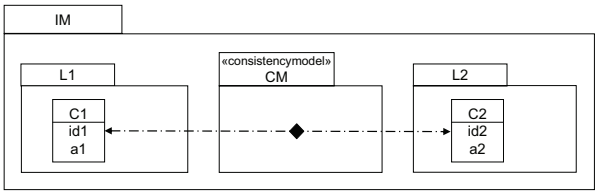

Figure 3: Integration Model Encompassing Legacy Models and a Consistency Model

the integration model: if attribute values :C1.id1 and :C2.id2 are equal, it makes sense to talk about the consistency of objects :C1 and :C2; they are consistent if and only if :C1.a1 is equal to :C2.a2. Using OCL-syntax, we may state this as follows.

Context C1 **inv**:

C2.**allInstances**->**forAll**(self.id1 = id2 **implies** self.a1 = a2)

While it seems possible to do consistency modeling with OCL alone(at least with minor extensions [CWD00], [GKR99]) this will not benefit from the visual strengths of UML.

A.3.3 Consistency Model

We term the part of the integration model that is not part of any legacy model the *consistency model*. The package containing the consistency model is stereotyped ⟪*consistencymodel*⟫, as shown in Fig. 3.

With UML (XMI), we may put modeling elements into separate packages. It is for example possible to put the description of an association into a package separate from the packages of the connected classes. It is up to the modeler to store the consistency model in a separate, stereotyped package.

At *consistency test time*, an instance of the integration model is instantiated. Instances of legacy models are prefabricated and will be inserted as parts of the integration model instance. The test environment then automatically instantiates the consistency model. The consistency model can be seen as a declaration: instances of consistency model elements are in a sense derived from the legacy instances and the declaration.

Let us take a closer look at the OCL expression from the previous section, the core of which can be written as:

e1.id1 = e2.id2 **implies** e1.a1 = e2.a2

where e1 is of type C1 and e2 is of type C2. Conceptually, if e1 and e2 have the same id value, they must have the same attribute value to be consistent; and if their id values are different, consistency is not questioned. This expression is an instance of a more general pattern:

<match> **implies** <consistency test>

The first part of the expression defines *what to test* for consistency, and the last part defines *what is required for consistency* to hold. This separation seems sensible even if the specification of what to test may not be trivial: which attributes are fundamental (Aristotle's Law of Identity)? And what if the values of the identifying attributes are inconsistent? The full power of OCL can be used when defining the matching. Since OCL expressions can include operation-calls (query operations), advanced functional libraries can be applied if they are available in the test environment, e.g., spatial operations in a geographical system.

A.3.4 MDA and the Consistency Model

The notions of Platform-Independent Model (PIM) and Platform-Specific Model (PSM) are central to OMG's MDA initiative; a PIM is a model where (some) technical details have been abstracted away. A PIM may be mapped to a PSM, which is closer to implementation. In short, MDA advocates a software development approach where abstract models are mapped (manually or automatically) to less abstract ones until implementation is achieved. If a legacy system has been developed in accordance with MDA, there will be several models that are candidates for consistency modeling. In this case, a decision of which abstraction level to use for the consistency model has to be made.

For the consistency testing to be correct, the user data must be at the same level as the integration model. Both user data and the integration model may be subject to mapping to achieve this "compatibility." This type of mapping may not be trivial: assume for instance that the integration model consists of two PIM legacy models and a consistency model that contains at least one OCL constraint that references elements in both PIMs; further, assume that the two legacy systems has been implemented on different platforms. If the consistency testing is to be performed at the "implementation level," the mentioned constraint will contain parts that must be mapped to one platform and other parts that must be mapped to another platform.

As a full discussion of this type of mapping is beyond the scope of this paper, we simply assume that the legacy models in question are *implementation models* [FK02] and that the consistency testing is applied to instances of implementation models.

A.4 Modeling Consistency

In a consistency expression, <match> determines whether there is a relation between objects; and if so, <consistency test> decides the state of this relation: *consistent* or *not consistent*. All the presented techniques separate the determination of which objects to test and the actual consistency test. The matching is modeled with binary UML associations, which we term *consistency associations* (c-assoc). To separate consistency associations from other associations, the stereotype ⟪c-assoc⟫ is introduced—if allowed by the UML tool at hand, a dash-dotted line can be used as demonstrated in this paper. A class that is part of the consistency model is called a *consistency class* (c-class, stereotype ⟪c-class⟫). We have selected the following techniques as a starting point for our research:

- use of c-assoc between legacy classes (arbitrary multiplicity)
- use of c-assoc between legacy classes (arbitrary multiplicity), and an association class connected to the c-assoc
- use of c-classes that can be associated with several legacy classes through the use of c-assoc (arbitrary multiplicity on the legacy class end, only 0..1 or 1 on the c-class end)
- use of c-assoc between legacy classes (arbitrary multiplicity) and use of c-classes that can only be connected to one legacy class with an c-assoc (multiplicity is limited to 1 on the legacy class end, 0..1 or 1 on the c-class end)

For all techniques, the c-assoc and the constraint connected to the c-assoc (a missing constraint is the same as having a constraint that is always *true*) play a key role when the consistency model is being instantiated. Logically, all possible instances of a c-assoc are considered when the consistency model is being instantiated. If the constraints on the c-assoc are met, the link is kept (if the c-assoc goes to a c-class, object of this class is created as needed). Note that this instantiation policy yields the consistency model with the maximum number of links. We have one important limitation

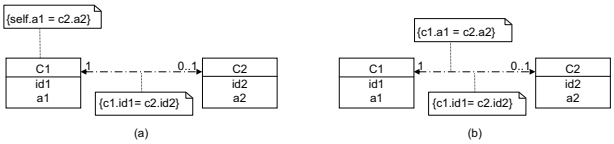

Figure 4: Connection of Constraint to Class (a) and Association (b)

on the constraints: *circular references between constraints must not occur.* Without this restriction several consistency model instances may be possible.

The different modeling techniques we consider will be tested on two examples. The first was given in Sect. A.3.2. The second concerns a situation were one legacy system contains descriptions of apartments and another contains descriptions of buildings—see Fig. 5. The size of the floor space of a building should be the same as the total floor space of its apartments; the number of apartments that is given as an attribute in class Building should be equal to the number of apartments with the same building id (attribute bId). One building should have at least one apartment, and an apartment should belong to exactly one building. This simple example is sufficiently illustrative for our purposes. The following sections gives more details on the selected techniques.

A.4.1 Only Association and Constraints

The UML standard [OMG03c] states: "A constraint is a semantic condition or restriction expressed in text. In the metamodel, a Constraint is a BooleanExpression on an associated ModelElement(s), which must be true for the model to be well formed." It is thus correct to attach a constraint to an association; OCL is not mentioned, and later on, OCL-invariants are only mentioned for classes, types, and stereotypes. But we assume that it is legal to connect a constraint to an association.

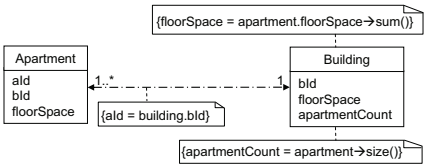

Figure 5: Building with Apartments

Fig. 4(b) shows an example where it is not possible to separate the matching from the actual consistency test. This observation reveals that this technique is not suited. In Fig. 4(a), the matching is connected to the c-assoc, and the consistency test is connected to one of the classes. The multiplicity is also of significance—in this example, an object of type C2 must be attached to an object of type C1 (with matching id); otherwise, there is an inconsistency. The next case is shown in Fig. 5.

As already mentioned the "matching part" is not just an invariant, it is also a production rule. When consistency is to be tested, the consistency associations will be instantiated, the instantiation policy will be to create all links that satisfy the match. If the specified multiplicity is broken (wrong number of links), it represents a consistency violation, which will be reported. Ideally, the matching should discriminate all links that are logically *wrong*, and it should include all the *right* ones. This would be the perfect match. A matching that is not discriminating enough may not be a problem: the links may not be used, or consistency tests that use them may always evaluate to true.

A.4.2 Consistency Modeled with Association Class

This technique extends the technique demonstrated in Fig. 4(b). An association class has been introduced to describe the consistency; Fig. 6 gives an example.

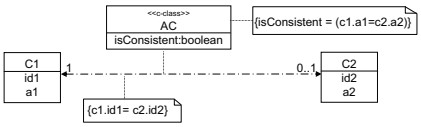

Figure 6: Constraints Connected to An Association Class

The constraint placed on the association corresponds to the matching.

The consistency test is formulated as an OCL expressed invariant on the association class; the attribute isConsistent, termed a consistency attribute, must be true if the considered objects are to be regarded as consistent. In a more complex situation, the consistency check can be structured into several invariants, distributed over several consistency attributes. A consistency attribute can be used in the reporting process, and it can also be referenced in other constraints.

Fig. 7 offers a solution for the second example. One weakness of this solution is that the consistency attributes will be calculated once for each

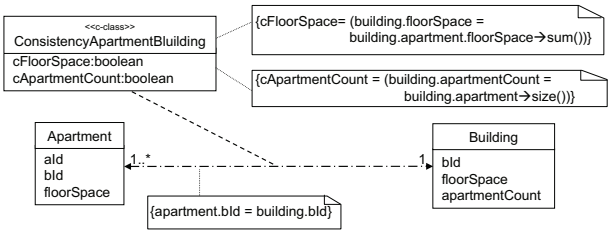

Figure 7: The Apartment / Building Problem Solved with An Association Class

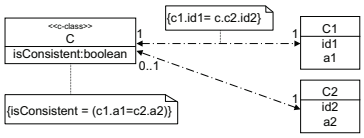

Figure 8: Use of An Ordinary Class Instead of An Association Class

apartment of a building when once for each building would suffice.

A.4.3 Ordinary Class as Consistency Class

In Fig. 8, a c-class connects legacy classes. This technique has many similarities with the one presented by Friis-Christensen and Jensen [FCJ03], where the aim is to integrate multiple representations of the same entity. But there are also differences. Whereas they place matching rules in a separate compartment of the c-class, we express the matching as constraints on the c-assoc.

Fig. 9 concerns the second example. The multiplicity on the apartment side in Fig. 9 demonstrates that several apartments are involved; the weakness found when association classes were used has been eliminated.

Yet another technique is demonstrated in Fig. 10. For each c-class, there is now exactly one c-assoc to a legacy class, and there can also be c-assoc's between legacy classes. It is also possible to put a constraint on the c-assoc from c-class to legacy class. This modeling technique seems to be simple and compact.

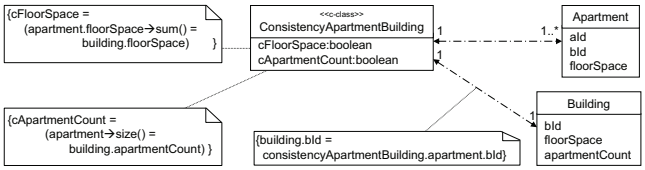

Figure 9: Use of Ordinary Class: The Building / Apartment Example

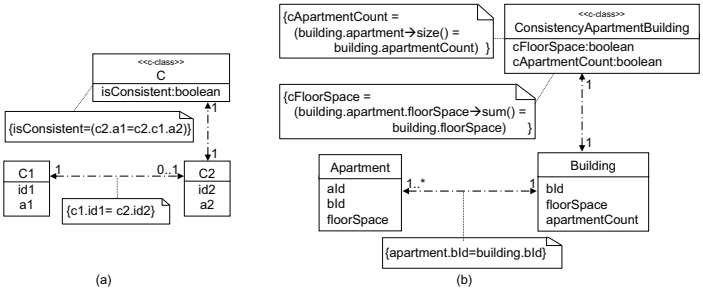

Figure 10: Consistency Classes with Only One Association to Legacy Class

A.5 Selected Solution

Next, we consider which modeling technique to choose and how to consistency check user data.

A.5.1 Which Technique to Choose

Having introduced several modeling techniques, we proceed to compare these in order to understand their relative merits. A more exhaustive study, including a study of combinations of the techniques, is left for future research. The following evaluation criteria are important.

1. How easy is it to apply the technique and comprehend the resulting models?
2. Is it possible to make an interpreter (or compiler) for the models produced?
3. Is it possible to interpret the models efficiently?
4. Can good reports be made?

Criterion 1 We feel that all techniques are simple and rather intuitive; much of the required skill comes down to understanding OCL. But it seems that inserting c-assoc's directly between legacy classes is quite *natural*. It is not possible to avoid complex dependencies. Rather, the best one can do is to express them so that they are easily understood. The use of consistency attributes and c-assocs makes it possible to partition complex OCL constraints into manageable pieces.

Criterion 2 Fulfillment of this point rests upon the possibility of instantiating the consistency model. This instantiation is later demonstrates for one technique; the other modeling techniques do not introduce any new complexity that cannot be solved by some extra mechanism for keeping track of intermediate results.

Criterion 3 A complex consistency model together with large amounts of user data can result in a combinatorial explosion, making the testing impossible in practice. Even if the presented techniques introduce only a limited number of new modeling elements, there will be practical limitations when it comes to the amount of user data to process. The technique that uses

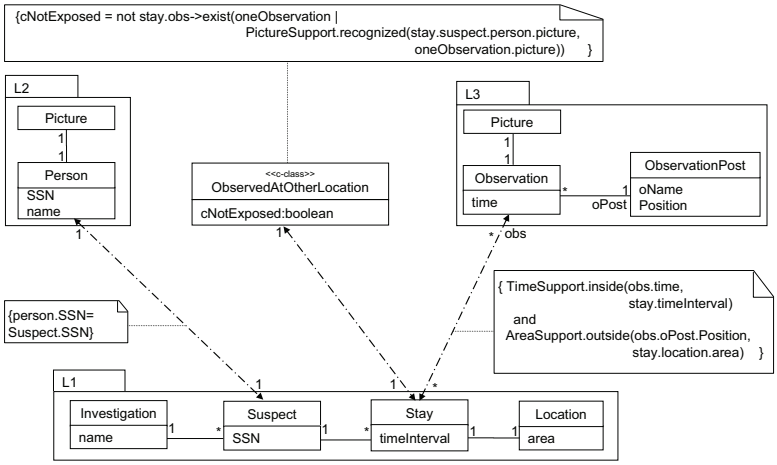

Figure 11: Has the Suspect Been Elsewhere?

association classes seems to produce models that lead to much redundant testing. The last modeling technique allows the c-assoc links to be reused.

Criterion 4 Reporting is not elaborated upon in this paper. However, we note that the consistency attributes may play an important role in the reporting process.

The last modeling technique presented seems to be the best, and it is our choice. But we see the need for gaining more experience with the techniques, and a longer-term goal is to establish a framework that allows us to investigate the merits of the techniques.

A more complex example is given in Fig. 11. Here, legacy system L2 contains pictures of persons. Legacy system L3 records observations done at different observation posts. Legacy system L1 contains information about police investigations. Instantiation and checking of the consistency model will expose a person who claims to have been one place, but has been observed in another place at the same time.

Classes TimeSupport, AreaSupport, and PictureSupport are part of the test environment; the operations of these classes that are used are all class scoped.

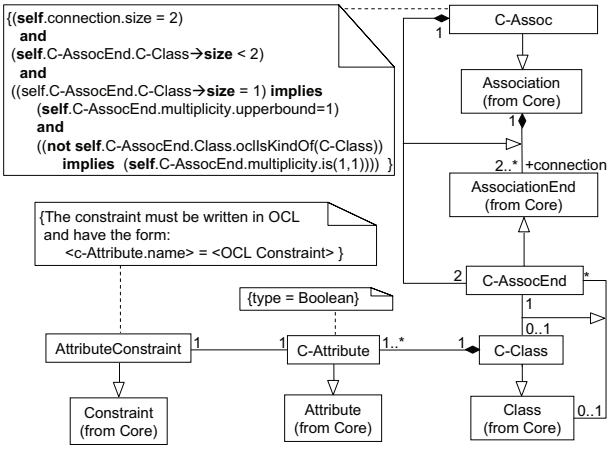

Figure 12: Metamodel

A.5.2 Consistency Testing Using the Chosen Technique

Fig. 12 presents a metamodel that illustrates which modeling elements to use and the constraints on them. In practice, the modeler must conform to the metamodel if the test is to be executed correctly. The metamodel also gives meaning to the stereotypes c-assoc and c-class.

We note that the c-assoc constraint is not shown in the metamodel, but it is important and will be described shortly. We proceed to explore the two main uses of the c-assoc:

1. It associates two legacy classes.
2. It associates a c-class and a legacy class.

Use of c-assoc's with legacy classes Legacy classes may belong to the same or different legacy systems. The c-assoc is similar to an ordinary association. Similarities include that role names can be inserted and that multiplicities can be selected freely. The only difference compared with an ordinary association is the c-assoc constraint: this constraint is a bit "more" than an ordinary association constraint. Like an ordinary constraint, it must be true for all links instantiated from the corresponding c-assoc; but,

in addition, the c-assoc constraint is essential when instantiation of the c-assoc takes place. All possible links between objects of the two legacy classes are considered: the links that do not meet the c-assoc constraint are rejected; the rest are kept. If the number of links for an object do not meet the specified multiplicity, the cardinality is erroneous, and a consistency violation is reported. If a c-assoc constraint is absent, it will be the same as a c-assoc that is always true.

Use of c-assoc's with legacy and c-classes When a c-assoc associates a c-class and a legacy class, the multiplicity is "1" on the legacy class side and "1" or "0..1" on the c-class side.

The consistency checks are connected to a c-class through the consistency attributes. There can be many consistency attributes for each c-class. The value of a consistency attribute is given by the corresponding attribute constraint, which is of the form:

<c-Attribute.name> = <OCL constraint>

The <c-Attribute.name> is the name of the corresponding consistency attribute. The attribute constraint will always be true (as an invariant should). In addition, it is used when the attribute obtains its value. The value of the <OCL constraint> will be calculated, true or false, and the consistency attribute is given this value.

The association constraint connected to this c-assoc plays a role when instantiating the c-assoc and the c-class. Given an object of the legacy class, an object of the c-class will be created if the association constraint is met. If the multiplicity is "1" on the c-class side and no c-class object can be created because the constraint on the c-assoc cannot be meet, the cardinality is wrong, and a consistency violation is reported.

Cyclic References A c-assoc may be referenced in attribute constraints and other c-assoc constraints. A constraint attribute may be referenced in attribute constraints that are attached to other c-classes (this is actually slightly too strict) and in c-assoc constraints. However, there is one limitation: the mentioned references must not be circular.

Consistency Model Instantiation While most of the logic has already been described, one question remains, namely that of instantiation order. Since there can be references between the elements of the consistency model, this order cannot be arbitrary. The order can be decided by building a

dependency graph (which will typically permit several orders). The nodes of the graph are obtained as follows.

1. A c-assoc that associates two legacy classes together with its constraint defines one node (node type 1).

2. An attribute constraint together with the c-assoc that associates its c-class and a legacy class defines a node (node type 2). If there are several attribute constraint for a class, there will be several nodes concerning the same class.

The edges arise from the navigations through c-assoc's and the references to the consistency attributes. If a node has a navigation that uses the c-assoc of another node, then there will be an edge from the first node to the second. Also, if one node has references to a constraint attribute (node of type 2) then there will be an edge from the first node to the second.

Since there are no cyclic references, the graph will be an acyclic directed graph. The instantiation can be done by selecting a node that fulfills the following: it has no edges pointing to nodes that have not already been instantiated. All instances of the selected node are created (e.g., if the node represents a c-assoc, all links with fulfilled constraints will be created).

The whole consistency model has been instantiated when there are no more nodes to instantiate. Use of the dependency graph ensures that all elements referenced in a constraint are present at instantiation time.

Instantiation of the first kind of nodes has already been explained. When instantiation of a node of type 2 is carried out, all legacy objects of the "right" kind are considered. The instantiation will occur in two ways:

- The c-class and corresponding c-assoc have not yet been instantiated. If the c-assoc constraint is met, an object of the c-class is created and linked to the legacy object by an instance of the c-assoc. The value of the consistency attribute is calculated, and the attribute is given this value. At this point, an object of the c-class has been created and linked to the legacy object; one consistency attribute has obtained its value.

- The c-class and corresponding c-assoc have been instantiated. The value of the consistency attribute is calculated, and the attribute is given this value.

A.5.3 Interpretation of OCL Expressions

The checking and evaluation of OCL expressions is done by an OCL interpreter. Building an interpreter or adapting an existing one for our purposes is achievable, as the object structure is stable and *ordinary*. An example of a rather similar application is found in the UML Specification Environment (USE) [GR02], where expressions written in OCL are used to specify integrity constraints on class diagrams. A model can be animated to validate the specifications; snapshots can be taken, and for each snapshot, the OCL constraints are checked automatically. Other tools for OCL includes the Dresden OCL Toolkit [HDF00] and OCLE [Com].

A.6 Summary and Research Directions

This paper has demonstrated how the full power of OCL as a declarative language can come to play in a setting where the consistency of semantically overlapping data sources is to be specified and checked. The proposed modeling technique is based on standard OCL and a small subset of UML's visual modeling elements. It is possible to use an ordinary UML tool to model the consistency. A consistency test environment is described. The use of XMI enables the integration of models and data of quite different origins; for this to happen, conversion to XMI must take place. The framework described needs a set of XMI-conversion tools to be in place; relation database schemas can easily be mapped to UML models, and it does not seem to be difficult to convert data stored in relational databases to XMI.

Because the amount of XML-data is growing rapidly, an investigation of how XML schemas and data fit into this framework is of great interest. The roles of ontologies [BKK+01] and the semantic web [KCH+01] have yet to be investigated; automatic generation of consistency models may be an option.

Appendix B

Towards a Data Consistency Modeling and Testing Framework for MOF Defined Languages

Jan Pettersen Nytun[1,2], Christian S. Jensen[1,3] and Vladimir A. Oleshchuk[1]

[1]Faculty of Engineering, Agder University College
Grooseveien 36, N-4876 Grimstad, Norway
[2]Department of Informatics, University of Oslo
P.O.Box 1080 Blindern, N-0316 Oslo, Norway
[3]Department of Computer Science, Aalborg University
Fredrik Bajers Vej 7E, DK-9220 Aalborg Øst, Denmark

Proceedings of Norsk Informatikkonferanse NIK'2003
Institutt for informatikk, Universitetet i Oslo, Norway, November 24-26, 2003

This appendix presents the paper: *Towards a Data Consistency Modeling and Testing Framework for MOF Defined Languages* [NJO03], with coauthors Christian S. Jensen and Vladimir A. Oleshchuk.

The paper can be seen as a continuation of Appendix A [NJ03]; it focuses on the architecture of a framework for integration of models.

A slightly modified version of the [NJO03] is presented below:

The number of online data sources is continuously increasing, and related data are often available from several sources. However accessing data from multiple sources is hindered by the use of different languages and schemas at the sources, as well as by inconsistencies among the data. There is thus a growing need for tools that enable the testing of consistency among data from different sources.

This paper puts forward the concept of a framework, that supports the integration of UML models and ontologies written in languages such as the W3C Web Ontology Language (OWL). The framework will be based on the Meta Object Facility (MOF); a MOF metamodel (e.g. a metamodel for OWL) can be input as a specification, the framework will then allow the user to instantiate the specified metamodel.

Consistencies requirements are specified using a special modeling technique that is characterized by its use of special `Boolean` class attributes, termed consistency attributes, to which OCL expressions are attached. The framework makes it possible to exercise the modeling technique on two or more legacy models and in this way specify consistency between models. Output of the consistency modeling is called an integration model which consist of the legacy models and the consistency model. The resulting integration model enables the testing of consistency between instances of legacy models; the consistency model is automatically instantiated and the consistency attribute values that are false indicates inconsistencies.

B.1 Introduction

The Semantic Web [BLHL01] aims at giving well-defined meaning to web content, in this way allowing automatic reasoning about, and processing of, web content. An important aspect of supporting this is the provisioning of appropriate knowledge representation [Sow00] languages, which remains an active area of research. Prominent examples of such languages include the Resource Description Framework, RDF [BG], the DAML+OIL [HPSvH02] language, which integrates the US DARPA Agent Markup Language and

the European OIL effort and is an extension of the RDF Schema, and DAML+OIL's successor, the World Wide Web Consortium's Web Ontology Language (OWL) [BG03].

Somewhat unrelated to this, the Unified Modeling Language (UML) is being used widely for conceptual modeling in the development of software systems. It may be noted that UML has substantial semantic overlap with knowledge representation languages such as those just mentioned, although there are also differences [OMG03e, KCH+01].

The Object Management Group (OMG) recently issued a request for proposals [OMG03e] that seeks:

- A standard, Meta Object Facility (MOF) 2.0 [OMG03h] compliant metamodel for Ontology Definition (ODM).
- A UML 2.0 [OMG03d] (UML for short) Profile that supports reuse of UML notation for ontology definition.
- A mapping from the ODM to the profile.

The OMG request also seeks a language mapping for the ODM to OWL. There are good reasons to reuse the UML notation for ontology definition [BKK+01]. For example, the graphical notation of UML is well tested and tools exist that support UML.

There is a trend towards the use of languages that are tailored for special problem domains and also towards integration of different languages (as indicated by the latest request for proposals from OMG); in an OMG context this can be done by using the extension mechanisms of UML, definitions of UML profiles, and also definition of new MOF metamodels. Tools that support definition and application of this type of languages are largely missing.

This paper takes the first steps towards defining a framework for experimenting with the integration of UML and knowledge representation languages. The framework should contain components that can be assembled to form different tools.

Our selected application is consistency testing of user data; the objective is to ensure consistency among semantically related data, but with different models (schemas) that might have been expressed in different languages like UML and OWL. For data sources with semantically related models, one simple consistency rule could be: *two objects (entities) with the same identity must have the same values stored for corresponding attributes; otherwise, they are not consistent with each other* (e.g., one data source claims that

Bob has income of 10.000 an another lists earnings of 50.000). Figure 13 offers an overview of our approach to consistency testing of user data.

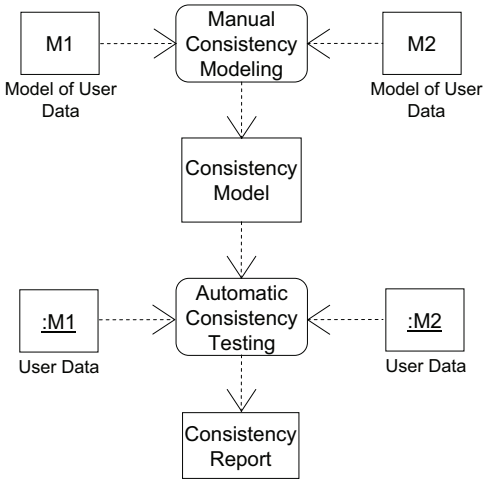

Figure 13: Consistency Modeling Overview

Given models `M1` and `M2` for two data sources, a consistency model is defined manually. The consistency model explicitly states constraints that must be fulfilled in order for instances of the two models to be consistent with each other. The consistency is defined at the "model level"; automatic consistency testing is done on the user data with help of the consistency model (which is instantiated automatically). The user data, depicted as `:M1` and `:M2`, must be instances of model `M1` and model `M2`, respectively. As can be seen from Figure 13, we need a *consistency modeling tool* and a *consistency testing tool*.

This main body of this paper offers a more detailed description of the consistency modeling and testing in a pure UML context, a more complex example would involve usage of both UML and OWL in defining the consistency model.

Some metamodeling tools are available in the literature [Inc03, Met06]. The consistency modeling and testing approach espoused in this paper is based on the results presented in [NJ03].

This paper is structured as follows. Section B.2 specifies what the frame-

work should support together with preliminary design considerations. In Section B.3, a consistency modeling technique is described in the context of UML; and the section also briefly describes how consistency testing of user data can be performed. Finally, Section B.4 offers a short summary and conclusions.

B.2 Data Integration Framework

In this section, we first describe the OMG metamodel architecture and how to represent user data and model. Then a non-exhaustive list of requirements to the framework is given, and finally initial design and implementation considerations are presented.

B.2.1 Use of the OMG Meta-Model Architecture

The OMG advocates a four-layer metamodel architecture [OMG03c] where MOF constitute the top level (level M3). The UML metamodel resides on the next highest level (level M2) and can be seen as an instance of MOF. When system's developers design a model using UML (level M1), the developers instantiate the metaclasses of the UML metamodel. In our context, only the small subset of the UML metaclasses that typically get instantiated on class diagrams are of interest. The run-time instances (user data) are found at the lowest level (level M0). The user-defined model has been instantiated to obtain these instances.

The OMG recommendations [OMG03e] state that the Ontology Definition Metamodel (ODM) should be an instance of MOF; this places the ODM at the same level as the UML metamodel—see Figure 14(a).

There is a semantic overlap between the UML metamodel and the ODM, but they are not subsets of one another, and a combination of the two might be worth investigating. Figure 14(b) illustrates a situation where the UML metamodel and the ODM are combined.

