Language Family Engineering
— with Features and Role-Based Composition

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

ingereicht von

Dipl.-Inf. Christian Wende
geboren am 01. April 1982 in Hoyerswerda

Gutachter:
Prof. Dr. rer. nat. habil. Uwe Aßmann
(Technische Universität Dresden)

Prof. Dr. Colin Atkinson
(Universität Mannheim)


Dresden im Januar 2012
Abstract

The benefits of Model-Driven Software Development (MDSD) and Domain-Specific Languages (DSLs) wrt. efficiency and quality in software engineering increase the demand for custom languages and the need for efficient methods for language engineering. This motivated the introduction of language families that aim at further reducing the development costs and the maintenance effort for custom languages. The basic idea is to exploit the commonalities and provide means to enable systematic variation among a set of related languages.

Current techniques and methodologies for language engineering are not prepared to deal with the particular challenges of language families. First, language engineering processes lack means for a systematic analysis, specification and management of variability as found in language families. Second, technical approaches for a modular specification and realisation of languages suffer from insufficient modularity properties. They lack means for information hiding, for explicit module interfaces, for loose coupling, and for flexible module integration.

Our first contribution, Feature-Oriented Language Family Engineering (LFE), adapts methods from Software Product Line Engineering (SPLE) to the domain of language engineering. It extends Feature-Oriented Software Development (FOSD) to support metamodeling approaches used for language engineering and replaces state-of-the-art processes by a variability- and reuse-oriented LFE process. Feature-oriented techniques are used as means for systematic variability analysis, variability management, language variant specification, and the automatic derivation of custom language variants.

Our second contribution, Integrative Role-Based Language Composition, extends existing metamodeling approaches with roles. Role models introduce enhanced modularity for object-oriented specifications like abstract syntax metamodels. We introduce a role-based language for the specification of language components, a role-based composition language, and an extensible composition system to evaluate role-based language composition programs. The composition system introduces integrative, grey-box composition techniques for language syntax and semantics that realise the statics and dynamics of role composition, respectively.

To evaluate the introduced approaches and to show their applicability, we apply them in three major case studies. First, we use feature-oriented LFE to implement a language family for the ontology language OWL. Second, we employ role-based language composition to realise a component-based version of the language OCL. Third, we apply both approaches in combination for the development of SumUp, a family of languages for mathematical equations.
I would like to thank my supervisor Uwe Aßmann. He gave me the opportunity to spend my time on the interesting and challenging topic of language engineering. Uwe inspired my early ideas on role-based language composition and helped me in shaping them to a real research topic. He was a steady source of new inspiration, helpful advice, surprising insights, and far-reaching objectives. I’m very thankful for the excellent scientific and personal atmosphere he creates in his group.

I want to thank my colleague Florian Heidenreich for his indispensable contributions to this thesis. His steady interest in my research and the intensive discussions about features, product lines, and metamodelling contributed a lot to my perception and knowledge of these topics. His contributions to model-driven product-line engineering, his tool FeatureMapper, and our joint work in the project feasIPLe are an inevitable foundation to feature-oriented language family engineering. In addition to Florian, Jendrik Johannes and Mirko Seifert had a huge impact on my research. Jendrik’s work on grammars in Reuseware prepared what later became our joint project EMFText. Mirko’s amazing perfectionism transformed EMFText to the stable tool it is today and the solid foundation my research on model-driven language composition is based on. The joined publications with Florian, Jendrik and Mirko, taught me a lot about scientific and collaborative writing and opened my mind for many topics beyond my thesis. On the other hand, the three became close friends during the countless hours in the office and our joint trips to conferences or project meetings. We always mixed business and pleasure and had the best and worst ideas during endless nights in pubs in Dresden-Neustadt.

In addition, Birgit Demuth, Claas Wilke, Matthias Bräuer, Steffen Zschaler, and Sven Karol had a great influence on this thesis. Birgit introduced me into the world of research during my diploma thesis. Together with Claas, and Matthias she inspired the OCL case study during our joint work on the Dresden OCL Toolkit. Steffen worked with me on the early design of role-based language composition and taught me a lot about research and scientific writing. Steffen’s and Sven’s critical thinking and their constructive comments helped me in revising and refining the contributions of this thesis.
Lots of the research in this thesis was done in the research projects feasiPLe and MOST. I want to mention Henrik Lochmann, who shared my passion for enhanced language integration in feasiPLe. I want to mention Srdjan Zivkovic. We worked together on the MOST TOPF case study and developed a family of ontology-based software development tools. I also want to mention Fernando Silva Parreiras, Tobias Walter, and Gerd Gröner who worked with me on the integration of metamodels and ontologies in OWLizer and taught me a lot on the application of ontologies in software engineering. I thank Yuting Zhao, Jeff Z. Pan, and Edward Thomas who introduced me to the details of ontology reasoning with TrOWL and helped in realising the OWL case study.

I had the privilege to supervise Beata Dryzmajllo, Nils Thieme, Henri Mühle, Michael Thiele, and Johannes Mey during their diploma theses. They did some excellent work and contributed a lot to the design, realisation, and extension of the role-based language composition system introduced in this thesis. I thank them for our productive joint work and their patience with me.

Many thanks to Uwe, Florian, Mirko, and Jendrik for their invaluable comments on the written text. They spent their valuable time to comment draft versions of this thesis. I also acknowledge the effort of the various, anonymous reviewers of the publications related to this thesis. They helped a lot to improve the presentation of our ideas.

Finally, but not lastly, I want to thank my family: my parents Marion and Frank who always encouraged and supported me, my brother Martin, who tolerates his bothering brother since many years, my great-grandfather Hans who taught me to be passionate about learning new things, and my great-grandmother Hildegart who taught me serenity. I am deeply indebted to my love Nicole and our little daughter Hannah. They suffered most during the last weeks of finishing this thesis. Nicole took care of Hannah during my endless “overtime hours” and Hannah took care of keeping Nicole in a positive mood. You never doubted my abilities and encouraged me during the hardest parts of this work. I love you.

Christian Wende
December 2011
This thesis is based on a number of publications. We first enumerate peer-reviewed publications that directly relate to contributions of this thesis, i.e., feature-based language family engineering and role-based language composition. Then we enumerate peer-reviewed publications that provide or concern foundations of each contribution. Finally, we enumerate tutorials related to topics of this thesis and non-peer-reviewed publications. In co-authored publications, the author contributed a large part and in particular the parts this thesis is based on or relates to.

**Peer-reviewed Publications on Feature-Oriented Language Family Engineering**


**Peer-reviewed Publications on Role-Based Language Composition**


**Peer-reviewed Publications on Feature-Oriented Development**


**Peer-reviewed Publications on Language Engineering and Language Family Engineering**


Tutorials at International Summer Schools and Conferences


Technical Reports Related to this Thesis


### Contents

List of Figures .................................................. xv

List of Tables .................................................... xix

List of Listings .................................................. xxi

1. Introduction .................................................... 1
   1.1. The Omnipresence of Language Families .................. 2
   1.2. Challenges for Language Family Engineering ............... 5
   1.3. Language Family Engineering with Features and Role-Based Composition 7

2. Review of Current Language Engineering ..................... 13
   2.1. Language Engineering Processes .......................... 13
      2.1.1. Analysis Phase ........................................ 13
      2.1.2. Design Phase .......................................... 14
      2.1.3. Implementation Phase ................................. 14
      2.1.4. Applicability in Language Family Engineering ........ 15
      2.1.5. Requirements for an Enhanced LFE Process .......... 15
   2.2. Technical Approaches in Language Engineering .......... 16
      2.2.1. Specification of Abstract Syntax ..................... 16
      2.2.2. Specification of Concrete Syntax ..................... 29
      2.2.3. Specification of Semantics ............................ 36
      2.2.4. Requirements for an Enhanced LFE Technique .......... 42

3. Feature-Oriented Language Family Engineering .............. 45
   3.1. Foundations of Feature-Oriented SPLE .................... 45
      3.1.1. Introduction to SPLE ................................ 45
      3.1.2. Feature-Oriented Software Development ................ 47
3.2. Feature-Oriented Language Family Engineering .......................... 52
  3.2.1. Variability and Variant Specification in LFE .......................... 52
  3.2.2. Product-Line Realisation, Mapping and Variant Derivation for LFE 57
3.3. Case Study: Scalability in Ontology Specification, Evaluation and Application ........................................... 63
  3.3.1. Review of Evolution, Customisation and Combination in the OWL Language Family ........................................... 63
  3.3.2. Application of Feature-Oriented Language Family Engineering for OWL ........................................... 65
3.4. Discussion ........................................................................ 86
  3.4.1. Contributions .................................................................. 87
  3.4.2. Related Work .................................................................. 90
  3.4.3. Conclusion ..................................................................... 93

4. Integrative, Role-Based Composition for Language Family Engineering 95
  4.1. Foundations of Role-Based Modelling ........................................... 95
    4.1.1. Information Hiding and Interface Specification in Role Models 96
    4.1.2. Loose Coupling and Flexible Integration in Role Composition 97
  4.2. The LanGems Language Composition System ............................. 98
    4.2.1. The Language Component Specification Language ..................... 98
    4.2.2. The Language Composition Language ...................................... 104
    4.2.3. Techniques of Language Composition ...................................... 110
  4.3. Case Study: Component-based OCL ......................................... 132
    4.3.1. Role-Based OCL Modularisation .......................................... 134
    4.3.2. Role-Based OCL Composition ............................................. 140
  4.4. Discussion ........................................................................ 142
    4.4.1. Contributions .................................................................. 142
    4.4.2. Related Work .................................................................. 145
    4.4.3. Conclusion ..................................................................... 148

5. LFE with Integrative, Role-Based Syntax and Semantics Composition 153
  5.1. Integrating Features and Roles ............................................... 154
  5.2. SumUp Case Study .................................................................. 154
    5.2.1. Motivation ..................................................................... 155
    5.2.2. Feature-Oriented Variability and Variant Specification ............. 155
    5.2.3. Role-Based Component Realisation ....................................... 156
    5.2.4. Feature-Oriented Variability and Variant Evolution .................. 158
    5.2.5. Model-driven Concrete Syntax Realisation ............................... 159
    5.2.6. Model-driven Semantics Realisation ....................................... 159
    5.2.7. Role-Based Composition and Feature Mapping ......................... 169
    5.2.8. Language Variant Derivation ............................................. 170
  5.3. Conclusion ........................................................................ 171

6. Conclusion ................................................................. 175
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Phenomena of language development, their motivating force and results.</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>LFE with features and role-based composition.</td>
<td>8</td>
</tr>
<tr>
<td>1.3</td>
<td>Combination of role-based language composition and feature-oriented LFE.</td>
<td>11</td>
</tr>
<tr>
<td>2.1</td>
<td>EMF implementation of MOF three-layered metamodelling architecture.</td>
<td>19</td>
</tr>
<tr>
<td>2.2</td>
<td>GXL approach for metamodelling for graph-based languages.</td>
<td>21</td>
</tr>
<tr>
<td>2.3</td>
<td>OWL ontologies applied in metamodelling and modelling.</td>
<td>24</td>
</tr>
<tr>
<td>3.1</td>
<td>Generic process for SPLE.</td>
<td>46</td>
</tr>
<tr>
<td>3.2</td>
<td>FOSD process for SPLE.</td>
<td>47</td>
</tr>
<tr>
<td>3.3</td>
<td>Feature-oriented process for LFE.</td>
<td>52</td>
</tr>
<tr>
<td>3.4</td>
<td>Variability in language features of DL language family.</td>
<td>53</td>
</tr>
<tr>
<td>3.5</td>
<td>Variability on language families used in software development activities.</td>
<td>54</td>
</tr>
<tr>
<td>3.6</td>
<td>Variability for language families tooling built with EMFText.</td>
<td>55</td>
</tr>
<tr>
<td>3.7</td>
<td>Variability of metamodelling languages in Eclipse/EMF.</td>
<td>57</td>
</tr>
<tr>
<td>3.8</td>
<td>Categorisation of exemplary variability dimensions wrt. abstraction and</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>metamodelling level.</td>
<td></td>
</tr>
<tr>
<td>3.9</td>
<td>Combination of FeatureMapper and EMFText for variability mapping and</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>variant derivation in LFE.</td>
<td></td>
</tr>
<tr>
<td>3.10</td>
<td>Application of FeatureMapper for mapping variability to metamodel elements</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>in LFE.</td>
<td></td>
</tr>
<tr>
<td>3.11</td>
<td>Application of variant editor in variant specification for OWL2 language</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td>features.</td>
<td></td>
</tr>
<tr>
<td>3.12</td>
<td>Application of custom editor derived for OWL2 language variant.</td>
<td>75</td>
</tr>
<tr>
<td>3.13</td>
<td>Variability for OWL2 reasoning infrastructure.</td>
<td>78</td>
</tr>
<tr>
<td>3.14</td>
<td>Application of FeatureMapper for mapping variability to TrOWL configuration</td>
<td>79</td>
</tr>
<tr>
<td></td>
<td>files in LFE.</td>
<td></td>
</tr>
<tr>
<td>3.15</td>
<td>Variability for OWL2 application in MOST TOPF tool environments.</td>
<td>83</td>
</tr>
</tbody>
</table>
List of Figures

3.16. Generic architecture for ontology-based MDSD tool environments. . . . . . 84
3.17. Application of FeatureMapper for mapping variability to components of
the MOST TOPF architecture in LFE. ........................................ 85
3.18. Variant model for SAP MOST TOPF variant. ................................ 85
3.19. Demonstrator for SAP case study derived from MOST TOPF. .......... 86
4.1. Overview of the LanGems language composition system. ................... 99
4.2. Syntactic and semantic interface of the language metamodel and their
relation to concrete syntax and semantics. .................. 99
4.3. Introduction of explicit interfaces for metamodel composition in role-based
metamodelling. ............................................................. 100
4.4. Metamodel of the basic Ecore metamodelling language. .................... 101
4.5. Metamodel of the LanGems LCSL. ........................................ 102
4.6. Control flow between integrated language components for role operation
and augmentation bindings. ................................. 103
4.7. Metamodel of the LanGems LCL. .......................................... 106
4.8. Overview of the LanGems composition process. ................................. 112
4.9. Exemplary instance of LanGems composition process. ...................... 113
4.11. Integrative composition pattern for EMF to implement statics of role com-
position. ............................................................. 116
4.12. Integrated, object-oriented metamodel of composed formFlow language. . 117
4.14. Textual editor generated from integrated formFlow syntax. ........... 119
4.15. Concepts of GMF specification languages mapped to an exemplary GMF
editor. ............................................................. 121
4.16. Process for GMF graphical concrete syntax composition in LanGems. 122
4.17. Example of GMF graphical editor composition for CPNs. ................. 124
4.18. Process for Java operational semantics composition in LanGems. .... 126
4.19. Exemplary wizard dialogue derived from a formFlow specification. ... 132
4.20. Evolution, customisation and combination in the OCL language family. 134
4.21. Modularisation of OCL grammar using grammar inheri-
tance [Akehurst 2007]. ................................................. 135
4.22. OCL core component. .................................................. 137
4.23. Editor generated for OCL with temporal and initial extensions. 142
5.1. Process for LFE with features and integrative, role-based composition. 154
5.2. Feature model for SumUp variability specification. ...................... 155
5.3. Variant specifications for custom SumUp variants. ......................... 156
5.4. SumUp language components. .......................................... 157
5.5. Refined feature model for SumUp variability specification. ........... 159
5.6. Guided refinement of SumUp variant specification for Martin. ......... 160
5.7. CPN component for semantics specification of sheet component. ....... 163
5.8. CPN component for semantics specification of math component. ....... 164
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.9</td>
<td>Process for CPN semantics composition in LanGems</td>
<td>166</td>
</tr>
<tr>
<td>5.10</td>
<td>Integrative composition pattern for CPN to implement dynamics of role</td>
<td>168</td>
</tr>
<tr>
<td></td>
<td>composition</td>
<td></td>
</tr>
<tr>
<td>5.11</td>
<td>Exemplary application of CPN composition pattern</td>
<td>169</td>
</tr>
<tr>
<td>5.12</td>
<td>Exemplary role bindings in SumUp composition program</td>
<td>170</td>
</tr>
<tr>
<td>5.13</td>
<td>Reduced role bindings for Christian’s SumUp variant</td>
<td>171</td>
</tr>
<tr>
<td>5.14</td>
<td>Editors derived for custom SumUp variants</td>
<td>172</td>
</tr>
<tr>
<td>6.1</td>
<td>Contribution I: Feature-oriented LFE process</td>
<td>176</td>
</tr>
<tr>
<td>6.2</td>
<td>Contribution II: Role-based language composition technique</td>
<td>177</td>
</tr>
</tbody>
</table>
1.1. Exemplary selection of families of modelling languages and their founding phenomena. ........................................... 4