Our aim is to establish a framework where different types of languages and mixtures of languages can be investigated. The focus will be on languages that are defined by metamodels or, more specifically, MOF defined languages. If successfully implemented, the framework might be characterized as a framework for integration of MOF-based languages.

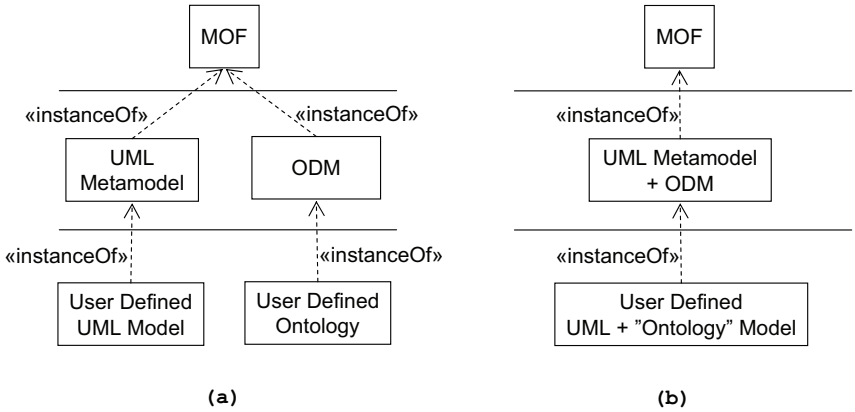

Figure 14: ODM Relative to the Metamodel Architecture

B.2.2 Representation of Model and Model Instance

UML 2.0 introduces the metaclass `InstanceSpecification`, which can be used to model an instance of another model element. An instance of `InstanceSpecification` can for example be used to illustrate an instance of a class (an object) or an instance of an association (a link between objects). As a concrete example, given a class `Person` (an instance of metaclass `Class`), `InstanceSpecification` can be instantiated to illustrate an instance of class `Person`; this is sometimes referred to as a snapshot (a run-time instance at a specific time) of the object. An `InstanceSpecification` will have a reference to the classifier that is the classifier of the represented instance. Consequently, it is possible to have a model (at level M1) that describes both a snapshot of user data and the corresponding metadata. When the user defines the consistency model, only metadata (models) matters; when the consistency testing is performed, both data and metadata must be present.

XML Metadata Interchange (XMI) [OMG02] is an interchange format that can be used on models/data from all the four levels of the OMG metamodel architecture; XMI is a natural choice when it comes to storing and exchanging models and data.

The proposed framework should be able to visualize instances of models, e.g., visually pinpoint inconsistencies exposed by the consistency testing. The mentioned use of `InstanceSpecification` will make this possible.

B.2.3 What the Framework Should Support

The list that follows briefly states central functionality expected from the proposed framework.

- Support definition of MOF metamodels, e.g., guide the combination of two metamodels and resolve possible conflicts. The definition of MOF metamodels can also be done using tools such as UML2MOF [net03b], which transforms UML models (conforming to UML Profile for MOF) into MOF metamodels. Also, standard tools from IBM [Rat03] have plug-ins that allow this.
- Offer users the possibility to load an MOF-specified metamodel.
- Offer users the possibility to instantiate the loaded metamodel. For example, if the loaded metamodel is the UML metamodel then the user is given the possibility to make UML models (which is done by instantiating the loaded UML metamodel); or if the loaded metamodel is ODM, the user is given the possibility to define ontologies.
- Import and export of models based on XMI and the UML Diagram Interchange Specification [BJMF02].
- A UML model is typically represented as an instance of the UML metamodel, but an SQL schema is not. Transformation of an SQL schema to an UML model is rather straightforward. A transformation that is even more likely to be necessary concerns the user data that have to be represented as instances of metaclass `InstanceSpecification`.
- Specific features that support the modeling of consistency and automatic consistency testing. Section B.3 offers more detail.

B.2.4 Implementation of the Framework

The implementation will be a set of components that can be assembled to form different tools. Figure 15 shows a set of interconnected components that together form a modeling tool. For example, if the component named *Metamodel Defined with MOF* is the ODM, the component named *MOF Based Modeling Tool* will give the user the ability to define an ontology that is made persistent with the help of the *Model Repository* component. Since modeling is to be done visually, the tool needs to know how to display the specific ontology elements. The *Concrete Syntax Definition* component will support this, although how this is to be achieved has yet to be investigated.

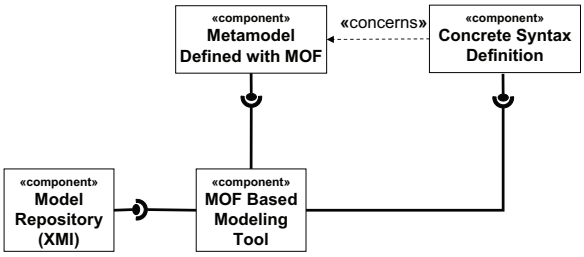

Figure 15: A UML Component Diagram Showing a General MOF Based Modeling Tool

An important implication of the framework being based on the four-layer metamodel architecture is that formal Meta Object Facility descriptions of abstract syntaxes must be a understood; this understanding is incorporated (hard coded) into the the component named *MOF Based Modeling Tool*. Figure 15 only offers an abstract picture, and further investigation is necessary.

The Eclipse Platform [Dri01,DFK+03] is designed for building integrated development environments; it has a plug-in architecture that makes it suitable for extensions, and several useful plug-ins are already present. The UML tool Rational XDE from IBM [Rat03] is built on the Eclipse Platform. Our framework could be built by making the right plug-ins for the Eclipse Platform. NetBeans IDE [net03a] is a similar framework and is also a candidate for use in implementing such frameworks.

B.3 Example Application of the Modeling Framework

Our consistency modeling and testing approach is presented in [NJ03]. This section presents an example and propose a component architecture to achieve the desired functionality.

Figure 13 offers an overview of our approach. The consistency modeling is to be done with the Object Constraint Language (OCL) [OMG03c] and a selected subset of UML modeling elements:

- Association
- OCL constraint
- Association class
- Class
- Class attribute

The output of the *consistency modeling* is an *integration model* where the two legacy (in this context, "legacy" means "pre-existing") models (`M1` and `M2`) have been integrated and the desired consistency has been expressed explicitly. We term the part of the integration model that is not part of any legacy model the *consistency model*—see Figure 16.

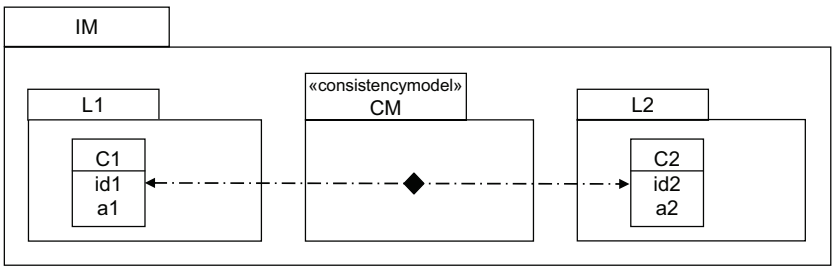

Figure 16: Integration Model Encompassing Legacy Models and a Consistency Model

We assume that the modeling activity is manual. Next, *consistency testing* is done automatically with the consistency model and legacy data (`:M1` and `:M2`) as inputs. Some processing of the user data might be necessary since they are to be represented with the help of metaclass `InstanceSpecification`. The output of the consistency testing activity is a report describing the consistency violations that were encountered.

B.3.1 Consistency Modeling Example

Figure 17 visualize three legacy models. Legacy model `M2` relates pictures to persons, legacy model M3 concerns observations done at different obser-

vation posts and legacy model `M1` concerns information about police investigations.

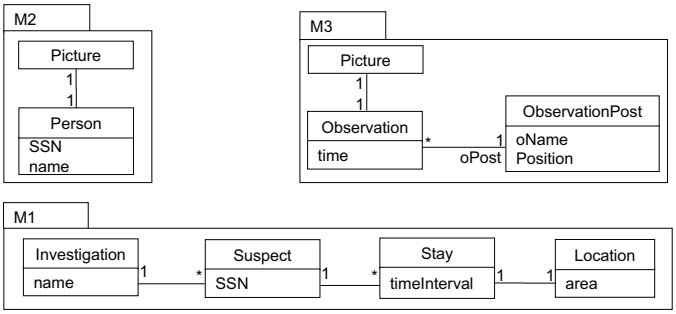

Figure 17: Three Legacy Models

From the perspective of the police, the following question is of interest:: *Has the suspect lied about his whereabouts?* A suspect is exposed as lying if he claims to have been in one place, but has been observed at the same time from an observation post located elsewhere..

In Figure 18, a consistency model has been inserted. The dashed-dotted line between class `Person` and class `Suspect` represents an association—we term it a *consistency association*. This association is used for linking a suspect with a picture of the suspect. The OCL constraint attached to the association: `person.SSN = Suspect.SSN` ensures that an object of class `Suspect` can only be linked to a correct object of type `Person` (the two objects must represent the same person).

Assume that classes `TimeSupport`, `AreaSupport`, and `PictureSupport` are part of the framework; the operations of these classes are class scoped and can be used inside OCL expressions.

The class `Stay` is used to record where a suspect claims to have been during a specific time interval. The consistency association between `Stay` and `Observation` is used to link a "stay" with observations done at the same time at different locations.

Class `ObservedAtOtherLocation` is a *consistency class* (stereotyped as `c-class`). The model prescribes that each `Stay` object must be linked to an `ObservedAtOtherLocation` object (multiplicity one-to-one). The constraint on the attribute `cNotExposed` prescribe the value `true` if no inconsistency has been exposed regarding where the suspect claims to have been and

observations done; if there is an inconsistency then `cNotExposed` must be `false`. Attribute `cNotExposed` is an example of what we call a *consistency attribute*.

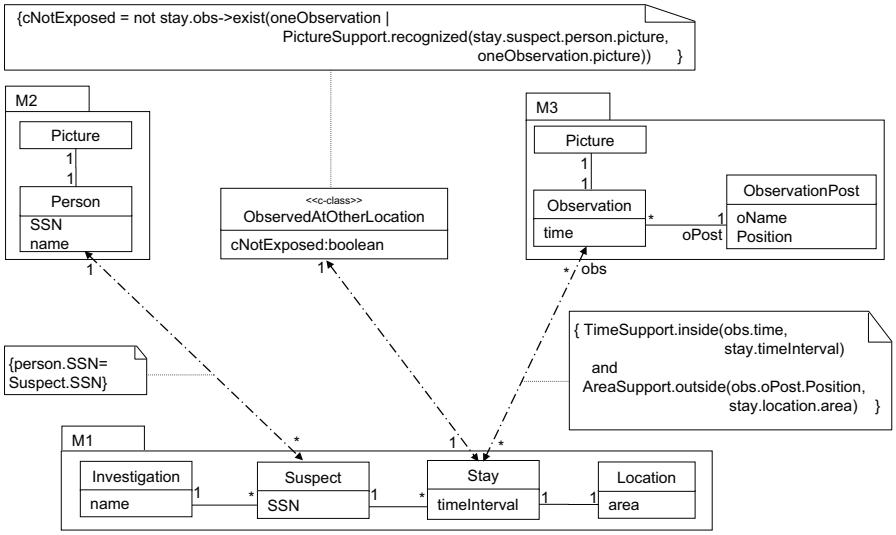

Figure 18: Consistency Model: Has the Suspect Been Somewhere Else?

The integration model can be made with an ordinary UML tool (except for the use of the dashed-dotted line, a stereotype `c-association` might be used instead).

At *consistency test time* an instance of the integration model will be instantiated. Instances of legacy models (the user data) are prefabricated and will be inserted as parts of the integration model instance. The test tool then automatically instantiates the consistency model. The consistency model can be seen as a declaration: instances of consistency model elements are in a sense derived from the legacy instances and the declaration. The constraint `person.SSN = Suspect.SSN` can function as a sort of production rule: for each pair of a `Person` object and a `Suspect` object, the constraint can be evaluated; and if it is fulfilled, a link can automatically be created. The rest of the consistency model can be instantiated in the same way.

A closer look at the constraint on `cNotExposed` reveals navigations through all the consistency associations. As a consequence, instantiation of this at-

tribute must be performed last. The constraints, the consistency associations, and the attributes of the consistency classes must not be dependent on each other in a cyclic way—if they are, it might not be possible to do the automatic instantiation. The order of instantiation can be decided by building a dependency graph, see reference [NJ03] for details.

The attributes of the consistency classes are used when the consistency report is generated, e.g. if `cNotExposed` equals `false` then there is a consistency violation.

B.3.2 Consistency Test Tool

A preliminary design of the consistency test tool is presented in Figure 19. The component named *UML Model and Data Repository* provides the integration model and the legacy data.

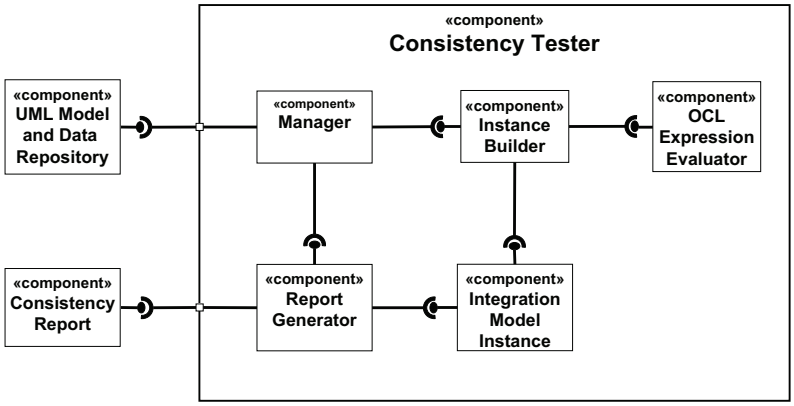

Figure 19: UML Component Diagram Showing the Consistency Tester

Looking inside the *Consistency Tester* component, we find the subcomponent *Instance Builder* that builds the consistency model instance which is represented by subcomponent *Integration Model Instance*. To build the instance, evaluation of OCL-expressions are necessary; this is done with the help of the *OCL Expression Evaluator* subcomponent.

The subcomponent *Report Generator* uses the subcomponent *Integration Model Instance* and produces a consistency report.

As mentioned above the integration model can be made with an ordinary UML tool, but an ordinary UML tool will allow cyclic references; a more sophisticated tool could prevent this. Obviously the framework presented above (section B.2) is a candidate for making such a tool. In [NJ03] a metamodel for consistency models is proposed. This metamodel could be input to the general MOF-based modeling tool presented above.

B.4 Summary and Conclusions

There is momentum in industry and academia towards the integration of UML and knowledge representation languages. A recent request for proposals issued by the Object Management Group is a clear indication of this (e.g. [OMG03e] and [OMG03f]). We have started the development of a tool (or framework) that will support such an integration: if successfully developed, the tool can be used to define diagrams that simultaneously incorporate both UML and "ontology features." The tool is meant as a MOF metamodeling tool, meaning that a MOF metamodel can be input as a specification. The tool then allows the user to instantiate the specified metamodel.

This paper has demonstrated how the full power of OCL as a declarative language can come to play in a setting where the consistency of partially overlapping data sources is to be specified and checked. The modeling technique proposed in the paper for consistency specification is based on standard OCL and a small subset of UML's graphical modeling notation. In future work, the reasoning possibility typically offered by knowledge representation languages will be included.

The proposed tool may be seen as representing a step in the direction towards the creation of a language that possesses the full power of UML and knowledge representation languages. The described consistency modeling and testing, which is an application of the framework will function as a practical demonstration.

Appendix C

Representation of Levels and Instantiation in a Metamodelling Environment

Jan Pettersen Nytun[1,2], Andreas Prinz[1] and Andreas Kunert[3]

[1]Faculty of Engineering, Agder University College
Grooseveien 36, N-4876 Grimstad, Norway
[2]Department of Informatics, University of Oslo
P.O.Box 1080 Blindern, N-0316 Oslo, Norway
[3]Department of Computer Science, Humboldt Universität zu Berlin
Unter den Linden 6, 10099 Berlin, Germany

Proceedings of NWUML'2004, 2nd Nordic Workshop on the Unified Modeling Language, pages 1-17
August 19-20, 2004, Turku, Finland, TUCS General Publication No 35, AUgust 2004
Jan Pettersen Nytun was not involved in the creation of Section C.5 and Andreas Kunert was only involved in the creation of this same section (Section C.5)

This appendix presents the paper: *Representation of Levels and Instantiation in a Metamodeling Environment* [NPK04], with coauthor Andreas Prinz and Andreas Kunert.

The paper presented in the previous appendix (Appendix B) discussed how to represent metalevel instances, this paper continues the discussion and propose a technique for handling instantiation. The paper is presented below:

In the scope of meta-modeling it is important to consider descriptions sometimes as a model and sometimes as a metamodel, e.g. the UML metamodel which is a metamodel for UML and at the same time a model in terms of MOF. For this handling to be easy, this article describes a metalevel representation that includes both aspects. It covers the most important relations within the MOF framework starting with plain objects and relations. A prominent role plays the instantiation, which is the only connection between levels. The article explains how instantiation is represented and which kind of constraints are related to it. This way it is possible to bring metamodeling to its basic semantics.

C.1 Introduction

There is a trend towards the use of languages that are tailored for special problem domains and also towards integration of different languages; in an OMG context this can be done by using the extension mechanisms of UML, definitions of UML profiles, and also definition of new MOF [OMG03h] metamodels. Tools that support definition and application of this type of languages are largely missing. Some of the goals of the SMILE (Semantic Meta-model-based Integrated Language Environment) project are to design and implement a framework that can be used to do metamodeling, integration of different languages and even support for the definition of new meta-metamodels different from MOF.

Our SMILE project targets all the levels of the OMG four-layer metamodel architecture, and we are proposing a basic representation to be used on all levels. We call our representation FORM after its main construct `Form` [3]. The term *carrier* from the telecommunication field has many similarities with our representation since all types of objects and classes are to be carried (represented) with it.

[3] We also consider FORM to mean "Fantastic Organized Representation of Models".

All levels, also the lowest level, will be covered in our basic representation; this lays the ground for executable UML with "metadata facilities" and it also makes it possible to specify how two neighboring levels fit together as described later.

As an example we discuss how FORM can be used to "carry" the different levels of the metamodeling architecture of UML. Afterwards we consider how instantiation relates to FORM.

Except for instantiation, semantics is not considered, but will be dealt with at a later stage. Even without full support for semantics the framework has interesting applications like analyzing static data (e.g. a snapshot of a running system) and check if an instance is consistent with a given model. To do these types of applications we would also need OCL. Relating OCL and XMI with FORM is briefly discussed.

This paper is structured as follows. Section C.2 gives an introduction to metamodeling and representation of metadata, followed by an overview of related work in Section C.3. We introduce our FORM representation in Section C.4. In Section C.5 we discuss why this simplified representation is able to cover all the things already within XMI, hence covering all that is necessary for models. We define the semantics of instantiation within Section C.6 and offer some conclusions and directions for further work in Section C.7.

C.2 Metamodeling

Today the metamodeling approach is a common way of organizing models, a way that involves descriptions on levels placed on top of each other. The concepts of one level have corresponding descriptions on a next level (metalevel, level above). Stated differently, a level is a model and the level below is an instance of this model. In relation to object-oriented programming the lowest level contains the objects of a running system, while the classes reside on the next lowest level. Traditionally, the BNF notation has been used to describe a programming language, this would be the next level. The top level would be a definition of BNF[4]. These 4 levels correspond to the 4 levels of the four-layer metamodel architecture of OMG, but here visual UML models are used instead of BNF. See Fig. 20 for an overview.

The advocated architecture is based on *strict metamodeling* which means that all elements on one level are instantiated from the level directly above. It seems natural to view the *instantiation logic* as operating with 3 levels,

[4]Please note that we do not need a level above BNF, because BNF can describe itself.

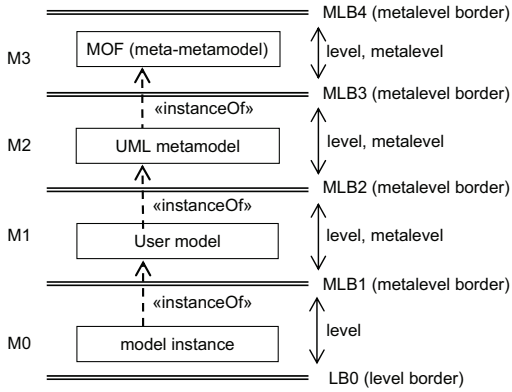

Figure 20: The four-layer metamodel architecture

e.g. you have a description of what a class is, you have a class (e.g. class Person) and you have an object. The definition of the class concept must of course be done with the help of yet another level.

MOF and UML do both offer support for object-oriented concepts and the core parts of MOF and UML are structurally equivalent. Since MOF is used to define itself, the level above MOF (`M4`) can be seen as MOF once more; one can imagine an infinite number of MOF levels - we have *infinite regression*. MOF is defined by self referencing, "ending up with" class `Class` which is an instance of itself.

The syntax of UML has been described in a notation independent way; this *abstract syntax* defines the elements of UML and how they relate to each other. There is also an agreement on the *concrete syntax* of UML. The concrete syntax of a simple class is a rectangle with the class name inside the top compartment and optional compartments for attributes and operations. An object is described as a rectangle with a top compartment containing an optional object name and then after a colon its class, the complete text is underlined (e.g. `Bob:Person`); an object can also have a second compartment for slots with values (attribute values).

You can only display modeling elements by using concrete representation; the abstract syntax is usually defined with class diagrams together with OCL [OMG03d] constraints. An instance of the UML metamodel can be shown as an object diagram or as a class diagram; the class diagram being a visual interpretation of the underlying object-graph. In a sense the class

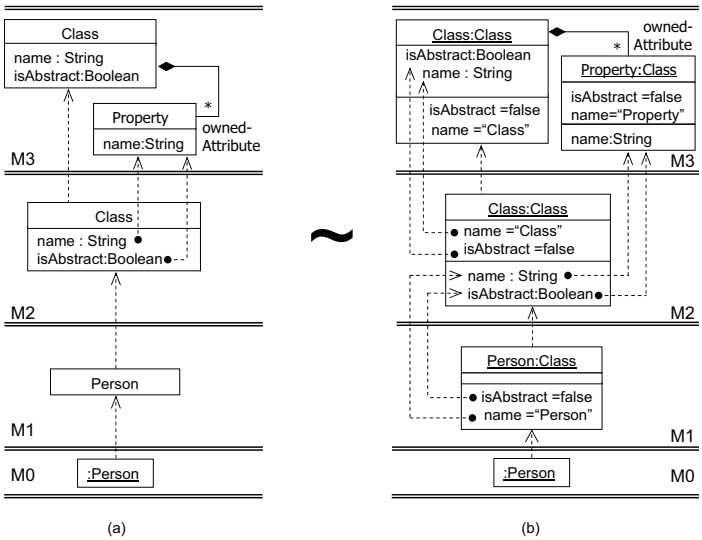

Figure 21: Different representations of the same structure.

diagram notation is using syntactic sugar, e.g. an attribute is shown inside a compartment and not as an instance of `Property`. A class diagram is a natural choice when you see a level from the level below, while an object diagram shows how the level has been instantiated from the level above.

A part of the MOF (abstract syntax) is shown in Fig. 21(a) on level M3, it deals with the structure of classes and states that a class can own attributes (properties). Fig. 21(a) shows how the abstract syntax has been instantiated to form the UML concept `Class` with attributes `name` and `isAbstract` on level M2; the class `Class` on level M2 is an instance of class `Class` of level `M3` and the attributes are instances of class `Property`. Similarly, the class `Person` on level M1 is an instance of the UML concept of `Class`, and on level M0 there is even an instance of the class `Person`. Fig. 21(b) show the same with *clabjects*. In [AK02, AK00b] an entity with class and object nature is called a clabject, in fact *all classes are clabjects*. The clabject-notation allows you to see the attributes and the slots of a class, this can be shown in separate compartments as done in the figure. All the classes of Fig. 21(a) are represented as clabjects in Fig. 21(b); Fig. 21(b) is more explicit: for instance the `name` attribute of class `Class` on level M3 has been instantiated to a slot with value `‘‘Class"` on level M2, class `Class` on level M2 can be instantiated since slot `isAbstract` has value `false`, the attribute `name` is an instance of `Property`.

We can make this representation even more explicit by also showing the relations between the classes and their attributes as in Fig. 22. Here we have shown all attributes as separate `Property` objects being associated with their defining class. Please note that for full completeness we also would have to insert a definition of the association between `Class` and `Property` on level `M3`[5].

C.3 Related Work

Metamodeling has been discussed for a long time, to mention a few articles: [AK02] describes the unification of the class and object facets (clabject); the same article also presents an elegant enhancement of the instantiation mechanism to allow definitions to transcend multiple levels; [CEK00] aims at improving the metamodeling architecture of UML.

[5]The definition would be like: `Class:Class` (already on `M2`) linked to `:Property` linked to `:Association` linked to `:Property` linked to `Property:Class` (already on `M2`). Both the links on `M2` could now have an `instanceOf` relation to `:Association` on level `M3`. For the sake of understandability we have omitted these parts on the figure.

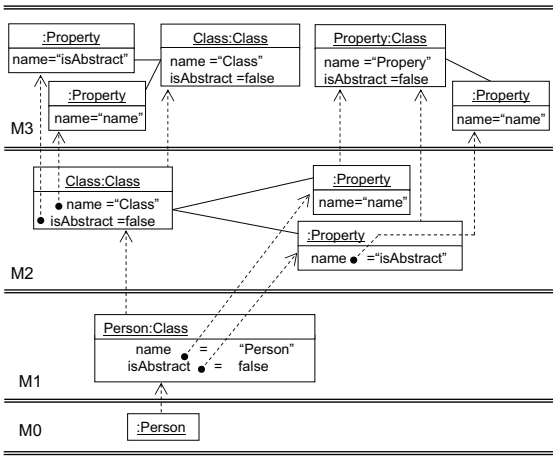

Figure 22: Representing the same underlying structure as Fig. 21, but more explicit.

There are several metamodeling repositories based on the Java Metadata Interface (JMI) [Jav02] specification. JMI is based on MOF and makes it possible to create and access metadata; this can be done at design time or runtime by using the reflective JMI API or via a set of generated metamodel-specific APIs. A tool like Metadata Repository (MDR) [Mat] can import a model represented in XMI and automatically produce JMI interfaces for accessing the metadata and also automatically provide implementation of the JMI interfaces.

The Eclipse platform [Dri01] is designed for building integrated development environments (IDEs). It provides already a working IDE for Java development and there are plug-ins for drawing UML diagrams. The Eclipse Modeling Framework (EMF) [Ecl04] includes a metamodel (Ecore) that is different from MOF; there is an EMF-based implementation of the UML 2.0 [Ecl07] metamodel. The functionality of EMF is quite similar to MDR.

All the mentioned Java-based approaches are using the class/interface concept of Java to implement the metaclasses, and the semantics are not handled separately as we plan to do.

The Coral modeling Framework [Iva02] is not based on Java but on the Python programming language; the meta-metamodel (MOF) is hard coded but different metamodels can be installed automatically; there is currently

no support for the lowest level. The framework support some advanced features, like transaction control of model updates and scripting of queries in Python. This approach is quite similar to the Java approaches, but here the class mechanism of Python has been used.

One major source of inspiration has been the UML and MOF specifications; inspecting these documents made it clear that a level can be represented as a graph consisting of connected objects; the concepts related to describing a class must then also be represented by objects, e.g. an attribute is an instance of `Property` linked to another object representing the class. Also by examining the possible instantiations described (e.g. an association is to be instantiated as a link), we got a picture of how the `instanceOf`-relation applies. We are designing a metamodeling framework, and are concerned about how two single levels can be connected in a correct way. Those types of questions are not answered by a repository-based approach. In our approach we separate out the representation, and then allow attaching of different semantics.

C.4 FORM: Our Basic Representation

As our aim is to represent models the same way as metamodels, we propose in this section a basic representation for models called FORM, which can be used independently of the level of the models. Our proposed representation is based on the following observation: *Every part of a level in the metamodel structure is an instance of something at the level above.*

Generally an instance has been instantiated from the level above, but the top level is special: everything at this level can be seen as an instance of something residing on this same level. This can be seen as using the same level for describing itself.

The metamodeling environment can be restricted to objects; in a sense "object" is the lowest common denominator of all levels. Since our metamodeling approach is object-oriented, we need to represent the following information:

- Objects.
- Slots owned by objects.
- Slot values and their type.
- Links between objects.

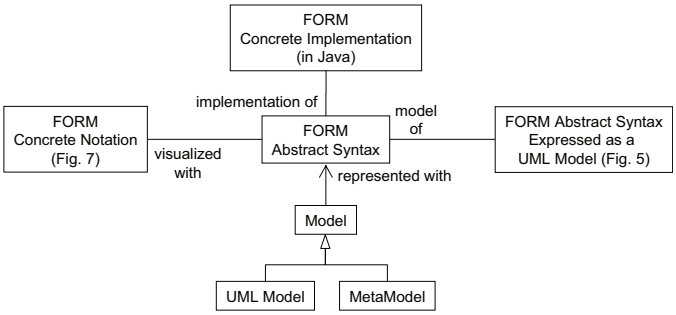

Figure 23: FORM Context

- Every object knows its class.
- Every slot knows its attribute.
- Every link knows its association.

One metalevel is an *understanding* (meaning, semantics) of how the level below can look like; this understanding is necessarily coded into the structure of the level. To make such a structure one needs a carrier - a substance that can be formed into a coded understanding. We are used to papers and pencil, our FORM representation on the other hand is meant for computers.

As we can see in Fig.23, FORM is providing an abstract representation of models (e.g. UML models). We can describe the constituents of this representation as done in Fig. 24, where we define physical building blocks that can be put together to form complex structures, much like how atoms can be put together to form molecules.

The metaphor above used the term physical which in this context can be seen as the same as concrete, one can choose to see the computer objects as existing physical objects [MMPr93]. “Abstract information” must be coded in concrete representation; the coded information is forming a pattern (structure) that might be interpreted by humans or machines. A computer can interpret in the sense that it can transform and operate on the structure.

It is of course problematic to talk about abstract representations and even to handle them. Therefore we also need a concrete (physical) representation. For our implementation this can turn out to be computational objects, e.g. Java objects. For the sake of this article, we do also need a

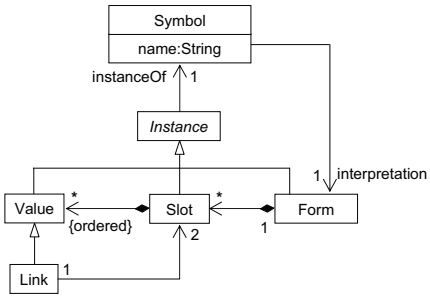

Figure 24: FORM - the basic representation

representation that can be put onto paper. We define such a representation in section C.4.2.

C.4.1 Definition of FORM

Fig. 24 shows a model of our basic representation; where we have `Form`, `Slot`, `Value` and `Link` as four special types of instances. Objects will be represented as instances of `Form`, links as instances of `Link` and slots as instances of `Slot`.

`Form` will be involved in the representation of both objects and classes; it seems that `Form` is a good name since it has the following two meanings: *"the shape and structure of something"* and *"a mold for shaping something"*. We also consider a value to be an instance, this is further discussed in section C.6.

An instance of class `Symbol` (typically the name of what it represents), will have a link to the `Form`-instance it symbolizes. The mentioned `Form`-instance, together with connected slots and possibly some other linked `Form`-instance, can be seen as an interpretation of the symbol. Instead of letting an instance be linked directly to what it is an instance of, we have an interface of Symbols between levels. This allows us to handle levels separately from their adjoining levels and only bringing them together when necessary. It is easy to relate the model to a metalevel border: the interpretation of a symbol is on the upper side of the border and the "instances of the symbol" are on the lower side of the border. The symbols of the instantiable forms of one metalevel constitute the interface towards the level below.

Fig. 25 shows how the levels can be seen as separate components; each component provides an interface which can be thought as the symbols on the

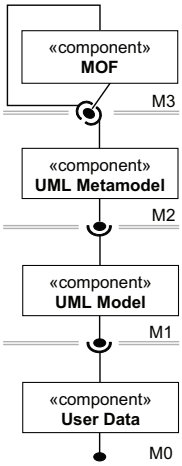

Figure 25: Viewing the levels as components

lower border of the level it represents. On the other hand, each component is requiring an interface which can be seen as the upper border of the level it represents. The MOF-component is special since its required interface is the same as the interface it provides. Also level `M0` (User Data) is different since it provides only an empty interface.

C.4.2 The FORM Notation

In order to be able to visualize the (abstract) FORM representation, we need some notation. Of course it would be possible to use some kind of UML notation, because UML includes all aspects that are necessary here. However, as our notation is used for representing models and metamodels, it usually leads to confusion if we use a known notation. Therefore we use the symbols shown in Fig. 26 for visualizing the elements of our abstract representation. We visualize a reference as a line connecting the two involved entities (arrows are used to indicate the direction of the reference; if there are two references going opposite ways we omit the arrow-head). The λ inside `Symbol` instance is to be replaced with the actual name of the `Symbol`; the λ inside `Value` instance is to be replaced with the value (e.g. `5`).

In Fig. 27 our mechanism is used with the example of Fig. 21. As we can see, a (UML) class is represented as a `Form`, a slot as a `Slot`, and so on.

Element	Concrete Visual Representation
Symbol instance	λ
Form instance	
Slot instance	•
Link instance	
Value instance	λ
Connection (Reference)	

Figure 26: The visual symbols of the FORM-notation.

It is obvious that this again is just an even more explicit representation, the structure represented is still the same[6]. From Fig. 27 it can be clearly seen that our representation is capable of representing all levels in a uniform way using only the few notational elements given in Fig. 26.

C.4.3 `InstanceSpecification` in MOF 2.0 and UML 2.0

The object notation with the `instanceOf`-relations introduced in figure 21 is not defined in [OMG03h, OMG03g, OMG03i]. The concrete syntax of `InstanceSpecification` is quite similar, but the purpose of `InstanceSpecification` is not to relate an object to its class at the level above. The UML specification [OMG03g] states: *"An instance specification is a model element that represents an instance in a modeled system... An instance of an InstanceSpecification is defined in a model at the same level as the model elements that it illustrates*".

An extract of the UML metamodel is shown in figure 28. As we can see, there will be no reference to the level above when it comes to instances of `InstanceSpecification` or any of the other classes. Our conclusion is that `InstanceSpecification` does not fill our needs, we must have references to the level above.[7]

[6]Please note that the simplifications done in the earlier figures are inherited and also that some symbols are duplicated.

[7]Author comment given when the dissertation was compiled: Figure 28 shows an association from `InstanceSpecification` to `Classifier` and this may be interpreted as a reference to the level above when instantiated; the two levels involved would then be related by ontological instance of relations. However, it does not seems like ontological levels

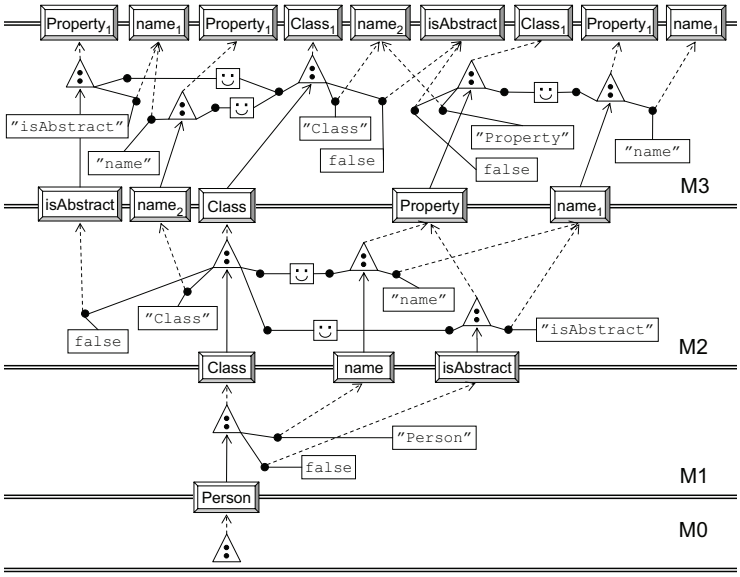

Figure 27: FORM representing the same underlying structure as Fig. 21

C.5 XMI

In this section we want to discuss if the FORM representation is sufficient to represent models stored using XMI and describe how a production of XMI files starting from models represented by FORM looks like. In SMILE we plan to use XMI as the exchange format. The main advantage is that XMI is a widely accepted standard, so we are able to exchange our models with modeling tools from other vendors. This is especially important in the project startup phase, where we can use XMI documents created by other tools as test samples.

XMI is a standard for exchanging and storing models using XML. The current version is 2.0 released by the OMG on May 2, 2003. The standard defines a number of production rules which are used to read/store models in/from XML files. The main production rules are:

are considered in the citation given: "...an InstanceSpecification is defined in a model at the same level as the model elements that it illustrates" – this is somewhat confusing!

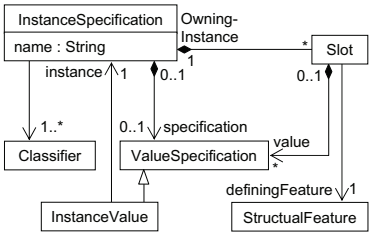

Figure 28: `InstanceSpecification` from the UML 2.0 metamodel

- production of XML Schemas starting from an object model
- production of XML documents compatible with XML Schemas
- production of XML documents starting directly from objects
- reverse engineering from XML to an object model

The first three methods are used for saving model data while the last one is needed for restoring. There are at least two ways of getting an XML file out of a model. The longer one is:

- Create an XML Schema which represents the metamodel (in older version of the XMI standard Document Type Definitions (DTD) were used instead of XML Schema).
- Create an XMI File according to the XML Schema containing the model.

The shorter way consists only of one step:

- Directly create an XMI File representing the model.

As you can see, the creation of XML Schemas representing the meta-model is optional. It is recommended nonetheless because it allows you to validate your model (stored in an XMI file) against the meta-model without having to decode the XMI file first. Moreover it gives you a possibility to define unique representations of certain model elements when the XMI standard allows multiple ways of encoding them.

However, it is still possible to store invalid models (according to the meta-model) in a valid XML file (according to the XML Schema produced out of the same meta-model) due to some semantic constraints of the meta-model

which cannot be expressed in the XML Schema. Thus XML Schemas of meta-models provide you necessary, but not sufficient criteria for validating models stored in XMI.

If we have a model stored using the FORM representation, we can create an XMI file using the following rules:

Every model element stored as `Form` in FORM becomes an XML element in XMI. This XML element is named like the `Symbol` the `Form` is described by, and has an attribute `xmi:id` whose value is unique among the saved forms (we will use `xmi` as the name for the namespace `http://www.omg.org/XMI` in the following examples). A `Form` implementing a computer (defined in the metamodel) would be written like the following:[8]

```
<computer xmi:id="c01"/>
```

Attributes, stored in `Value`s and referenced via `Slot`s in FORM, can be represented in XMI in two different ways. They can either become XML attributes of the corresponding XML element (the XML element of the `Form` the attribute belongs to) or XML elements that belong to the XML element of the owner. In the first case, the name of the `Symbol` describing the texttttSlot is represented by the name of the XML attribute. In the second case, it becomes the name of the XML element.

Considering the computer mentioned in the example above had an attribute called os (which is stored using a `Slot` implementing the `Symbol` os). The XMI representation of two computers running two operating systems could be represented by the following XMI fragment (both possibilities are shown):

```
<computer xmi:id="c01" os="linux"/>

<computer xmi:id="c02">
     <os>unix</os>
</computer>
```

Associations are represented by `Links` in FORM. Like attributes they can also be stored using either XML attributes or XML elements. An XML attribute is named like the association (i.e. the `Symbol` it is described by) and carries the `xmi:id` of the referenced element(s) as value. An XML element is named in the same way and it contains the referenced element(s) in the `href` attribute. If we add an association `user` from computers to

[8] `c01` is just an automatically generated identity of this entity. It is not needed in this small example.

persons in our example, we can represent two computers which are both used by the same two persons like that:

```
<person xmi:id="p01">
<person xmi:id="p02">

<computer xmi:id="c01" user="p01 p02"/>

<computer xmi:id="c02">
     <user href="#p01"/>
     <user href="#p02"/>
</computer>
```

Apart from some minor details the transformation process from a FORM model to an according XMI representation is rather simple. However, the process of restoring XMI data is much more complicated.

It is obvious that we are able to import XMI files formerly exported by ourselves. It is harder to import XMI files from other vendors.

The biggest problem is that XMI allows different representations of the same model construct. Moreover there are some XMI constructs that we do not plan to use, but we have to expect tools from other vendors using them. However, we can represent all these constructs using FORM.

The only exception to this statement we know are the so-called vendor extensions. These are extension elements in an XMI file that can be used to store additional data which is only important for the tool which exports (and reimports) the XMI file (e. g. graphical modeling tools can use vendor extensions to store the screen positions of model elements).

The standard behaviour of XMI applications is to ignore all unknown vendor extensions. In SMILE we plan to deal with vendor extensions the same way. This is not only unavoidable due to the fact that the data in vendor extensions is unstandardized, but also acceptable since vendor extensions should not contain model data.

To summarize this section we can say that we are able to convert every model from a FORM representation to an XMI representation and vice versa. This means that FORM is sufficient to store any kind of models.

C.6 Instantiation

The most interesting action with metamodeling is instantiation, meaning creating a level using the information on the next higher level. However,

metamodels are declarations; the act of instantiating a metamodel is not described, and only the set of possible correct results is described[9].

Therefore we will discuss in this section the problem of deciding if two given single levels can be seen as neighboring levels. What are the semantic consequences of connecting two levels and what must the underlying structure support to achieve a "sound" connection?

The discussion will be concerned with two adjacent levels, a lower level (object/instance level) and an upper level (meta level). We consider *basic instantiation patterns* for all the elements of our FORM representation, i.e.

- Instance, see C.6.1
- relation between Symbol and Instance, see C.6.1
- different kinds of Instances, see C.6.3
- relation between Form and Slot, see C.6.2
- relation between Link and Slot, see C.6.2
- relation between Slot and Value, see C.6.2
- built-in values, see C.6.4.

We only consider the symbol interfaces related to the instantiation, i.e. the upper interface of the lower level (client interface) and the lower interface of the upper interface (server interface). In our approach, we do not have any information coded "into the symbols". This way the names do not provide semantics.

C.6.1 Basic Instantiation - Matching Symbols

Any instance on any level has to be related to a defining `Form` on the level above. The most basic pattern describes this: an `Instance` on the object level relates to a `Form` residing on the meta level through matching `Symbol`s of their related interfaces, i.e. level borders. Two `Symbol`s in different borders are considered to be the same `Symbol` if they have equal names, assuming that the `Symbol`s of each border are unique.

Fig. 29(a) shows two levels that are to be connected; the `Symbol`s are represented as $\lambda 1$ and $\lambda 2$, for a match to occur these `Symbol`s must be the same. Assuming $\lambda 1$ and $\lambda 2$ are equal then the result of matching can be seen in Fig. 29(b).

[9]The MOF specification has some instructions on how to go about and also includes a factory-class.

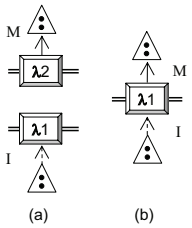

Figure 29: Connecting a `Form` to its description.

If some `Symbol`s of the server interface are not merged with `Symbol`s of the client interface, then it simply means that there are no "instances of" these `Symbol`s, which is no problem. If some `Symbol`s of the client interface are not merged with `Symbol`s of the server interface then some instances of the object level are still without definition, which means that the matching was not completely successful - some `Symbol`s are unresolved.

If we assume that this rather uncritical matching of `Symbol`s has created a situation where all `Symbol`s of the client interface are matched, then we have to check that the descriptions on the meta level fits to the instances on the object level. This is described below.

C.6.2 Instantiation of Links

The only possible information that can be given on the meta level so far is structural information in the sense of connections between entities. There are three kinds of connections defined in Fig. 24, ignoring the connections with `Symbol`s which already have been taken care of.

The connections we are talking about are not the links which are instances of `Link` as part of FORM; the connections we are talking about are the most basic ones and they must be supported by the underlying system (how they are physically represented is up to the underlying system - in Java such links could be implemented as references).

What is the relation between `Link` and a connection? It is clear that we need some kind of description of an object level connection. This description should be given on the meta level, and according to the possibilities in FORM it has to be a `Link` instance. However, as `Link` can only be connected to a `Slot`, a "complete" link is represented as (form-)slot-link-slot(-form), and connections will be used to bind those entities together (e.g. a connection from a `Slot`-instance to a `Link`-instance).

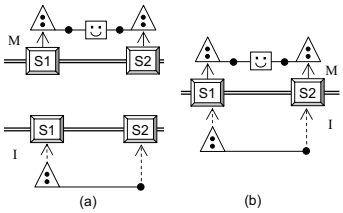

Figure 30: Relating a connection to its description.

Fig. 30(a) shows two levels that are to be related; the `Symbol`s match and the two levels connected can be seen in Fig. 30(b). On the upper level of Fig. 30(b) we find the pattern: "slot-link-slot" which is what we choose to see as a description of a connection. On the lower level we find the connection between a form and a slot. A connection on the lower level must always have a *corresponding* instance of the slot-link-slot pattern on the upper level meaning that the slot-link-slot pattern is to be found on the upper level between the descriptions of the two involved entities.

If this condition can not be met we can not correctly connect the levels, the description does not fit to the instance. The same logic also applies for a connection from a `Slot` to a `Value`, and for a connection from a `Link` to a `Slot`.

C.6.3 Different kinds of Forms

It seems that so far all connections on the lower level are always represented by the same slot-link-slot pattern on the upper level. This is not really sufficient since the three kinds of connection are different, e.g. a slot-link connection is different from a slot-value connection. In order to provide support for this we attach an information to the `Form`s stating which kind of instance can be related to them. In order to also have this information visible on the interface, we in fact attach it to the corresponding `Symbol`s. In Fig. 31 we have added an enumeration `InstanceKind` to distinguish the different kinds of instances a Form can create.

The description type (or instance type) can not be assigned arbitrarily. Take as a complex example the instantiation of `Link`: To do this a rather huge structure has to be present on the upper level (a complete description of an association). If this structure is not in place we end up with having a description that does not make sense. The support of these things is done

by the framework and is not described in this article.

C.6.4 Handling Values

The only special kind of values we have introduced so far are the `Links`. However, in real systems it is also necessary to handle primitive data like integers or strings. Looking at UML we find the concept of *primitive type*, which is called `PrimitiveType` (or just `Primitive`); instances of `PrimitiveType` are primitive types like `Integer` or `String`. Instances of these types are values of the domain they represent, e.g. `5` is an instance of `Integer`. The number `5` is an example of what they call a `PrimitiveValue`. A primitive type is special since it is implemented or built in by the underlying infrastructure; it is made available for modeling and is accessible at all times and at all levels. Its semantics cannot be found on a level in the metamodel structure - it has no *"relevant substructure (i.e. it has no parts)"* as they state in [OMG03d]. Assume that `5` is used on level `M1`; `5` could then be seen as an instance of the type `Integer` which resides on level `M2`; `Integer` could then be seen as an instance of `PrimitiveType` which resides on level `M3`, and finally `PrimitiveType` could be seen as an instance of `Class`.

However, a basic value has already a defined semantics, which can be related to the definition of its (proto)type on the level above. The same is not true for the other instances, they all have an explicit structure as their semantics.

In our approach, however, we do not have the possibility to give special semantics to names - all semantics should be given explicitly or be built-in. Therefore we introduce a special kind of `Value` for the basic values called `BasicValue`. We also introduce a special kind of `Form` for the basic types called `BasicType`. Both basic types and basic values are not characterized by their internal structure, but they carry their semantics already with them. In a way, they have an "external structure"[10]. Handling of external structure is only possible to be built-in, which is what we do here.

The only interesting information about basic values and basic types when it comes to instantiation is their relation: When is a basic value an instance of a basic type? This structural conformance check has two parts. First, the defining form of a basic value has to be a basic type. Second, the basic value should be within the range of the basic type.

[10] As the research on abstract data types has shown it is possible to turn every external structure into an internal structure. However, as was also shown by the ADT community, this is in general very error-prone and should be avoided if there are other possibilities.

The first requirement is easily handled by introducing a new `InstanceKind` for basic values. The second requirement is handled by introducing a type checking function for basic types, i.e. BasicType::membercheck(Value) → Boolean. The result of all these additions is seen in Fig. 31.

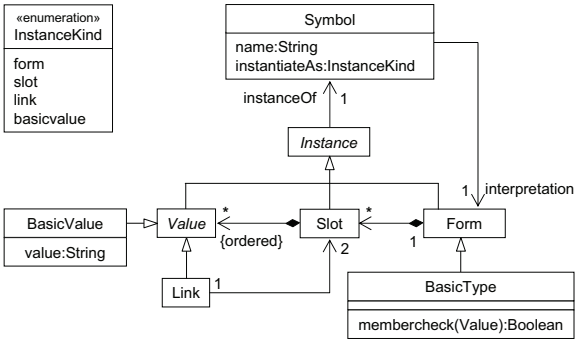

Figure 31: FORM with instantiation information

This handling conforms very much to the schema of UML. If we repeat the example with `5: 5` is a `BasicValue` instance of the type `Integer` which resides on level `M2`; `Integer` is an `BasicType` instance of `PrimitiveType` which resides on level `M3`; `PrimitiveType` would be a `Form` which is an instance of `Class`. The difference is that we are giving the semantics bottom-up instead of top-down as within UML. In other words, it is completely arbitrary which class the basic types are collected in because we do not attach meaning to names.

C.6.5 Derived Consistency with OCL

The above checks are enough for checking all the semantic things that are defined on the basic model. However, sometimes it is necessary to define some more checks. This is done using OCL, i.e. by attaching formulas to the forms which should be checked on the level below.

We take the view that this is still another application of the same instantiation pattern for the OCL parts. This means the OCL formulas are considered to be the templates and are instantiated as values on the level below. In this context the OCL formulas are considered well-formedness rules and the implied semantics is that these values should all be `True`. However,

it is possible to consider any other use of the values generated by the OCL formulas.

We want to give the semantics of the OCL formulas directly on the OCL metamodel, not within the implementation. This way it is possible to consider OCL as just an additional module allowing a way of semantics description. In a similar way also dynamic semantics can be defined as a separate module.

C.7 Conclusions and Research Directions

In this article, we have described FORM, the abstract syntax of a language for representing levels in a metamodeling environment. An instance of FORM will be an object-graph were objects have slots with values and the objects are linked to each other by a *link value* (instance of `Link`). `Form` is a powerful construct, an instance of `Form` is always an object, but it can also be used to represent a class, a class for classes and so on.

This is achieved using the very basic structures present in all the models and also within the UML and MOF. We have taken this to the extreme and removed all special cases in order to get a level independent representation.

This way all entities are understood to be created according to some object or pattern on the level above. Similarly, all `Form`s have the power to create objects on the next level. In this article, we have tried to give a better understanding of the process of instantiation, or the semantics of it.

Using our approach we can represent all the things possible to be represented using XMI, i.e. all models. This representation does not yet include any semantics apart from the instantiation.

We envision to provide ways to attach semantics to the meta model elements making it possible to use this semantics on the next level to describe static and dynamic properties, textual and graphical representations or exchange formats.

Appendix D

Accessibility Testing XHTML Documents Using UML

Terje Gjøsæter[1], Jan Pettersen Nytun[1], Andreas Prinz[1] and Merete S. Tveit[1]

[1]Faculty of Engineering, Agder University College
Grooseveien 36, N-4876 Grimstad, Norway

Proceedings of NWUML'2005, 3rd Nordic Workshop on the Unified Modeling Language
August 29–31, 2005, Tampere, Finland

This appendix presents the paper: *Accessibility testing XHTML documents using UML* [GNPT05], with coauthors Terje Gjøsæter, Andreas Prinz and Merete S. Tveit.

The paper presented in Appendix A described modeling and testing of consistency requirements; the paper of this chapter describes another application of the metamodeling framework. The paper is presented below:

This paper handles modeling and test of accessibility requirements for web documents. We propose to use metamodeling with UML and OCL for this task. Our own environment within the SMILE project has proposed a basic representation that can be used on all levels in a metamodeling architecture; this representation is used for representation of the elements of the metamodel, models and model instances. We show the use of OCL formulas to express simple and advanced accessibility requirements.

D.1 Introduction

Access to web content for all is crucial for building the information society. Information on the web should be accessible to all users, independent of disabilities or choice of web browser. Within the EIAO[11] (European Internet Accessibility Observatory) project [EIA], we want to improve the accessibility of web content by providing measurement data about accessibility. However, it is not straightforward to measure accessibility, because this is very subjective. One main task within the EIAO project is to formalize the informal and subjective requirements.

In order to tackle the problem from a higher level and for making sure the formalization is understood by the experts, a pilot project called MEBACC has been established in cooperation with the Norwegian Directorate of Primary and Secondary Education [UTD]. The aim of the project is to create a prototype for an Open Source tool for accessibility checking of web documents and web based teaching material. The approach within the MEBACC project is to model the requirements and the measurement policy explicitly. The project is integrated with the ongoing research on metamodeling at Agder University College (SMILE project) and the EIAO project. An important part of MEBACC is to define web document models representing the relevant standards for use in conformance testing. This paper will show how UML can be used for this purpose. A subset of the XHTML specifi-

[11] EIAO has been co-funded since Sept. 2004 by the European Commission under contract number 004526.

cation will be represented as a web document model in UML. XHTML is chosen because it has stricter requirements for structure than HTML; it is therefore easier to create model instances from an XHTML document than from a HTML document.

The SMILE project is built around a kernel that can represent models on arbitrary levels of the metamodeling hierarchy. We will also show in this article how this SMILE basic representation (called MATER) looks like for the case of UML-like models as used here. In fact, the use of UML is taken here just as one possible representation of the metamodel and the model, because this is a familiar notation. The MATER model abstracts from this kind of representation, such that any notation could be used (e.g. UML).

Within MEBACC, we also built a prototype to show the relevance of our theoretical results. The prototype is simple while still advanced enough to prove the potential of our approach. Carefully chosen subsets of XHTML, OCL and the WCAG 1.0 accessibility guidelines are supported in the prototype.

The article is structured as follows. In Section 2, we give background information about web accessibility measurements in the scope of the EIAO project. Section 3 deals with metamodeling in general and with our SMILE project. In Section 4 we describe our approach and give a small example to demonstrate how accessibility is modeled using OCL and UML. We conclude the paper in Section 5.

D.2 Measuring accessibility

There are defined some standards to measure web accessibility. The WCAG guidelines presented in Section 2.1 is one standard that will be used within the EIAO project. EARL reporting (Section 2.2) is another standard that will be used to evaluate web pages against the guidelines from WCAG.

D.2.1 The WCAG guidelines

The WCAG Web Content Accessibility Guidelines [W3C99] is produced as a part of W3C Web Accessibility Initiative [W3C06], and explains how to make web content accessible to people with disabilities. The WCAG 1.0 includes fourteen guidelines, or general principles of accessible design. The guidelines discuss accessibility issues and provide accessibility design solutions, and they address typical scenarios that may pose problems for users with certain disabilities. Each guideline includes a list of checkpoints which explain how the guideline applies in typical content development scenarios.

D.2.2 EARL reporting

EARL (the Evaluation And Report Language) [EAR] is a language to express test results. Test results include bug reports, test suite evaluation and conformance claims. EARL is in a RDF based framework for recording, transferring and processing data about automatic and manual evaluations of resources.

EARL expresses evaluations about all sorts of languages and tools, and could be used to evaluate web pages and web sites against WCAG, and then generate an accessibility report corresponding to the test results.

D.2.3 The EIAO project

The European project EIAO [EIA] (European Internet Accessibility Observatory) will assess the accessibility of European web sites and participate in a cluster developing a European Accessibility Methodology. The assessment will be based on the WCAG developed by W3C. The project is carried out in a co-operation among 10 partners in a consortium co-ordinated by Agder University College Norway.

EIAO is carried out within the Web Accessibility Benchmarking (WAB) Cluster together with the projects [EAM] and BenToWeb [BEN], co-funded by the European Commission. The cluster consists of 24 partner organisations in Europe.

Among its planned output is a set of Web accessibility metrics, an Internet robot "ROBACC" for automatic collection of data on Web accessibility and deviations from Web accessibility standards, and a data warehouse providing on-line access to measured accessibility data.

EIAO is defining an extensible plug-in architecture in cooperation with W3C and the European WAB Cluster. This architecture will allow exchange of web accessibility assessment modules among different applications. The test modules that are produced based on accessibility models, may implement the EIAO interface, and thereby use the ROBACC crawler of EIAO as a vehicle for testing of a large number of web sites.

D.3 Metamodeling using SMILE

Our metamodeling approach is done inside the SMILE metamodeling framework; this section describes and introduces some of the basic concepts of the SMILE metamodeling framework.

The SMILE project targets all the levels of the OMG four-layer metamodel architecture; this implies definition of an object representation. FORM [NPK04] was the first definition proposed; it was meant to be used on all the levels of a metamodel architecture and it included the `instanceOf`-relation between elements of to adjacent levels. FORM allowed two levels to be tested for "adjacency" (can one be seen as the model for the other). Its successor MATER (Model All Types and Extent Realization) has been extended with a deep instantiation mechanism; this has been done by supporting definitions of patterns that span multiple levels. Since this paper has another focus instantiation patterns are not described here. MATER is more flexible than FORM allowing different "styles" of metamodeling.