2.1. Formalisms and technical spaces for metamodelling. ......................... 17
2.2. Alignment of terminology for metamodelling formalisms and technical spaces. ....................................................... 26
2.3. Subclassing ambiguity: subtyping vs. inheritance. ......................... 27
2.4. Formalisms and technical spaces for textual syntax specification. ....... 30
2.5. Formalisms and technical spaces for graphical syntax specification. ...... 33
2.6. Formalisms and technical spaces for semantics specification. ............ 37

3.1. Feature relationships and their semantics in propositional logic. ........... 49
3.2. Classification of variability dimensions. .................................. 58
3.3. Selection of languages from EMFText zoo applicable for variability mapping. .................................................. 61
3.5. Variability of MOST Case studies wrt. common requirements for tool environments. ............................................... 82

4.1. Mapping of Steimann’s role features (feature numbers refer to [Steimann 2000b]) to modularity properties desired for language modularisation. .................................................. 97

5.1. Concepts and Syntax for CPNs. ............................................. 161

6.1. Overview of case studies used for qualitative evaluation of our approach. . 179
List of Listings

3.1. Excerpt of OWL2 metamodel definition in TextEcore syntax. .......... 70
3.2. Excerpt of OWL2 syntax definition in EMFText syntax. ............... 71

4.1. Role-based specification of statechart language component in LCSL. . 105
4.2. Role-based specification of form language component in LCSL. ....... 108
4.3. Role-based composition of formFlow language in LCSL ............... 109
4.4. Role-based composition of logging for formFlow language in LCSL . 110
4.5. Role-based specification of logger language component in LCSL ....... 110
4.6. Role-based composition of CPN language in LCSL .................. 125
4.7. Pattern to generate AspectJ aspect for role binding .................. 128
4.8. Pattern to generate AspectJ pointcut-advice for role operation binding binding. ...................................................... 128
4.9. Pattern to generate AspectJ pointcut-advice for before binding of augmentation. ................................................ 129
4.10. Pattern to generate AspectJ pointcut-advice for after binding of augmentation. .............................................. 129
4.12. AspectJ aspect generated for role binding of natural State to role LoggingArtifact defined in Listing 4.4. ......................... 130
4.11. AspectJ aspect generated for role binding of natural Form to role Activity defined in Listing 4.3. ........................... 131
4.13. Realisation of priorities in syntax specification of OCL math component. 138

5.1. Excerpt of SumUp composition program in LCSL ...................... 167

A.1. Role-based specification of OCL core component in LCSL .......................... 183
A.2. Syntax specification of OCL core component. ................................. 184
A.3. Role-based specification of OCL logic component in LCSL ................. 185
A.4. Syntax specification of OCL logic component. ................................. 186
List of Listings

A.5. Role-based specification of OCL math component in LCSL .................. 187
A.6. Syntax specification of OCL math component ................................. 187
A.7. Role-based specification of OCL collection component in LCSL ......... 188
A.8. Syntax specification of OCL collection component ........................... 189
A.9. Role-based specification of OCL messages component in LCSL .......... 190
A.10. Syntax specification of OCL messages component ........................... 190
A.11. Role-based specification of OCL tuple component in LCSL ............. 191
A.12. Syntax specification of OCL tuple component ............................... 191
A.13. Role-based specification of OCL string component in LCSL ............ 192
A.14. Syntax specification of OCL string component ............................... 192
A.15. Role-based specification of OCL classifiercontext component in LCSL . 192
A.16. Syntax specification of OCL classifiercontext component ................. 193
A.17. Role-based specification of OCL attributecontext component in LCSL . 193
A.18. Syntax specification of OCL attributecontext component ................. 194
A.19. Role-based specification of OCL operationcontext component in LCSL . 194
A.20. Syntax specification of OCL operationcontext component .................. 195
A.22. Syntax specification of OCL oclpackage component ........................ 195
A.23. Role-based specification of OCL initial component in LCSL .......... 196
A.24. Syntax specification of OCL initial component .............................. 196
A.25. Role-based specification of OCL temporal component in LCSL .......... 197
A.26. Syntax specification of OCL temporal component ............................ 198
A.27. Composition program for complete OCL including temporal and initial extensions in LCL .................................................. 198

B.1. Role-based specification of SumUp sheet component in LCSL .......... 203
B.2. Syntax specification of SumUp sheet component .............................. 203
B.3. CPN-based semantics specification of SumUp sheet component .......... 204
B.4. Role-based specification of SumUp math component in LCSL ............. 204
B.5. Syntax specification of SumUp math component .............................. 205
B.6. CPN-based semantics specification of SumUp math component .......... 206
B.7. Role-based specification of SumUp conditional component in LCSL .... 209
B.8. Syntax specification of SumUp conditional component ........................ 210
B.9. CPN-based semantics specification of SumUp conditional component .... 210
B.10. Role-based specification of SumUp units component in LCSL ........... 213
B.11. Syntax specification of SumUp units component ............................ 213
B.12. CPN-based semantics specification of SumUp units component .......... 214
B.13. Role-based specification of SumUp variables component in LCSL ...... 214
B.14. Syntax specification of SumUp variables component ........................ 214
B.15. CPN-based semantics specification of SumUp variables component ..... 215
B.16. Role-based specification of SumUp strings component in LCSL .......... 216
B.17. Syntax specification of SumUp strings component .......................... 216
B.18. CPN-based semantics specification of SumUp strings component ...... 216
B.19. Composition program for complete SumUp language in LCL ............. 217
Introduction

“Language is the picture and counterpart of thought.”
– Mark Hopkins, 1841.

“Ein Ding verstehen, heißt nicht, seine Bewegung und seine Materie beschreiben, sondern seinen Sinn einsehen.”

To understand a thing, does not mean to describe its movement and material, but to realise its meaning.

The development of software systems has always involved particular languages to prescribe the structure and behaviour of software systems. In computer science, Language Engineering describes the attempt to establish systematic means for the design and implementation of such languages. The investigation of topics like formal language theory [Chomsky 1965], compiler construction [Hopcroft 1979], lexical, syntactic or semantics analysis, and code generation [Aho 1985] shaped the early decades of research in language engineering. The theoretical and practical results of this research provide a solid technical foundation for the application of languages in all areas of computer science ranging from hardware description to cloud computing.

A current trend of increased diversification and extension of language applications is indicated by the growing interest in topics like Domain-Specific Languages (DSLs) [van Deursen 2000], Model-Driven Engineering (MDE) [Schmidt 2006], Model-Driven Development (MDD) [Atkinson 2003, Selic 2003], or Model-Driven Software Development (MDSD) [Stahl 2006, Greenfield 2004]. DSLs contribute language features customised for a specific domain, i.e., custom types of expressions that capture a particular domain abstraction, a specific syntax, or specific evaluation characteris-
Chapter 1. Introduction


To serve the growing demand for custom languages, this thesis treats the topic of Language Family Engineering (LFE). Language families describe the attempt to enhance the efficiency of language engineering by transferring the idea of product families or product lines from software to languages. Software Product Line Engineering (SPLE) [Pohl 2005] enables a more systematic management of variable and common requirements and a more efficient reuse and customisation of realisation artefacts among a set of related software products. The application of the SPLE idea for language families requires an adaptation of SPLE and language engineering approaches to match the particular challenges of LFE. This is our aim for this thesis. We motivate the idea of language families, identify challenges for LFE, investigate existing techniques for language engineering for issues that complicate their application on language families, introduce a comprehensible approach for LFE, and evaluate the conceptual, technical, and qualitative contributions of this approach in exemplary case studies.

1.1. The Omnipresence of Language Families

The growing number of languages developed and the model- and language-centric development methods introduced with MDSD, MDE, and MDD led to a new way of using and perceiving languages: (1) Languages are no longer used in isolation. Multi-dimensional methods [Ossher 2002] for system specification motivate the combination of multiple, orthogonal languages for specifying software systems, e.g., [Kienzle 2009, Atkinson 2010, Johannes 2010]. (2) Languages are used in a more agile way. They are adapted, customised, and extended more often and more rapidly. (3) Languages can be developed more efficiently. This enables faster language evolution and contributes to language diversification. (4) Dedicated languages are considered beneficial for an ever-growing number of applications.

These observations led to the notion of language families. The term is borrowed from the discipline of comparative linguistics of human languages. There, a language family is defined as a group or set of languages that are derived from a common ancestor the so-called proto-language [Hewes 1973, Fitch 2010]. Based on this relationship, languages of the same family share common features.

The omnipresence of language families in computer science is supported by the number of scientific publications that suggest language families for various domains (cf. Table 1.1). In analogy to linguistics, these publications use the term language family to denote a set of formal languages that share a common core but vary in certain features.
1.1. The Omnipresence of Language Families

The variation of language features is induced by the different requirements of the domains that they are applied to.

This common definition describes how a language family appears at a single point of time. To identify the challenges for methods and techniques for LFE, it is, however, important to also understand how a language family develops over time. By considering both the influence of different application domains and time, we identified the following three sufficient phenomena of language development for founding language families:

**Language Evolution** The phenomenon of language evolution can be observed for every language that is continuously developed over a longer period of time [Levenez, E. 2011]. It creates different versions of a language that result from the maintenance and extension of language features over time. Fig. 1.1 illustrates the continued evolution of languages like OCL [OCL 2010a], UML [UML 2009], or MOF [MOF 2006]. Further examples are the development of languages like C++ [Stroustrup 2007], Java [Gosling 1996, Gosling 2000, Gosling 2005], or OWL [OWL 2009a].

**Language Customisation** The phenomenon of language customisation can be observed when a single language is adapted for different domains. It creates language variants or dialects. These variants are derived from a shared core language. The objective of language customisation is to share the common features of the core language, but also to address the particularities of the specific domain. Fig. 1.1 exemplifies language customisation by the extraction of the OCL language from the UML standard and the specialisation and extension of OCL for a particular domain e.g., constraints using temporal logics (TOCL [Ziemann 2003]). Further examples are other members...
<table>
<thead>
<tr>
<th>Family</th>
<th>Description</th>
<th>Phenomena</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADL Family</td>
<td>ADL is a family of languages for describing software architectures. It can be customised to support different levels of abstraction and different styles of architecture.</td>
<td>Customisation</td>
</tr>
<tr>
<td>EOL</td>
<td>The Epsilon Object Language (EOL) consists of a set of languages for model management.</td>
<td>Combination</td>
</tr>
<tr>
<td>MOF</td>
<td>The MetaObject Facility (MOF) standard contributes metamodelling languages (e.g., Essential MOF (EMOF), Complete MOF (CMOF)) with different complexity and expressiveness.</td>
<td>Evolution, Customisation</td>
</tr>
<tr>
<td>OCL</td>
<td>The Object Constraint Language (OCL) standard defines a language to define constraints for object-oriented modelling languages. OCL has steadily evolved, was combined with different languages, and customised for different domains.</td>
<td>Evolution, Customisation, Combination</td>
</tr>
<tr>
<td>OWL</td>
<td>The Web Ontology Language (OWL) contributes a set of ontology languages with different expressiveness and reasoning efficiency. It is also used in combination with rule languages like Semantic Web Rule Language (SWRL) [SWR 2004].</td>
<td>Evolution, Customisation, Combination</td>
</tr>
<tr>
<td>QVT</td>
<td>The Query/View/Transformation (QVT) standard contributes an imperative and two declarative languages for model transformations.</td>
<td>Customisation</td>
</tr>
<tr>
<td>UML</td>
<td>The Unified Modeling Language (UML) standard contributes a family [Atkinson 2001] of integrated languages (e.g., state charts, class diagrams, sequence diagrams) to design software systems.</td>
<td>Evolution, Combination</td>
</tr>
<tr>
<td>VML*</td>
<td>The VML* language family contributes a set of imperative languages for variability realisation in software product lines. Each family member contributes a dedicated language that is customised for a particular modelling language.</td>
<td>Customisation, Combination</td>
</tr>
<tr>
<td>WebDSL</td>
<td>WebDSL contributes a family of languages for the design of web applications. It includes languages for user interface specification, data modelling, access control, or data validation. These languages are interrelated and integrated, i.e., specifications in different languages can refer to each other.</td>
<td>Combination</td>
</tr>
</tbody>
</table>

Table 1.1.: Exemplary selection of families of modelling languages and their founding phenomena.
Challenges for Language Family Engineering

For decades, language engineering was focused on the formal foundations of techniques and tools to automate particular tasks of language implementation. Given a formal language specification, language engineering tools like scanner, parser, or compiler generators can derive custom processing tools for system specifications written in a particular language [Aho 2006]. The importance of the generative language engineering tools for today’s language engineering is without any doubt. However, the phenomena that create language families impose additional challenges for LFE:

Challenges from Language Evolution: The phenomenon of language evolution is motivated by the need for continued language development over time. It results in different language versions. A challenge for language evolution is to enable efficient reuse of language features among different versions and to provide means to easily remove, add and revise language features.

Challenges from Language Customisation: One motivation for language customisation is a diversification of domains a DSL is applied to. Another motivation for customisation is when a language covers a number of domains and is meant to be
Chapter 1. Introduction

Partitioned into sub-languages. Customisation results in variations among the features of a language. It results in a number of custom language variants sharing a common core. Such customisation raises the challenge of a **systematic management of variation and commonalities** and requires means to implement **variants of language specifications**.

**Challenges from Language Combination:** Language combination is motivated by the need for interaction among different languages and their features. Such interaction enables multi-dimensional system specifications while providing an adequate abstraction for each dimension. Language combination results in a family of languages with implicit and explicit interdependencies. This introduces the challenge to **integrate different languages and their abstractions for interoperability** while preserving the independence of the individual languages.

Solving the challenges induced by language evolution, customisation and combination, requires a revision of both methodical and technical aspects of language engineering.

Historically, methodologies for language engineering are focused on the implementation of the different processing tools for a single language [Aho 2006]. Different authors introduce approaches to systematise the language engineering process [Klint 2005, Mernik 2005] and to strengthen the analysis phase in language engineering [Thibault 1999, van Deursen 2002, Mernik 2005]. Unfortunately, these processes are rather abstract and the analysis phase is not or just loosely coupled to language design and implementation. Furthermore, language engineering processes are designed in a rather linear top-down approach. We consider both issues a major limitation wrt. LFE. For evolution, customisation, and combination in language families, it is vital to keep track of language features identified during the analysis phase, their interdependencies, and their relation to design and implementation artefacts.

1.3. Language Family Engineering with Features and Role-Based Composition


Although some of these approaches provide advanced compositional properties, they lack a separation of means for module specification and module integration. From research in software composition [Aßmann 2003], we learned that the distinction of a component model for module specification and a composition language for module integration makes composition approaches more comprehensible and flexible. As its hard to anticipate the changes required for language evolution and language customisation and the potential scenarios for language combination, a potent technique for module integration is required.

Furthermore, most concrete syntax and semantics approaches lack an integration with current abstract syntax metamodelling approaches. They contribute a different, proprietary technical space and are, thus, hard to integrate in a comprehensive language engineering approach.

In summary, an approach for LFE is missing that provides advanced variability management among the consecutive phases of language engineering and that supports integrative composition for the different technology spaces involved in language engineering.

1.3. Language Family Engineering with Features and Role-Based Composition

The issues of state-of-the-art language engineering that were outlined in the previous section have to be addressed in two ways. First, a process has to be developed that enables systematic management of variability and reuse in language engineering. Second, techniques have to be provided that enable the implementation, customisation and integration of reusable language components. To address both, this thesis contributes a comprehensive approach for LFE based on features and role-based composition. As depicted in Fig. 1.2, our approach is based on the historical foundations of language engineering and its extensions to modular language engineering. It emphasises two fundamental principles regarding the language engineering process and technique, respectively:

**The Principle of Variability** The principle of variability recognises the strong need for variability inherent in the above challenges for LFE and aims at providing a process for the continuous analysis, management and realisation of variability. Such techniques are expected to provide conceptual means and a systematic approach for the implementation of evolving, customisable and combined language families.

**The Principle of Integrative Composition** The principle of composition aims at fostering the composition of languages from self-contained, reusable and interchangeable language components. A compositional LFE approach is considered to improve the evolution of language families by enforcing explicit interfaces between individual components, to help language customisation by enabling flexible integration and reuse of components, and to enhance language combination by providing means
to combine multiple languages for multi-dimensional system specifications. We argue, that the need for a potent module integration technique motivates a grey-box composition system as introduced in [Aßmann 2003]. It combines means to define composition interfaces as found in black-box component models with the power of invasive component integration [Henriksson 2007].