D.3.1 MATER - Model All Types and Extent Realization

MATER defines a uniform way of representing metadata and object information in a metamodeling environment. This uniform representation is a *level independent representation*, meaning that all levels can be represented with the help of one common mechanism.

MATER is not meant to be a metamodel or a meta-metamodel, it is meant to be "the substance" that is used when a level of the metamodel architecture is made; this proposed basic representation takes care of what [GKP98] calls *inter-level* instantiation and [AK02] calls the *physical classification*. The conceptual model of MATER is object-oriented; when instantiated an object graph will be the result.

If a metamodel for relational databases is defined, the model level will define the layout of tables, and the information level will consist of actual tables. In the SMILE metamodeling framework the information level will be an object graph that can be mapped to actual tables, the object graph will have a structure that logically correspond to the tables.

Fig. 32 presents the conceptual model of MATER in UML (how to handle basic types is left out, but [NPK04] demonstrate how this can be done).

The metamodel border between two levels is seen as an interface composed of symbols (instances of Symbol) which from the level below represent the instantiable elements of the level above (e.g. names of classes). One metalevel together with the upper and lower interfaces constitutes a manageable module. An instance of Symbol that does not reside on the border is abstract and will have no instances.
A `Slot`-instance can keep one or more values (e.g. a number); a special type of value is the link-instance which can connect two or more `Slot/Substance`-instances. A `Link` instance can represent a reification of an association in-

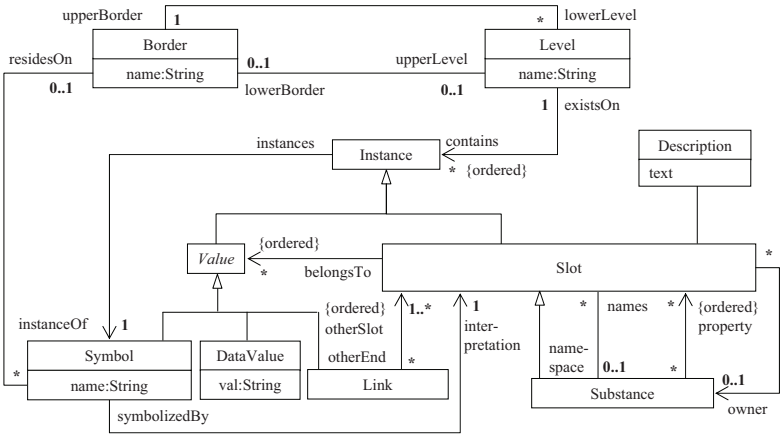

Figure 32: The conceptual model of MATER.

stance (which in UML is called a link); the Link instance makes it possible to have an `instanceOf`-relation from the "link" to the association which has been instantiated. The links" of an object graph made by instantiating the conceptual model are called connections; this are not considered objects and must be supported by the underlying software (e.g. references in Java). This problem is discussed in [Atk97] which have the following statements:

> ... it is possible to reify (i.e. view as objects) links and associations so that they can be modeled as objects and clabjects respectively... The difficulty in reifying links is not in working out how to view them as objects, but in knowing when to stop viewing them as objects...To break this potentially infinite regression it is necessary to identify certain kinds of links as implicit or primitive links which will not be stored as objects.

`Substance` is a specialization of `Slot`; instances of `Substance` can keep values and have other substance as property; the `owner` and `property` associations used together can define compositions, while the `namespace-names`-association obviously is meant for modeling namespaces. `Description` is meant for additional semantic information.

There is not full agreement on what object-orientation includes and consequently the conceptual model of MATER is one approach; the conceptual

model of MATER is kept small but still powerful enough to allow flexible modeling.

In object-oriented metamodeling the essential object-oriented concepts must in some way be stated since abstract syntaxes are described with class diagrams. It might be possible to use the underlying representation in such a way that it directly supports a specific object-orientated concept, e.g. that it supports objects with slots. Other concepts might be modeled more indirectly where we as humans must study the structure of several levels to make an adequate interpretation. The following is a list of the object-oriented concepts considered and how they can be supported in the MATER approach:

Object: The underlying representation supports this. A class will be described with the help of objects.

Slot: The underlying representation supports this.

Link: The underlying representation supports this.

Multiplicity: Must be modeled explicitly

Identity: A symbol can be used to identify a Slot.

Namespace: `Slot/Substance` has a special association for modeling namespace hierarchies; the members of a namespace instantiate this association to reference the `Substance` which function as a namespace; a member can be a new namespace. A `Slot`-instance that is member of a namespace must have a `Symbol`-instance as value, this value function as a name. The default namespace is the level which means that all symbols that are not part of another namespace must be unique. It is up to the metamodeler to model the namespace.

Composition: In UML 2.0 class `Property` has a boolean attribute called `is- Composition`; an instance of `Property` typically becomes a property (attribute or "association end") of the owing class (instantiated from `Class`); `isComposite` will be a slot of the `Property`-instance; if the `Property`-instance is an association end and this slot has value true then this is indicated by showing a filled diamond; an object instantiated from such a class will be a container for the object referenced by the slot or value contained in the slot. In UML 2.0 one has to look at the level above to see if something is a composite or not. In MATTER composition can be modeled as done in UML 2.0; additionally

the `owner-property` relations (composite-part) can be used to show the composition where it actually occurs.

Concrete class: Is not directly supported and must be modeled by the metamodeler. If something is a class then it can be instantiated to objects on the next level - the metamodeler models this with instantiation patterns, e.g. a metaclass will be specified on one level; instantiated on the next level to a class; which again can be instantiated to an object on the next level. Examining the levels and how they relate shows what are classes; the names used are irrelevant.

Abstract class: Same as concrete class but the symbol will not be placed on the border but reside "freely on the level".

Property (attribute and association end): Is not directly supported and must be modeled by the metamodeler. If something is a property then it can be instantiated to slots on the next level.

Association: Is not directly supported and must be modeled by the metamodeler. If something is an association then it can be instantiated to links on the next level.

Inheritance: Is a description technique; which means that from the "object-level" it looks like ordinary class-instantiation and it is only by examining the model level that the use of inheritance will be revealed. Inheritance is not directly supported.

Packages are a way of grouping elements and defining namespaces. One might consider one level as a package which defines the default namespace. To simplify the presentation packages are not included.

D.3.2 MATER with set notation

Fig.33 shows a concrete notation for MATER; it has similarities with the notation used when visualizing sets, but here extended with meta-information.

Fig.34 shows how an object of class `Person` and class `Person` can be modeled, corresponding UML notation is also supplied (at level `M2` only UML notation is shown). Class `Person` is on level `M1` and the object on level `M0`. Note how class `Person` is modeled as composite for its property called `name`, the `owner` and `property` associations of basic representation has been used to model this (both being instantiated). Multiplicity information is included for the `name` property of class `Person` (it is set to 1). Note how

Element	Concrete Visual Representation
Border	
Symbol	λ
Value instance	λ
Link instance	
Slot (not Substance)	
Substance	
Relations: Instance.instanceOf and Symbol.interpretation	
Relations: Slot.Value, Link.Slot, Slot.owner, Substance. property	
Composition: when Slot s2 is property of Substance s1 and s1 is owner of s2 then s2 is shown inside s1.	
Relation: at the same time Slot.Link and Link.Slot	
Relation: names-namespace	

Figure 33: A set-like concrete notion for MATER.

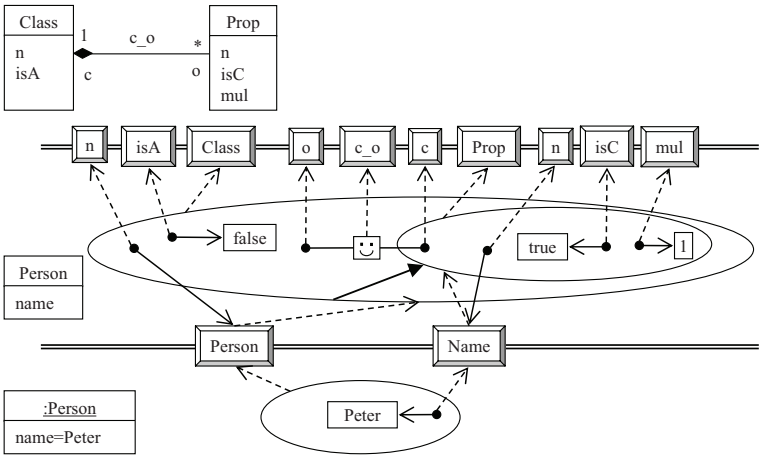

Figure 34: Two levels with MATER, concrete syntax as shown in Fig.33

namespace information has been modeled by the arrow (the one with the filled arrow head) going from the `Person`-substance to the `name`-substance (the description of the `name` property of class `Person`); the value of the `n`-slot (the symbol called `name`) is used as a name in the namespace; class `Person` function as the namespace, in effect all properties of the class have to have unique names.

D.4 Modeling accessibility for XHTML with UML and OCL

The XHTML standard is represented as a UML model called the web document model. The planned tool will take a set of web documents as input and instantiate those into web document model instances based on the web document model. If this instantiation is successful, the web documents are considered valid.

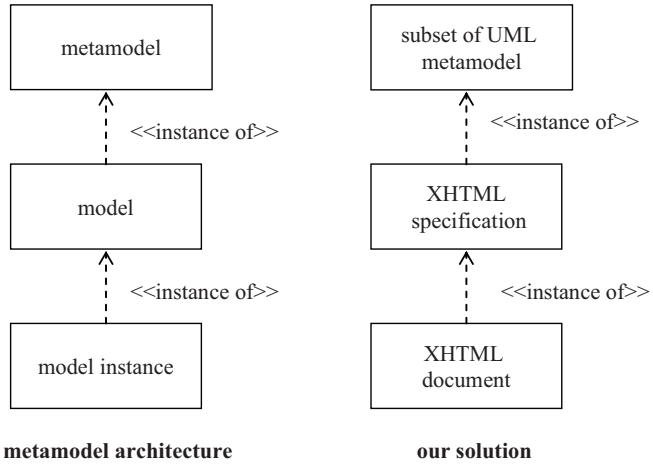

Figure 35: Our metamodeling architecture

OMG operates traditionally with a four-layer metamodel architecture [OMG]. For our purpose it is sufficient with three levels shown in Fig. 35. The elements of Fig. 35 are explained below.

Metamodel: The top level is a metamodel that describes the concepts that will be used when defining the XHTML-standard as a model. This

metamodel could be the concepts of XML, but we chose an object-oriented approach. We select a subset of the UML metamodel that includes: class, property, association generalization and composition. OCL will work well on such a subset.

Model: The middle level will be an object-oriented representation of a subset of the XHTML-standard itself and can be seen as an instance of the top level. It is at this level the accessibility modeling with OCL is performed; OCL accessibility constraints are attached to the modeling elements and can later be evaluated on the model instance level.

Model Instance: An XHTML-document is transformed to be an instance of the model (level above); the OCL accessibility constraints are evaluated and a report is generated that states to which degree the web document fulfils the accessibility demands.

The EARL Evaluation and Report Language will be used for reporting deviations from standards and accessibility requirements. The instantiation technique of MATER will be used when building the metamodeling architecture.

D.4.1 The Metamodel

For our experiments with web accessibility we have developed a very simplistic metamodel as shown in Fig. 36. The metamodel is compatible with UML in that it is just a very restricted MOF (a simplified subset of the UML metamodel kernel), and it is compatible with SMILE as SMILE allows representing it using MATER.

D.4.2 The Web Document Model (subset of XHTML)

The XHTML standard is represented as a UML model (the middle level) called the web document model. The planned tool will take a set of web documents as input and instantiate those into web document model instances based on the web document model. If this instantiation is successful, the web documents are considered valid.

OCL is a powerful language that offers first order predicate logic on object graphs. OCL expressions can include function-calls with elements of the graph as parameters. The prototype includes a hard coded subset of OCL and a set of functions that have been specifically made to do accessibility testing; the functions are available in the evaluation environment and can be used in OCL- expressions.

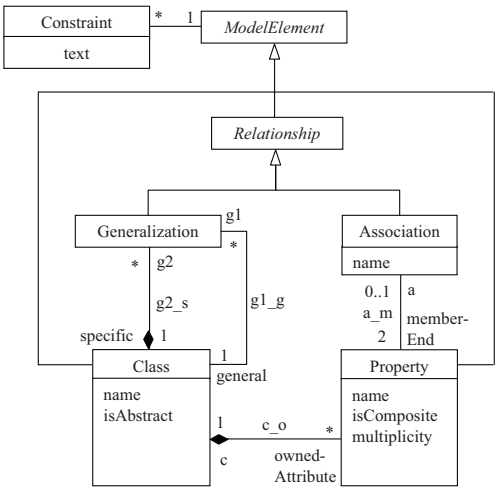

Figure 36: A minimal reflexive metamodel

OCL or OCL-like constraints will be added to the web document model to model accessibility requirements. If there is a valid instance-of relationship between the model instance and the accessibility model, the tested documents are considered accessible.

We have created a model of a large enough subset of the XHTML 1.0 transitional specification to cover the sample web document. Some more complicated parts of the specification will require OCL constraints, such as the requirement that you must have either HTTP-EQUIV or NAME, but not both, as attributes to a META tag (OCL constraint is not shown in this model).

Accessibility constraints: For the XHTML model given above we formulate three constraints that are derived from the WCAG guidelines.

1. Each image has to have a valid alt tag associated with it. Guideline 1 Provide equivalent alternatives to auditory and visual content describes how content developers can make images accessible. Some users may not be able to see images, other may use text-based browsers that do not support images, while others may have turned off support for images. The guidelines do not suggest avoiding images as a way to improve accessibility. Instead, they explain that providing a text

equivalent if the image, which serves the same purpose, will make it accessible. An image in HTML has an alt-tag which is used to provide text equivalents.

2. You must have either HTTP-EQUIV or NAME, but not both, as attributes to a META tag. This is in fact not a WCAG requirement but a static constraint for XHTML.

3. The color of the text should have enough contrast with the surrounding color. Guideline 2 Dont rely on color alone ensures that text and graphics are understandable when viewed without color. Checkpoint 2.2 says that it is important to ensure that the foreground and background color combinations provide sufficient contrast when viewed by someone having color deficits or when viewed on black and white screen.

These constraints are formulated in OCL as follows.

```
1. Context Img
   Inv:  libAcceptableAltTag( alt )

2. Context Meta
   Inv:  name.size() > 0 xor http-equiv.size() > 0

3. -- We define an auxiliary recursive function that
   -- finds the body.
   Context Block
   def:  getBody(b :  Block) :  Set(Body) =
   if b.body->size() = 1 then body
   else b.getBody(composite)
   endif
   Context Font
   Inv:  libAcceptableContrast(textColor,
   getBody(this)->any(true).background)
```

The functions `libAcceptableAltTag` and `libAcceptableContrast` are library functions.

D.4.3 The Model Instance

For the presentation we use a simple web page as shown in Fig. 38. We use this web page to evaluate accessibility requirements according to the OCL formulas given.

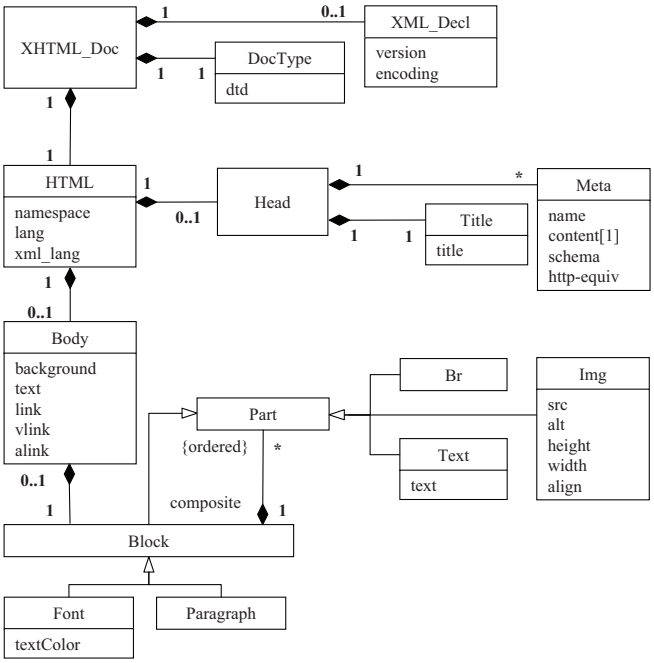

Figure 37: The web document model

The web document shown above is a (slightly broken) XHTML document, with the following source:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
   Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
   xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
   xml:lang="en" lang="en">
<head>
   <meta http-equiv="Content-Type"
   content="text/html;charset=utf-8" />
   <meta name="generator" content="gedit" />
```

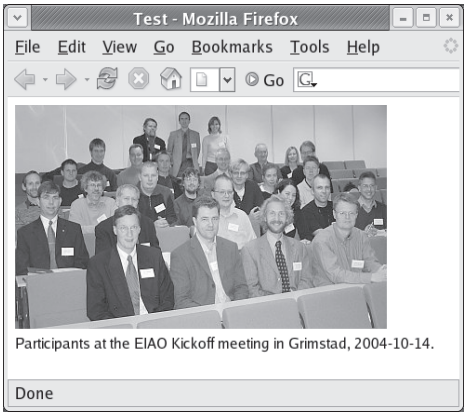

Figure 38: A web sample web page

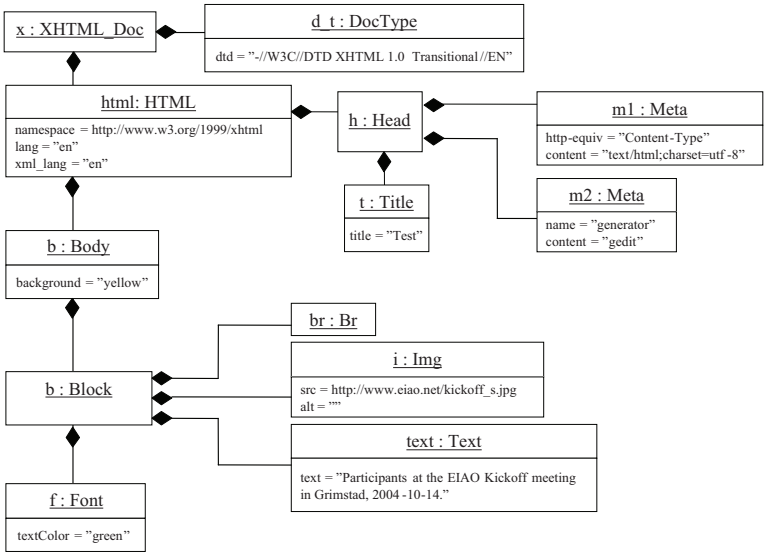

Figure 39: The sample web page as instance of the web page model

```
   <title>Test</title>
</head>
<body bgcolor="yellow">
   <img src="http://www.eiao.net/kickoff_s.jpg"
      alt="" />
   <br />
   <font color="black">
      Participants at the EIAO Kickoff meeting in
      Grimstad, 2004-10-14.</font>
</body>
</html>
```

The document has an empty alt-tag. This is in conflict with the WCAG 1.0 Guideline 1: "Provide equivalent alternatives to auditory and visual content." If the alt-tag was missing, the document would also be in conflict with the XHTML specification.

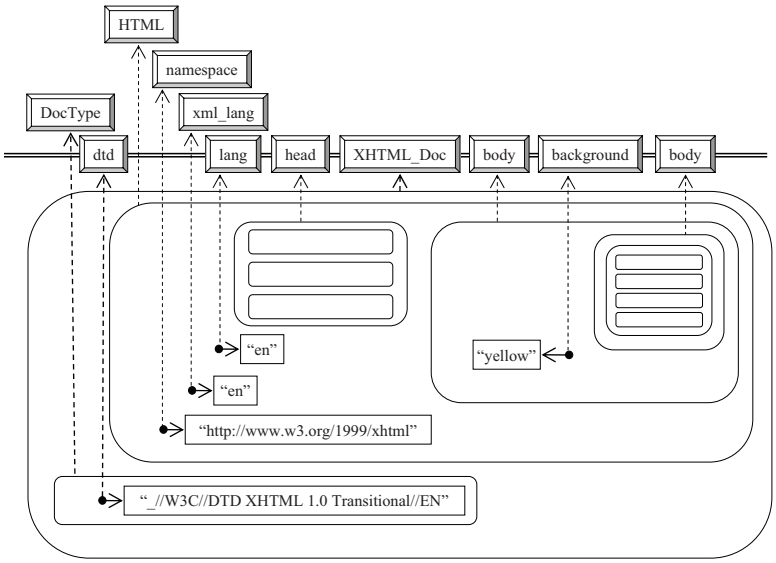

Figure 40: The instance of the web page model using the notation of MATER

When evaluating the OCL constraints on these instances, we have to check the following conditions.

1. The first condition is applicable in `Img` context. The only instance of this kind is the object i. So we have to apply `libAcceptableAltTag()`, which will give the result `false`. So this requirement is not fulfilled.

2. The second condition is applicable in `Meta` context. We have two meta objects `m1` and `m2`, and for both the condition is fulfilled. So there is no problem here.

3. The third condition refers to the Font context. For the font object f in our example we have to check `libAcceptableContrast(black,yellow)` which yields the result `true`. So this requirement is also fulfilled.

Fig. 40 uses the notation of MATER and shows the same as Fig. 39. Fig. 40 is not showing all the details: the instanceOf-relation for the links is not shown, and also the details of head and body are not shown.

D.5 Conclusion and further work

Accessibility requirements and web technology are constantly evolving. High level modeling of accessibility requirements can support more rapid generation of new test modules and improve the understanding of the accessibility barriers for web documents. UML seems to be the natural choice for this modeling. However, we need to take into account the limitations of using OCL to define accessibility constraints; we may need to extend OCL to be able to perform some more advanced accessibility tests. The basic representation and instantiation technique from the SMILE project is very useful when it comes to implementation of these ideas.

Starting from the prototype tool we have created, we will extend the subsets of HTML and WCAG 1.0 covered. We will integrate a complete OCL interpreter into the SMILE framework, such that it is possible to express more constraints than just simple comparisons. Moreover, we will extend the library of functions needed to do sensible checks for accessibility.

Appendix E

Modeling Accessibility Constraints

Terje Gjøsæter[1], Jan Pettersen Nytun[1], Andreas Prinz[1],
Mikael Snaprud[1] and Merete S. Tveit[1]

[1]Faculty of Engineering, Agder University College
Grooseveien 36, N-4876 Grimstad, Norway

ICCHP 2006 10th International Conference on Computers Helping People
with Special Needs
July 12-14, 2006, University of Linz, Austria

This appendix presents the paper: *Modeling Accessibility Constraints* [GNP+06], with coauthors Terje Gjøsæter, Andreas Prinz, Mikael Snaprud and Merete S. Tveit.

The approach presented here is different from the one presented in Appendix D - this approach is more practical and robust. The paper is presented below:

This paper describes the combination of research topics from two projects focusing on web accessibility testing, and on metamodeling and high-level specification languages. This combination leads to a novel approach to accessibility assessment that will improve the understanding of accessibility issues, and explore the potential for generating executable accessibility tests based on accessibility constraint models.

E.1 Introduction

Access to web content for all is crucial for building an inclusive information society. Information on the web should be accessible to all users, independent of disabilities or choice of web browser. In order to improve accessibility, the EIAO [EIA] project evaluates web content according to accessibility. This will raise the awareness for accessibility, and hopefully in the long run increase accessibility.

However, the evaluation of web pages is only trustworthy if the measurements are transparent to the users so they can understand how a ranking is established. We see two supplementary ways to achieve transparency: 1) the consistent use of Open Source software, and 2) explicit models of tests.

The SMILE [SMI] project is concerned with explicit representation of semantic information, such as description of formulae or algorithms. This is done using a metamodeling approach with an explicit description of the semantics of a language using a model of the language.

Building on experiences and results from the EIAO project, we apply the methodology of the SMILE project in the area of the EIAO project. Describing the issues with a metamodel-based approach [MEB], we will be able to model accessibility aspects directly. This work will result in a platform for experiments with accessibility constraints.

E.2 The EIAO project

The EIAO project is designed to develop a European Internet Accessibility Observatory (EIAO). It comprises a set of Web accessibility metrics, an Internet robot for automatic collection of data on Web accessibility and deviations from Web accessibility standards, and a data warehouse providing on-line access to measured accessibility data. In this way, the EIAO project will contribute to better e-accessibility for all citizens in Europe by providing a systematic and automatically updated overview of the current compliance with Web accessibility guidelines. EIAO is carried out within the Web Accessibility Benchmarking (WAB) Cluster [WAB] together with the projects Support-EAM [EAM] and BenToWeb [BEN], also co-funded by the European Commission[12]. The cluster consists of 24 partner organizations in Europe.

Web accessibility is on the European agenda through the eEurope action plans 2002/2005 on European and on national level. eGovernment policies in many European countries require compliance with WAI [W3C06] standards, and the private sector increasingly has to deal with Web accessibility. Existing automatic Web site tests are mainly used to evaluate and improve single Web sites. Benchmarking of, for example, all public sector Web sites or large collections of Web sites are not available. The objective of the European Internet Accessibility Observatory EIAO is to develop a framework for an automated Web accessibility assessment. The EIAO assessment system can provide valuable input to European benchmarking and can effectively support Web accessibility analysis and policies. Together with the EU projects BentoWeb and Support-EAM, the EIAO project forms a cluster that develops a Unified Web Evaluation Methodology (UWEM) [UWE].

E.3 The SMILE Project

The SMILE project accepts the current challenge of semantic information processing. Providing a methodology combined with a framework, SMILE allows us to enrich data with semantic interpretation (context knowledge); this is a key technology for providing tools for IT-based information processing. Using an extended project-specific metamodel approach, SMILE supports turning low-level information into higher-level information in new contexts by generating tools that produce and process this higherlevel in-

[12]The project is co-funded by the European Commission DG Information Society and Media, under the contract IST-004526

formation.

The capability of the SMILE methodology and of the framework will be demonstrated by solving problems in different application domains. The entirety of the identified practical problems forms the requirement base of SMILE; the solution of these problems will prove SMILEs feasibility.

The SMILE methodology can be seen from the meaning of its name: Semantic Metamodel-based Integrated Language Environment. The parts of this name have the following meaning in the scope of SMILE.
Semantic refers to the underlying idea to represent the semantics of languages and language constructs in an explicit way. This builds on the existing language description frameworks (e.g. grammars) and extends then to cover all aspects of modern computer languages.
Metamodel-based means that the approach is relying on modeling on all levels. We do not only use models for the actual systems, but also for the languages to be used for describing the systems and even for languages describing languages, etc. The important point is that the high-level descriptions of systems and languages should be complete. Our approach is **Integrated**, because we handle all the information in the same way, be it languages or systems. We describe the meaning of the constructs unambiguously in the (meta)model, and use this high-level description to generate tools.
Languages are in the focus of our project, because most of the activities can be related to languages. We are concerned with formal languages, and try to lift the level of formality so that all important aspects of the languages can be captured.
We build a complete **Environment**, such that the tools used within the SMILE project are finally built using the SMILE technology. A very important issue is that all parts of the SMILE description capabilities will be supported by tools: tools handling the description (reading, generating, editing) and tools transforming the description into other formats (e.g. code generation).
Currently we have succeeded in building the basic framework of SMILE according to [NPK04] together with a simple user interface. Further work will include the creation of all the other components necessary for semantic modeling.

E.4 The Approach

The main idea is based on a combination of the accessibility rating according to EIAO with the modeling approach of SMILE. Modeling is especially important for web content, because accessibility requirements and web technology are constantly evolving. High level modeling of accessibility requirements can support more rapid generation of new test modules and improve the understanding of the accessibility barriers for web documents in a wireless environment.

UML models with OCL-like constraints [OMG03d] can be used to model requirements for accessibility testing [TGN05]. Accessibility constraints will be attached to web document models that represent the relevant standards; such models guide the accessibility testing of web documents that are seen as instances of the mentioned models.

One approach to accessibility testing of XHTML documents was presented in [GNPT05]. The approach defines a three-level metamodel architecture where each level is to be represented in the SMILE-framework:

Metamodel A simple metamodel that defines the most basic object-oriented concepts like class, property and composition (a small subset of the UML metamodel).

Model A subset of the XHTML-specification [XHT05] represented as a UML model instantiated from the metamodel. The model is extended with OCL constraints that define accessibility requirements to be fulfilled at the model instance level.

Model Instance An XHTML-page is represented as an instance of the model. The OCL accessibility constraints are evaluated at this level to expose accessibility violations.

The EARL Evaluation and Report Language [EAR] developed by W3C, will be used for reporting deviations from standards and accessibility requirements.

The approach presented here is different from the one presented in [GNPT05]; it is meant to be more robust when it comes to deviations from the XHTML standard and it does not require a full model of the XHTML specification. The approach is based on the observation that an XHTML-document can be seen as a tree if we ignore external and internal links. Links could be handled with ordinary associations on the model level and links on the model instance level, but we have postponed this issue for a

later version. By doing this simplification only the following object-oriented concepts are needed: class, property and composition.

The following subsections presents the constraint modeling technique by giving an example and how to implement the tool is also briefly described.

E.4.1 The XHTML-document and the Accessibility Constraints

We used the three constraints defined in [GNPT05] as examples of accessibility constraints. These constraints are derived from the WCAG 1.0 Guidelines [W3C99]:

1. Each image has to have a valid alt tag associated with it to provide text equivalents.

2. Only one of HTTP-EQUIV or NAME is allowed as attributes to a META tag.

3. The color of the text should have enough contrast with the background color. Checkpoint 2.2 in the WCAG says that it is important to ensure that the foreground and background color combinations provide sufficient contrast when viewed by someone having color deficits or when viewed on black and white screen.

These constraints are formulated in OCL as follows:

```
1. Context Img Inv:  libAcceptableAltTag( alt )

2. Context meta Inv:  name.size() > 0 xor
   http-equiv.size() > 0

3. Context font
   Inv:  libAcceptableContrast(textColor,
   getBody(this)->any(true).background)
   -- We need an auxiliary recursive function that
   -- finds the body for this definition.
   Context block
   def:  getBody(b :  Block) :
   Set(Body) =
   if b.body->size() = 1 then body
   else b.getBody(composite) endif
```

The functions `libAcceptableAltTag` and `libAcceptableContrast` are library functions defined in the framework.

We use the web page with the following XHTML code as example:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en"
>
<head>
<meta http-equiv="Content-Type"
content="text/html;charset=utf-8" />
<meta name="generator" content="gedit" />
<title>Test</title>
</head> <body bgcolor="yellow">
<img src="http://www.eiao.net/kickoff.jpg" alt="" />
<br />
<font color="black">Participants at the EIAO Kickoff
meeting in Grimstad,2004-10-14.</font>
</body>
</html>
```

The first constraint will expose that the document has an empty `alt`-tag which is in conflict with the WCAG 1.0 Guideline 1. The second constraint will test the instances of `meta` and find that this constraint is fulfilled.
The third constraint refers to the `font` context; the call `libAcceptableContrast("black","yellow")` will give the result true; so this requirement is fulfilled for the `font`-element in the document.