Our vision is to implement a LFE approach based on these principles. This thesis documents our attempt in achieving this vision. It is structured as follows.

**Review of State-of-the-art (Chapter 2)**

To evaluate the capabilities and identify shortcomings of existing language engineering approaches wrt. LFE, we provide a comprehensive review of state-of-the-art. In Chapter 2, we, first, review existing processes for language engineering wrt. their applicability in LFE. Second, we investigate the state-of-the-art in language implementation techniques. Finally, we derive detailed requirements for a comprehensive LFE approach to address the shortcomings found for state-of-the-art techniques and processes.

**Feature-Oriented Language Family Engineering (Chapter 3)**

To implement the principle of variability, we suggest to adapt techniques and processes found in SPLE [Pohl 2005] for LFE. SPLE provides means for the systematic analysis and implementation of commonalities and variability among families of related software systems. It aims at enhancing reuse while providing customised software for particular applications. The SPLE approach Feature-Oriented Software De-
1.3. Language Family Engineering with Features and Role-Based Composition

Development (FOSD) [Apel 2009] employs so-called feature models for variability analysis, management and realisation in software systems.

**Hypothesis I:** It is feasible to construct a feature-oriented approach for LFE. Such an approach provides enhanced means for analysing, documenting, and managing variability in language families.

To prove this hypothesis, we adapt FOSD to introduce a continuous reuse- and variability-oriented process for **Feature-Oriented LFE**.

The OWL language family is used to investigate the applicability, benefits and challenges of LFE with features. OWL introduces a language for encoding knowledge about specific domains and reasoning rules that allow for deriving implicit knowledge. The manifold of domains that ontology languages are applied to, requires language variants with different expressiveness and reasoning efficiency. This led to an ad-hoc language family. The feature-oriented management of variability for OWL is motivated by the need for a more flexible adaptation of ontology languages for different use cases [Stuckenschmidt 2001], a more efficient configuration of custom reasoning infrastructures in ontology evaluation [Thomas 2010], and the automatic derivation of language tooling (e.g., dedicated parsers, printers, editors, and reasoners) for the practical application of custom OWL variants.

**Role-Based Language Composition (Chapter 4)**

To implement the principle of integrative composition, we suggest the introduction of roles for abstract syntax metamodelling. As discussed in [Reenskaug 1996], a “[...] role model is an object-oriented model of an object structure and represents a bounded part of the real world or of an interesting concept. It models patterns of collaborating objects as a similar structure of collaboration roles”. We assume the abstract syntax metamodel of a language to be composed of a number of such role-based object structures, where each structure encapsulates the realisation of a particular language feature.

**Hypothesis II:** It is feasible to construct an approach for role-based, integrative composition of language components at the level of abstract syntax. Such an approach enhances the modularity properties of language realisation and provides means to integrate arbitrary technical spaces involved in language realisation.

To prove this hypothesis, we design and implement a role-based language composition system by extending current metamodelling languages with roles. We also investigate the realisation of integrative, role-based composition techniques for concrete syntax and semantics technical spaces.

The OCL language family is used to evaluate the applicability, benefits and challenges of role-based language composition in a practical context. OCL was released in 2000 as a constraint language for models defined with the UML. The standardisation of its textual syntax and its set-based semantics promoted a wide adoption of OCL in academia and
industry. Although not planned from the beginning, the evolution and the demand for customisation and combination made the OCL a language family by accident. This motivated a number of extensible implementations of OCL [Akehurst 2004a, Bräuer 2007, MDT 2011, Kolovos 2008a, Wilke 2010]. Role-based language composition is meant to further enhance OCL modularisation [Wende 2010]. It is expected to reduce complexity in language evolution [Akehurst 2004a], to prepare the integration of new language extensions in a systematic manner [Akehurst 2004a, Wende 2010], and to help the combination of OCL with other languages.

Integrative Semantics Composition with Coloured Petri Nets (Chapter 5)

Chapter 4 introduces a foundation for the static aspects of role-based language composition by describing a normalisation of roles and role composition to an object-oriented metamodel. However, the application of role-based language composition does also affect dynamic aspects of language realisation. To give a formal foundation for dynamics of role composition, a semantics formalism is required that enables integrative semantics composition and preserves the modularity properties of roles and role composition. We argue that Coloured Petri Nets (CPNs) as introduced in [Jensen 1987, Jensen 2007] provide a formal foundation for such approach. They introduce rich modularity concepts like port places and substitution transitions and provide an integrative, grey-box composition technique for dynamic semantics.

**Hypothesis III:** It is feasible to extend role-based language composition with language semantics composition based on CPNs. Such extension provides a foundation and an applicable approach for integrative, semantics composition in LFE.

In Chapter 5, we prove Hypothesis III by extending our role-based language composition system with CPN-based semantics composition. The applicability and benefits of the extended approach are evaluated in a case study.

Combination of Feature-Oriented LFE and Integrative, Role-Based Syntax and Semantics Composition (Chapter 5)

As depicted in Fig. 1.3, feature-oriented LFE and integrative, role-based composition of syntax and semantics can be combined to a comprehensive approach for LFE. LFE with features and role-based composition, therefore, specialises the two common phases of SPLE: domain engineering and application engineering to **language family engineering** and **language variant engineering**, respectively.

During the phase of language family engineering, a language family is decomposed in a set of language features, i.e., distinctive characteristics of language functionality. The objective of this step is the identification of reusable and variable language features and the specification of their interdependencies. Afterwards, the found variability is implemented in the syntax and semantics of role-based language components.
1.3. Language Family Engineering with Features and Role-Based Composition

In the phase of language variant engineering, a custom language variant is derived from the language family. Therefore, the concrete features required for the language variant are specified. Finally, the variant derivation step applies role-based language composition to derive an integrated implementation of syntax and semantics of a custom language variant.

*Hypothesis IV:* It is feasible to combine features and role-based composition. Such a combination implements a comprehensive approach to LFE with further benefits.

In Chapter 5, we prove this hypothesis using the SumUp language family. SumUp introduces a set of DSLs for the specification of mathematical equations. We evaluate the benefits of the combined approach and discuss open challenges for LFE.

**Conclusion and Outlook (Chapter 6)**

In summary, this thesis provides the following contributions:

0. An evaluation of existing language engineering approaches for their applicability in LFE, *(Foundations)*

I. A novel, feature-oriented process for LFE, *(Hypothesis I)*

II. A novel, role-based approach for integrative language composition, *(Hypothesis II)*

III. A novel, CPN-based approach for the integrative composition of dynamic language semantics *(Hypothesis III)*, and

IV. A qualitative evaluation of feature-oriented LFE with integrative, role-based syntax and semantics composition using practical case studies *(Hypothesis IV)*.

In Chapter 6 of this thesis, we conclude the details of these conceptual, technical and qualitative contributions and discuss topics for future work.
This chapter provides a comprehensive review and evaluation of current methods and techniques for language engineering for their applicability in LFE. In Section 2.1, we discuss existing processes and methodologies for systematic language engineering, evaluate their applicability, and derive concrete requirements for an enhanced LFE process. In Section 2.2, we analyse the technical spaces involved in language engineering, investigate the state-of-the-art for challenges wrt. the realisation of language families, and conclude explicit requirements for an enhanced LFE realisation technique.

2.1. Language Engineering Processes

Traditionally, approaches to implement languages originated from the technical spaces involved in language design and implementation. They do not introduce a particular process or methodology, but solely focus on tasks that are required to provide these artefacts. The lack of a systematic approach towards language engineering was identified by various authors [Klint 2005, Mernik 2005]. In a comprehensive survey of language engineering techniques, Mernik et al. [Mernik 2005] stress the importance of a analysis and design phase preceding the implementation of DSLs. In the following, we review the survey of Mernik et al. and recent work on language engineering processes wrt. the typical engineering phases analysis, design, and implementation.

2.1.1. Analysis Phase

During the analysis phase of a language engineering process, it is important to identify the basic language concepts and their interrelations in the domain the language is meant to be applied to. A conceptualisation of a domain can be either descriptive or prescriptive [Seidewitz 2003], i.e., either describe all that is known for the domain or give a detailed, prescriptive specification of how a domain is to be implemented. We consider the language description resulting from the analysis phase a descriptive conceptualisation. Mernik et al. suggest the application of descriptive methods from ontology
Chapter 2. Review of Current Language Engineering

engineering [Falbo 2002, Guizzardi 2002] and from domain analysis [Kang 1990] to enable a first identification of language concepts and their interrelations. The application of domain analysis is also suggested in [Thibault 1999]. In accordance to Deursen and Klint, the “prerequisite for the design of a DSL is a detailed analysis and structuring of the application domain” [van Deursen 2002] that may be provided by feature models. Deursen and Klint also provide a generative approach to derive a prescriptive UML class diagram from a given feature model. Mernik et al. conclude that, besides these first attempts, clear guidelines and a systematic approach for bridging the descriptive analysis and subsequent prescriptive phases of language engineering are missing.

2.1.2. Design Phase

The objective of the language design phase is to develop a prescriptive specification that describes the syntax and semantics of the language. For this phase, Mernik et al. distinguish formal and informal specification methods. Informal methods basically use natural language and exemplary DSL programs. This is considered a straightforward but also very vague way of language design. Informal specifications can neither be explicitly related to the artefacts produced in the analysis phase, nor do they provide means for validation of the language specification. In contrast, formal methods employ formalisms to prescribe abstract and concrete language syntax and semantics. Formal descriptions are considered a solid foundation for an automatic validation and implementation of DSLs using analysis and generation techniques, respectively. As the capabilities required for the specification of concrete syntax, abstract syntax and semantics are very different, specific formalisms for each asset exist. For an in-depth survey of the state-of-the-art formalisms and tools, we refer to Section 2.2.

2.1.3. Implementation Phase

The implementation phase has the objective to contribute an executable implementation of a language and corresponding language tooling. Mernik et al. and others (e.g., [Fowler 2005]) distinguish internal and external DSL implementations. Internal languages are implemented by reusing and optionally extending an existing host GPL to express domain-specific concepts. External implementation approaches introduce a new separate language implementation. For evaluating external languages, external implementation approaches typically employ transformations (compilation or interpretation) that map DSL constructs to GPLs.

In accordance to [Mernik 2005, Fowler 2005], the main benefits of internal implementation approaches are that existing tools and infrastructure of the host GPL can be reused and developers do not need to learn a new, specific syntax. On the other hand, the syntax is restricted and may not be optimal wrt. to the domain conceptualisation. In contrast, external implementation approaches can provide custom syntax and means for domain-specific analysis, validation, error reporting of language expressions. In general, the effort for external implementation approaches is considered higher [Mernik 2005, Fowler 2005]. However, we experienced [Heidenreich 2009a] that this effort is strongly reduced by using
2.1. Language Engineering Processes

generative tools to derive external language implementations and language tooling. Given that a formal specification approach was used during the design phase, such tools enable an automatic or semi-automatic derivation of an external language implementation. For an in-depth discussion of the available automation, we refer to Section 2.2.

2.1.4. Applicability in Language Family Engineering

This review of current state-of-the-art in language engineering processes shows their strong focus on language design and implementation. For these phases, various formalism, techniques and tools are available. Unfortunately, the analysis phase and its artefacts are not or just loosely coupled to design and implementation. Furthermore, language engineering processes are designed in a rather linear top-down approach.

We consider both issues a major limitation in particularly wrt. LFE. For evolution, customisation, and combination in language families, it is vital to keep track of language features identified during the analysis phase, their interdependencies, and their relation to design and implementation artefacts. An in-depth variability analysis and specification for language requirements is vital to foster reuse in language evolution, customisation, and combination. Such variability analysis should cover language variability dimensions beyond language expressiveness. The suggested application of methods from domain analysis might be a good starting point. In addition, we argue for a methodical and technical approach for continuously coupling variability analysis with the phases language design and implementation. This is meant to enable a requirements-driven, automated derivation of customised language variants. Such derivation process should exploit the initial variability specification to provide language engineers with guidance in language customisation. Furthermore, LFE requires an engineering process that enables an iterative and continued refinement, extension and maintenance of language families.

2.1.5. Requirements for an Enhanced LFE Process

From the issues discussed above, we derive explicit requirements for an enhanced development process (DP) for language families:

**DP 10: Variability Analysis and Specification** Systematic language evolution, customisation and combination requires a development process to provide means for variability analysis and specification.

**DP 20: Different Abstraction and Granularity** Language evolution, customisation and combination happens at different levels of abstraction and granularity. The development process should be flexible and applicable to these different levels.

**DP 30: Continuous Process Application** Described variability needs to be coupled to language design and implementation. The development process should, therefore, provide technical means to explicitly interconnect variability specification and implementation.
Chapter 2. Review of Current Language Engineering

DP 40: Technology Agnostic Language design and implementation involves different and numerous formalisms and techniques. The development process should be able to cope with such technical diversity.

DP 50: Guided Language Customisation The customisation of variants in LFE should exploit the descriptive variability specification to guide language engineers.

DP 60: Automated Variant Derivation The derivation of custom variants in LFE for a given variant specification should be automated given the connection of variability specification and implementation.

DP 70: Iterative Refinement Evolution, customisation and combination in LFE are considered iterative activities. The development process needs to support an iterative refinement and extension for variability specification and variability realisation.

Chapter 3 describes a feature-oriented process for LFE that is meant to address these requirements.

2.2. Technical Approaches in Language Engineering

The formalisms and tools used for language design and implementation are a determining factor for the applicability of a language engineering approach for the implementation of language families. In this section, we review the most common formalisms and tools currently available.

To distinguish different categories of formalisms and tools for language engineering, we follow the traditional distinction [Aho 2006, Selic 2010] of language constituents into concrete syntax, abstract syntax, and semantics. As discussed by Favre [Favre 2007], this distinction is founded by the two most important functions of languages: to communicate and to reason about things. While the concrete syntax of a language focuses on the communication function, language semantics serves the reasoning function. Between those, abstract syntax provides means to store the information represented by concrete syntax and evaluated by semantics in a normalised format [Aho 2006, Harel 2000].

For each of the three constituents, there are a number of language engineering approaches based on particular formalisms and implemented in particular tools. Each formalism, the according concepts, the related knowledge, the provided tools, and their functionality form a particular technical space [Bézivin 2005]. The state-of-the art in technical spaces for each constituent is discussed in the Sections 2.2.1–2.2.3.

2.2.1. Specification of Abstract Syntax

Historically, the specification of abstract syntax was strongly related to the specification of concrete syntax. Abstract syntax trees were considered the result of syntactic analysis of language expressions written in concrete syntax. Consequently, the abstract syntax of
Table 2.1.: Formalisms and technical spaces for metamodelling.

<table>
<thead>
<tr>
<th>Formalism</th>
<th>Technical Spaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object-Oriented</td>
<td>UML, MOF, EMOF, CMOF, Ecore, KM3, Kermeta, Netbeans MDR, MOFLON, JetBrains MPS, Metaedit+ GOPPRR, Microsoft Oslo</td>
</tr>
<tr>
<td>Typed, Attributed Graphs</td>
<td>GXL, KOGGE, AGG, PROGRES, GrUML</td>
</tr>
<tr>
<td>Ontologies</td>
<td>OWLizer, OntoDSL, OWLText, Hybrid-MDSD</td>
</tr>
</tbody>
</table>

a language was understood as plain data structure that was instantiated in the semantic actions annotated to its concrete syntax specification [Aho 2006].

The emergence of MDSD, MDE, and DSLs as well as the accompanying interest in metamodelling strengthened the significance and perception of abstract syntax (or language metamodels) as first class entity in language design and implementation [Selic 2010]. This led to a number of technical spaces for metamodelling that enable the definition of language metamodels, provide generic tools to edit, validate or store models, and support the generation of tooling for models. Table 2.1 concludes the formalisms that constitute a foundation for metamodelling technical spaces and names concrete technical spaces implementing each formalism. In the following, we discuss these technical spaces in detail and evaluate their applicability for a modular specification of abstract syntax in language families.

**Specification Formalisms for Abstract Syntax**

**Object-Oriented Metamodelling** Object-oriented approaches to metamodelling are based on the foundations of object orientation and typically involve a three-level metalayer architecture. Metamodelling languages provide concepts like classes, inheritance, associations and attributes, as typically found in object-oriented programming (e.g., Java, C#) and object-oriented modelling languages (e.g., UML [UML 2009]). These can be used to describe a language’s abstract syntax.