E.4.2 The Three Level Metamodel Architecture

Inspecting an XHTML-document, or more generally an XML-document, reveals that you have elements, some elements have attributes and some have content (elements inside another element). In our approach an XML-document is related to a model instance and a model in the following way:

XML Element An element in an XML-document is seen as an object of a class with the same name as the element-name.

XML Element Attribute An element attribute in an XML-document is seen as a slot with the same value as the value specified for attribute; the slot is seen as an instance of a property with same name as the attribute; the property is a property of a class with the same name as the element containing the attribute.

XML Element Content The content of an element is in a composition-relation to the containing element; the content is seen as part of the element.

There is no special translation of hyperlinks. Fig. 41 shows how the chosen XHTMLdocument is represented as a model instance, also the model and metamodel is shown.

The translation schema demands a metamodel that can support class, property and composition. The presented metamodel fulfils this demand and is an extremely simple reflexive metamodel[13].

Multiplicity is not defined; we simply see the multiplicity as fixed to zero-to-many on the part side of a composition and zero-to-one on the composite side of the composition (this can be specified as well-formedness rules in OCL). Constraints can be attached to classes in the form of text, e.g. constraint number two (described above) will be attached to class Meta on the model level! The metamodel defines that compositions are named; this is ignored at the model and model instance level because no names are given.

E.4.3 The Process of Defining and Evaluating the Constraints

The metamodel is defined in advance and it is this metamodel that dictates the translation schema and conceptually it defines our world view. Obviously the metamodel does not have to be in place (built as a complete level in the framework) unless it is accessed which is not the case in the presented approach; as far as we know other metamodeling frameworks would require that the meta-model is in place, but not the SMILE framework.

On the model level only the classes, properties and compositions referenced by the constraints need to be modeled by the user, e.g. the user only needs to define class `meta` with attributes `name` and `http-equiv` if constraint two (see above) is the only one to be defined.

We have developed a reader that reads an XHTML-document and converts it to a model instance as defined above; the reader is at the same time building a model, e.g. an element named `meta` gives a class `meta` on the model level as described in the subsection above. When the reader has done its job, the predefined model (which contains the constraints) is merged with the model that has been automatically created; at this point the constraints are tested on the model instance and a report describing the accessibility violations is created.

[13] A metamodel is reflexive if it defines all the entities it uses for its own definition

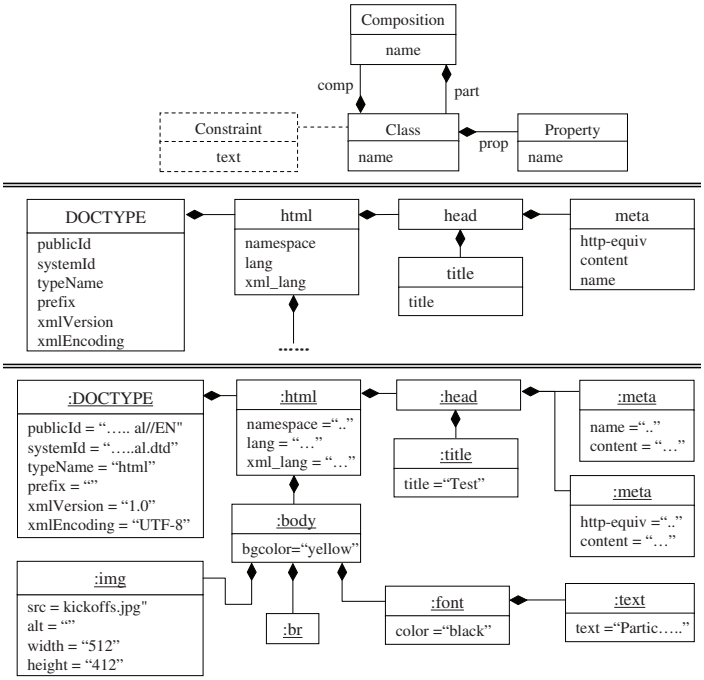

Figure 41: The Metamodeling Architecture Presented with the help of UML (the model level is only partly shown)

E.4.4 Benefits

Modern techniques for doing semantic validation of XML data are in nature declarative and typically based on XML-technology, e.g. Schematron [Jel02] based on XPath, and xLinkit [NCEF02] based on XPath. Formalisms based on XSLT and XPath are considered hard to write, understand and maintain (see [CBC05]).

Schematron is more suitable for simple rules. Therefore we used Schematron in the first release of EIAO for expressing the basic building blocks of the accessibility tests. In order to express the higher-level tests, we will use a higher-level formalism, probably based on OCL.

[CBC05] was published at the same workshop as [GNPT05], it describes a general approach for semantic validation of XML data based on metamodeling; the paper describes some strengths that also apply to our approach. Please find below a non-exhausting list of the advantages.

- Only one formalism is used (UML/OCL) and the constraints are specified at a high level of abstraction.
- It is partly graphical (UML).
- Deviations from the XHTML standard are allowed.

Some more benefits are added by our approach:

- We allow testing of XHTML-documents against constraints on fragments of models, i.e. we do not need the full specification of the XHTML standard.
- The parts of a document that conform to common deviations can be tested for accessibility; this is done by including model elements describing common deviations and attaching accessibility constraint to those elements.
- We apply special library functions in the OCL accessibility constraints, e.g. functions for testing foreground against background colors.

Our project is at the time of writing not yet completed (some important parts of OCL are not yet supported). In particular we want to investigate extensions of the navigation possibilities in OCL, e.g. navigating through objects of types that were unknown at model time (deviations from the standard), e.g. looking for background colors defined by elements of unknown type. However, we think we can already claim with some confidence that this approach to web accessibility assessment allows a flexible description of accessibility criteria and their evaluation.

E.5 Conclusion

Accessibility requirements and web technology are constantly evolving. High level modeling of accessibility requirements can support more rapid generation of new test modules and improve the understanding of the accessibility barriers for web documents. UML seems to be the natural choice for this modeling. The basic representation and instantiation technique from the SMILE project is very useful when it comes to the implementation of these ideas.

Starting from the prototype tool we have created, we will extend the subsets of XHTML and WCAG 1.0 covered. We will integrate a complete OCL interpreter into the SMILE framework, such that it is possible to express more constraints than just simple comparisons. Moreover, we will extend the library of functions needed to do sensible checks for accessibility.

This work was the result of successfully bringing together two relatively unrelated projects at AUC. The synergies between the SMILE and EIAO projects proved fruitful for the high-level description of accessibility requirements.

Appendix F

Automatic Generation of Modeling Tools

Jan Pettersen Nytun[1], Andreas Prinz[1] and Merete S. Tveit[1]

[1]Faculty of Engineering, Agder University College
Grooseveien 36, N-4876 Grimstad, Norway

Second European Conference on Model Driven Architecture Foundations and Applications
July 10-13, 2006, Bilbao, Spain

This appendix presents the paper: *Automatic Generation of Modeling Tools* [NPT06], with coauthors Andreas Prinz and Merete S. Tveit.

The paper focuses on design of a metamodeling framework and several existing frameworks are described. The paper is presented below:

Higher-level modeling is considered to be the answer to many of the problems computer science is faced with. In order to do modeling, it is necessary to use proper tools. This article is about modeling tools and how they can be generated automatically out of (modeling) language descriptions. Language descriptions in turn are given in meta-models. In this article, we define a terminology for aspects of meta-models and check how they are supported by existing metamodeling tools. In particular we look at semantic aspects of the meta-models.

F.1 Introduction

Information technology is spreading more and more into all areas of daily life, leading to an ever increasing amount of information and applications of a very high complexity. Traditional methods of software production and data handling cannot cope with this ever increasing complexity. New ways of complexity handling take higher levels of abstraction and describe systems using models. In particular, OMG puts forward their idea of a model-driven architecture (MDA) [OMG03a] which focuses on software development by means of high-level models. We will use the term MDD (model driven development) in the sequel to denote an approach taking high-level descriptions for the generation of low-level results, e.g. executable code. For an effective application of MDD it is necessary to use models that fit their application domain, which means to use domain specific languages (DSLs) or domain specific adaptations of languages.

This leads to the problem of the development of DSL tool support. The currently existing tools support common multi-purpose languages, but are not particularly adapted to a specific domain. On the other hand, developers insist on integrated development environments to be effective in their daily work. In this context, the choice is either to take a not fitting language with a good tool support or to use a well fitting language with no tool support. Of course, none of these alternatives is satisfactory.

So the problem is to provide tool support for modeling languages. There are basically two ways to achieve this, either by manually building such tools or by having higher-level tools that generate modeling tools. In any

case it is necessary to have a description of the language first. We will call such a description of the language a meta-model. There are varying levels of accuracy when it comes to describing meta-models and also a whole range of tools that support parts of this tool production by automation. Of course, also meta-models are just a special kind of models and for their handling we again need (meta-)modeling tools. This closes the circle and we can apply the same reasoning on the next level. So in all the levels we have the need of powerful tools that are able to handle models of different kind.

In this paper we will focus on this need for generating modeling tools. We will first in section F.2 look at the different requirements coming for these tools. Then we will look at a new class of integrated tools claiming to support the complete description of languages in Section F.3. Section F.4 concludes the paper.

F.2 Meta-modeling and Tool Production

A modeling tool is a tool that is able to handle models of a certain kind. The description of the model kind is given by a meta-model, or in simpler cases by an abstract grammar or even by a concrete grammar.

[GSCK04] defines metamodeling as: *... the construction of an object-oriented model of the abstract syntax of a language.* However, in our article we use the term meta-model in a wider sense: *A meta-model is a model that defines a language completely including the concrete syntax, abstract syntax and semantics.*

The current situation of meta-model use is characterized by the following observations.

- Meta-models are usually *not given explicitly*, but are built-in into the tools that provide them; this can be seen as a sort of hard coded implementation of a meta-model. In particular there is no direct relation between an external meta-model and the representation in the tool.

- Meta-models *change over time.* Tool builders adapt their meta-models along with their tools and do only provide means to align with their own old versions.

- Meta-models *are not standardized.* Although several organisations, in particular OMG, try to publish standards for meta-models, the standards are far from being formal and implementations deviate more or less severely from the standards.

This leads to the fact that users are bound to one tool at a time. They are allowed to import models from other tools, but then they are again encapsulated. On the other hand, most metamodeling tools provide a set of basic facilities that are the same and some advanced facilities that are specific. A user is usually not able to combine the positive parts of different tools.

In this section we will be looking at tools, aspects of meta-models and how tools and meta-models are related.

F.2.1 Aspects of Meta-models

The meta-model can have several aspects that are to be covered by the modeling tool. In figure 42 we have shown the essential parts of a meta-model. There is no complete agreement about these parts, but in most contexts the same or similar parts are identified. In the picture, we have shown the following parts.

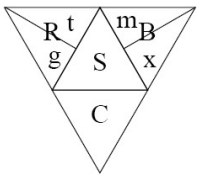

Figure 42: Structure of a Meta-model

Structural information for the meta-model including all the information about which concepts exist in the domain and how they are related. An example of this would be a MOF class diagram. In our understanding this part does just include very simple structural properties and not more advanced concepts that rely on the use of constraints.

Constraints giving additional information about the structure in that they identify the allowed structure according to additional logical constraints. This will include first-order logic constraints (e.g. written in OCL) as well as multiplicity constraints. In classical compiler theory these are collected under the name of static semantics and in a meta-model context they are called well-formedness rules.

Representation description includes model serialization syntax and information about how the models are to be (re)presented to the user.

The textual grammars (concrete textual syntax) are well understood in terms of compiler theory. When it comes to graphical grammar (concrete visual syntax), there is less agreement and also some open research topics.

Behavior description describes how the model is used. This item includes execution of the model as well as mappings. By mapping we will understand a relation between the model itself and another representation, e.g. in another language. A typical example would be a compiler from UML to Java, or mapping from PIM to PSM. An execution is the real run of the model, which is of course only possible if the model is executable. A typical example here would be a run of a UML state diagram.

In the picture given, the structure is the central aspect and all the other parts relate to the structure. This is quite clear for the constraints, which need the structure to be meaningful, but also for the representation and the semantics. Most language descriptions do currently follow this approach, i.e. defining a structure first and attaching all the static and dynamic semantic information to this basic structure.

When we take a step back, we will notice that the representation as well as the semantics are not that closely bound to the structure. In fact, several modeling languages use the same representation in order to represent similar things and also the semantics is largely comparable although the internal structure might be different.

For this situation, the MVC (model-view-controller) pattern is better suited. This means in our case that the connection between the representation and the structure and between the semantics and the structure is not direct, but mediated via a controller. This will allow to associate both with each other as shown in figure 43.

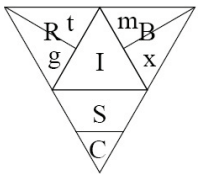

Figure 43: Decoupling the Structure of a Meta-model

In the new structure, the middle is just connecting the parts as described below.

Integration gives the connections between the different model parts. Each part forms a unit of its own, e.g. the syntax is described separately without reference to the basic structure of the language. Afterwards, the integration allows to connect these separate parts.

Please note that the explicit connections are already implicitly present in figure 42. We have just extracted them explicitly in order to allow a better handling of model descriptions. In the following we will ignore the explicit connections and use the figure 42 as a reference.

F.2.2 Tools as Meta-model Implementations

We have discussed in the previous section how meta-models describe the possible models to be handled. When we now look at tools, we can see that tools have the same property as meta-models. They also define what kind of models are allowed, how they look and what you can do with them. This way, a tool can be considered a special meta-model as shown in figure 44.

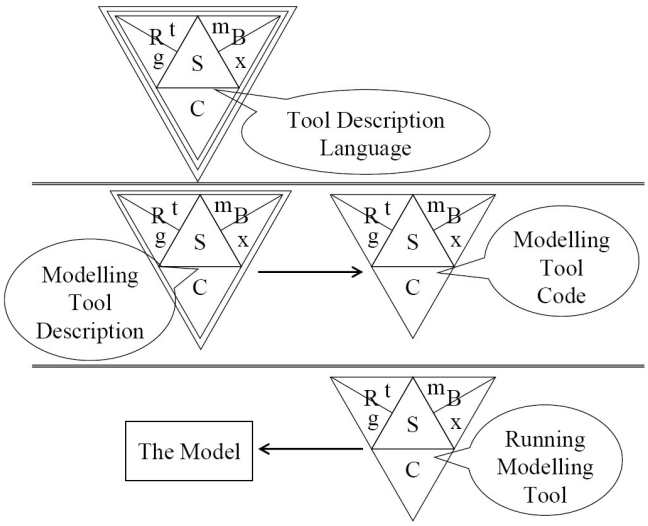

Figure 44: Tools and Meta-models

The meta-model gives a description of the tool, which in turn can be transformed into tool code. This code has then to be executed in order to be a running tool, which then can handle a model.

Figure 44 does also match nicely with the OMG 4-level architecture. The model would here stay on the level 1 (models), the tool code and the meta-model would be on level 2 (meta-models) and then we do also have languages described on level 3 (meta-meta-model).

This meta-meta-aspect goes into the next level of description. In order to have (formal) description of the meta-model, we need a (formal) meta-meta-model which can be used to provide this description. Alternatively, we can use known ad-hoc solutions, e.g. using a low-level programming language for doing the description. Of course, all the aspects identified in figure 42 for a meta-model have to be supported on all levels. For the tool, there should be code for each of them; in the meta-model we need a description for each of them and in the language level we need a language for each of them.

F.2.3 Tool Production Requirements

When it comes to tools that produce modeling tools, we will look at the following requirements:

Generativeness: As we speak about tools that produce modeling tools, the most important requirement is that they are able to automatically produce the tool. In figure 44 this amounts to the mapping from the meta-model to the tool code.

High-level Description: The descriptions are more easily handled when they are given in a high-level notation. This means that a tool should provide high-level notations for the different modeling language aspects. This is reflected in the figure 44 by the top-level layer.

Completeness amounts to the coverage of the different aspects introduced in the previous section. A good meta-tool will allow the expression of all important aspects of a modeling language. This requirement is reflected in figure 44 by the amount of the meta-model aspects that are covered. Please note that completeness is applicable for the tool, for the tool description and for the tool description language.

Conformance to Standards is given in this respect very easily when the tools are produced automatically from the corresponding standards documents. For this to be possible the standards documents have to be given in a formal way.

User friendliness: Of course, for generated tools there is also the aspect of user friendliness. As we focus on the very generation of the tools, this end-user aspect is out of our focus.

When we look at the requirements, we see that they are all completely covered in the two upper levels of figure 44. Therefore we will use these two levels as the reference for comparing several metamodeling tools in the next section. There, we just identify which aspects are supported and if they are described formally or built-in. If they are described formally, we check if they have a high-level notation or if they are given using a low-level language. The template for the comparison is therefore the two upper levels of figure 44.

F.3 Some Meta-modeling Frameworks and Tool Production

In this section, we will compare different metamodeling frameworks according to the structure presented in the previous section.

F.3.1 MDA Meta-modeling

Karl Frank [Fra05a] states the following:

> At the core of MDA are the concepts of models, of meta-models defining the abstract languages in which the models are captured, and of transformations that take one or more models and produce one or more other models from them.

Since OMG introduced MDA in 2001, much work has been done in defining this approach with proposed specifications and implementations. Please find below some specifications that together cover all langues aspects of figure 44:

- For serialization: XMI [OMG07a] based on XML and UML 2.0 Diagram Interchange Specification [OMG05d].
- For concrete textual syntax: Human-Usable Textual Notation [OMG04].
- For concrete graphical syntax: Human-Usable Graphical Notation [OMG04].
- For transformations: Query/View/Transformation Specification [OMG05a] which also has a reference implementation.

- For execution: Action Semantics [OMG05c] (no concrete syntax defined).
- For constraints: OCL [OMG05b].
- For abstract syntax: MOF [OMG03h].

When it comes to tool production the specifications are important with respect to "input" and "output", e.g. code conforming to the Action Semantics specification [OMG05c] might be produced as output and run on some UML virtual machine.

The QVT [OMG05a] might turn out to be important since the jobs a tool does in many respects can be seen as transformations.

Today there is no single tool or coherent set of tools producing a family of tools that conforms to the listed specifications.

For MDA to work in practice models have to be unambiguous and their semantics have to be precisely defined - UML does not fully comply with this demand [GSCK04].

In many respects UML has been defined as a general modeling (programming) language (but without fully described semantics) - a DSL, on the other hand, is specific (by definition), such that a UML tool might not be the right tool for expressing statements in a DSL (considering a DSL a subset of UML).

If the UML tool allowed advanced configuration (e.g. excluding parts of the UML language), supported the extension mechanism of UML (profiling), then the UML could be set up as a DSL tool; but even this might not work well in all cases since UML after all is a predefined language based on some language design decisions - this is the opposite argument of the "missing semantic" argument, UML might be to specific in "the wrong way"! It seems harder to reject MOF in the context of defining DSLs (which is done in [GSCK04]); if some semantic is missing then add it!

F.3.2 XMF-Mosaic

XMF-Mosaic from Xactium is a platform for building tailored tools that should provide high level automation, modeling and programming support for specific development processes, languages and application domains. The tool is implementing a layered executable metamodeling framework called XMF that provides semantically rich metamodeling facilities for the design of languages. This way the Mosaic platform is realizing the Language Driven Development (LDD) process presented by Xactium in [CESW04]. LDD is a

model-driven development technology based on MDA [OMG03a] standards, and it involves adopting a unified and semantically rich approach to describe languages. A key feature of the approach is the possibility to describe all aspects of a language in a platform-independent way, including their concrete representation and behaviour. The thought is that these language definitions should be rich enough to generate tools that can provide all the necessary support for use of the languages, such as syntax-aware editors, GUI's, compilers and interpreters.

XMF provides a collection of classes that form the basis of all XMF-Mosaic defined tools. These classes form the kernel of XMF and are called XCORE. XCORE is a MOF-like meta-metamodeling language, and it is reflexive, i.e. all XCORE classes are instances of XCORE classes. XMF provides an extensive language for describing language properties called XOCL (eXtensible Object Command Language). XOCL is built from XCORE and it provides a language for manipulating XCORE objects. In addition to XCORE, XMF provides a collection of languages and tools defined in XOCL.

The general architecture of a tool or a language built using XMF-Mosaic is as follows:

Structure At the heart of most XMF-Mosaic tools is a meta-model, in XMF called the domain model. This meta-model describes the structure of the concepts in a language or in a domain. The language for building the structure is XCORE.

Constraints For adding constraints to the domain model, XMF-Mosaic supports a constraint language based on OCL. It is also possible to create instances of the domain model and test them against their constraints.

Representation This is also called the user-interface model in XMF, and describes the concrete representation of the concepts in the domain model. For this purpose XMF-Mosaic provides XBNF, which is a grammar definition language for defining the textual syntax, and XTools which is used to specify the concrete graphical representation of a language and to model user interfaces.

Behaviour The language XOCL is used to build executable tools with executable semantics. XMF-Mosaic also supports the representation of model-to-model transformation and model-to-code mappings, including generation of Java from XCore models and XML serialization of

models. The language XMap is a pattern-based language that is used to write model-to-model transformations.

According to this, XMF-Mosaic is fully covering all the aspects of the template in figure 44.

F.3.3 Coral

Coral [AP04] is a meta-model independent framework, which means that it positions itself at the top of the OMG's meta-model architecture and then creates a meta-meta-model interface. In figure 45 it is shown which parts of the template (see fig. 44) Coral supports by indicating them in grey. It was a bit problematic to describe Coral according to the template, because the tool is not fully documented.

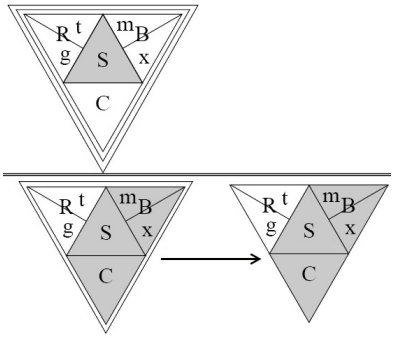

Figure 45: Aspects supported by Coral

Coral is divided into two main components: the kernel and the graphical user-interface. The kernel is implementing a model repository. This repository could be seen as a program library or an application framework that is used to manage models described in the user-defined modeling language. This model repository is based on a specific modeling language, Simple Metamodel Description Language, which defines the structure of all modeling languages in Coral. SMD can be seen as analogue to MOF, but SMD contains some extensions to deal with models described in multiple modeling languages. When Coral needs the definition of a modeling language, the SMD model for this language is loaded and converted to a meta-model internally. This way Coral provides full support for all structural aspects of meta-models.

The graphical user-interface in Coral can be used to view and edit models manually. The kernel and the graphical user-interface are independent. This means that the kernel can manage and transform models even if the user-interface cannot render them graphically.

Currently Coral is coming with some predefined modeling languages, such as UML 1.1, UML 1.3, UML 1.4 and UML 1.5, and also the XMI-DI 2.0 [OMG05d]. Coral can load and save models and meta-models using XMI 1.0 and XMI 2.0 format. It is also possible to load and save models containing diagram interchange information using XMI-DI, and this format is also used to represent diagrams. When it comes to interactive graphical support, this is missing, and support for every diagram must be written explicitly.

One feature in Coral is the possibility to query and modify models at runtime. This is done by creating Phyton wrappers around the Coral kernel, which is written in C++. Model transformation can be written as Phyton programs with separate phases for precondition, query and modification and post conditions. The Phyton interface in Coral makes it possible to query models in a very similar way to OCL [OMG05b], thus allowing constraints and transformations and executions to be expressed. Because there is no specific language to express these things, but just Python modules, we have not indicated these parts in the top-most language layer.

F.3.4 Software Factories

Software Factories are described in [GSCK04] in the following way:

> A software factory is a product line that configures extensible development tools like Microsoft Visual Studio Team System (VSTS) [Mic06] with packaged content and guidance, carefully designed for building specific kinds of applications ... the software factory schema specifies which DSLs should be used and describes how models based on these DSLs can be transformed into code and other artefacts, or into other models ... the software factory template ... provides the patterns, guidance, templates, frameworks, samples, custom tools such as DSL visual editing tools, scripts, XSDs, style sheets, and other ingredients used to build the product ... When configured with the software factory template, VSTS becomes a software factory for the product family.

Software Factories are promoted by Microsoft and can be based on tools like VSTS - which is a tool that lets you develop Microsoft .Net Framework applications. In .Net many different languages can be used; compilation is done to a common binary language (IL) which can be executed by the same runtime engine. The .Net approach gives integration of different general purpose languages (e.g. C# and C++) and this seems to be a good starting point for the development of a DSL framework.

The Software Factories method describes a MDD approach that is not based on UML or MOF; it opposes the MDA which is based on UML and claims it to give insufficient support to development of DSLs.

A comprehensive example is given in [GSCK04]; the following list describes the elements that constitute a DSL:

1. Abstract syntax graphs instantiated from meta-models and also abstract syntax trees instantiated from context-free grammars.

2. Layout information instantiated from concrete syntax. Concrete syntax is described with annotations on meta-model elements, e.g. class Identifier has annotation: `[$shape: TextBox]`.

3. Serialized abstract syntax graphs and layout information which conforms to defined serialization syntax. Serialization is not based on XMI, which is seen as too strongly coupled to the target language meta-model and also hard to read; they advocate the following: "..., the XML syntax should be designed on a language-by-language basis, so that the language designer has the flexibility to change the mapping to accommodate different rates of change on either side."

4. Well-formedness rules defined with some "OCL-like" language.

5. Trace-based semantics describing what happens during execution; this semantics is described with a meta-model attached to the meta-model of the DSL; well-formedness rules can be attached in the "normal way"; a concrete syntax for the trace-based semantics is described (as above).

Software factories do also demonstrate how a meta-models can be broken down to parameterized language elements, called language design patterns, that can be glued together in different configurations - this gluing is considered a special case of model mapping.

Item 2 above describes how graphical layout can be attached; OMG has a different approach [OMG05d] which seems to be more flexible since it

defines a separate graph for the graphics - a graph that will be connected to the abstract syntax graph, e.g. one element of the abstract syntax graph might be represented with several nodes in the concrete syntax graph.

The arguments concerning the rejection of XMI (item 3) seems hard to follow, e.g. change from one XMI version to another can be performed by some (simple) transformation. It seems likely that this approach will lead to yet another standard!