The UML [UML 2009] is a very widespread language for conceptual modelling. It provides a family of modelling languages to model software systems. A lightweight approach to metamodelling with UML is to use the profile mechanism [Fuentes-Fernández 2004]. It provides means to define custom DSLs based on UML diagram notations [Abouzahra 2005, Selic 2007, Wimmer 2007]. This enables the reuse of existing UML tooling for various domains. However, the approach is also limited to DSLs whose abstract syntax and semantics is compatible to the structure, representation and semantics of standard UML [Selic 2007]. In addition, the UML specification introduces the so-called infrastructure library. It provides all metamodelling concepts that are used to define the UML languages. This infrastructure library can be consid-
Chapter 2. Review of Current Language Engineering

... an object-oriented metamodelling language [UML 2009] and provides a conceptual foundation for OMG metamodelling languages like MOF [MOF 2006].

The MOF specification is an attempt of the Object Management Group (OMG) to standardise object-oriented metamodelling. It defines a three-layered architecture for metamodelling (cf. Fig. 2.1) that is based on the MOF metamodelling language. The MOF metamodelling language is located at the topmost layer (M3). It can be used to define (instantiate) language metamodels, e.g., the metamodel of a statechart language (M2), or the metamodelling language itself (M3), using an object-oriented abstraction. A particular language metamodel is again instantiated to represent concrete models (M1) to, for instance, describe a concrete state chart.

The OMG specification distinguishes two variants of MOF: EMOF and CMOF. EMOF contributes the minimal core of metamodelling concepts. CMOF extends EMOF for more sophisticated metamodelling [MOF 2006]. A concrete implementation of EMOF is the Eclipse Modelling Framework (EMF) [Steinberg 2008]. As depicted in Fig. 2.1, EMF contributes the Ecore metamodelling language (M3) — an adaptation of EMOF — that allows to specify language metamodels (M2). A metamodel is defined using an EPackage that contains a number of classes (EClassifier, EClass) capturing a language’s domain-specific modelling concepts. Classes are arranged in inheritance hierarchies (eSuperTypes), can be connected using associations (EReference), and can own attributes (EAttribute) to store values of a primitive data type (EDatatype). Furthermore, it is possible to contribute simple structural restrictions, e.g., association or attribute cardinalities (lowerBound and upperBound attributes for EAttributes and EReferences). Such constraints affect all models which instantiate elements of the metamodel (M1). The EMF contributes graphical and tree-based editors that ease the definition of metamodels in Ecore. Furthermore, a code generator is provided that derives a Java-based metamodel implementation and a tree-based editor for instantiating metamodels.

The advanced tool support, the widespread use, and huge community makes EMF a prominent technical space for metamodelling used in academia and industry. In addition to Ecore, there are also other metamodelling languages integrated with the EMF.

- The KM3 Metamodelling Language (KM3) [Jouault 2006] was defined as a simplification of MOF and contributes a dedicated textual notation to specify metamodels. Compared to the graphical and tree-based editors provided for Ecore this syntax eases the definition and adaptation of metamodel specifications. In terms of expressiveness, KM3 can be compared with Ecore.

- The metamodelling language Kermeta [Drey 2010] is also compatible with Ecore and MOF. Kermeta contributes means to imperatively specify metamodel semantics [Muller 2005]. Furthermore, Kermata enables the definition of metamodel aspects which allow for extending existing metamodels in a crosscutting way.
2.2. Technical Approaches in Language Engineering

Figure 2.1.: EMF implementation of MOF three-layered metamodeling architecture.
There are also open-source implementations of the MOF standard not coupled to EMF, they contribute their own technical space:

- The NetBeans MDR [Matula 2008] was developed by Sun to provide a repository for the Netbeans Integrated Development Environment (IDE). It’s development seems to be discontinued since 2009.

- MOFLON [Amelunxen 2008] is a MOF-compatible environment for the implementation of MDSD tools and tool adapters. MOFLON puts a strong emphasis on complete standard compliance with MOF version 2.0. Especially, the implementation of advanced MOF modularisation concepts like package merges makes MOFLON an interesting candidate for modularity in abstract syntax specification.

Besides these standardised or freely available implementations of object-oriented metamodelling languages, there is a number of proprietary technical spaces for object-oriented metamodelling that are used in industry:

- The JetBrains Meta Programming System (MPS) provides a language to define abstract syntax structures. It basically supports metamodelling concepts similar to Ecore (classes, properties, references, cardinalities) [Dmitriev 2005]. The MPS tool provides a special editing approach that uses a tabular presentation of models. This eases the extension of the concrete syntax of languages [Völter 2011]. However, the metamodelling language of MPS has no particular means to enhance modularity for abstract syntax specifications.

- MetaEdit+[plus] is a platform for the development and implementation of domain-specific modelling tools [Kelly 1996]. It is based on the GOPP RR metamodelling language [Kelly 2008] that uses concepts like graphs, objects, properties, relationships, and roles. While named differently, the GOPP RR concepts correspond to the metamodelling concepts commonly found in MOF-based languages [Kern 2008]. GOPP RR introduces a role concept to denote the ends of relationships. This usage differs from the term role as used in this thesis. The most distinctive feature of MetaEdit+[plus] is that the GOPP RR concepts are universally used to represent both metamodels and models. Hence, MetaEdit+[plus] does not rely on the strict meta-layers introduced in MOF.

- Microsoft Oslo [OSL 2009] is a metamodelling and modelling platform introduced for the .NET environment. It provides the MSchema language to define types and properties for metamodels, the MGraph language for model representation, and the MGrammar language for the definition of custom concrete syntax for model representation. In terms of expressiveness, Oslo can be compared to MOF-based metamodelling languages [Bruneliere 2010, Hillner 2010]. Recently, Oslo was moved to the SQL Server project and is now named SQL Server Modeling Services. Its continuation in the context of MDSD is uncertain.
2.2. Technical Approaches in Language Engineering

**Figure 2.2.:** GXL approach for metamodelling for graph-based languages.

Metamodelling for Graph-based Languages A second category of metamodelling approaches is based on the formalism of graph theory. Graphs are considered a very general representation for information in tool repositories. Especially typed, attributed graphs are experienced useful in metamodelling. A typed attributed graph consists of a set of vertices connected by a set of edges. Each vertex and each edge is assigned a certain type. Furthermore, edges and vertices can hold attributes with values of a given data type. Using these concepts arbitrary models can be represented [Ebert 1995].

To specify the set of available vertex, edge and attribute types and, thus, valid graphs for a given language, again some kind of metamodelling approach is required. Fig. 2.2 illustrates the kind of layered metamodelling typically found for graph-based languages. The illustration relates to the Graph eXchange Language (GXL) approach [Winter 2001]. GXL is meant to provide an exchange format for graph-based tools. It is based on typed, attributed graphs and uses eXtensible Markup Language (XML) to exchange graph-based models and metamodels. The number of layers found in GXL resembles the 3-layered architecture found for MOF-based languages. On the topmost layer (M3), we find a metamodelling language—the Graph Metaschema—that is used to define metamodels of graph-based languages. It uses concepts of typed, attributed, graphs (GraphClass,
NodeClass, EdgeClass, and AttributeClass) for metamodelling. GXL uses a UML-based syntax to render metamodels. The language metamodel, or Type Graph as its often called for graph-based languages, is specified at (M2) using the concepts in the metaschema. At the same metamodeling layer a generic Graph Model is located. It defines the concepts that are available to specify system models in a graph-based representation. It is also based on concepts of typed, attributed graphs (Graph, Edge, Node, Attribute). At the lowest layer (M1), these generic concepts are instantiated to represent a concrete system model. The concrete node, edge and attribute types in this graph refer to types defined in the Type Graph and need to conform to their structural relationships and constraints.

A similar layering and comparable concepts can be found in most technical spaces for graph-based languages. Prominent technical spaces for metamodelling that build on the formalism of graph theory are introduced in the following.

- In [Ebert 1995], the authors use Entity-Relationship (ER) diagrams for the specification of type graphs (metamodels). Entities define vertex types, relationships define edge types, and ER attributes describe attributes for edges or vertices. The relations encoded in the ER diagram add further structural constraints that restrict valid models (e.g., what kind of vertices can be connected with what kind of edges). The approach was implemented in the KOGGE metamodelling environment [Ebert 1996].

- The PROgrammed Graph REwriting Systems (PROGRES) approach introduces a graph-based language and a programming environment for the implementation of software systems and software engineering tools [Schürr 1998]. It contributes a proprietary object-oriented metamodelling language to define type graphs. Its expressiveness can be compared to common metamodelling approaches like EMOF. Furthermore, the PROGRES language provides sophisticated means for the declarative specification of derived attributes and edges, a rule-based language to define graph queries or graph rewrite rules, and enables an imperative specification of graph transformations.

- The approach presented in [de Lara 2007] contributes a formalisation for inheritance in metamodelling for graph-based languages. It is again based on the foundations of typed attributed graphs [Ehrig 2004]. For type graphs, abstract vertex types and inheritance for vertex types are introduced. The resulting metamodelling language combines the expressiveness of object-oriented metamodelling approaches and the strong theoretical foundation provided by graph theory. The approach is implemented in the AGG environment that is applied to various MDSD scenarios [Taentzer 2004].

- The Graph Unified Modelling Language (GrUML) is introduced in [Ebert 2008]. It is a graph metamodelling language that was derived from the UML. GrUML contains (only) those metamodelling concepts of the UML which can be interpreted for graph-based languages. Classes are used to describe vertex types, associations
to describe edge types, specialisation and generalisation enable the definition of type hierarchies, and UML attributes are used to define the attributes of vertices or edge types, respectively.

The relation of metamodeling approaches for graph-based languages to ER, UML, and EMOF shows that they share the expressiveness of object-oriented metamodeling approaches. To enable extensions and reuse during metamodeling, the discussed approaches support inheritance for vertex and edge types.

**Ontology-based Metamodelling** Ontologies (as used in computer science) can be applied to formally describe knowledge about a domain. A broadly applied approach to formalise ontologies is Description Logic (DL) [Baader 2003]. OWL is a famous family of ontology languages based on DL. Given a concrete OWL ontology, reasoning can be applied to pose queries, check its consistency, and to derive implicit knowledge. This enables a broad application of ontologies to enhance expert systems, guidance systems, databases, or search engines by exploiting semantic metadata. Furthermore, ontologies are envisioned to contribute semantics support that is currently missing for standard metamodeling languages [Atkinson 2003, Happel 2006]. This vision motivates a number of technical spaces for metamodeling based on ontology technology.

Ontology languages typically model both domains and systems in that domain using the same language. As depicted in Fig. 2.3, it consists of only two distinct layers. The upper layer defines the ontology language (e.g., OWL) that is used to model both domains and systems and corresponds to (M3). It contributes concepts (OWLClass, DataProperty, ObjectProperty, Description) to describe intensional knowledge, i.e., the terminology to structure knowledge in the application domain, in the TBox (Terminology box). OWL also provides rich concepts to classify inheritance hierarchies. A given OWLClass can be specified to be a subclass of a Conjunction or Disjunction of other OWLClasses. Such classes can be referenced directly (ClassAtomic), or defined by complex PropertyRestrictions on object or data properties. All extensional knowledge, i.e., the elements of a particular system in the domain, is described using concepts like Individual, DataPropertyFact, ObjectPropertyFact, Literal. They built the so-called ABox (Assertional box) of the ontology.

In the lower layer, these language concepts are instantiated to represent both a domain and a system in the respective domain in the ontology TBox and ABox, respectively. In contrast to the previous metamodeling approaches, metamodel (M2) and model (M1) are not located on different layers, but reside in the same one. Furthermore, they are not connected by an instance of relationship, but by simple references (type reference).

A key challenge for employing ontologies in metamodeling is the mismatch between Closed World Assumption (CWA) and Open World Assumption (OWA). Metamodels and models provide a prescriptive representation of a particular domain under the assumption that the given information is closed, i.e., everything that is not explicitly specified in the model is assumed to be false. In contrast, ontologies provide a descriptive representation of a domain. They assume that the information given about the system under study is open. This means that everything that is not explicitly specified may be considered true or false.
Chapter 2. Review of Current Language Engineering

Research on Semantic Web Enabled Software Engineering (SWESE) [Happel 2006] tries to bridge this mismatch [Gaševic 2004] and investigates ways to exploit ontology reasoning [Atkinson 2003] services in metamodelling. The bridging approaches typically rely on the conceptual correspondence of ontology TBoxes with metamodels and ABoxes with models. In the following, we conclude a number of approaches for ontology-based metamodelling.

- The HybridMDSD approach discussed in [Bräuer 2008] employs ontologies to combine different languages. Therefore, HybridMDSD defines an new upper ontology for software systems. This upper ontology is used to build an ontology containing information about relationships between elements in different languages. To enable
2.2. Technical Approaches in Language Engineering

reasoning under CWA, HybridMDSD automatically adds closing axioms to the ontology. Finally, standard reasoning can be applied to ensure the completeness and consistency of models specified using the integrated languages.

- The OWLizing approach we introduced in [Walter 2010] discusses integration and transformation bridges to exploit ontologies in metamodeling. OWLizer implements an integration bridge for Ecore metamodels and OWL language elements used to specify ontology TBoxes. This integration provides means to annotate metamodels with ontology-based constraints. Second, OWLizer provides a transformation bridge that transforms Ecore-based models to the ontology ABox and automatically closes the ontology. This enables the application of conventional reasoners to evaluate ontology-based constraints and validate models using CWA.

- The OntoDSL approach described in [Walter 2009] integrates the OWLizer to implement a language engineering approach. OWL-based reasoning is employed for defining basic metamodel constraints and to realise constraint evaluation, inconsistency explanation, or guidance in modelling. This demonstrates the applicability and benefits of ontologies in metamodeling.

- The OWLTText approach [OTe 2011] combines OWLizer, Ecore metamodels, and the parser generator EMFText [Heidenreich 2009a]. A tight integration of these approaches contributes a fully generative approach to derive advanced, syntax- and semantics-aware editors for textual DSLs. It demonstrates the feasibility and benefits for integrating multiple technical spaces in metamodeling.

Compared to other metamodeling approaches ontologies excel in categorising concepts in hierarchies. They provide advanced means to model rich inheritance structures. Regarding other metamodeling concepts, the expressiveness of ontologies can be compared to object-oriented and graph-based approaches.

Applicability in LFE

We aim at reducing the effort for language realisation by an enhanced reuse of parts of language specifications and implementation artefacts within language families. Research in component-based software engineering states that such reuse requires a sensible design of self-contained, reusable and interchangeable components and a dedicated system for component composition [Aßmann 2003, Szyperski 2011].

Due to its pivotal role in representing language expressions derived from concrete syntax and evaluated by language semantics, abstract syntax is considered the essential artefact for modularity of language components and language composition systems. For the independence and reusability of components, we consider the following modularity properties important:

Information Hiding The principle of information hiding [Parnas 1972] is meant to ensure the encapsulation of the implementation details of a given component. This avoids that other components rely on particularities of the implementation details
and need to be changed whenever these details change. Information hiding, thus, enhances the stability of the overall component system if individual components evolve.

**Component Interfaces** Component interfaces are strongly related to the principle of information hiding. The interface of a component is meant to enable the communication with other components. It should only expose the parts of the component that are necessary to realise (required interface) and access (provided interface) its functionality. Component interfaces are vital for component customisation and extension.

**Loose Coupling** The principle of loose coupling is meant to avoid direct interdependencies of components. Components that are independent of other components are easier to evolve, reuse and customise.

**Flexible Integration** To interconnect components using their interfaces, flexible and expressive means for component integration are needed. Such integration is meant to resolve interoperability issues for components and to bridge incompatibilities in their interfaces. As not all evolution, future customisation, and combination scenarios can be anticipated, an integrative, grey-box composition technique that enables invasive component integration is needed.

In the following, we evaluate the above introduced metamodelling approaches wrt. these modularity properties. To enable a consistent discussion, Table 2.2 maps the metamodelling concepts used in the different metamodelling formalisms spaces to a common terminology.

For the metamodelling formalisms discussed above, we distinguish three basic means to achieve modularity and reuse in metamodel specifications: *module import, module merging* and *aspect orientation*. Some means are provided by every formalism, some means are only implemented in particular approaches.