It is hard to get an overview of tool support (tools that makes tools) when it comes to software factories since a product line is put together in a somewhat ad hoc way and since there is no specialized complete framework (as we know of) for supporting the software factory method - only more general frameworks with some pre-made components. On the other hand, [GSCK04] and articles like [Jac04] present a vision that includes full language support (a fulfillment of all the aspects of figure 44).

F.3.5 More Examples

Of course, the idea to generate language processing tools out of language descriptions is not new. The first attempts were grouped around the idea to generate grammar handling tools out of grammars. They have been successful in the area of lexical handling (e.g. [LS]) and in the area of parsers (e.g. [PQ95], [Joh]). It was quickly clear that these properties did not fully describe a language and several other approaches have been defined to capture the complete range of language aspects. However, none of these has had real success.

Currently, there are several initiatives towards the idea of a more complete language handling coming from different starting points. We have a closer look at two of them.

Intentional Software [Int, CE00] is an attempt to use the informal descriptions of a software in order to generate code from them. This way, the *intent* of the code is still visible later and the connection to the real code stays alive. It is very difficult to get deeper understanding of their technology from the publicly available information. What we have seen is that they allow the definition of languages that capture the intent at the level that the developer has meant it. Then they apply tools that make these descriptions valuable, i.e. they are transformed to code. It is not visible which kind of description languages are used in order to describe languages.

Meta-Programming System [MPS06] is coming from JetBrains. The

name does already say that this is a tool for meta-programming. They state "MPS is an implementation of Language Oriented Programming [Dmi04], whose goal is to make defining languages as natural and easy as defining classes and methods is today. The purpose is to "raise the level of abstraction", which has been a major goal of programming since the first assembly language was born." This way they also allow the definition of languages and the generation of tools out of the descriptions. It is not easy to see what languages they use for language description and which aspects they cover. On their website it is possible to get a pre-release of their tools for experiments.

GMF The Eclipse Graphical Modeling Framework (GMF) [GMF] is a promising open-source technology based on the Eclipse Modeling Framework (EMF) and the Graphical Editing Framework (GEF). One purpose of GMF is to support definition and implementation of Domain-Specific Languages. EMF provides its own meta-model, called Ecore which is very similar to EMOF (a subset of MOF 2.0). EMF includes support for XMI 2.0 serialization and reflection APIs; support for OCL has also been added. GEF is an MVC-based framework to create graphical editors. GMF brides EMF and GEF; it supplies a set of tools that allows you to define and then automatically generate a graphical Eclipse-based modeling tool. GMF seems quit complete already and it will probably play an important role as a tool making tool. This new approach has not had the time to mature and it is left to see if it is flexible enough to meet the demands of tomorrow.

MetaEdit+ [Met05] is a commercial metaCASE tool developed by the company MetaCase Consulting, Finland. The tool consists of two parts: the Method Workbench and the CASE tool. The Method Workbench is a dialog based interface, which allows the user to define the language concepts, their properties, associated rules, symbols etc. To describe the language concepts, a metamodeling language called COPPRR is used. GOPPRR stands for Graph, Object, Property, Port, Relationship and Role, which are the meta-types used to describe modeling languages. The CASE tool MetaEdit+ follows the language definition given in the Workbench, and provides a modeling tool according to this specification. MetaEdit+ support automatic code generation for predefined and user-defined programming language. The predefined includes: Smalltalk, C++, Java, Delphi, SQL and CORBA IDL.

F.3.6 The SMILE Framework

The SMILE project [NPK04, GNPT05, NP04] started as an attempt to implement a technology that allows high-level language descriptions to be interpreted or compiled into real tools.

The basic idea of SMILE is the application of MDD to the language handling itself. This is done by using high-level descriptions of the languages for creating complete development environments. The descriptions are given in high-level languages, thus allowing the application of the SMILE principle to itself, which is usually called bootstrapping or self-reference. This idea came out of the success of this technology in the implementation of the SDL formal semantics [ITU99, EGG+01, Pri00].

For language modeling, the SMILE methodology takes three steps:

1. the description of structure and semantics,
2. the automated generation of specific repositories and tools, and
3. the use of the generated repositories and tools for concrete models.

This methodology is based on a combination of metamodeling for information structure description with technologies to describe the semantics of that information accordingly. These description techniques, covering different language aspects, have to be adopted and aligned to create a common language modeling framework. With this new technology that integrates structure and semantics, the SMILE toolset will be able to generate data repositories and language tools that reflect the given semantics.

The SMILE methodology will be supported by a domain-independent framework that provides language support for information structure and semantic descriptions, making SMILE applicable to the described domains. To describe the information structure SMILE will use existing standards to describe a repository, e.g. MOF or RDFS. In the area of semantics SMILE distinguishes between five kinds of semantics: Static semantics that is described with a condition language based on OCL, execution semantics that will be handled through the ASM method, transformations formally described by rules and two ways to describe concrete representations, textual and graphical. SMILE will provide a) languages to handle these semantics and b) implementations that allow the generation of tools (model checkers, transformation engines, model editors and parsers) from descriptions in these languages.

The SMILE approach is best understood by looking at the meaning of the project abbreviation, which is Semantic Model-based Integrated Language

Environment. These parts stand for the following concepts.

Semantic: SMILE acknowledges the importance of explicit semantic descriptions in all places of the technology. The current approach to have informal descriptions of parts of modeling languages (most prominently the dynamic aspects) is not fitting the state of the art. There is enough knowledge about how language semantics can be formalized and there are even tools that can transform such explicit semantics descriptions into real tools (interpreters or compilers).

Model-based: The whole approach of SMILE is focused on the idea to handle models. Not only the descriptions of the software are models, but also the descriptions of the languages and the languages to describe them and even the generated code. In order to handle these models in a unified way, a basic model representation is used allowing to capture models internally. This is detailed below.

Integrated: The integration within SMILE starts with the unified model representation. Every bit of information in SMILE is handled in a similar way. This is achieved by using a basic instance representation with an explicit interface between metamodeling levels. This means, SMILE follow a strict metamodeling approach without connecting the levels to each other by default. In SMILE, a model can be connected to different meta-models if the interface between them allows this coupling.

Language: The most prominent examples of using SMILE are languages. In fact, in SMILE a model is just a kind of a language and vice versa. Therefore, the concentration on languages is not that special, because everything is a language in the end.

Environment: The final aim of SMILE is providing a complete modeling environment, which would also be a metamodeling environment. Moreover, the SMILE technology does also an easy integration of external modeling or metamodeling tools. The SMILE implementation is started in the scope of the Eclipse [DFK+03] platform using EMF [Gri03].

The SMILE project is still in its early phases and is not yet completely implemented. A basic representation called MATER (see [GNPT05]) was defined that allows the representation of any model (and meta-model) independently of the corresponding meta-model. This is possible since SMILE

has the complete information about the model and the meta-model encoded into structural properties. This allows models to be connected to different meta-models in SMILE.

F.4 Concluding Remarks

In this article, we have defined a terminology for the comparison of environments that generate modeling tools. This framework is very heavily related to metamodeling. There are several current initiatives to create such an environment, and although very few results exist so far, we can conclude that almost all approaches focus on the same aspects of languages, namely structure, constraints, representation (textual and graphical), and behavior (mapping and execution).

Despite these striking commonalities, there are also several differences, that mostly relate to the semantics of the parts. In all MDA-related approaches a fixed exchange format (XMI) is taken as part of the structure semantics. Software factories argue that this is not needed and will use a specific format defined for each language instead. This kind of reasoning is understandable when one thinks of the many versions of XMI and that they do not really achieve the goal of exchangability. However, we still think that in an ideal setting a basic exchange format should be defined independently of the concrete language. This is taken into account in the SMILE framework in that we consider also the semantics of structural information to be given by the description language of structural information, and a general way of exchange can be described there in terms of textual representation. This way, it is just a special case and would also be possible the same way in software factories.

Another difference are the concrete languages put forward for expressing the different aspects of meta-models. Surprisingly, very concrete languages are used, although they are defined based on meta-models. There is not much work in integrating these different formalisms. Only the SMILE project tries to tackle this problem, but they are at the very beginning of their work.

Finally, it remains to be said that almost all approaches take the language structure for granted and do not allow handling of changes to the meta-model. As these approaches are that similar, it would be a very good idea to allow them to integrate, i.e. that there are ways to use the models of one approach also in another approach.
The plans described in the different environments sound very promising and

could lead to a completely different way of software development, once they are fully implemented.

Appendix G

A Generic Model for Connecting Models

Jan Pettersen Nytun

Faculty of Engineering, Agder University College
Grooseveien 36, N-4876 Grimstad, Norway

First International Conference on Software and Data Technologies -
ICSOFT 2006, Setubal, Portugal

This chapter presents the paper: *A Generic Model For Connecting Models In A Multilevel Modeling Framework* [Nyt06], written without any coauthor and presented at ICSOFT 2006.

This paper uses the same application as presented in Appendix A [NJ03]; it describes how to achieve a solution in a multilevel modeling environment. A model for connecting models on the same and on different levels is defined. The paper is presented below:

In science and elsewhere models are weaved together forming complex knowledge structures. This article presents a generic way of connecting models with model borders both vertically and horizontally in a multilevel modeling framework. One model can be connected vertically to several models allowing a model element to be an instance of several metaclasses and different views can then be managed in an integrated way. Models at the same level can also be connected by defining the correspondence between model elements.

The idea behind the approach is to break model architectures down to elementary building blocks so that all parts that might be of interest become explicit and accessible.

G.1 Introduction

In this article some of the ideas behind a metamodeling framework called Semantic Integration World Animation (Siwa) is presented; this framework is under development at Agder University College and it is meant for learning and experimentation; it is an offspring of the SMILE project [NPK04] which is more directed towards integration of existing language technologies.

MOF metamodel [OMG03h] architectures have a pyramid structure, while a Siwa architecture is like a directed graph with models as nodes. The graph is not cyclic except maybe for the top models (e.g. level `M3` in the UML metamodel architecture).

OMG has issued a request for revision of MOF 2.0 [OMG06a], some of MOFs restrictions are becoming increasingly burdensome. MOF does not allow properties to have an independent existence and multiple classification is not possible - Siwa can be used without these restrictions. It will also be possible to specify architectures that are not complete, e.g. that a metamodel is missing; this opens up for data analysis, reasoning about models and in some cases the framework might automatically suggest a metamodel.

Some models are static structures and some are executable models. We

call the executable models *semantic engines*, some sematic engines are presented but they are not the main issue in this article.

A model is to a large extent defined by the role it plays in relation to what it models; two basic roles are defined by Thomas Kühne [Küh05]: *token* and *type*. A token model captures the singular aspects, while a type model captures the universal aspects of what it models. A class `Building` might capture the universal property that buildings have owners. An object that models one specific building is a token model for this building, e.g. it might capture the name of the owner. The focus of this article is a technique for connecting models, the following *model configurations* are to be supported:

Vertical (type model) This is the type model role which spans two model levels (some would call this the `instanceOf`-relation). Several models can be type models for the same model instance; this is not supported by MOF.
There are variations of this relationship, e.g. a model instance might actually have been instantiated from the model or the model is describing only some aspects of the model instances.

Horizontal (token model) We see the need for a model-to-model relationship which do not span a level border, but is between two models that are considered to be on the same level. Several models can in different way and with different level of granularity and detail model the same thing; these models are related with this relationship.

Fig. 46 demonstrates, as we understand it, both token and type model roles (the UML notation has been used in an ad hoc fashion).

The transitive property of the token model role can also be seen in Fig. 46: the `TMBuilding` class is a token model for class `Building` which is a token model for "the concept of a building", consequently `TMBuilding` is also a token model for "the concept of a building". It seems to be a growing agreement [Küh05, Fav04a] that a metamodel is a type model for another model which again is a type model for its model instance. These models are forming a stack structure where you don't have the same transitivity as for the token model role. Subclassing is a transitive relation and should not span a level border [Küh05, Fav04a].

UML has no diagram type that truly spans several (metamodel) levels; UML *object diagrams* shows *instance specifications* (instances of metaclass `InstanceSpecification`) and also classes are allowed; an object diagram

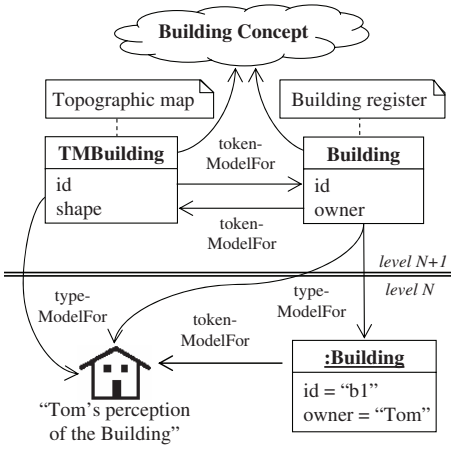

Figure 46: Example of token and type model

is placed on the model level (`M1`). As the name indicates, an instance specification is a specification and it might in fact specify properties of several instances at the model instance level (`M0`). According to this understanding, an object diagram is correctly placed on `M1` because an instance specification is not "truly" in a horizontal relation to an instance on `M0`. In our view, if a model is in a horizontal relation to another model, then both models are modeling "exactly the same specific thing" even if the number of details and precision might be different; the models should consequently be placed on the same level since they model the same thing.

The idea behind our approach is to break model architectures down to elementary building blocks so that all parts that might be of interest become explicit and accessible; the framework should of course allow the user to view an architecture at different levels of granularity and with different concrete syntaxes that hide the underling complexity.
Using the example in Fig. 46: object `:Building` is a structure that contains a slot called `id` with value `‘‘b1"` and a slot called `owner` with value `‘‘Tom"`; we consider `Building`, `id` and `owner` to be symbols that forms an *upper border* to the type model containing class `Building`; we actually have two borders (or border sides), one for each models being connected, this allows different number of symbols at the borders and it allows symbols to have different names.

Sec. G.2 presents our multilevel (meta-)modeling framework and explains how models are connected. In Sec. G.3 we mention some related work. Sec. G.4 presents an example of how Siwa can be used to do testing of data consistency. We summarize and describe some research directions in Sec. G.5.

G.2 The Siwa Approach

Lately the term megamodel has been used to name a model of MDE itself, e.g. by Bézivin and Favre [BJV04, Fav05]. Such a megamodel describes the concepts of MDE - concepts like model, metamodel and transformation. Fig. 47 shows our preliminary megamodel which has been inspired by Favre [Fav05].

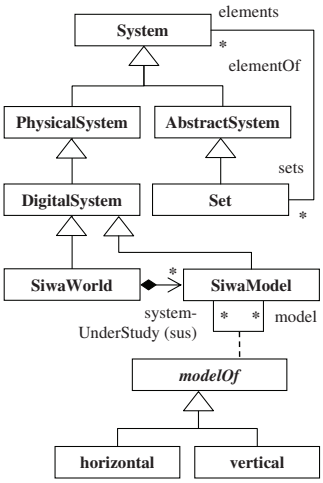

Figure 47: Megamodel

As we can see from Fig. 47 basically everything is a *system*. In [FBJV05] a system is described as a group of interacting, interrelated, or interdependent elements that form a complex whole.
Abstract systems can only be described since they are not to be found in the *concrete*; *physical systems* are concrete and manifested "in reality". We consider a *computer system* to be a special type of *physical system* since

they are manifested in computer hardware.

The Siwa framework supports the notion of levels like you find it in metamodeling architectures defined by OMG. Atkinson and Kühne [AK01] have earlier used the term *multilevel metamodeling*, we prefer the term *multilevel modeling framework* since an arbitrary number of levels will be supported including metamodel levels. We call a multilevel metamodel architecture defined in the framework for a *Siwa world*. Fig. 47 defines a Siwa world as a special type of digital system. A Siwa world is composed of Siwa models which also are considered to be special types of digital systems. The `modelOf` relation comes in the two generic types: vertical and horizontal.

A `Siwa model` can be a model for an abstract or a physical system which is not part of a Siwa world. This possible relation is not depicted in Fig. 47 since it can not be explicitly represented in the framework.

Favre presents briefly the notion of *static* and *dynamic system* in [Fav05], a Siwa world also has these two aspects which we call: Model All Types with Extent Realization (MATER) and Play Activations and Transformations with Extent Realizations (PATER). The focus of this article is MATER and in the following subsections MATER is presented with the help of UML notation and examples. PATER is touched in Sec. G.4 when some sematic engines are described.

In Subsec. G.2.1 we present how to represent the internal structure of a Siwa model, in Subsec. G.2.2 we describe how Siwa models can be connected to constitute a Siwa world (a multilevel model architecture).

G.2.1 Representing One Model

Fig. 48 presents the part of MATER that is used when one model is to be represented, it is meant to be used on all the levels of a Siwa world. There are several similarities between this part of MATER and MOF [OMG03h] as an instance model, e.g. an object can be represented as an instance of `Structure` containing instances of `Slot` with values representing properties of the object and `Descriptor` can be used for type information as described later. Links between objects can be represented as instances of `Link`. The property/owner associations can be used to represent relations between sets; since there is no reference to a `Descriptor`, instantiation of these associations might in some cases lead to ambiguous situations when it comes to finding their description on the level above. Fig. 48 has two types of symbols:

Descriptor This symbol type is used to indicate classification and it is used to establish a vertical relation, e.g. structure describing a building called `b1` might have a descriptor called `Building`.

Identifier An instance of `Identifier` is labeling a part of the model being defined and functions as an identifier for this structure, e.g. an identifier `b1` might reference structure that describes a building with that id. Another example would be an identifier `Building` referencing a structure that describes a class `Building`. Several identifiers might reference the same structure; in some cases this means that there are synonyms.

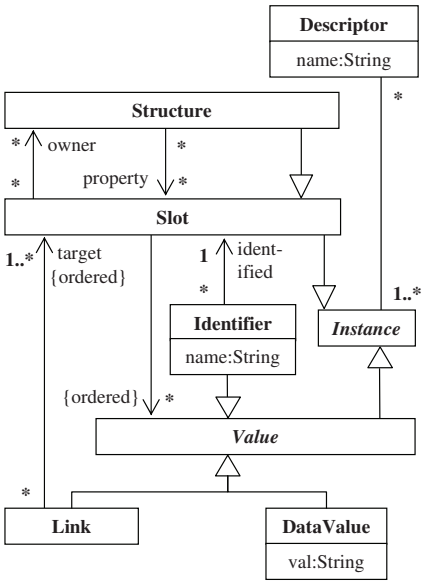

Figure 48: Part of MATER: the internals of a Siwa model

For us as humans the symbols are typically telling what a model is about, from the "framework point of view" only what has been formalized and represented in MATER "does matter".

An example of how an object of type `Building` can be represented is given in Fig. 50. In Fig. 50(a) the object is shown in UML notation, Fig. 50(b) shows how MATER can be instantiated to represent the same. Fig. 50(c) is showing an overview of the `Building`-object in an ad hoc notation where the structure is hidden except for the symbols (two *bor-*

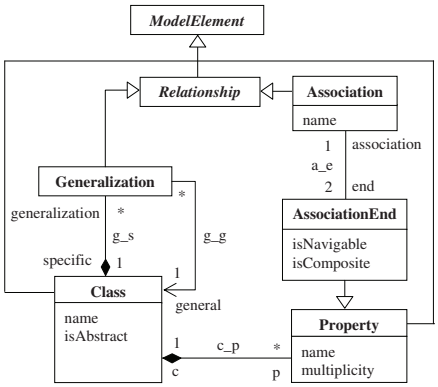

Figure 49: The top metamodel of the examples

ders are shown, marked `U` and `L`, this concept will be explained below). Fig. 49(a) shows the top model which is called Class-MM. It describes important object-oriented concepts like: abstract and concrete class, property, multiplicity, association and generalization.

Fig. 51(a) shows a simple class called `Building` with a property called `id`; Fig. 51(b) describes how Class-MM can be instantiated to get the class and then Fig. 51(c) demonstrates how MATER can be used to describe the class (the `:Property`-object is not shown). As we can see from the figure the number of model elements is huge - it correspond approximately to the number one would get if the UML metamodel was instantiated.

In the object-oriented literature, and also in this article, the difference between a *class as a set* (something abstract) and the *description of a class* is often confused[14]. As a consequence of being abstract: it is not possible to "point to" a class and say "there it is", but it might be possible to point to the instances of a class and also to a description of a class[15]. MATER does not have *class* as a built-in construction - there is no model element in Fig. 48 called `Class`, but it is possible to describe classes (e.g. Fig. 51(c)).

Seeing a class as an object is not in conflict with the UML metamodel architecture since a UML model can be seen as composed of objects instan-

[14]In our view reification is merely to establish a descriptions of a concept.

[15]The terms *abstract class* and *concrete class* used in object-oriented programming is something else, in that context a concrete class means that there are objects that are direct instances of the class which is not the case for abstract classes.

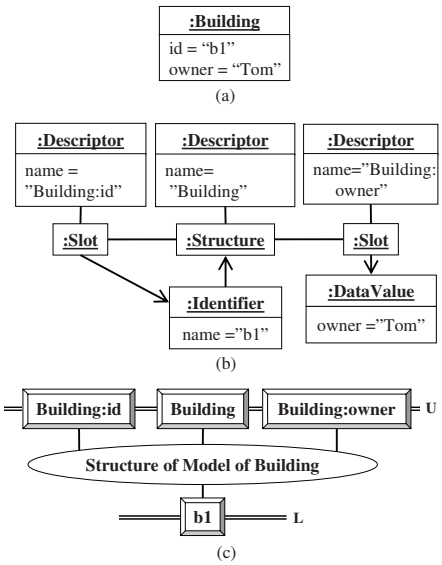

Figure 50: Example of how to represent an object in MATER

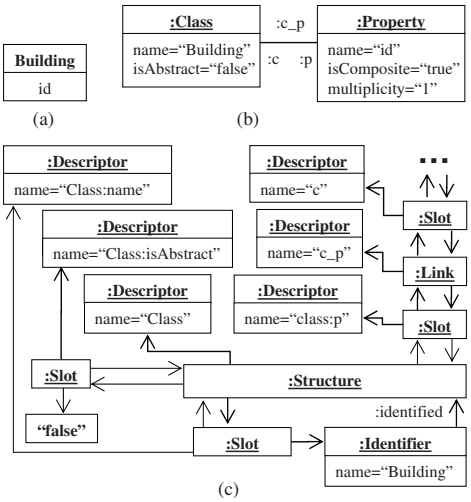

Figure 51: A part of the description of class `Building`

tiated from the UML metamodel (see [AK02] for more on this class/object nature); the UML metamodel can again be seen as composed of objects instantiated from MOF and MOF can be seen as composed of objects instantiated from itself.
A class `Building` in a UML class diagram is an instance of class `Class` of the UML metamodel, we understand that `Building` will be a class when we read about the semantics of class `Class` [OMG06b]: *A class is a type that has objects as its instances...The instances of a class are objects.* From this description we understand that a whole UML metamodel architecture can be depicted as an object diagram, which is known from the literature (e.g. [NPK04] and [GFB05]).

MATER offers several ways to model the same thing and it is not "strongly" constrained - this is deliberate and opens for experimentations. It is not discussed in the article but it will be possible to configure the framework with the help of some OCL-like language, e.g. enforce *strict metamodeling* [AK00b].

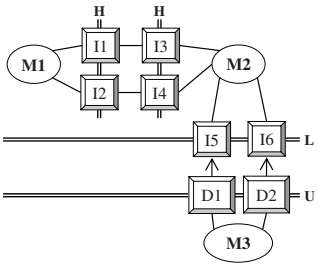

Figure 52: Sketch of connected models

G.2.2 Connecting Models

Fig. 53 extends MATER and adds the possibility to connect models. A Siwa model can contain borders; two models are connected by connecting two borders, one from each model. When the models to connect are on the same level both borders will be of type `HorizontalBorder` (Fig. 53(c)), when the models are on different levels the border on the lower level is of type `UpperBorder` (Fig. 53(b)) and the border of the upper model is of type `LowerBorder`.

Fig. 52 offers a sketch where a model called `M3` is connected to a model `M2` that resides on a level above. `M3` has an instance of `UpperBorder` (marked with letter U) containing two instances of `Descriptor` called `D1` and `D2`; these two symbols are connected to instances of `Identifier`, `I5` and `I6` respectively; `M2` has an instance of `LowerBorder` (marked with letter L) containing both `I5` and `I6`. Fig. 52 is also demonstrating how model `M1` and `M2` residing on same level are connected by two instances of `HorizontalBorder` (marked with letter H).

The vertical association in Fig. 53(b) has multiplicity `0..1` on the `Identifier` side, this means that incomplete architectures, as claimed in the introduction, are possible. The claim that a model can have several metamodels is justified by allowing several upper borders for one and the same model.

Fig. 54 shows in more detail an example (same example as in Fig. 46) of how MATER can be instantiated to connect models, (a) shows how `TMBuilding` and `Building` residing on the same level are connected, while (b) shows how `Building` and `:Building` residing on different levels are connected.

This way of connecting models is an extension to what we have presented before; [NPK04] presents a solution where two models would share a common

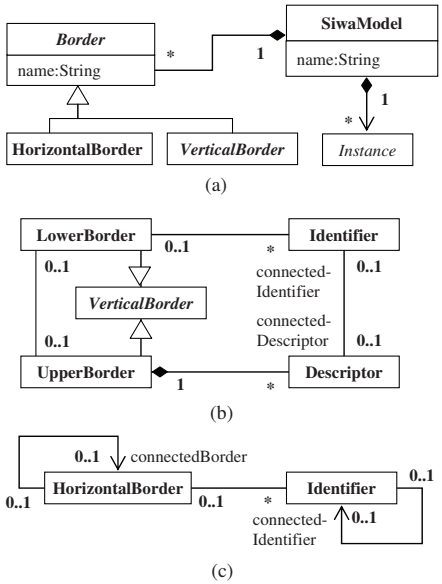

Figure 53: Part of MATER: connecting Siwa models

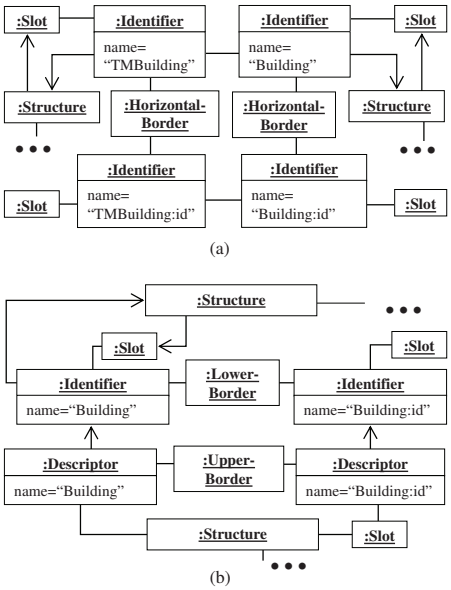

Figure 54: The example (incomplete) of Fig. 46 in MATER

border instead of having one border for each model to be connected; also the connecting of models on the same level is new.

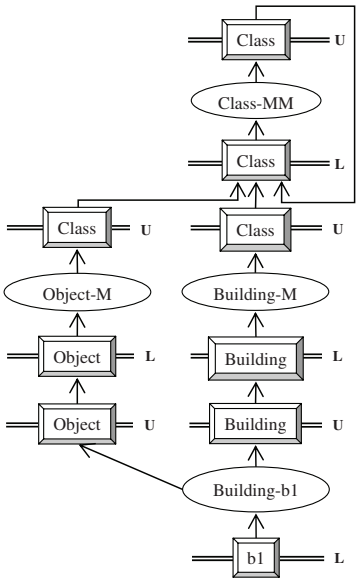

Figure 55: Architecture of building example (`TMBuilding` is not included)

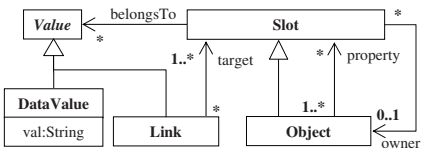

Figure 56: Model Object-M in detail

In Fig. 55 some of the models described above are put together to form a multilevel architecture. The only model that has not been mentioned before is the one named Object-M; as can be seen in Fig. 56 it simply defines an object as a structure having slots with values and links to other objects. The instance of `Structure` representing the building object has both `Building` and `Object` as descriptor.

Model Object-M is introduced to demonstrate that a model can have several type models. Object-M could be used to define a more general XML format than if Building-M was used.

G.3 Related Work

Today there is much interest in the use of metamodels, e.g. in MDA [OMG03a], MDE [Fav04b], LDD [Fow05], DSL [GSCK04].

The part of Siwa presented in this article, which is mainly the static part, can be used as a starting point in all the mentioned fields.

There are several other metamodeling frameworks, to mention a few: MetaEdit+ [Met06], Coral [AP04], XMF [CESW04], EMF [Ecl04] and MPS [MPS06].

Rondo [MRB03] is a programming platform for generic model management and it includes high-level operators used to manipulate models and mappings between models. AMW [FBJ+05] goes further and allows extensible mappings. AMW [FBJ+05] is a generic model weaver that allows the specification of correspondences between model elements from different models - models are in this way connected with a model, e.g. correspondence *Equals* might be established between the two `id` attributes of classes `TMBuilding` and `Building` of Fig. 46.

Our approach has similarities with the aforementioned works, but we have not found a framework that allows models and model levels two be connected so freely as our approach does, e.g. the weaving of models described above can be achieved simply by introducing another model with borders to the models to be weaved; this new model will describe the structure of the correspondences, semantic engines can then be defined to handle the semantics of the correspondences; a somewhat related example is given below.

G.4 Legacy Data Consistency As Example

Our article [NJ03] focused on the consistency problems that occur when previously uncoordinated, but semantically overlapping data sources are being integrated. The paper presented techniques for modeling consistency requirements using OCL and other UML modeling elements. The paper also considered the automatic checking of consistency in the context of one of the modeling techniques. This section presents an outline of how Siwa can

be applied to implement one of these techniques and how automatic testing of consistency can be performed.

G.4.1 Consistency Modeling And Testing

Fig. 57 shows an integration of two legacy models, where one is a description of apartments (class `Apartment`) and the other a description of buildings (class `Building`). The consistency requirements are as follows:

1. The number of apartments that is given as a property in class `Building` should be equal to the number of apartments with the same building id (attribute `bId`).

2. One building should have at least one apartment, and an apartment should belong to exactly one building.

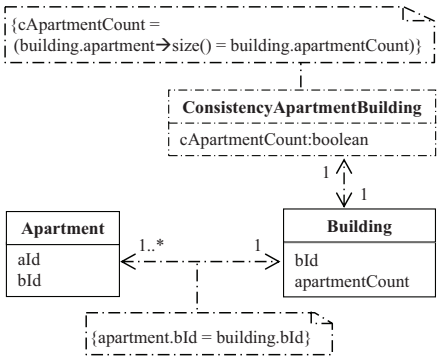

Figure 57: Consistency between `Apartment` and `Building`

The elements with dash-dotted line style in Fig. 57 constitute what we call a *consistency model*, this model is manually made by the user. When the consistency model is established, consistency testing of legacy data can be performed automatically. In our case there is one legacy data source with information about buildings and one about apartments. Consistency testing results in a report revealing which legacy data that do not fulfil the consistency requirements.
The association (Fig. 57) between `Apartment` and `Building`, including the attached invariant expressed in OCL, constitute consistency requirement

two. When testing is performed on the legacy data, a link is created between an `Apartment` and a `Building` instance if the invariant is fulfilled; if the multiplicity on the association is broken, this is reported in the consistency report.
Consistency requirement one is specified with help of class `ConsistencyApartmentBuilding`, property `cApartmentCount` and its attached invariant. During testing instances of type `ConsistencyApartmentBuilding` are created and linked to `Building` instances; slot `cApartmentCount` will be set to the value that fulfils the invariant; if the value is `false` then a consistency violation has occurred and will be reported. Note that links between `Building` and `Apartment` instances are traversed when the values of `cApartmentCount` slots are set. From this example we can understand that standard OCL-statements are used as production rules when the consistency model is being automatically instantiated.

G.4.2 Implementation In Siwa

In Siwa the consistency model can be seen as an instance of the declarative domain specific language described by the metamodel given in Fig. 58.

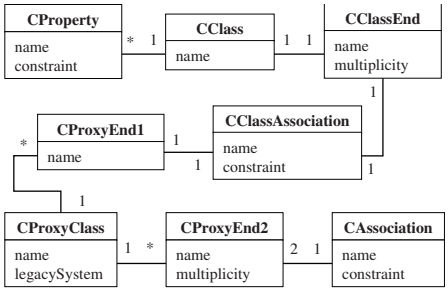

Figure 58: Consistency modeling metamodel

In brief: The legacy classes `Building` and `Apartment` have *proxy classes* to represent them in the consistency model; a proxy class is an instance of `CProxyClass`. Class `ConsistencyApartmentBuilding` (Fig. 58) is an instance of `CClass`; its property `cApartmentCount` is an instance of `CProperty`, where the value of slot `constraint` is the text:

cApartmentCount = (building.apartment→size() = building.apartmentCount)

The association between `ConsistencyApartmentBuilding` and `Building` is

represented as an instance of `CClassAssociation` going between the building proxy class and `ConsistencyApartmentBuilding`.
The association between `Building` and `Apartment` is represented as an instance of `CAssociation` going between the two proxy classes; the value of slot `constraint` for this instance is the text:

apartment.bId = building.bId

Fig. 59 gives an overview of the complete architecture. The lowest level can be seen as one contiguous model composed of legacy data and a consistency model instance. The legacy models are on the other hand not changed - the borders towards the consistency model can be extracted automatically. From a model management point of view this is considered an advantage since it gives few models to manage (remember that the lowest level can be produced automatically at will).

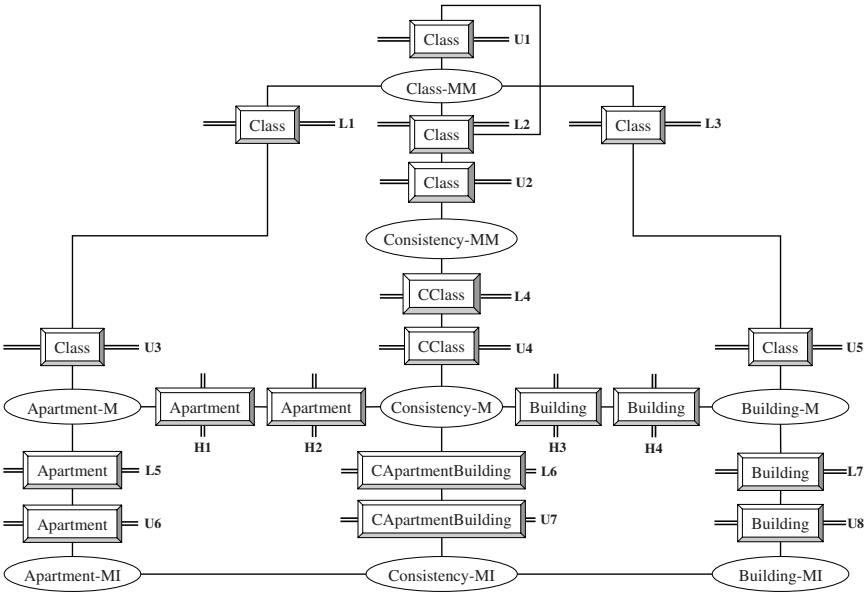

Figure 59: Architecture example: consistency modelling and testing

The format of this article does not give room for presenting a complete picture of how Siwa solves the problem at hand, but below is some information about how the consistency model instance is automatically created by

a semantic engine.
A semantic engine is a Siwa model that exhibits behavior. A special type of semantic engines can be attached to borders, they are called *border engines* and are typically involved in instantiation. For simplicity we can assumed that such engines are programmed in Java since this is our implementation language. The *metamodeler* has made a border engine and attached it to border `L4` (Fig. 59). Border `U4` and `L6` is created by the engine when the modeler decides to make a consistency model. The engine is also attaching a premade border engine at `L6`, it is this engine that automatically produces the consistency model instance and the consistency report when triggered. This last engine is an adapted implementation of the algorithm presented in [NJ03].

This way of establishing the semantics can be seen as a specific way of implementing *deep characterization*; some see deep characterization as a natural part of metamodeling [Küh05].

G.5 Summary And Research Directions

The MATER model presented in this article represent our understanding of what we mean by a multilevel model architecture; we have tried to make an explicit representation of all elements that constitutes such an architecture. Complex and advanced concepts can then be built in a natural way by combining these defined building blocks.

MATER can be seen as a metamodel by itself, but we choose to see it as the physical carrier for multilevel modeling architectures. This view allow us to specify top models as we see needed, e.g. a top model that gives properties separate existence.

The following is a list of features that in our view makes Siwa a promising and unique framework:

- It is not strongly coupled to the instantiation found in its implementation language, this allows a model to have several type models each offering different and useful information about the model.
- It is extremely generic which makes it adaptable to many different modeling needs, e.g. it might allow separate existence of properties.
- It is possible to have incomplete architectures, e.g. XML documents might be loaded for analysis, and then a model might be produced au-

tomatically [GNP+06]. This is typically not possible in other frameworks due to their strong coupling to instantiation in the selected implementation language.

Parts of our framework are already implemented in the Eclipse framework [DFK+03]. The first prototype is implemented by defining the MATER model as a UML model in Eclipse and from this we create an EMF Model; this looks like a trick since we end up with having all the Siwa model levels at one EMF level [PNCW06], but it gives us a jump-start and it automatically produces a lot of useful code.

Appendix H

The Object-oriented Paradigm and Some Basic Philosophy

This appendix gives an informal introduction to the object-oriented way of thinking and the basis of modeling.

Often the object-oriented development paradigm is presented with references to philosophers like Aristotle and Plato – this is no coincidence, philosophy is a potent source of knowledge when doing computer science, e.g., language as a topic by itself is essential to both philosophy and computer science. Also, defining ontologies – which is an old philosophical discipline – can be seen as modeling. The close relation between philosophy and modeling (metamodeling) is exploited in temporary works, e.g., Alfons Laarman and Ivan Kurtev states the following [LK09]:

> The philosophical theory presented here gives us a set of concepts to build a new metalanguage...
> In this paper we use an ontology called *Four-category ontology* (FCO) that can be traced back to Aristotle and is also used in several contemporary works on Formal Ontology...
> The philosophical theory presented in this section gives us a stable and well-founded set of concepts to start with building our metalanguage.

H.1 Seer, Seeing, and the Seen

In our normal state of mind we perceive the world as composed of objects with properties. The state of an object is given by the values of its properties; one property may be that an object can be linked in some specific way to another object (e.g., a flying object placed in the gravitation field of earth). Some properties seems to be more intrinsic than relative, like you being the object and the level of your blood pressure[16] bing the property. An object may interact with other objects exhibiting behavior which is governed by its state. All types of objects conform to this general description – and it defines the object-oriented world view.

What does it mean that we perceive the world as composed of objects – does it mean that the world is composed of objects or is this "only" our perception? The object-oriented view is dualistic in the sense that you have *subject* and *object*. It is not meaningful to talk about an object without

[16] Blood pressure of course depends on the outside pressure; there is a lot to the statement "everything is relative", and some philosopher even call the physical reality for *the relative*.

a subject; it is the subject that experiences "something"[17] as an object; something becomes an object if it appears before our *inner eye* as something separate from its surroundings. Reality has the potential of being perceived as composed of objects, and our perception "works" since it allows us to "navigate in reality".

Some objects are more equal than others, which makes it possible to classify objects by their differences and similarities. The notion of a *class* is built on this understanding; discrimination is done by some established demands on properties of the objects in question[18].

A *concept* is often considered to be an abstract idea generalized from particular instances; the instances may be physical entities or concepts. One may argue that a concept is described with (symbolic) references to other concepts and because of this we have a non-useful self referential situation, however, our physical experiences can be referenced and descriptions of "abstract things" can in this way be grounded in the physical. The term concept is not always indicating a set of instances, e.g., the mental representation of Lassie (seen as an individual dog) is also considered to be describing a concept and in this case there is a one-to-one relation between the particular instance and the concept. The Four-category ontology[19] is used in some later works as a basis for modeling [Gui05, DHHS01]; this ontology have two main entities: *Individuals* and *universals*. Individuals can further be divided into *substantials* and *moments*; a substantial is typically represented as an object in UML. Moments are dependent on other individuals for existing and they correspond to slots and links. If a complete ontology is to be made, then change or time would typically be included; an object could then persist while its *inherent* moments may come and go. We understand from this, since Lassie (the dog) appears as a continuously changing entity, that even the concept of Lassie must have a certain "flexibility" so that it can continuously be connected to the dog Lassie.

The meaning of a new concept can be defined in terms of concepts which already have a well-defined meaning, e.g., as a specialization of an already known concept by describing additional properties (e.g., some additional behavior); this may be how a *subclass* was established; classes can also be overlapping; in this way complex hierarchies (taxonomies) can be defined.

[17] Below I have not always quoted the terms *something* and *thing* even if used in "an imprecise manner".

[18] This is not meant to be a political manifesto despite the terminology – but please note that discrimination is based on choice and selecting an object is like giving it a property: One of the chosen ones.

[19] (This ontology can be tracked back to Aristotle.)

Typically, when "looking" at an object, we find that it is *composed of parts*; focusing on a part makes that part an object, e.g., the head as part of a human body. So an object can be seen as one object even if it is composed of several parts that again can be seen as objects. The parts of a composite (a whole) are related, and this makes a whole more then the sum of its separate parts, e.g., the parts of a chair are not arbitrarily put together.

The reality is complex, "things" in reality are extremely entangled with each other; senses and instruments are limited; consequently our perception of the world is (extremely) limited.

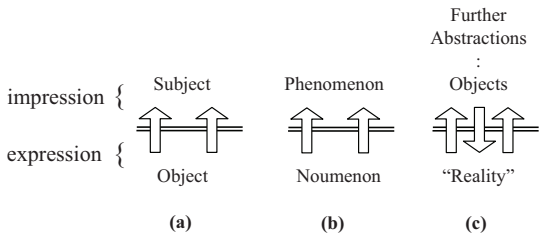

Figure 60: Subject and object.

Is objectivity possible? The definition of the word *objective* may involve phrases like "observer independent" and "existing independently of mind". Fig. 60(a) shows an observation (done by some instrument/sense organ) where the subject (observer) does not effect the observed object. According to Immanuel Kant we can not know *the thing-in-itself* (noumenon), the subject can only know the *phenomenon* which can be understood as the "physical impact" the noumenon has on the subject (Fig. 60(b) shows the terminology used by Kant). Fig. 60(c) shows a more true story than Fig. 60(a): An observer always causes a perturbation of some kind onto the object being observed. The subject must be in some sort of interaction with reality to receive information; the influence of the observer generates a sort of response from the thing being observed – the response generates the impression experienced by the observer. One should also be aware that all subjects (e.g., human minds) comes with an individual context and that the objects "constructed" will be "colored" by this context! Even if an observer independent observation seems impossible, the following statement is considered true: There is "something" in the outer world that in a "homomorphic (isomorphic) way" correspond to the objects that appear before our inner eye.

H.2 Form and content

A variation of the whole-part relation can often be found by examining an object at a "finer granularity" exposing smaller parts, e.g., the head is composed of cells, a cell is composed of molecules and so on. Another example would be text written on paper; first we realize that the text makes some statement about something and this is typically not about the ink on the paper; in this case ink on paper functions as *medium*; the ink/paper is again made up of molecules which gives a more detailed description of the medium. The language used making the text can also be understood as a medium used to carry the description given by the text[20]. All media used must be rooted in physical reality and all descriptions need a medium, even *the stuff that dreams are made of*[21] is physical in some way.

A description is about something physical (concrete) or about something you imagine, something abstract – this is called the *content of the description*. The physical reality by itself is not a description of something else[22] – "it describes itself". Every medium in use (included languages), is rooted in the physical reality, and consequently it communicates a physical state – this state is called the *form of the description*. The form can often be viewed at different levels that all contributes to the form appearing, e.g., a running computer program can be understood at a level which involves electronics and it may also be seen at a level involving byte code.

Content and form are intermingled and how they relate is still discussed by philosophers, e.g., a specific medium is giving some quality to the description that other media can not give.

Content implies a language, and by "stretching" the notion of being a language, one may say that the physical reality consists of statements made in an "immanent language" – this is the language that physics as a science is trying to capture and externalize. The term "implemented language" may be used instead of immanent language, but it is problematic to use this term since we do not know if there is a specification behind the "implementation".

This view gives the understanding that all we relate to are statements in languages which we in some way understand partially or fully – and, that all things existing are statements in some language[23]. One may say that

[20]The understanding, i.e., seeing a language as a medium, is used to make the ideas presented more understandable; it is based on the believe that the term medium appears to be less abstract than the term language and more easy to understand.

[21]From Hamlet by W. Shakespeare

[22]Ignoring any metaphysical objections.

[23]Bypassing any metaphysical objections like the claim that there is an *absolute formless*

this view constitutes a *language-oriented world view*.

The language-oriented world view is at a more general level than the programming paradigms (e.g., the object-oriented paradigm, the event-oriented paradigm, the rule-oriented paradigm, the function-oriented paradigm, etc.) which all are based on languages.

Returning to the example above, i.e., text written in ink on paper, the ink/paper medium may be seen as a *representational language* (*embedding language*) for the text; the representational language is used in a specific way to code sentences in the language represented, and consequently interpretation is needed to get the sentences (which in this case is the meaning). To get the *semantics* of the sentences interpretation is again needed.

To get the semantics of sentences in a language an interpreter doing an interpretation is always needed. A language is typically built on top of another language forming stacks of languages and it is not possible to make a language without already having a language[24].

H.3 Is Concept the Same as Class?

In common use "abstract" is a relative term – something is more or less abstract in relation to something else – the most concrete are then the things that are seen as physically existing. Abstraction is the mental process of abstracting out things that are considered to be more important than the ones ignored. Abstraction often involves generalization; this gives abstractions that describes several instances.

The meaning triangle, shown in Fig.61(a), was discussed by Aristotle and it describes conceptualization; as an example: The *referent* may be a concrete person with name Peter, the name Peter is a *symbol* (also called *represent*) that stands for this person, the concept or idea of this person is found at the top of the triangle. Another example of concept would be: The "person concept" (all words represent concepts); the word `Person` could be the symbol and "real" persons would be referents.

It seems that John F. Sowa [Sow00] consider a concept to be neural excitations in the brain that relate the symbol to its object; I can not fully agree on this – a concept should be considered something abstract and not something concrete as brain excitations; I consider the brain excitations to

reality (non-relative).

[24] In the process of boot strapping a computer language, another language is at least initially needed.

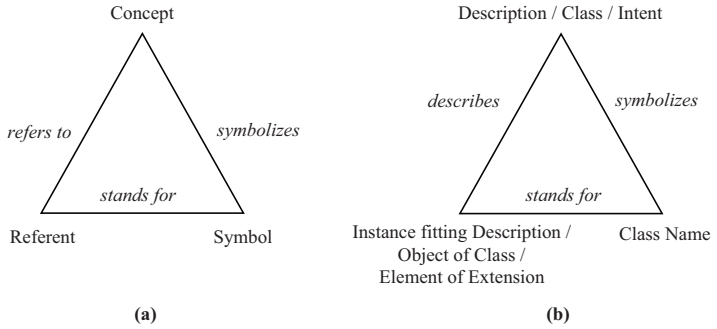

Figure 61: The meaning triangle (a) and a variant of it (b)

be descriptions/processing that give the "right linking of symbol to referent".

All concept are by themselves abstract, it is not possible to "point to" a concept and say "there it is", but it may be possible to point to the instance(s) of a concept and also to a description of a concept.

The term description is here seen as something physical, e.g., a (concrete) paper map describing a terrain or a (concrete) painting of goddess Saraswati.

Concepts are the "ingredients" in intellectual human communication; the following is meant as a schematic description: A symbol (e.g., "chair") is used as a starting point, by exchanging descriptions of what the symbol means (e.g., a chair is something to sit on) a common meaning is established or a disagreement on what is meant by the symbol is exposed; in this process of communication one assumes that there is a *truth which is language (form) independent*[25] and it is this truth (right understanding) that we call a concept.

Often we are able to use some "sort of logic" and see that two descriptions are describing the same concept, in other cases we may agree that one description is giving more information about one aspect of the concept than another description. In mathematics the concepts are typically precisely described but in other domains this is not the case, e.g., the concept of a chair may have conflicting descriptions – some may claim that it must be manmade and others not.

Nominalism is defined as [Her01] :

[25]This is not meant as a complete description of the issue – after all this has been discussed in philosophy for centuries !

> ...the doctrine holding that abstract concepts, general terms, or universals have no independent existence but exist only as names.

The discussion so far should have made it clear that concepts do not have their own separate existence, one may stretch the term exist and say a concept exists in the form of descriptions. Another view, advocated by Aristotle, is to see a concept (class) as existing via its instances and not independently of them. "Separating" reality into objects (without more classification) is demanding a (mental) process, separating the objects into categories adds further complexity to the process of selection/discrimination.

Both the description of a physical phenomenon and of a concept may be stored on a computer – in this respect there is no difference.

Some physical phenomenon like "a game character" may only exist on a specific computer, the description and its existence may in this case be so tightly connected that removing the description removes the phenomenon altogether; however, from the "computer point of view" there are no difference.

Claiming that concepts are mere names (like just "an empty sound") is "taking it to far". Visioning a world which includes time but no reoccurring patterns – that is hard or impossible since "no thing" would appear! Our world is not a world of total chaos, we manage by using our creative minds to abstract and reason. "Concepts are" potential ways of understanding the world, by agreeing on some logic and context we have ways that will lead us to common concepts. The mentioned logic is not enough – there is also a human element of choice involved, a creative factor, consequently the "conceptual worlds" are many (this is particularly evident when it comes to politics, religion, etc.).

Fig. 61(b) shows a modified triangle, here concept has been replaced by a description of referent – in a sense this triangle is more directly applicable when it comes to computers since only descriptions can be stored and manipulated. (Fig.61(b) correspond to the viewpoint of John F. Sowa [Sow00] where the description is found in a humans brain.)

Is concept (ignoring the "one-to-one type of concepts") the same as class? James Rumbaugh et al define a class as [RJB05]:

> The *descriptor* for a set of objects that share the same attributes, operations, methods, relationships, and behavior. A class represents a concept within the system being modeled.

A descriptor is defined as:

> A model element that describes the common properties of a set of instances, including their structure, relationships, behavior, constraints, purpose, and so on.

From this we understand that class is not the same as concept – a class is a description of what is the intent or the meaning of the concept. In this view a class and a copy of the class are not the same class, but they describe the same concept in a similar way.

In addition to a description of the referents, different types of information may be attached to a class:

- Descriptions related to the class itself, like who made the class.
- Descriptions about the concept itself, e.g., people can be classified as *melancholic* where melancholic by itself is an instance of *personality type* (notice that a person that is a melancholic is not an instance of personality type).

Additionally there is always a linguistic aspect to a description since it is done in a language.

Where does *type* fit in this picture? One view is to see a type as a description (specification) without a *physical implementation* [RJB05]; physical implementation can in this context be interpreted as having the description in a programming language; in this view a type is a "bit more abstract" than class.

Philosophy contains many different and competing views on reality; *contemporary philosophical realism* (metaphysical realism) advocates the view that there is a reality that is completely independent of our mental activities. Another view is *constructivist epistemology* which advocates a view where all our knowledge is considered to be "constructed", and that it does not necessarily reflect any external reality.

Today *phenomenology* has a strong position, according to Husserl phenomenology is (collected from [Smi07]): "...the reflective study of the essence of consciousness as experienced from the first-person point of view". Husserl is considered to be the founder of contemporary phenomenology. Central to phenomenology is the understanding that consciousness is directed towards some object, in phenomenology this "aboutness" ("directedness") is called *intentionality*. In *classical Husserlian phenomenology*, there is a distinction

between the represent (the mental object) and the thing it presents (referent) – as it is for Kant. The human experience of a thing goes via particular concepts, images, etc. – this "direction towards the thing" makes up the content or the meaning of the experience. An example that may explain this view is given by Friedrich Ludwig Gottlob Frege [26]: "The morning star" and "the evening star" both refer to the same object (Venus) but express different content (or meaning; Frege used the term *sense*; Husserl used the term *noematic sense*) since the same object is presented in different ways.

Phenomenology does not rule out objectivity even if experiences are done by subjects, e.g., in a mathematical system, which includes axioms and deductive rules, two different persons can derive the same trustful results.

In performing phenomenology, Husserl proposed "bracketing" the question of the existence of the natural world – this fits the view presented above: A modeling framework is about descriptions; what is being described is only "important to the framework (tool)" if there are references in the description to other descriptions found in the framework, e.g., an object and a reference to its class.

For man computer can be seen as an extension to the brain; computer and brain have different strengths but they are both used to memorize and manipulate descriptions. There are many theories when it comes to *philosophy of mind*, e.g., weak *materialism* proclaiming that each type of mental state corresponds to or is identical with a type of brain state. Another theory being *functionalism*, focusing on what brains do [Smi05]:

> ...Instead, mind is what brains do: Their function of mediating between information coming into the organism and behavior proceeding from the organism. Thus, a mental state is a functional state of the brain or of the human (or animal) organism. More specifically, on a favorite variation of functionalism, the mind is a computing system: Mind is to brain as software is to hardware; thoughts are just programs running on the brain's "wetware".

There are several other theories, but it seems hard to come around that at some level, neural activities and the states of the brain relate to the descriptions that occur as the contents of consciousness.

However, as a modeling framework developer not all philosophic issues have to be settled:

[26] Frege precedes modern phenomenology.

A modeling framework is necessarily about phenomena or appearances – discussing if there is something like thing-in-itself and realism contra a more constructivist view, is not needed when seeing the content of a modeling framework as descriptions. Whether a description is describing something existing or not, that question is left to the modeler!

Most object-oriented programming languages offer a class construct that includes functionality that allows objects conforming to the class to be created (instantiated) – this functionality is in some way coded into the construct. This work takes another approach: It sees the functionality of instantiation as what makes a description into a class. Also, object creation can be done in different ways, allowing the explicit attachment of (varying) instantiation functionality gives flexibility and a more "open solution". Another, essential functionality is the one that allows objects to be classified – this functionality may also be explicitly attached to the class (description).

References

[AK00a] Colin Atkinson and Thomas Kühne. Meta-level independent modeling. In *International Workshop Model Engineering (in Conjunction with ECOOP'2000)*. Cannes, France, June 2000.

[AK00b] Colin Atkinson and Thomas Kühne. Strict Profiles: Why and How. In *<<UML>>2000 - The Unified Modeling Language, Advancing the Standard*, volume 1939 of *Lecture Notes in Computer Science*. Published by Springer, 2000.

[AK01] Colin Atkinson and Thomas Kühne. The Essence of Multilevel Metamodeling. In *UML 2001 - The Unified Modeling Language: Modeling Languages and Applications*, volume 2185 of *Lecture Notes in Computer Science*, pages 19–33. Published by Springer, 2001.

[AK02] Colin Atkinson and Thomas Kühne. Rearchitecting the UML infrastructure. *ACM Transactions on Computer Systems (TOCS)*, 12(4):290–321, October 2002.

[AK03] Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation, September 2003. IEEE Software.

[AK05] Colin Atkinson and Thomas Kühne. Concepts for Comparing Modeling Tool Architectures. In *MoDELS*, pages 398–413, 2005.

[AP04] Marcus Alanen and Ivan Porres. The Coral Modelling Framework. In *Proc. of the 2nd Nordic Workshop on the Unified Modeling Language NWUML'2004*. Turku Centre for Computer Science, Finland, 2004.

[ASS96] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, second edition, 1996.

[Atk97] Colin Atkinson. Meta-modeling for Distributed Object Environments. In *Enterprise Distributed Object Computing*, pages 90–101. Published by IEEE Computer Society, 1997.

[Bar99] Roman Barták. Constraint programming: In pursuit of the holy grail. In *Proceedings of Week of Doctoral Students (WDS99)*, pages 555–564, 1999.

[Bar07] Roman Barták. On-line guide to constraint programming. 2007. http://kti.mff.cuni.cz/~bartak/constraints/, Accessed: July 26, 2010.

[BC87] Jean-Pierre Briot and Pierre Cointe. A Uniform Model for Object-Oriented Languages Using the Class Abstraction. In *IJCAI*, pages 40–43, 1987.

[BEN] `http://bentoweb.org/`.

[Ber97] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2 edition, 1997.

[Béz04] Jean Bézivin. In search of a basic principle for model driven engineering. *CEPIS, UPGRADE, The European Journal for the Informatics Professional*, V:21–24, 2004.

[Béz05] Jean Bézivin. On the unification power of models. *Software and System Modeling (SoSym)*, 4(2):171–188, 2005.

[BG] D. Brickley and R.V. Guha. Resource Description Framework Schema Specication 1.0. Technical report, W3C Consortium. http://www.w3.org/TR/2000/CR-rdf-schema-20000327.

[BG01] Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *ASE*, pages 273–280, 2001.

[BG03] D. Brickley and R.V. Guha. OWL Web Ontology Language: Overview. Technical report, W3C Consortium, W3C Working Draft: http://www.w3.org/TR/owl-features/, March 2003.

[BHJ+05] Jean Bézivin, Guillaume Hillairet, Frédéric Jouault, Ivan Kurtev, and William Piers. Bridging the MS/DSL Tools and the Eclipse Modeling Framework. In *Proceedings of the International Workshop on Software Factories at OOPSLA*, 2005.