<table>
<thead>
<tr>
<th>Common</th>
<th>Object-oriented</th>
<th>Graphs</th>
<th>Ontologies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module</td>
<td>Package, Schema</td>
<td>GraphClass, Schema Class, Type, Object, Role</td>
<td>Ontology (TBox) Class, Concept</td>
</tr>
<tr>
<td>Concept</td>
<td>Class, Type, Object, Role</td>
<td>NodeClass, VertexClass, Type</td>
<td></td>
</tr>
<tr>
<td>Reference</td>
<td>Reference, Association, Relationship</td>
<td>EdgeClass</td>
<td>ObjectProperty, Property</td>
</tr>
<tr>
<td>Attribute</td>
<td>Attribute, Property</td>
<td>AttributeClass</td>
<td>DataProperty, Property Property</td>
</tr>
<tr>
<td>Subclassing</td>
<td>Inheritance</td>
<td>Inheritance</td>
<td>Inheritance</td>
</tr>
</tbody>
</table>

Table 2.2.: Alignment of terminology for metamodelling formalisms and technical spaces.
2.2. Technical Approaches in Language Engineering

Inheritance Subtyping

<table>
<thead>
<tr>
<th>Intention</th>
<th>Inheritance</th>
<th>Subtyping</th>
</tr>
</thead>
<tbody>
<tr>
<td>share commonalities</td>
<td>(attributes, references)</td>
<td>reuse generic behaviour by anticipated variation</td>
</tr>
<tr>
<td>Information Hiding</td>
<td>no hiding</td>
<td>hiding of private information</td>
</tr>
<tr>
<td>Reuse Granularity</td>
<td>within components</td>
<td>over component borders</td>
</tr>
</tbody>
</table>

Table 2.3.: Subclassing ambiguity: subtyping vs. inheritance.

Module Import. The application of *modules and module import* is the most common practice to implement modularity in metamodeling. Modules are used to encapsulate sets of related concepts. To construct languages from reusable modules, module import and direct references or subclassing between concepts in different modules is used. This is considered harmful for several reasons: First, the connection and integration of language modules is expressed as part of the module specification. As the connection specification depends on the combination of modules that are integrated, single modules are more difficult to reuse. The inheriting or referencing module needs to know the module it inherits from or references to. The modules are strongly coupled and, thus, not maintainable and usable individually.

In addition, subclassing combines the characteristics of *subtyping* and *inheritance*. This ambiguity is discussed very controversially [Bracha 1992, Taivalsaari 1996]. While subtyping expresses that objects of a subtype can be used wherever their supertype is expected (i.e., in references defined between the types), inheritance describes that concrete features (i.e., attributes, references and operations) defined by an (abstract) superclass are propagated to all subclasses. As concluded in Table 2.3, this mixes two intentions in metamodel design: reuse of generic behaviour by anticipated variation (subtyping) and reuse by sharing of common properties (inheritance). The combination of both in subclassing breaks the principle of information hiding between language modules since inherited attributes and references can be accessed and altered in arbitrary ways. As a consequence, language modules provide no means to define explicit interfaces. We propose subtyping as the mechanism of choice to interconnect language modules while inheritance should be restricted as reuse formalism within modules. A second aspect discouraging subclassing as the sole reuse mechanism for modular language engineering is that it operates on the granularity of single concepts. We, however, consider reusable language modules to consist of a set of related metamodel concepts.

Module Merging. The concept of *module merging* is introduced in [MOF 2006] as another modularisation and reuse mechanism especially for object-oriented metamodels. It combines all concepts of a merged module with the merging module. In addition, the concepts in the merging module are extended by all references and attributes of similarly named concepts in the merged module. A similar mechanism is used when an ontology imports other ontologies and concepts in both ontologies share the same name. Although
Chapter 2. Review of Current Language Engineering

the definition of merge semantics in [MOF 2006] lacks a precise formalisation [Zito 2006], module merge is applied in language composition [Emerson 2006] and to build the UML language family.

The mechanism of module merging allows a flexible extension of language modules. However, merging also breaks the independence and reusability of language modules for two reasons. First, there is no way to ensure information hiding between merged modules. Second, modules have to be compatible. The concepts to be merged have to share the same name and all references, containment structures, visibilities, cardinalities, or constraints of the merged packages need to be compatible [Zito 2006]. This requires a global alignment of all modules and disables any reuse or invasive module integration among different language families. As the coupling of language modules is only implicitly defined by name correspondence, we consider package merge hard to comprehend by language developers. Merged languages may suffer from unexpected side-effects [Emerson 2006] and easily break when modules are refactored and concepts are renamed.

Aspect-orientation. The application of aspect-orientation is meant to provide another alternative to realise modular metamodels. The approach is, for example, implemented in the Kermeta [Drey 2010] language. It enables the specification of aspect modules that extend an existing metamodel module with new concepts or add references, attributes, operations, or constraints to existing concepts. Aspect modules can, thus, be used to modularise and integrate individual aspects of a language implementation. Aspects can not only extend modules, but also override their static or dynamic semantics. Furthermore, concepts in an aspect module that are meant to be merged with concepts from the extended module are marked explicitly. This enhances the comprehensibility and predictability of the merge results for language developers. Considering the independence of the language modules the approach implemented in Kermeta shares the drawbacks of module merging: Aspects break information hiding and module integration needs to be specified as part of the aspect module. Thus, each aspect module is strongly coupled to the module it is meant to affect. It has to refer to classes of the extended module and relies on its structure and semantics. This restraints module independence and reusability. Aspects provide an integrative, white-box composition technique, that lacks means for describing explicit composition interfaces.

From this review of state-of-the-art formalisms for abstract syntax specification, we see that they provide rich means for specifying a language’s abstract syntax. Ontology-based approaches provide an expressive metamodelling language to define sophisticated concept hierarchies. However, they suffer from the discrepancy between OWA and CWA. Graph-based languages provide means for prescriptive specification and benefit from their formal foundation. Although GXL attempts to integrate the plethora of proprietary graph-based approaches, it does not achieve the same standardisation level and industrial application as metamodelling approaches like MOF, EMOF, or Ecore.

Our analysis also reveals that the discussed approaches do not provide sufficient means to achieve the desired modularity properties for language components. These observa-
tions motivate an application and extension of current metamodelling approaches and their rich tooling to introduce a compositional metamodelling approach. In contrast to current modularisations, we envision language components that are defined independent of each other and can later be composed to an integrated language or language family. Well-defined language components with explicit interfaces and an integrative composition technique are required to provide the modularity properties introduced above.

2.2.2. Specification of Concrete Syntax

The concrete syntax of languages describes how language expressions are presented to the user. The possible representations are manifold. The specification of textual concrete syntax is one of the classical disciplines of (theoretical) computer science. With the rise of graphical modelling languages, diagrammatic syntax gained importance, and even the tree-based model editors found in current modelling tools (e.g., EMF) can be considered a concrete syntax.

All concrete syntax formalisms have in common that they are explicitly or implicitly related to the abstract syntax of a language. Parsers transform a textual syntax into an abstract syntax model that instantiates the language metamodel. Diagram editors use specific graphical primitives to distinguish entities of abstract syntax models wrt. their metamodel class. Also tree-editors combine a graphical representation of the containment associations between model entities with a textual and form-based representation of entity attributes and references.

In the following, we discuss the details and particularities of common technical spaces for the specification of textual and graphical concrete syntax. Furthermore, we evaluate their applicability in LFE.

Specification Formalisms for Textual Concrete Syntax

Most systematic approaches to specify textual concrete syntax are grammar-based approaches. They originate from the research of Noam Chomsky [Chomsky 1965] on grammars for formal languages. A grammar consists of a set of production rules that specify all syntactically valid expressions that can be build from a language’s alphabet. Given a formal grammar, parser generators can automatically generate a software tool (parser) that transforms a textual language expression to a parse tree or an abstract syntax model conforming to the language metamodel.

A very common category of grammars are context-free grammars. They are typically specified using Extended Backus-Naur Form (EBNF) [Aho 2006]. The growing complexity of programming languages and the advent of DSLs intensified research on modular or extensible grammars and a number of different approaches have been developed that allow for a modular definition of parsers. Table 2.4 concludes the state-of-the-art in the specification of textual concrete syntax. In the following, we discuss these formalisms and technical spaces and evaluate their support for modularity in LFE.

- Context-free grammars are the most popular and widely-used formalism for specifying textual concrete syntax. Their formal foundation and the efficient parsing
Table 2.4.: Formalisms and technical spaces for textual syntax specification.

<table>
<thead>
<tr>
<th>Formalism</th>
<th>Technical Spaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context-free grammars (Inheritance,</td>
<td>Yacc, Bison, SableCC, PolyGlot, JavaCC,</td>
</tr>
<tr>
<td>Delegation)</td>
<td>ANTLR, LISA</td>
</tr>
<tr>
<td>Context-aware scanning</td>
<td>LETOS, Copper, lexer states</td>
</tr>
<tr>
<td>Scannerless Generalised LR Parsing</td>
<td>SDF</td>
</tr>
<tr>
<td>Parsing Expression Grammars</td>
<td>Rats!</td>
</tr>
<tr>
<td>Model-driven parser generators</td>
<td>EMFText, Xtext</td>
</tr>
</tbody>
</table>

algorithms available let to a plethora of parser generators for context-free grammars (e.g., [Johnson 1975, Donnelly 1995, Gagnon 1998, Nystrom 2003, Copeland 2007, Parr 2007]). Basic modularity in context-free grammars can be achieved by so-called grammar inheritance. Here, a grammar extends another. That means the production rules of both grammars are merged. Furthermore, the extending grammar can add, refine, replace, and remove individual production rules [Mernik 2000, Nystrom 2003]. Grammar inheritance is, for instance, implemented in the ANTLR [Parr 2007], LISA [Mernik 1999], or the Polyglot [Nystrom 2003] parser generator.

- The LETOS tool introduced in [Bosch 1997] is based on Delegating Compiler Objects (DGOs). DGOs are self-contained language units consisting of a lexer, a parser and a parse graph. DGOs apply a mechanism called parser delegation to modularise context-free language grammars. The parser of one DGO can delegate the evaluation of a token stream to another parser that runs until it reaches the end of its syntax specification. In addition, parsers can extend, reuse and refine existing parsers. This enables sophisticated ways for modularisation and reuse in concrete syntax specifications.

- The Copper tool introduced in [van Wyk 2007] aims at improving the implementation of extensible languages. A drawback of context-free grammars is that they are not closed under composition. The merging of grammars used to realise grammar inheritance may lead to syntactic and lexical ambiguity. Such ambiguity results in indeterminism in the generated parsers. Copper implements checks to detect such ambiguity for the merged grammars, implements a context-aware scanner to address the threat of lexical overlaps, and provides means to manually add disambiguation functions to handle parsing ambiguities. Context-aware scanning can also be achieved by lexer states [Clark 2000].

- Scannerless Generalised LR Parsing, as, for example, implemented in the Syntax Definition Formalism (SDF) [Heering 1989], completely avoids lexical ambiguity by eliminating the scanner. In addition, ambiguous grammars are parsed into sets of alternative abstract syntax models that are later disambiguated by so-called post-parse filters. SDF comes with a set of generic filters, but for special disambiguation
custom filters have to be provided.

- Packrat parsers (e.g., Rats! [Grimm 2006]) are also scannerless and based on Parsing Expression Grammars (PEGs) [Ford 2004]. Production rules for PEGs are written in an EBNF dialect. It replaces the alternative construct ("|") with an ordered choice ("/`). All PEGs are deterministic and closed under composition which makes them beneficial for modular syntax specifications. Packrat parser support only a restricted subset of context-free languages.

- Approaches like EMFText or Xtext [Efftinge 2006] aim at providing advanced means for implementing textual concrete syntax for languages specified using EMF. They can be understood as technical space bridges [Bézivin 2005] between the EMF metamodelling technical space and a technical space for concrete syntax specification. EMFText and Xtext both implement model-driven parser generators that work with a model-based representation for context-free grammars. They integrate the ANTLR parser generator and add means to parse language expressions into EMF-based models. Both provide means for grammar inheritance and add sophisticated checks to detect and resolve syntactic and lexical ambiguity. Furthermore, they automatically generate an advanced textual editor for a given language grammar and a language printer that transforms EMF-based models back to textual language expressions.

Applicability for LFE

The modularity of concrete syntax specifications is determined by the compositional characteristics of the underlying formalism. For formalisms that are not closed under composition, grammar composition may lead to ambiguities. Ambiguities can occur in the lexer, when token definitions are in conflict, or in the parser, when productions overlap. In the following, we review the introduced formalisms wrt. these issues of concrete syntax composition.

Inheritance and delegation are the basic mechanisms to combine context-free grammars. Unfortunately, context-free grammars are not closed under composition. Here, scanner and parser ambiguity is typically handled in one of the following ways. First, ambiguities are reported and removed during grammar combination using token and rule overriding. This is, for instance, done in Copper or EMFText. Second, SGLR parsers [Heering 1989] can be generated that omit scanning and build ambiguous abstract syntax models that are later disambiguated by additional post-parse filters.

Another approach is to use grammar classes that are closed under composition (e.g., PEGs). This alleviates the need to handle ambiguities, but also restricts the supported syntax structures, e.g., regarding left-recursive productions.

Our current research on combining parser generators for context-free grammars and frameworks for MDSD demonstrates the potential to extend the generation processes to tools that work on textual concrete syntax beyond parsers. Approaches like EMFText and Xtext provide means to generate and refine a sophisticated tool infrastructure consisting of parsers, printers, editors, debuggers, static semantics analysers, builders, and
other tools. These tools employ existing parser generators and their means for grammar inheritance. In addition, they add tools that analyse grammar specifications for ambiguities in rules and tokens. They also provide special production rules to implement left-recursion in expressions and to realise operator prioritisation [Ete 2011].

Given the strong relation of grammars to abstract syntax, we envision a different approach to grammar modularisation in LFE. The basic idea is to first enhance the modularity properties for abstract syntax metamodeling as discussed above. This enables the definition of independent language components. Concrete syntax can then be defined component-wise against these abstract syntax components. When abstract syntax components are combined to an integrated language, their concrete syntax needs to be integrated to. Here, we can benefit from the powerful MDSD tool machinery (e.g., model transformations) for an analysis and generative composition of the grammar specifications of multiple language components. We argue that their advanced sophisticated generative capabilities and their seamless integration with metamodeling formalism and MDSD tooling suggests an application of model-driven parser generators like EMFText or Xtext for such an approach.

**Specifications Formalisms for Graphical Concrete Syntax**

A specification of graphical syntax for a language consists of a set of graphical elements that are used for visualisation and a mapping of these elements to abstract syntax concepts. Besides these two shared constituents the specification formalisms for graphical concrete syntax are very different. In Table 2.5, we distinguish three general categories of formalisms: model-driven approaches, grammar-based approaches, and graph-based approaches. In the following, we distinguish these categories, present exemplary technical spaces and evaluate their support for modularity in LFE.

**Model-driven approaches** for the specification of graphical syntax are typically found in current metamodeling tools. They provide means to specify a graphical representation for concepts defined in the language metamodel. Model-driven approaches are bound to a concrete metamodeling framework or tool. In the following, we discuss some exemplary approaches:

- The Graphical Modelling Framework (GMF) [Gronback 2009] is a widely used model-driven approach for generating graphical editors for EMF-based languages. The generation of GMF-based editors is configured by a set of dedicated specification models that customise the graphical elements (.gmfgraph), the mapping of the available graphical elements and particular metamodel classes (.gmfmap model) and the editor tool palette (.gmftool model). All models can be distributed among several files which provides means for modular syntax specifications.

- The EuGENia approach introduced in [Kolovos 2010] aims at reducing the complexity experienced in specifying graphical editors with GMF. EuGENia provides means to directly annotate Ecore metamodel concepts with their respective graphical visualisation. Given this annotation, it automatically generates the corresponding GMF models (.gmfgraph, .gmfmap, and .gmftool) and employs the GMF code
2.2. Technical Approaches in Language Engineering

<table>
<thead>
<tr>
<th>Formalism</th>
<th>Technical Spaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model-driven approaches</td>
<td>GMF, EuGENia, MetaEdit+, GEMS</td>
</tr>
<tr>
<td>Grammar-based approaches</td>
<td>positional grammars, relational grammars, constraint multiset grammars</td>
</tr>
<tr>
<td>Graph-based approaches</td>
<td>DiaGen, GenGE, TIGER</td>
</tr>
</tbody>
</table>

generator to derive a graphical editor. Due to the coupling to Ecore metamodels, EuGENia shares their modularity characteristics.

- The MetaEdit+ metamodeling tool [Kelly 1996] provides the so-called Symbol Editor — a graphical tool to specify the graphical representation for the objects, roles, and relationships of a GOPPRR-based metamodel [MetaCase 2009] (cf. Section 2.2.1). Graphical representations can be built using vector-based symbols (e.g., rectangles, circles, lines, Bezier curves) and custom bitmaps. The defined representations are then used to represent model elements in the MetaEdit+ Diagram Editor. As graphical representations are specified against metamodel concepts, the modularity of graphical syntax specifications corresponds to the modularity of the metamodel they refer to with MetaEdit+.