[BJMF02] Marko Boger, Mario Jeckle, Stefan Mller, and Jens Fransson. Diagram Interchange for UML. In *<<UML>>2002*, volume 2460 of *Lecture Notes in Computer Science*, pages 398–367. Published by Springer, October 2002.

[BJV04] J. Bézivin, F. Jouault, and P. Valduriez. On the Need for Megamodels. In *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[BKK+01] Kenneth Baclawski, Mieczyslaw K. Kokar, Paul A. Kogut, Lewis Hart, Jeffrey Smith, William S. Holmes III, Jerzy Letkowski, and Michael L. Aronson. UML for Ontology Development. *Knowledge Engineering Review*, 2001.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.

[BSM+04] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. Addison-Wesley, 2004.

[Cap94] Nigel P. Capper. The Impact of Object-oriented Technology on Software Quality: Three Case Histories - Technical. *IBM Systems Journal*, March 1994.

[CBC05] Dan Chiorean, Maria Bortes, and Dyan Corutiu. Semantic Validation of XML Data A Metamodelling Approach. *NWUML 2005*, August 2005.

[CCR08] Jordi Cabot, Robert Claris, and Daniel Riera. Verification of UML/OCL Class Diagrams using Constraint Programming. *IEEE International Conference on Software Testing Verification and Validation Workshop*, 0:73–80, 2008.

[CE00] K. Czarnecki and U. W. Eisenecker. *Generative Programming. Methods, Tools, and Applications*, chapter 11: Intentional Programming. Addison Wesley, 2000.

[CEK00] Tony Clark, Andy Evans, and Stuart Kent. A Feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-modeling Approach. Technical Report version 1.0, pUML, http://www.puml.org/, 2000.

[CESW04] Tony Clark, Andy Evans, Paul Sammut, and James Williams. *Applied Metamodelling. A Foundation for Language Driven Development.* Xactium, 2004. Available at `http://www.xactium.com`.

[Cho57] Noam Chomsky. *Syntactic structures.* Mouton, 1957.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems (TOCS),*, 3(1):63–75, February 1985.

[Coa92] Peter Coad. Object-oriented patterns. *Commun. ACM*, 35(9):152–159, 1992.

[Coi87] Pierre Cointe. Metaclasses are First Class: The ObjVlisp Model. In *OOPSLA '87: Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 156–162, New York, NY, USA, 1987. ACM Press.

[Com] Computer Science Research Laboratory of Babes-Bolyai University of Cluj-Napoca Romania. http://lci.cs.ubbcluj.ro/ocle/, Accessed: July 26, 2010.

[Com94] CDIF Technical Commitee. *CDIF Framework for modeling and extensibility, Electronic Industries Association, EIA/IS-107.* September 1994.

[CW93] Stefano Ceri and Jennifer Widom. Managing Semantic Heterogeneity with Production Rules and Persistent Queries. In *19th International Conference on Very Large Data Bases*, pages 108–119, Dublin, Ireland, 1993.

[CWD00] Miro Casanova, Thomas Wallet, and Maja D'Hondt. Ensuring Quality of Geographic Data with UML and OCL. volume 1939, pages 225–239, 2000.

[DCG+89] Peter J. Denning, Douglas Comer, David Gries, Michael C. Mulder, Allen B. Tucker, A. Joe Turner, and Paul R. Young. Computing as a Discipline. *Commun. ACM*, 32(1):9–23, 1989.

[Den99] Peter J. Denning. Computer science: The discipline, 1999.

[DFK+03] Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellermann, and Pat McCarthy. *The Java Developer's Guide to Eclipse*. Addison-Wesley, 2003.

[DHHS01] W. Degen, B. Heller, H. Herre, and B. Smith. GOL: Toward an Axiomatized Upper-level Ontology, In Proceedings of the International Conference on Formal Ontology in Information Systems. Volume 2001, FOIS '01. ACM, New York, NY, 34-46, October 2001.

[Dmi04] Sergey Dmitriev. Language oriented programming: The next programming paradigm. *onBoard*, (1), November 2004. http://www.onboard.jetbrains.com/is1/-articles/04/10/lop/, Accessed: July 26, 2010.

[DMN70] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. SIMULA 67 Common Base Language. Technical report, Norwegian Computing Center, Oslo, Norway, October 1970.

[Dri01] Tobin A. Driscoll. *Object Technology International, Inc., Eclipse platform A universal tool platform*. 2001. http://eclipse.org.

[EAM] `http://www.support-eam.org/`.

[EAR] `http://www.w3.org/TR/EARL10/`.

[Ecl04] Eclipse Project. *EMF, Eclipse Modeling Framework* . http://eclipse.org/emf/, Accessed 2004.

[Ecl07] Eclipse Project. *UML2, EMF-based UML 2.1 Metamodel Implementation*. http://eclipse.org/uml2/, Accessed 2007.

[EGG+01] Robert Eschbach, Uwe Glässer, Reinhard Gotzhein, Martin von Löwis, and Andreas Prinz. Formal Definition of SDL-2000: Compiling and Running SDL Specifications as ASM Models. In *Abstract State Machines 2001: New Developments and Applications*. J.UCS Special issue vol. 7, no. 11, 2001.

[EIA] http://www.eiao.net.

[Fav04a] Jean-Marie Favre. Foundations of Meta-Pyramids: Languages Vs Metamodels, 2004. http://megaplanet.org/-jean-marie-favre/papers/FoundationsOfMeta-PyramidsLanguagesVsMetamodels-EpisodeIIStory-OfThotusTheBaboon.pdf, Accessed: July 26, 2010.

[Fav04b] Jean-Marie Favre. Foundations of model (driven) (reverse) engineering, 2004. http://megaplanet.org/jean-marie-favre/papers/FoundationsOfModelDrivenReverse-EngineeringModelsEpisodeIStoriesOfTheFidusPapyrusAnd-OfTheSolarus.pdf, Accessed: July 26, 2010.

[Fav04c] Jean-Marie Favre. Towards a Basic Theory to Model Model Driven Engineering. In *3rd Workshop in Software Model Engineering (WiSME 2004)*, 2004. http://megaplanet.org/jean-marie-favre/papers/-TowardsABasicTheoryToModelModelDrivenEngineering.pdf, Accessed: July 26, 2010.

[Fav05] Jean-Marie Favre. Megamodelling and Etymology. In *Dagstuhl Seminar 05161 on Transformation Techniques in Software Engineering*, 2005. http://megaplanet.org/jean-marie-favre/papers/-MegamodellingAndEtymologyAStoryOfWordsFromMED-ToMDEViaMODELInFiveMilleniums.pdf, Accessed: July 26, 2010.

[FBJ+05] Marcos Didonet Del Fabro, Jean Bézivin, Frédéric Jouault, Erwan Breton, and Guillaume Gueltas. AMW: a generic model weaver. In *Proceedings of the 1re Journe sur l'Ingnierie Dirige par les Modles (IDM05)*, 2005. http://atlanmod.emn.fr/www/papers/IDM_2005_weaver.pdf, Accessed: July 26, 2010.

[FBJV05] Marcos Didonet Del Fabro, Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. Applying Generic Model Management to Data Mapping. In *Proceedings of the Journes Bases de Donnes Avances (BDA05)*, 2005. http://atlanmod.emn.fr/www/papers/CR_2005_BDA-_weaving[V1.0].pdf, Accessed: July 26, 2010.

[FCJ03] Anders Friis-Christensen and Christian S. Jensen. Object-Relational Management of Multiply Represented Geographic Entitites. In *SSDBM 2003*, July 2003.

[FCJNS05] Anders Friis-Christensen, Christian S. Jensen, Jan P. Nytun, and David Skogan. A conceptual schema language for managing multiply represented geographic entities. *Transactions in GIS*, 9(3):345–380, June 2005.

[FHL+98] E.D. Falkenberg, W. Hesse, P. Lindgreen, B.E. Nilsson, J.E. Han Oei, C. Rolland, R.K. Stamper, F.J.M. van Assche, A.A. Verrijn-Stuart, and K. Voss. A framework of information system concepts. the frisco report. Technical report, 1998.

[FK02] Martin Fowler and Scott Kendall. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Published by Addison-Wesley, 2st edition, 2002.

[Fow05] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? Technical report, 2005. http://www.martinfowler.com/articles/language- Workbench.html, Accessed: July 26, 2010.

[Fra03] David Frankel. *Model Driven Architecture*. OMG Press, 2003.

[Fra05a] Karl Frank. A Proposal for an MDA Foundation Model. An ORMSC White Paper V00-02 ormsc/05-04-01, Object Management Group (OMG), 2005. http://www.omg.org/docs/ormsc/05-04-01.pdf, Accessed: August, 2005.

[Fra05b] David Frankel. MDA Journal: Eclipse and MDA. March 2005. Available at: http://www.bptrends.com/.

[Fre87] Frederick P. Brooks Jr. No silver bullet - essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[FV09] Marcos Didonet Del Fabro and Patrick Valduriez. Towards the Efficient Development of Model Transformations Using Model Weaving and Matching Transformations. *Software and System Modeling*, 8(3):305–324, 2009.

[GFB05] Martin Gogolla, Jean-Marie Favre, and Fabian Büttner. On Squeezing M0, M1, M2, and M3 into a Single Object Diagram. In *Proceedings of the MoDELS'05 Conference Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends, Montego Bay, Jamaica, October 4, 2005*, Technical Report LGL-REPORT-2005-001, pages 1–14. EPFL, 2005.

[GGL05] Lars Grunske, Leif Geiger, and Michael Lawley. A graphical specification of model transformations with triple graph grammars. In *In First European Conference Model Driven Architecture - Foundations and Applications, number 3748 in Lecture Notes in Computer Science*, pages 284–298. Springer, 2005.

[GH05] Ralf Gitzel and Tobias Hildenbrand. A taxonomy of metamodel hierarchies. Research report, Department of Information Systems, Universitt Mannheim, 2005.

[GKP98] R. Geisler, M. Klar, and C. Pons. Dimensions and Dichotomy in Metamodeling. In *Proceedings of the Third BCS-FACS Northern Formal Methods Workshop*, September 1998.

[GKR99] S. Gaito, S. Kent, and N. Ross. A Meta-model Semantics for Structural Constraints in UML. *Behavioural Specifications for Businesses and Systems*, pages 123–141, September 1999.

[GMF] *Eclipse Graphical Modeling Framework*. http://www.eclipse.org/gmf, Accessed: July 26, 2010.

[GNP+06] Terje Gjøsæter, Jan Pettersen Nytun, Andreas Prinz, Mikael Snaprud, and Merete Skjelten Tveit. Modelling Accessibility Constraints. In *ICCHP*, volume 4061 of *Lecture Notes in Computer Science*, pages 40–47. Springer, 2006.

[GNPT05] Terje Gjøsæter, Jan P. Nytun, Andreas Prinz, and Merete S. Tveit. Accessibility Testing XHTML Documents Using UML. In *Proc. of the Nordic UML Workshop*. University of Tampere, Finland, 2005.

[GOS07] Ralf Gitzel, Ingo Ott, and Martin Schader. Ontological Extension to the MOF Metamodel as a Basis for Code Generation. *Comput. J.*, 50(1):93–115, 2007.

[GPHS06] Cesar Gonzalez-Perez and Brian Henderson-Sellers. A powertype-based metamodelling framework. *Software and System Modeling*, 5(1):72–90, 2006.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80. The Language and its Implementation*. Addison-Wesley, Reading, 1983.

[GR02] Martin Gogolla and Mark Richters. Development of UML Descriptions with USE. In *1st Eurasian Conf. Information and Communication Technology (EURASIA'2002)*, volume 2510 of *Lecture Notes in Computer Science*, pages 228–238. Published by Springer, 2002.

[Gri03] Catherine Griffin. Using EMF. Technical report, 2003. http://www.eclipse.org/articles/-Article-Using EMF/using-emf.html, Accessed: July 26, 2010.

[GSCK04] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, 2004.

[Gui05] G. Guizzardi. Ontological Foundations for Structural Conceptual Models, Telematica Instituut, Enschede. 2005.

[Har93] Juris Hartmanis. Some observations about the nature of computer science. In *FSTTCS*, pages 1–12, 1993.

[HDF00] H. Hussmann, B. Demuth, and F. Finger. Modular Architecture for a Toolset Supporting OCL. In *<<UML>>2000*, volume 1939 of *Lecture Notes in Computer Science*, pages 278–293. Published by Springer, October 2000.

[Her01] *The American Heritage Dictionary: Fourth Edition*. Houghton Mifflin Company, 2001.

[Hes06] Wolfgang Hesse. More Matters on (Meta-)Modelling: Remarks on Thomas Kühne Matters. *Software and Systems Modeling (SoSyM)*, 5(4):387–394, December 2006.

[HH] Peter Hut and Zef Hemel. Assembling Classes at Runtime. http://www.zefhemel.com/upload/-AssemblingClassesAtRuntime.pdf, Accessed: July 26, 2010.

[HMPR04] A. R. Hevner, S. T. March, J. Park, and S. Ram. Design Science in Information Systems Research. *MIS Quarterly*, 28(1):75–106, 2004.

[Hof80] Douglas R. Hofstadter. *Gödel, Escher, Bach: an Eternal Golden Braid*. Published by Vintage Books, 1980.

[HPSvH02] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. Reviewing the Design of DAML+OIL: An Ontology Language for the Semantic Web. In *AAAI/IAAI*, pages 792–797, 2002.

[Inc03] Honeywell Inc. *DOME (the DOmain Modeling Environment)*. 2003.

[Ing78] Daniel H. Ingalls. The smalltalk-76 programming system design and implementation. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 9–16, New York, NY, USA, 1978. ACM Press.

[Inm96] W. H. Inmon. The data warehouse and data mining. *Commun. ACM*, 39(11):49–50, 1996.

[Int] Intentional Software. Technical report. http://intentsoft.com/.

[ITU99] ITU-T. SDL - ITU-T Specification and Description Language, Formal Semantics. ITU-T Recommendation Z.100, Annex F, 1999.

[Iva02] Ivan Porres. A Toolkit for Manipulating UML Models. Technical Report 441, Turku Centre for Computer Science. Technical report, January 2002.

[Jac04] Jack Greenfield and Keith Short. Moving to Software Factories. July 2004. http://blogs.msdn.com/askburton/articles/232021.aspx, Accessed: July 26, 2010.

[Jav02] Java Community Process: JSR-40, Specification Lead: Ravi Dirckze, Unisys Corporation. Java^{TM} Metadata Interface(JMI) Specification. Technical report, http://java.sun.com/products/jmi/, June 2002.

[JB06] Frédéric Jouault and Jean Bézivin. KM3: A DSL for Metamodel Specification. In *FMOODS*, pages 171–185, 2006.

[Jel02] Rick Jelliffe. The Schematron Assertion Language 1.5. Specification. *Academia Sinica Computing Centre*, October 2002. http://xml.ascc.net/resource/schematron/Schematron2000.html, Accessed: July 26, 2010.

[JMI] Java Metadata Interface (JMI), Reference Implementation. Technical report. `http://ecommunity.unisys.com/`.

[Joh] Stephen C. Johnson. yacc – Yet Another Compiler-Compiler. Technical report. http://dinosaur.compilertools.net/yacc/index.html, Accessed: July 26, 2010.

[JW97] Ralph Johnson and Bobby Woolf. *Type Object*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[KA08] Thomas Kühne and Colin Atkinson. Reducing Accidental Complexity in Domain Models. *Journal on Software and Systems Modeling*, pages DOI: 10.1007/s10270–007–0061–0, 2008.

[Kab07] Jevgeni Kabanov. JavaRebel Brings Class Reloading to Java. *TheServerSide.com*, October 2007.

[KBJV06] Ivan Kurtev, Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. Model-based DSL Frameworks. In *OOPSLA Companion*, pages 602–616, 2006.

[KCH+01] Paul Kogut, Stephen Cranefield, Lewis Hart, Mark Dutra, Kenneth Baclawski, Mieczyslaw Kokar, and Jeffrey Smith. Extending UML to Support Ontology Engineering for the Semantic Web. volume 2185, pages 342–360, 2001.

[Ken02] Stuart Kent. Model driven engineering. In *IFM*, pages 286–298, 2002.

[Kic91] Gregor Kiczales. *The Art of the Metaobject Protocol*. MIT Press, July 1991.

[Kn05] Alexander Knigs. Model transformation with triple graph grammars. In *In Model Transformations in Practice Satellite Workshop of MODELS 2005, Montego*, 2005.

[KS07] Thomas Kühne and Daniel Schreiber. Can Programming be Liberated from the Two-level Style? – Multi-level Programming with DeepJava. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 229–244, NY, USA, 2007. ACM.

[Küh05] Thomas Kühne. What is a Model? In *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, 2005. http://drops.dagstuhl.de/opus/volltexte/2005/23 [2006-04-10].

[Küh06] Thomas Kühne. Matters of (Meta-) Modeling. *Software and Systems Modeling (SoSyM)*, 5(4):369–385, December 2006.

[Kum92] Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992.

[Kur05] Ivan Kurtev. *Adaptability of Model Transformations*. PhD thesis, University of Twente, 2005. http://wwwhome.cs.utwente.nl/~kurtev/files/thesis.pdf, Accessed: July 26, 2010.

[KWB03] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture - Practice and Promise*. 2003.

[LK09] Alfons Laarman and Ivan Kurtev. Ontological Metamodeling with Explicit Instantiation. In *SLE*, Denver, Colorrado, October 2009.

[LS] M. E. Lesk and E. Schmidt. Lex - A Lexical Analyzer Generator. http://www.w3.org/TR/xhtml2/, Accessed: July 26, 2010.

[Lud03] Jochen Ludewig. Models in software engineering. *Software and System Modeling*, 2(1):5–14, 2003.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley Longman Publishing Co., 1999.

[Mat] Martin Matula. *NetBeans Metadata Repository*. http://mdr.netbeans.org/MDR-whitepaper.pdf.

[MEB] `http://osys.grm.hia.no/mebacc/`.

[Met05] MetaCase. MetaEdit+. Version 4.0. Evaluation Tutorial. Technical report, MetaCase, 2005. http://www.metacase.com/support/40/manuals/-eval40sr2a4.pdf, Accessed: July 26, 2010.

[Met06] Metacase, 2006. MetaEdit+: http://www.metacase.com/.

[Mic06] Microsoft. Information on Visual Studio Team System. Technical report, Microsoft, 2006. http://lab.msdn.microsoft.com/vs2005/teamsystem, Accessed: 2006.

[MM03] J. Miller and J. Mukerji. Mda guide version 1.0.1. Technical report, Object Management Group (OMG), 2003.

[MMPr93] Ole Lehrmann Madsen, Birger MøllerPedersen, and Kristen Nygård. *Object-Oriented Programming in the Beta Programming Language*. Published by ACM Press, Addison-Wesley, 1993.

[MNPW10] Theo Dirk Meijler, Jan Pettersen Nytun, Andreas Prinz, and Hans Wortmann. Supporting Fine-grained Generative Model-driven Evolution. *Software and Systems Modeling*, 9(3):403–424, January 2010.

[MO95] James Martin and James Odell. *Object Oriented Methods: A Foundation.* Prentice Hall, Englewood Cliffs, NJ, 1995.

[MPS06] Meta programming system. Technical report, 2006. http://www.jetbrains.com/mps/, Accessed: July 26, 2010.

[MRB03] S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: A Programming Platform for Generic Model Management. In *SIGMOD*, pages 193–204, 2003.

[MS95] Salvatore T. March and Gerald F. Smith. Design and natural science research on information technology. *Decis. Support Syst.*, 15(4):251–266, 1995.

[NCEF02] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xLinkit: a Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology, 2(2)*, May 2002. http://xml.coverpages.org/xlinkitwp200102.pdf, Accessed: July 26, 2010.

[net03a] netBeans.org. *NetBeans IDE*. 2003.

[net03b] netBeans.org. *UML2MOF Tool, http://mdr.netbeans.org/-uml2mof, Accessed: 2005*. 2003.

[NJ03] Jan Pettersen Nytun and Christian S. Jensen. Modeling and Testing Legacy Data Consistency Requirements. In *UML 2003 - The Unified Modeling Language: Modeling Languages and Applications*, volume 2863 of *Lecture Notes in Computer Science*, pages 341–355. Published by Springer, October 2003.

[NJO03] Jan Pettersen Nytun, Christian S. Jensen, and Vladimir A. Oleshchuk. Towards a Data Consistency Modeling and Testing Framework for MOF Defined Languages. In *Norsk informatikkonferanse NIK'2003*, pages 149–158, http://www.nik.no/ (March 2004), November 2003. Published by Tapir akademisk forlag.

[NP04] Jan Pettersen Nytun and Andreas Prinz. Metalevel representation and philosophical ontology. In *Proc. of ECOOP workshop on Philosophy, Ontology, and Information Systems*. University of Oslo, Norway, 2004.

[NPK04] J. P. Nytun, A. Prinz, and A. Kunert. Representation of levels and instantiation in a metamodelling environment. In *Proc. of the Nordic UML Workshop*. Turku Centre for Computer Science, Finland, 2004.

[NPT06] Jan Pettersen Nytun, Andreas Prinz, and Merete Skjelten Tveit. Automatic generation of modelling tools. In *Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA*, volume 4066 of *Proceedings. Lecture Notes in Computer Science*, pages 268–283. Published by Springer, July 2006.

[Nyt06] Jan Pettersen Nytun. A generic model for connecting models in a multilevel modelling framework. In *ICSOFT*, pages 302–311. INSTICC Press, 2006.

[Ode94] James Odell. Power types. *JOOP*, 7(2):8–12, 1994.

[OMG] `http://www.omg.org`.

[OMG02] OMG Editor. *XML Metadata Interchange (XMI) Specification v1.2*. OMG Document. Published by Object Management Group, http://www.omg.org, January 2002.

[OMG03a] OMG. *Model Driven Architecture Guide, Version 1.0.1, OMG Document: omg/03-06-01*. Object Management Group, June 2003.

[OMG03b] OMG Editor. *Meta Object Facility (MOF) 2.0 Core Proposal, OMG Document:ad/2003-04-07*. 2003.

[OMG03c] OMG Editor. *OMG Unified Modeling Language Specification, Version 1.5*. OMG Document. Published by Object Management Group, http://www.omg.org, March 2003.

[OMG03d] OMG Editor. *OMG Unified Modeling Language Specification, Version 2.0*. OMG Document. Published by Object Management Group, http://www.omg.org, 2003.

[OMG03e] OMG Editor. *Ontology Definition Metamod.RFP, OMG Document: ad/2003-03-40*. 2003. http://www.omg.org/.

[OMG03f] OMG Editor. Production Rule Representation.RFP, OMG Document: br/2003-09-03. Technical report, 2003.

[OMG03g] OMG Editor. *Reponse to UML 2.0 Infrastructure RFP (OMG Document ad/00-09-01), 3rd revised submission*. OMG Document: ad/2003-03-01. Published by Object Management Group, http://www.omg.org, March 2003.

[OMG03h] OMG Editor. Revised Submission to OMG RFP ad/2003-04-07: Meta Object Facility (MOF) 2.0 Core Proposal. Technical report, April 2003. http://www.omg.org/.

[OMG03i] OMG Editor. *UML 2.0 Superstructure Specification*. OMG Document: ptc/03-08-02. Published by Object Management Group, http://www.omg.org, August 2003.

[OMG04] OMG. Human-Usable Textual Notation (HUTN) Specification Version 1.0. OMG Document: formal/04-08-01, Object Management Group, 2004. http://www.omg.org/.

[OMG05a] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Final Adopted

Specification. OMG Document: ptc/05-11-01, Object Management Group, 2005. http://www.omg.org/.

[OMG05b] OMG. *OCL 2.0 Specification*. OMG Document: ptc/2005-06-06. Object Management Group, June 2005.

[OMG05c] OMG. UML 1.4 with Action Semantics. OMG Document: ptc/02-01-09, Object Management Group, 2005. http://www.omg.org/.

[OMG05d] OMG. Unified Modeling Language: Diagram Interchange version 2.0. OMG Document: ptc/05-06-04, Object Management Group, 2005. http://www.omg.org/.

[OMG06a] OMG Editor. MOF Support for Semantic Structures Request For Proposal. Technical report, June 2006. http://www.omg.org/.

[OMG06b] OMG Editor. *UML 2.0 Infrastructure Specification*. OMG Document: formal/05-07-05. Published by Object Management Group, http://www.omg.org, March 2006.

[OMG07a] OMG. MOF 2.0/XMI Mapping Specification, v2.1.1. OMG Document: formal/07-12-01, Object Management Group, 2007. http://www.omg.org/.

[OMG07b] OMG Editor. *Unified Modeling Language: Infrastructure version 2.1.1*. OMG Document: formal/07-02-06. Published by Object Management Group, http://www.omg.org, February 2007.

[OMG07c] OMG Editor. *Unified Modeling Language: Superstructure, version 2.1.1*. OMG Document: formal/2007-02-05. Published by Object Management Group, http://www.omg.org, February 2007.

[OMG08] OMG Editor. *Common Object Request Broker Architecture (CORBA) version 3.1*. OMG Document. Published by Object Management Group, http://www.omg.org, January 2008.

[OMG09] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1. OMG Document: ptc/09-12-05, Object Management Group, 2009. http://www.omg.org/.

[OMG10] OMG Editor. *OMG Unified Modeling Language (OMG UML), Infrastructure V2.1.3*. OMG Document: formal/2010-05-03. Published by Object Management Group, http://www.omg.org, May 2010.

[PNCW06] A. Prinz, J. P. Nytun, L. Chen, and S. Wei. Integration of MATER and EMF. In *Proc. of the 4th Nordic Workshop on the Unified Modeling Language NWUML'2006*, 2006. `http://grimstad.hia.no/nwuml06/`.

[PQ95] T. J. Parr and R. W. Quong. ANTLR: A Predicated-LL(k) Parser Generator. In *Software – Practice and Experience*, number Vol. 25(7). ACM Press New York, 1995.

[Pri00] Andreas Prinz. *Formal Semantics for RSDL: Definition and Implementation*. PhD thesis, Humboldt-Universitt zu Berlin, June 2000.

[Rat03] Corporation Software Rational. 2003. http://www.rational.com.

[Ree02] Trygve Reenskaug. A Rudimentary UML Virtual Machine as a Smalltalk Extension. 2002. http://heim.ifi.uio.no-/~trygver/2002/uml-vm/uml-vm-04.pdf, Accessed: July 26, 2010.

[RFBLO01] Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen, and Nosa Omorogbe. The Architecture of a UML Virtual Machine. In *OOPSLA*, pages 327–341, 2001.

[RJB05] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Model Language Reference Manual, second Edition*. Published by Pearson Education, Inc., 2005.

[RSK91] Marek Rusinkiewicz, Amit Sheth, and George Karabatis. Specifying Interdatabase Dependencies in a Multidatabase Environment. *Computer*, 24(12):46–53, dec 1991.

[Sch94] Andy Schürr. Specification of graph translators with triple graph grammars. In *WG*, pages 151–163, 1994.

[Sei03a] Ed Seidewitz. *What Do Models Mean?, OMG Document: ad/03-03-31*. Published by Object Management Group, March 2003.

[Sei03b] Ed Seidewitz. What Models Mean. *IEEE SOFTWARE*, pages 26–32, October 2003.

[SMI] http://osys.grm.hia.no/smile/index.htm, Accessed: 2006.

[Smi05] David Woodruff Smith. Phenomenology. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2005.

[Smi07] David Woodruff Smith. *Husserl*. Routledge, 2007.

[Sow00] John F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Published by Brooks Cole Publishing Co., 2000.

[SS04] Steffen Staab and Rudi Studer, editors. *Handbook on Ontologies*. International Handbooks on Information Systems. Springer, 2004.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, Wien, 1973.

[TGN05] Merete S. Tveit, Terje Gjøsæter, and Jan P. Nytun. A UML Model of HTML for Accessibility Testing of Web Documents. In *Workshop on Web Accessibility and Metamodelling, Grimstad, Norway*, 2005.

[TvS02] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Published by Prentice Hall, 2002.

[UTD] `http://www.utdanningsdirektoratet`.

[UWE] `http://www.wabcluster.org/uwem05/`.

[W3C99] The World Wide Web Consortium W3C. *Web Content Accessibility Guidelines 1.0 (WCAG10), http://www.w3.org/TR/WCAG10/*. May 1999.

[W3C06] The World Wide Web Consortium W3C. *Web Accessibility Initiative (WAI), http://www.w3.org/WAI/*. 2006.

[WAB] http://www.wabcluster.org/, Accessed: July 26, 2010.

[XHT05] XHTML 2.0, W3C Working Draft. 2005. `http://www.w3.org/TR/xhtml2/`.

[YJ07] Joseph W. Yoder and Ralph Johnson. The adaptive object-model architectural style, accessed May, 2007. http://www.adaptiveobjectmodel.com/WICSA3/Architecture-OfAOMsWICSA3.htm.