- The Generic Eclipse Modeling System (GEMS) approach discussed in [White 2007] provides means to generate a graphical modeling tool that can be customised using Cascading Style Sheets (CSS) [Lie 1997]. A GEMS style sheet refers to concepts defined in the GEMS metamodel and uses a CSS-like notation to adapt its graphical representation. The visualisation style can also be dynamically adapted based on attribute values of model elements.

Grammar-based approaches commonly define the set of available visualisation elements in an alphabet. Furthermore, a grammar of production rules is used to define the mapping of metamodel concepts and the graphical alphabet. These production rules employ specific spatial relationships to construct complex visualisations from alphabet elements. The available spatial relationships vary for the different approaches. In the following, we describe some exemplary approaches:

- Positional grammars as discussed in [Costagliola 1993] are based on graphical symbols which are located at a concrete position on a grid. A production rule relates a non-terminal and a number of terminals that are in a relative spatial distance. Positional grammars can be translated to context-free grammars. This enables the adaptation and application of parser generators for textual languages. VLDesk [Costagliola 2006] is a tool to generate a graphical language editor, based on eXtended Positional Grammars (XPGs) [Costagliola 2004].

- Relational Grammars were introduced in [Crimi 1990] as extension of grammars for textual languages to provide means to formally describe visual notations. In
addition to the relation “on the left” that is typically used in textual grammars, relational grammars allow additional relations to be used in production rules. The authors of [Ferrucci 1994] describe the problem of parsing relational grammars as NP-hard and suggest a context-free subclass of relational grammars that can be parsed in polynomial time.

• In [Marriott 1994], constraint multiset grammars are introduced as means to generate parsers for graphical syntax. Each production in a constraint multiset grammar relates a concept of the language’s abstract syntax with a constraint that encodes a specific spatial layout on elements of the graphical alphabet. All visual elements and relationships found in a concrete model are then put into a database and constraint solvers are used to match the constraints and derive an abstract syntax model. The approach is, for example, implemented in the CIDER tool [Jansen 2004].

Graph-based approaches Graph-based approaches are based on the inherent graph structure found in most graphical languages. Both the graphical representation and the abstract model of a language expression (cf. Section 2.2.1) can be represented using graphs. Graph grammars are used to prescribe the construction rules for the respective graphs. Furthermore, a formalism to map both representations is needed. In the following, we discuss a number of exemplary approaches addressing this need.

• In [Rekers 1996], the authors propose a graph-based approach to map graphical representations to abstract syntax models. They distinguish the spatial relationship graph and the abstract syntax graph. The spatial relationship graph represents the structure of the graphical concrete syntax. It abstracts from absolute layout data, but works with more abstract graphical relations (e.g., touches, contains, left of). The abstract syntax graph represents the logical structure of the language expression (abstract syntax model). Both graphs can be defined in terms of graph grammars and are connected by so-called representation edges. Editing actions are expressed using production that adapt both the spatial relationship graph and the abstract syntax graph simultaneously.

• DiaGen [Minas 2002] is a tool for generating a graphical editor using graph grammars and graph transformations. A language expression in its concrete graphical representation is described in a representation graph. Each graphical element is represented in a typed node. Connections of graphical elements are represented by graph edges. The concrete syntax of the language is described in a graph grammar. It consists of a number of graph transformation rules. Each transformation rule consists of a graph pattern (left-hand side). The application of a particular rule replaces the matched left-hand side with the right-hand side and, thus, expands the graphical syntax. In addition, graph transformation rules describe the editing operations available for a particular language.

• The GenGEd approach is introduced in [Bardohl 1998]. It supports a generation of graphical editors based on algebraic graph transformations. The GenGEd tool
provides graphical editors to define a custom alphabet of a graphical language as well as rules that specify available edit operations and their impact on abstract syntax models. In the generated graphical editor, these edit operations can be applied. The AGG [Taentzer 2004] graph transformation tool is applied to transform the visual representation and the abstract syntax model accordingly.

- In [Ermel 2006], Ermel et al. introduce the TIGER system. It is a successor of GenGEd and follows a similar approach for generating graphical editors based on edit operations. In contrast to GenGEd, TIGER is tightly integrated with state-of-the-art metamodelling platform EMF and Eclipse technologies for editor implementation like Graphical Editing Framework (GEF) [Gronback 2009]. The visual alphabet for TIGER is defined in a model-driven way and graph transformation rules defined using AGG describe the mapping between a language metamodel and its graphical syntax.

**Applicability for LFE**

In general, specification mechanisms for graphical syntax suffer from similar issues as textual ones. Ambiguities in the graphical representation of language constructs result in difficulties to parse and understand language expressions.

We are not aware of work regarding the modularity of grammar-based approaches for visual languages. Their declarative nature and their relation to context-free grammars suggests that different sets of grammar specifications could be integrated by inheritance and delegation and that syntactic ambiguity is also an issue to be addressed. Research is still investigating the relation of the above mentioned grammar-based approaches to context-free grammars and mainly concerned with enhancing the expressiveness and efficiency of particular grammar classes. For an in-depth discussion and classification of grammar-based approaches for parsing of visual languages, we refer to [Marriott 1997] and [Costagliola 2004]. Tools that are using a grammar-based approach have not yet made their way to the mainstream and are not integrated with current metamodelling approaches.

For graph-based approaches, we also could not find any approach focusing on modularity in syntax specifications. The respective publications commonly discuss the specification of visualisations for complete languages. The transformations used to synchronise the graphical and the abstract syntax are typically given in a monolithic specification. Given recent results for the composition of graph- and model transformations [Kurtev 2006, Wagelaar 2008], it might be possible to develop compositional approaches for graph-based syntax specifications. Investigating this requires further research.

In analogy to formalisms for textual concrete syntax specifications, we consider model-driven approaches for graphical syntax specification most interesting for LFE. Frameworks like GMF enjoy a broad user base in academia and industry and are well integrated with object-oriented metamodelling approaches like EMF. Currently, GMF specifications can be modularised among different files. However, direct, fixed references between these
modules are necessary and break their independence. Again, we see a strong binding of language metamodels and graphical syntax specifications. This suggests a similar approach as discussed for textual concrete syntax. When language components are be defined independently at the level of abstract syntax, the enhanced modularisation can be exploited to achieve the desired modularity properties for specifications for graphical syntax. Given model-driven syntax specifications, MDSD tool machinery can be applied for a composition of multiple graphical syntax specifications.

2.2.3. Specification of Semantics

The semantics of a language defines the meaning of the concepts introduced in the language metamodel. Commonly, two ways of semantics evaluation are distinguished: interpretation and compilation. In this section, we discuss common, technical spaces for realising interpreters and compilers and evaluate their applicability in LFE.

Specification Formalisms for Semantics

Formalisms for the specification and implementation of semantics for languages cover a wide spectrum from very pragmatic implementation patterns for interpreters and compilers to approaches with a strong formal background. Table 2.6 concludes common categories of formalism for semantics specification and names concrete technical spaces implementing the respective formalism. In the following, we discuss their details.

In case of interpretation, language expressions are fed to a software program that dynamically evaluates language expressions. The interpreter pattern described in [Gamma 2002] introduces a systematic and pragmatic way of implementing language semantics in a manually-written interpreter. Basically, an interpreter simulates the behaviour of computer hardware. It uses an environment that stores data values in a key-value memory and manages a stack. The semantics of language expressions are implemented in instructions (implemented in the language the interpreter is written in) that manipulate the environment and perform computations on the memory and the stack.

In [Parr 2009], Parr discusses different patterns for interpreter implementations:

- Syntax-directed interpreters immediately evaluate language expressions during parsing. Therefore, grammar productions are annotated with semantic actions. These actions directly access the tokens matched in the respective production during parsing and implement its semantics using a GPL. Syntax-directed interpreters can, for example, be implemented using Java expressions and the ANTLR parser generator [Parr 2007].

- Model-based interpreters evaluate individual elements of the abstract syntax model resulting from parsing. Such evaluation is typically implemented using methods, functions or procedures that are implemented for the data structures generated from the abstract syntax metamodel or using the visitor pattern [Gamma 2002].

In case of compilation, the language expressions are translated to another language with an already implemented semantics. Similar to an interpreter, a compiler can also
### 2.2. Technical Approaches in Language Engineering

#### Table 2.6.: Formalisms and technical spaces for semantics specification.

<table>
<thead>
<tr>
<th>Formalism</th>
<th>Technical Spaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreter implementation patterns</td>
<td>Syntax-directed interpreter, model-based interpreter implemented in GPL</td>
</tr>
<tr>
<td>Compiler implementation patterns</td>
<td>Syntax-directed compiler, model-based compiler, rule-based compiler implemented in GPL</td>
</tr>
<tr>
<td>Natural semantics</td>
<td>Typol, CENTAUR, RML</td>
</tr>
<tr>
<td>Attribute grammars</td>
<td>Synthesizer Generator, FNC-2, Eli, Cocktail, LISA, JastAdd, Silver</td>
</tr>
<tr>
<td>Term rewriting</td>
<td>ASF+SDF, Stratego, RASCAL, TXL, ELAN, Maude, TOM</td>
</tr>
<tr>
<td>Action semantics</td>
<td>Actress, OASIS, Abaco</td>
</tr>
<tr>
<td>Abstract state machines</td>
<td>MAX, Montages</td>
</tr>
<tr>
<td>Model checking, -analysis, -synthesis</td>
<td>Coq, PROMELA, HUGO, Alloy, Isabelle, Isabelle/HOL</td>
</tr>
<tr>
<td>Graph- or model transformation</td>
<td>EProvide, Xtext, Xactium, QVT, ATL, ETL, JET, MOFScrip, EGL</td>
</tr>
<tr>
<td>Proprietary tools with semantics support</td>
<td>Elegant, RIGAL, OCS, CoSy, GENTLE, PIM, MetaEdit+, MPS</td>
</tr>
</tbody>
</table>


- Syntax-directed compilers are implemented in semantic actions for grammar productions. These semantic actions directly print code for the compiler’s target language. Due to the direct association of parsing and printing, syntax-directed compilers are restricted to the implementation of semantics for very simple languages. Languages with sophisticated control, delegation, and modularisation constructs typically require multiple passes to analyse and evaluate their semantics.

- To enable interpretation for more sophisticated languages, model-based compilers are suggested. They first derive the abstract syntax model of the program to compile. This abstract syntax model can then be analysed, optimised, and normalised. Finally, they employ a visitor to translate syntax elements of the input language to syntax elements in the compiler’s target language.

- In addition, Parr distinguished rule-based compilers. These also operate on abstract syntax models. In addition, they employ declarative specifications to describe the mapping of constructs in the input language to constructs in the output language. The explicit specification of mapping rules makes rule-based compilers extensible and prepares the exploitation of more formal approaches to specify language semantics.
Chapter 2. Review of Current Language Engineering

In both interpretation and compilation, two languages are involved. First, the source language that is meant to be semantically evaluated. Second, the target language that enables semantics implementation. A semantics specification can, thus, be considered a mapping between the source and the target language. In case of interpretation, the mapping is implemented in the interpreter using the target language. In case of compilation, the mapping is implemented as a transformation that translates expressions in the source language to expressions in the target language. In analogy to grammars for concrete syntax specification, theoretical computer science investigated means to formally define such mapping. Historically, three basic directions of semantics were distinguished [Winskel 1993].

- Operational semantics define the semantics of a language by describing its execution in computation steps on an abstract machine. Plotkin [Plotkin 1981] was the first to advocate the use of structural operational semantics. Here, inference rules describe how individual language constructs map to a computation step on the abstract machine. Given these inference rules, operational semantics are meant to prepare a straight-forward implementation of a semantics interpreter in a logic programming language like Prolog.

- Denotational semantics use mathematical representations (called denotations [Scott 1971]) to specify the semantics of language constructs. A program is considered a mathematical function mapping the programs input to its output. Denotational semantics are more abstract than operational semantics and not meant to prepare a direct implementation.

- Axiomatic semantics systems like Hoare logic [Hoare 1969] are used to proof the semantic correctness of a given program. Axiomatic semantics verify that for a given logical pre-condition and a program a given post-condition holds. The impact of particular language constructs is encoded in logical inference rules. Axiomatic semantics are more abstract than denotational semantics and typically only enable a partial verification of the correctness of a given program. They are not meant to fully describe a language’s semantics.

Conceptually, all three directions comply to the idea of specifying semantics in a mapping to a language with implemented semantics, i.e., operations of an abstract machine, mathematics, or logics, respectively. Based on these foundations, various more or less formal technical approaches emerged to generate compilers or interpreters from semantics specifications. This has two main advantages compared to pragmatic approaches for semantics implementation. First, it enables a more abstract, declarative specification of the mapping rules, rather than manual implementation of the translation process. Second, formal semantics specifications can be checked for completeness and correctness. In the following, we review common technical spaces and tools founding on formal semantics formalisms:

- Natural semantics are based on Plotkin’s structural operational semantics and were introduced in [Kahn 1987]. They define language semantics by a set of inference
2.2. Technical Approaches in Language Engineering

rules that relate to individual language constructs. Each rule consists of a number of premises and a consequence. If all premises hold, the consequence is proven.

The Typol language [Despeyroux 1984] enables the compilation of natural semantics specifications into Prolog to realise an interpreter implementation. Typol is included in the language implementation environment CENTAUR that provides additional means for the specification of syntax and the generation of language tooling (e.g., editors).

To enable the generation of an efficient compiler from a natural semantics specification, Pettersson introduced the language RML [Pettersson 1999]. Using a number of intermediate representations the inference rules of a natural semantics specification are transformed to C code. The resulting compiler is much faster than the aforementioned interpreters for natural semantics.

• The formalism of Attribute Grammars (AGs) was first introduced by Knuth in [Knuth 1968] as a technique to assign semantics to the productions of context-free grammars. AGs enable the specification of calculations on the elements of the abstract syntax model. Basically, synthesised attributes and inherited attributes are distinguished. Synthesised attributes are defined by calculations among values of other attributes. Inherited attributes are passed along the containment tree of the abstract syntax model. Semantics specifications using AGs can be checked for circularity or completeness.

The basic ideas of AGs were successively refined and extended. Such extensions include higher-ordered AGs [Vogt 1989], reference AGs [Ekman 2004], forwarding [van Wyk 2002], or collection attributes [Boyland 2005] and were implemented in a plethora of AG systems (e.g., Synthesizer Generator [Reps 1984], FNC-2 [Jourdan 1990], Eli [Gray 1992], Ag in Cocktail [Grosch 2002], LISA [Mernik 2002], JastAdd [Hedin 2003], or Silver [van Wyk 2008]).

To enable an easier application of AGs for the specification of semantics for modelling languages, we introduced the JastEMF approach [Bürger 2010]. It implements a transparent integration of JastAdd and EMF and indicates the potential of combining formal semantics approaches and metamodelling.

• Term rewriting is another approach to derive interpreters or compilers for a given semantics specification. A term rewriting system is configured with a set of term rewriting rules consisting of a left-hand side and a right-hand side. Whenever the left-hand side of a rewriting rule matches a given input term the matched term is replaced with the right-hand side of that term.

An application of term rewriting for implementing language semantics can be found in the algebraic specification formalism ASD+SDF [van den Brand 2001]. Here, language expressions are parsed to a term-based representation and language semantics are defined by term equations, i.e., rewrite rules among the terms. In the ASF-SDF system, term rewriting rules are transformed to a C program that implements the given semantics by term traversal and matching. In [Luttik 1997], the
Stratego system is introduced that enables a customisation of rewriting strategies for ASF+SDF to allow for more efficient traversal. The language RAS-CAL [Klint 2009] is introduced as successor of ASF+SDF. It combines a number of extensions to basic term rewriting like Stratego’s rewrite strategies or traversal functions [van den Brand 2003] to enhance the efficiency of term rewriting.

Other term rewriting systems used in semantics specification are TXL [Cordy 1991], ELAN [Borovansky 1998], Maude [Clavel 2002], or TOM [Balland 2007].

- **Action semantics** [Mosses 1992, Mosses 1996] were developed to enhance readability and conciseness of denotational semantics specification. They can be considered a combination of denotational and operational semantics. Action semantics use denotational semantic functions to map language constructs to so-called actions. The semantics of actions is defined operationally. An implementation of action semantics using ASF+SDF was discussed in [Mosses 2002]. Other approaches that enable the generation of compilers or interpreters from action semantics are Agent [Brown 1992], [Bondorf 1993], OASIS [Ørbæk 1994], or Abaco [de Moura 1999].

- The semantics formalism **Abstract State Machines (ASMs)** was introduced in [Gurevich 1995] as means to enable an operational specification of arbitrary algorithms. ASMs generalise finite state machines to work over arbitrary data structures. Finite state machines are restricted to reading from an input location, writing to an output location, and operate on restricted data types (states, input alphabet). Abstract state machines can read from and write to arbitrary locations and support arbitrarily complex data types. Consequently, ASMs can be applied for the specification of complex software and hardware systems [Börger 2010]. The formalisation of language semantics is another important application of ASMs (e.g., [Börger 1999, Börger 2005]). Tools for semantics implementation with ASMs are, for example, MAX [Poetzsch-Heffter 1994], Montages [Anlauff 1997], or AMMA [Di Ruscio 2006].

- Approaches for **model checking**, **model analysis**, or **model synthesis** are widely applied to verify the correctness of hardware and software systems. They typically translate a given specification into a formal representation and apply principles like logic deduction to reason over such representations and prove certain properties. There is a plethora of model checking, model analysis, theorem proving, and model synthesis tools that are based on various formal foundations like logics, finite state machines, or mathematical induction. Their ability to deduce additional knowledge from a given model, to synthesise models and to proof certain model characteristics enables their application for the realisation of language semantics. Among model checkers that were applied in semantics specification we find for example Coq [Terrasse 1995], PROMELA [Lilus 1999], HUGO [Schäfer 2001], Isabelle [Berghofer 2004], Alloy [Kelsen 2008], or Isabelle/HOL [Grönniger 2009].
2.2. Technical Approaches in Language Engineering

• The formalisms of graph transformation or model transformation can also be used in specifying language semantics. They are strongly related, as both can be applied for mapping graph-shaped abstract syntax models to the target language of a semantics specification. Graph transformation systems have a more formal background, whereas model transformation approaches enjoy a recent popularity due to the interest in MDSD.

Typically, model or graph transformations are specified using declarative transformation rules that are matched against models in the input language and construct models in the target language. As discussed by several authors [Corradini 2000, Engels 2000, Wachsmuth 2008], they can be used to define the mapping of language constructs to a semantics domain which is also called semantic anchoring [Chen 2005].

Tools that enable the generation of interpreters or compilers using graph or model transformations are, for example, EProvide [Sadilek 2009], Xtext [Efftinge 2006], or Xactium [Clark 2008]. Furthermore, various model-to-model transformation approaches (e.g., QVT [QVT 2011], ATL [Jouault 2008], ETL [Kolovos 2008b]) and model-to-text transformation approaches (e.g., JET [JET 2011], MOFScript [Oldevik 2006], EGL [Rose 2008]) can be applied.

In addition, there is a huge number of proprietary, commercial tools for compiler construction, e.g., Elegant [Augusteijn 1990], RIGAL [Auguston 1991], OCS [Justice 1993], CoSy [Alt 1994], GENTLE [Schröer 1997], PIM [Bergstra 1997], Metaedit+[Kelly 1996], or MPS [Dmitriev 2005]. They are either based on one of the aforementioned methods for semantics specification or follow a pragmatic approach for semantics realisation.

Applicability for LFE

In research on language semantics, modularity of semantics specifications is an important issue. Early semantics formalisms were criticised for bad modularity, now modular approaches are known for nearly all semantics formalisms, e.g., modular AGs [Dueck 1990, Ekman 2006, Sloane 2010], modular denotational semantics [Liang 1996], modular operational semantics [Mosses 2004], modular term rewriting [Ohlebusch 1995, Lüth 1997], modular action semantics [Doh 2003], or modular ASMs [Goos 2000]. This is just an exemplary selection that could be further refined.

Although some of these approaches announce a compositional technique for the integration of several language modules, we are not aware of a semantics approach that distinguishes means for module specification and module integration. Module specification deals with the implementation of semantics for a specific language feature. Module integration deals with the integration of several language modules for interoperability. As discussed previously, this distinction of a component model for module specification and a composition language for module integration is vital to achieve the desired modularity properties. The approaches discussed above target the modular development of single languages. Here, module integration is of minor importance, as language modules can be designed for compatibility and are not expected to be adapted for integration and reuse.
Language families strengthen the idea of reuse and variability and, thus, the importance of a composition language for integrating language modules for different context. Furthermore, most semantics approaches lack an integration with current metamodeling approaches like MOF. Instead they typically reside in a proprietary technical space. This complicates their application for developing language families with state-of-the-art tooling like EMF.


We envision a compositional approach for semantics implementation in LFE. In analogy to concrete syntax, semantics is defined in relation to abstract syntax. Consequently, we propose to couple semantics specification and integration to the component model and the composition language defined at the level of abstract syntax metamodels and employ MDSD tool machinery to handle semantics composition.

2.2.4. Requirements for an Enhanced LFE Technique

As discussed above, we envision an enhanced technique for compositional LFE. The approach should provide means for defining self-contained, reusable language components at the level of abstract syntax that provide the introduced modularity properties. It should, furthermore, enable the integration of formalisms for specifying concrete syntax and semantics component-wise. In the following, we derive concrete requirements for a language composition system to realise these objectives.

Generally, a composition system is a triple consisting of a component model, a composition language, and a composition technique [Aßmann 2003]. The component model describes how components—in our case language components—look like and how they can be accessed. A component model (CM) for language families need to address the following requirements:

**CM 10: Self-Contained Language Components** The component model needs to support the specification self-contained language components that are independent of each other. This is meant to ensure their isolated evolution, their exchangeability and their flexible reuse in different languages or language families.

**CM 20: Explicit Component Interfaces** Components need to provide means to specify required and provided interfaces to enable their later interconnection.

**CM 30: Information Hiding** Component independence requires information hiding between components. Thus, the component model needs to encapsulate the inner workings of each component. A component model with explicit interfaces also helps preserving the principle of information hiding.

**CM 40: Comprehensible Component Specification** The component model needs to support the different constituents of a language implementation (abstract syntax, concrete syntax, semantics) for individual components to be reusable and comprehensible.
2.2. Technical Approaches in Language Engineering

The composition language introduces the vocabulary used to describe concrete composition programs that specify the combination of several components to a system—in our case an integrated language. A language composition language (CL) needs to address the following requirements:

**CL 10: Flexible Component Binding** A language family is considered to consist of a set of language components. The language composition language needs to provide flexible means to specify the integration these components.

**CL 20: Component Integration** It can not be expected that the interfaces of predefined language components match perfectly. The language composition language needs to provide means to invasively integrate component specifications for interoperability using their required and provided interfaces.

**CL 30: Comprehensible Composition Specification** The composition language needs to support the different constituents of a language implementation (abstract syntax, concrete syntax, semantics) for a realisation of fully-integrated languages or language families.

And finally, the composition technique (CT) defines the technological mechanisms that actually realise the specified composition. The composition technique for language components needs to address the following requirements:

**CT 10: Extensibility** There is a vast number of formalisms for implementing the different constituents of a language. Selecting a concrete formalism is typically a trade-off between their strengths and weaknesses that strongly depends on the context. The composition technique for language families should be extensible with new composition operators to enable language composition in different formalisms.

**CT 20: Universality** Implementing, refining and maintaining composition operators is a recurring task. The composition technique should provide a universal, efficient approach to define composition operators for the different constituents of a language implementation.

**CT 30: Sustainability** Integrating formalism for language specification and the respective operators is still an effort. A language composition technique should be based on a stable, sustainable infrastructure that providers of different formalisms can agree on.

Chapter 4 describes a role-based language composition system that is meant to address these requirements.
In this chapter, we introduce a feature-oriented approach to LFE. It is meant to provide a comprehensible methodical framework to replace ad hoc language development with systematic language engineering that foster reuse and variability management in language families and corresponding tools.

This chapter is structured as follows. In Section 3.1, we introduce the foundations of SPLE and FOSD—a technique for explicit variability analysis and implementation in software systems. In Section 3.2, we motivate and discuss the adaptation and application of techniques from FOSD to variability specification and implementation in LFE. In Section 3.3, we demonstrate the application of feature-oriented LFE for implementing the OWL language family. Finally, in Section 3.4, we evaluate the benefits of feature-oriented LFE, discuss related work and describe open challenges.

3.1. Foundations of Feature-Oriented SPLE

In this section, we discuss the foundations and state-of-the-art in SPLE. We will concentrate on SPLE with features as it is proposed by FOSD. We highlight recent innovations in FOSD and introduce the FOSD tool FeatureMapper\(^1\). It contributes a novel approach for FOSD with arbitrary realisation languages and, thus, paves the path to feature-oriented LFE.

3.1.1. Introduction to SPLE

The insight that software systems for a particular application domain often share some similar functionality, but also vary on other parts has led software engineers to move from the development of single software systems to SPLE [Pohl 2005]. A Software Product Line (SPL) describes a set of related software systems that share a set of common features. A systematic sharing of such commonalities and the development of software systems in families increases reuse and is, thus, more efficient and cost effective [Pohl 2005].

\(^1\)http://www.featuremapper.org
SPLE investigates both the methodical and technical foundations for the development of SPLs. A SPL is typically built in two specification spaces: the problem space, and the solution space [Czarnecki 1998]. In the problem space, variability modelling languages, e.g., feature models [Kang 1990] or the Orthogonal Variability Model (OVM) [Pohl 2005], are used. The solution space consists of arbitrary realisation artefacts that describe the realisation of the SPL and all its features.

The SPLE process is concerned with specifying both spaces of a SPL and their mapping to enable an automatic derivation of custom product variants. Fig. 3.1 shows a generic view on the SPLE process. Instead of referring to concrete formalisms used in each process step, the generic process just names the «generic artefacts» produced in the respective step. The overall process is divided into two main phases called domain engineering and application engineering [Pohl 2005].

Domain engineering is concerned with the development of the SPL in general. It consists of three steps (S1)-(S3). First, the variability of the SPL has to be specified (S1) using a «variability model». Next, the implementation of the SPL members and their commonalities and variability needs to be specified (S2). This may be done in arbitrary specification formalisms and a number of «realisation artefacts». Afterwards, the mapping between «variability model» and (parts of) «realisation artefacts» needs to be described (S3). Therefore, a number of mapping approaches can be applied. In general, a «variability mapping» consists of a number of m:n relationships between elements of the «variability model» and (parts of) «realisation artefacts». This finalises the domain engineering phase.

Application engineering is concerned with the derivation of a custom product variant from the SPL. It consists of two additional steps. First, a «variant model» needs to be given (S4). It needs to conform to the «variability model» provided in (S1). Next, the product derivation step (S5) takes all the specification artefacts provided by (S1)-(S4) and automatically derives and deploys a specific product variant.

Although the process was illustrated in a stepwise manner, it is not strictly sequential.
3.1. Foundations of Feature-Oriented SPLE

In contrast, the two phases and their individual steps are typically executed concurrently. Their explicit interdependencies help developers to synchronise the development artefacts. For example, it is necessary to first finish the specification and realisation of a feature, before both can be mapped. However, both feature specification and realisation are expected to be refined and extended in further iterations of the SPLE process.

3.1.2. Feature-Oriented Software Development

The availability of appropriate tooling is vital in each step in the SPLE process. This led to the emergence of various methodologies and tools. FOSD is an attempt to implement SPLs with a focus on so-called features. It uses and extends the Feature-Oriented Domain Analysis (FODA) methodology [Kang 1990] to analyse, specify and implement features. As FOSD and feature models enjoy a very widespread use, a comprehensive methodology, and good tool support, they were selected as our foundation for variability analysis and management in LFE.

As depicted in Fig. 3.2, FOSD specialises the generic SPLE process (cf. Fig. 3.1) wrt. three steps. First, feature models are used as «variability models» during (S1) variability specification. Second, features and feature expressions are used as source elements in «variability mappings». Third, variant feature models are used as «variant models» in (S4) variant specification. In the following, we discuss the foundations and recent innovations for these three process steps. We also introduce the particularities of our FOSD tool FeatureMapper that motivates its application for LFE.

Feature-Oriented Variability Specification

In FOSD, a feature is considered a requirement on a software system from a stakeholder’s perspective that results in an increment in program functionality [Czarnecki 2000]. Kang et al. state that “ [...] features define both common aspects of the domain as well as differences between related systems in the domain. Features are also used to define the
domain in terms of the mandatory, optional, or alternative characteristics of these related systems.” [Kang 1990].

There exists a variety of feature modelling languages that are built around the basic notion of features and feature constraints. The first feature modelling language was introduced in FODA by Kang et al. [Kang 1990]. It denotes a hierarchically organised tree of features that is built and constrained by the following relationships:

**Parent-Child Relationship** A feature model has a root feature with a set of child features and each of the child features can again have children and so on. Using this relationship, feature models build a tree structure. Each parent-child relationship implicitly implies the inclusion of the parent feature when one of its child features is meant to be included in variant.

**Mandatory Features** A child feature is either mandatory or optional w.r.t its parent feature. Mandatory means that the feature needs to be included in every variant its parent is included in.

**Optional features** An optional child feature can be optionally included when its parent is selected for a variant.

**Alternative Features** A set of child features can be in an alternative relationship which means that exactly one of the features has to be included in the variant model if their common parent is included.

**Or Features** A set of child features can be in an or relationship which means that at least one of the features has to be included in the variant model if their common parent is included.

Besides these basic relationships, a feature model can be complemented by additional cross-tree relationships. They are used to express implications and conflicts between features that are not related hierarchically. FOSD typically distinguishes requires and conflicts relationships. A requires relationship between two features means that the inclusion of the first feature requires the inclusion of the second feature. A conflicts relationship between two features means that the inclusion of either feature prohibits the inclusion of the other feature. The requires and conflicts relationships can be considered two special variants of propositional cross-tree relationships [Batory 2005]. For their normalised representation and for more complicated constraints, we use the implies cross-tree relationship:

**Implies** A feature can imply a propositional term (using Negation ¬, Conjunction ∧, and Disjunction ∨) defined against all other features in the feature model to describe expressive cross-tree relationships.

A general way to describe the semantics of feature relations are propositional formulas [Batory 2005]. Table 3.1 introduces a graphical notation for the discussed relationships and concludes their semantics using propositional formulas.
3.1. Foundations of Feature-Oriented SLE

Table 3.1.: Feature relationships and their semantics in propositional logic.

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Visualisation</th>
<th>Propositional Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent-Child</td>
<td></td>
<td>( C \rightarrow P )</td>
</tr>
<tr>
<td>Mandatory</td>
<td></td>
<td>( P \rightarrow M )</td>
</tr>
<tr>
<td>Optional</td>
<td></td>
<td>(-)</td>
</tr>
<tr>
<td>Alternative</td>
<td></td>
<td>( P \rightarrow ((A_1 \land \neg A_2 \land \ldots \land \neg A_n) \lor (A_2 \land \neg A_1 \land \ldots \land \neg A_n) \lor \ldots \lor (A_n \land \neg A_1 \land \ldots \land \neg A_{n-1})) )</td>
</tr>
<tr>
<td>Or</td>
<td></td>
<td>( P \rightarrow (O_1 \lor \ldots \lor O_n) )</td>
</tr>
<tr>
<td>Implies</td>
<td></td>
<td>( F \rightarrow \text{Term} ) (with ( \text{Term} ) build using ( \neg, \land, \lor, ) and references to features)</td>
</tr>
</tbody>
</table>

In addition, various extensions to the original feature modelling constructs, e.g., cardinality-based features, feature groups [Czarnecki 2005b], and feature attributes [Czarnecki 2005c] were suggested. As their benefit and meaning is still discussed controversially in practice, our FOSD tool FeatureMapper restricts means to build feature models to the six relationships introduced above. The tree-shaped structure of feature models allows for a hierarchical nesting of variability specifications. Furthermore, the FeatureMapper allows for building a SPL using a number of feature models that each describe a particular dimension of variability.

Feature-Oriented Variability Mapping

To describe the relation of features to the solution space, FOSD applies so-called mappings. A mapping describes the interconnection of features to (parts of) realisation artefacts. This interconnection can be expressed in various means. In the following, we review the state-of-the-art in this respect.

Traditionally, feature mappings were mainly considered for programming languages. Language-specific annotation approaches like directives for the C preprocessor were used to markup features in source code. As they are supported by standard tooling and
support a very fine grained annotation, they enjoy a wide application in industry. Tools like Gears [Krueger 2001] or pure::variants [Beuche 2004] enabled a mapping of such code annotations to feature models. This enables a centralised organisation of features and more systematic variant management in SPLs. In [Kästner 2008], Kästner et al. introduce the tool CIDE. It implements a generic approach to map features to textual languages. CIDE uses an externalised mapping to avoid the pollution and obfuscation of source code with annotations. To enable the application of CIDE for a specific language, a language grammar has to be provided in a custom, CIDE-specific grammar format. Given that grammar, a parser for the language is generated that produces a tree-based representation of input programs. CIDE provides means to map features to this tree-based representation and to visualise the mappings in custom language editors.

On of the first approaches to combine product-line engineering with modelling languages was introduced in [Muthig 2002]. There, UML component diagrams were used to specify product-line architectures in a model-driven way. Variability information was annotated using UML stereotypes. Czarnecki and Antkiewicz [Czarnecki 2005a] extended such annotations with means to refer to features from the variability specification. They introduced the concept of so-called template models, i.e., models that describe the realisation of the complete SPL using various UML diagrams. Stereotypes where used to annotate feature-based presence conditions. Given a variant specification, the evaluation of these presence conditions reveals a variant-specific instantiation of the template model. This refined model can then be used to deploy the custom SPL member. The authors of [Ziadi 2006] also use annotations to annotate feature names in a template model and implemented a generic model transformation to derive a Variant-Specific Model (VSM) for a given feature selection. In principle, these approaches are language independent, but they require an adequate syntactic construct to annotate realisation artefacts with presence conditions or feature names.

The approach presented in [Botterweck 2007], employs model transformation languages to encode the mapping procedure. The authors use source models that describe the whole product line and define a generic model transformation that copies the full source model. To create custom product variants, the generic transformation is refined to adapt the copy process wrt. the features selected for a specific product variant. As model transformations work for arbitrary model-based languages the approach works for arbitrary realisation artefacts.

The approach introduced in [Zschaler 2009] extends existing languages to enable their application in variability implementation for SPLs. It describes a generative way to introduce variability constructs that enable a mapping of features to a set of modification actions in solution space models. The general mapping methodology is generic, but the mapping language has to be specialised for each new language to be supported.

In [Heidenreich 2007], we describe an approach to implement feature mappings using Graph Rewrite Systems (GRSs) and sets of edit operations on solution space models. As both, GRSs and the representation of edit operations are language independent, the approach can be employed to map features to arbitrary realisation artefacts. In the tool FeatureMapper [Heidenreich 2008b], we generalised the approach to implement a realisation-language independent approach to map feature models to arbitrary Ecore-
3.1. Foundations of Feature-Oriented SPL-based modelling or programming languages. Feature mappings are represented by a tuple consisting of a propositional feature expression and a number of model elements in the solution space which are mapped to the given feature expression. The set of tuples belonging to the SPL is stored in a so-called mapping model. This realises an externalised representation of the mapping information and, thus, eliminates the need for specific language constructs, tool extensions, or language extensions to represent mappings. Finally, an interpretative, generic variant derivation process evaluates the mapping model and derives a SPL variant.

Feature-Oriented Variant Specification

A feature-oriented variant specification is typically expressed as a subset of the features declared in the feature model. A variant model is required to conform to the variability constraints imposed by the feature model. This relationship of feature and variant models can be employed to check variant specifications for validity, but also to guide developers in customising product lines. Both tasks require means for systematic feature analysis [Kang 1990], i.e., reasoning on features, feature relationships and variant models.

Due to the semantic correspondence of feature relationships and propositional logic, a number of approaches employ Prolog [Mannion 2002, Beuche 2003], model checkers [Zhang 2004], the Alloy Analyzer [Sun 2005], SAT solvers [Batory 2005], CSP solvers [Benavides 2005], description logics [Wang 2005], BDDs [Czarnecki 2007], or proprietary approaches [van Deursen 2002, Segura 2008] for feature analysis. For a comprehensive survey, we refer to [Benavides 2006].

For feature analysis in the FeatureMapper, we integrated and extended the analysis approach discussed in [Wang 2005]. It implements a two stepped approach. First, feature models, their relationships, and variant models are transformed to an OWL-based representation. Second, ontology reasoners are used to answer concrete questions of feature analysis. This approach is seamlessly integrated with the FeatureMapper tool. For details on the implementation, we refer to [Zhao 2011].

During variant specification, the approach is used to find inconsistencies and to guide developers in resolving these. Whenever a feature is added to a variant, our approach triggers the transformation and reasoning process. If the reasoning process finds inconsistencies, these are translated to error annotations in the variant models. We also derive repair suggestions that guide developers in resolving the found errors. Another useful feature analysis service that our implementation provides is partial completion for incomplete variant specifications. It automates the inclusion or exclusion of features whose inclusion or exclusion status can be automatically derived from a given, incomplete variant model.

The approach is also used during the other phases of the FOSD process that deal with feature models. During variability specification, it checks the constraint definitions in feature models for consistency. Each relationship that is added to the feature model is immediately checked for satisfiability w.r.t to the existing features and relationships. During variability mapping, our approach interactively checks feature expressions that are meant to be mapped to the solution space to avoid unsatisfiable mappings.
3.2. Feature-Oriented Language Family Engineering

In the previous section, we introduced the idea of SPLE and its realisation in FOSD. Now, we elaborate how these ideas can be adapted and employed to contribute systematic means for variability analysis and variability management in LFE. Our approach can be considered a specialisation of the FOSD process depicted in Fig. 3.2 in terms of the «realisation artefacts» used to implement the solution space.

Fig. 3.3 depicts the refined version of the FOSD process applied for LFE. It refines the FOSD process in two steps. First, LFE requires particular «realisation artefacts» in the step of (S2) Product-line Realisation. These cover different artefacts for language specification like abstract syntax, concrete syntax, and semantics specification as well the implementation of language tooling. These realisation artefacts also need to be supported in (S3) Variability Mapping. A feature-oriented approach for LFE requires the ability to map features to (parts of) the aforementioned realisation artefacts.

Besides the depicted refinement, the specialisation of FOSD for language families also impacts the other phases of the FOSD process. In the following, we discuss the impact of this specialisation for the individual process phases. In Section 3.2.1, we discuss the phases related to the problem space of LFE. In Section 3.2.2, we investigate the impact on phases related to the solution space and the mapping of problem and solution space.

3.2.1. Variability and Variant Specification in LFE

Feature-Oriented variability and variant specification as introduced in the previous section is a universal approach that can be applied to model the solution space of product lines for arbitrary domains. In language families, variability occurs at very specific abstraction and metamodelling levels: First, variability may affect the particular expressions that are supported by a given family member. A number of language expressions that are considered to belong together is typically denoted a language feature. Second, if we analyse a wider context, variability in language families also affects other dimen-
3.2. Feature-Oriented Language Family Engineering

Variability of Language Features

Variability in the expressiveness of languages is often motivated by the need to address the requirements of different application domains. In fact, we find ad hoc approaches to classify language families in accordance to language features in various context, e.g., for the classification of the various dialects of the OWL language family. Baader et al. describe their syntactic and semantic expressiveness inductively by the DL constructors they provide for knowledge representation [Baader 2003]. Feature-Oriented variability specifications introduce a standardised and expressive methodology to leverage such ad hoc approaches. In contrast to just naming concepts, a feature model provides means to organise language features hierarchically and to explicitly represent various relationships among them. This not only helps a better classification of languages in the language family, but also leverages language customisation.

The feature model depicted in Fig. 3.4 describes the variability found for languages of the OWL language family as defined in [Baader 2003]. Each language needs to include all mandatory features, i.e., Concepts, Atomic Concept, Top, Bottom, Concept Constructors, Minimal, Union, Intersection, Atomic Negation, ValueRestriction, Limited Existential Quantification, Roles, and RoleConstructors. This minimal feature selection corresponds to the language ALC (Attributive Language with...
Chapter 3. Feature-Oriented Language Family Engineering

Variability of Languages in Software Development

Languages, i.e., customised members of language families, are used in various phases and activities of software development. A quite abstract variability dimension for language families is related to choosing and customising the concrete languages and language libraries employed for a particular software development activity. This variability dimension is considered interesting for LFE as it makes the variability wrt. development activities and alternative languages explicit and allows the definition of constraints on allowed language combinations.

The feature model depicted in Fig. 3.5 shows that some development phases, e.g., Deployment might be optional. Particular languages might be used alternatively for

Complements) that is considered a minimal base of the DL language family. By including, for example, the optional features Transitive Roles, Role Hierarchies, Nominals, Inverse Roles, Unqualified Number Restriction, and Datatypes one could configure the DL language $SHOIN(D^+)$ with the expressiveness of OWL DL [Baader 2003].

In addition to the conventional approach of building language variants by freely combining constructors, the feature-based approach supports additional constraints on their combination. For instance, the features Functionality, Unqualified Number Restriction, and Qualified Number Restriction are modeled as exclusive alternatives, or the selection of feature Complex Inclusion implies a selection of the features Transitive Roles and Role Hierarchies. Such constraints are important to explicitly express feature dependencies and can be employed to guide the customisation of OWL variants. Such customisation of OWL variants is mainly motivated by the trade-off experienced between language expressiveness and reasoning efficiency. Therefore, it is vital to apply OWL variants that match the expressiveness and efficiency requirements of a given use case.
a given development activity, e.g., one might choose to use either Java or C# for the implementation phase. Furthermore, there might be cross-dependencies between customisation options, e.g., the inclusion of JUnit for the Testing phase or Javadoc for Documentation requires the application of Java in the Implementation phase.

Such a variability specification might be used to specify the portfolio of languages and developer skills available within a software engineering company. A particular variant of this feature model is instantiated whenever a concrete development project is started. The variability binding might result in an initial configuration of the development infrastructure (IDE with respective languages and plug-ins), the suggestion of a project team with the respective skills, or the derivation of particular project management and documentation artefacts.

**Variability of Language Tooling**

The practical applicability of a language is not only influenced by its expressiveness, but also by the language tooling available to edit and evaluate language expressions. The advancement of software engineering from machine languages to higher level implementation languages and modelling languages was accompanied by an evolution of early software development tools to sophisticated tool environments [Isazadeh 1997, Wende 2011]. Advanced model-driven parser generators introduced in Section 2.2.2 provide means to generate or at least prepare implementation stubs for sophisticated editors and the associated tool environments. In EMFText, these generators are configured using a plethora of options in the syntax specification of a concrete language.

Fig. 3.6 depicts a feature model that specifies the variability provided by EMFText generators. The features Resource Plug-in and UI Plug-in represent the tooling generated for basic handling of language resources and for a graphical User Interface (UI) to
work with language resources, respectively. While the Resource Plug-in is mandatory, the UI Plug-in can be disabled which results in a headless language runtime. The resource plug-in contributes a Parser, a Printer and Reference Resolvers to derive a model-based representation from given language expressions. In addition, stubs for Post Processors, an Interpreter, or a Builder can be generated. Furthermore, the generated parser can support automatic Token Sorting, Backtracking or Memoisation. If the UI Plug-in is enabled, an Editor with Syntax Highlighting, Code Completion, and an Outline View are contributed. In addition, optional wizards and graphical tools to evaluate language expressions like a Launcher or a Debugger can be enabled.

Again additional feature relationships contribute constraints that limit the variability specified by the feature model. The selection of the feature Debugger requires the contribution of an Interpreter and the activation of the Launcher feature. This explicit variability specification helps language developers to comprehend the available options and to reflect their interdependencies when developing a new language and the corresponding tools with EMFText.

Variability of Metamodelling Languages and Tools

Metamodelling approaches provide advanced languages or language families and sophisticated means to formally specify analysis and transformation activities on models. Given such specification, according tools can be generated. They help to enhance both the quality and efficiency of implementing modelling languages. Our review of language engineering approaches in Section 2.2 revealed a number of alternative formalisms and approaches. Even when we focus at a single technical space like Eclipse/EMF, we find a number of alternatives to achieve particular metamodelling tasks. A feature-based classification helps to analyse the variability of metamodelling with EMF and to explicitly specify interdependencies of metamodelling languages and tools.

In the context of LFE, such classification is considered vital to categorise language specification approaches and their relationships. The feature model depicted in Fig. 3.7 specifies the variability found for a number of exemplary metamodelling approaches and tools to manage and process languages built using the Eclipse IDE and the EMF framework. It distinguishes languages and language families for Metamodelling, and Automation. Metamodelling Languages are categorised in terms of their function, i.e., specification of Abstract Syntax, Concrete Syntax, or Semantics. Each category names a number of features referring to alternative metamodelling languages available. Automation Languages are distinguished into languages for Consistency Checking and for Transformation. The Transformation languages are further divided into languages for Model to Model transformation and for Model to Text transformation.

The variability specification shows that there are a number of non-exclusive alternatives for particular metamodelling tasks. In general, the approaches can be freely combined as they typically only depend on the feature EMF that is mandatory anyway. The only implication goes from EuGENia to GMF since the EuGENia approach generates GMF models for further evaluation.

Some of the presented languages again build language families that share a common
3.2. Feature-Oriented Language Family Engineering

Eclipse/EMF

Concrete Syntax

Metamodelling Languages

EMF

Xtext

Abstract Syntax

Textual

Graphical

Consistency Checking

OCL

Xcheck

Transformation

QVT

Xtend

Model to Model

ATL

Mofscript

Model to Text

Xpand

ETL

EGL

EuGENia

Ontologies

JastEMF AGs

Java

Semantics

Evolutions

Ecore

Java

Ecore2Xtext

Ecore2JastEMF

Ecore2EuGENia

Figure 3.7.: Variability of metamodelling languages in Eclipse/EMF.

core language and runtime infrastructure. Each family member is specialised to realise particular metamodelling tasks. For instance, the Xtext language family (Xtext, Xcheck, Xtend, and Xpand) or the Epsilon language family (EuGENia, EVL, ETL, and EGL).

Categorisation of Variability Dimensions

The exemplary variability dimensions discussed in this chapter demonstrate the wide applicability of feature-oriented variability specification in the context of LFE.

As depicted in Fig. 3.8, these examples extend classical applications of FOSD to the metamodelling levels M2 and M3. To show that feature modelling scales for different metamodelling and abstraction levels, we discussed one example for each possible combination between metamodelling and abstraction level. The examples at M2 investigate variability within a single language family, whereas examples at M3 are concerned with variability at the level of tooling for LFE.

In Table 3.2, each variability dimension is classified in terms of the following characteristics:

Language Family Engineering: Describes the context of variability specification during the domain engineering phase.

Language Variant Engineering: Describes the context of variant specification during the product engineering phase.

Product: Describes the structure and capabilities of the resulting product.

3.2.2. Product-Line Realisation, Mapping and Variant Derivation for LFE

Due to the diversity of abstraction- and meta levels involved in LFE, we find a plethora of approaches used in the solution space of language families. Product-line realisation is
Table 3.2.: Classification of variability dimensions.

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Language Family Engineering</th>
<th>Language Variant Engineering</th>
<th>Variant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variability of Language Features</td>
<td>Specifies the variability found in a language family in terms of language features that describe an increment in expressiveness. This is typically done when designing new language families or for classifying existing ones.</td>
<td>The customisation of a concrete language configuration is typically done in the context of a given application scenario. The goal here is to customise a language variant that addresses the requirements of the scenario in terms of expressiveness and efficiency.</td>
<td>A custom language variant and implementation with language constructs that provide the required expressiveness and efficiency.</td>
</tr>
<tr>
<td>Variability of Languages in Software Development</td>
<td>Specifies the languages and language families available for specific software development activities in a particular company. This portfolio management is typically done for companies or development teams.</td>
<td>The customisation of a concrete language configuration is typically done during the initialisation of a new development project. Configuration is done by the respective project manager.</td>
<td>A custom configuration describes the setup of a technological and methodical project infrastructure.</td>
</tr>
<tr>
<td>Variability of Language Tooling</td>
<td>Provides an explicit specification of variability and interdependencies found in the generation options of the model-driven parser generator EMFText.</td>
<td>The customisation of these options is typically done when defining a syntax specification for a new language or language family. The objective here is to customise the generated language and tooling implementation wrt. the concrete requirements of the newly developed language (family).</td>
<td>Product derivation results in a custom language and language tooling implementation suited to the respective application context.</td>
</tr>
<tr>
<td>Variability of Metamodelling Languages and Tools</td>
<td>Specifies the variability found in metamodelling languages and language families for the Eclipse/EMF technical space.</td>
<td>The selection of approaches and tools is required when specifying and generating the desired tooling to implement a new language family.</td>
<td>Product derivation results in a custom setup for the Eclipse/EMF metamodelling infrastructure.</td>
</tr>
</tbody>
</table>
3.2. Feature-Oriented Language Family Engineering

Figure 3.8.: Categorisation of exemplary variability dimensions wrt. abstraction and metamodelling level.

considered the development of realisation artefacts that implement the commonality and variability of the language family and the respective tooling. We experienced that all realisation artefacts of a language family can be represented in a model-based way, i.e., they are themselves specified using a (meta)modelling language. This is a vital foundation to provide a comprehensible and practical approach to product line realisation, mapping and variant derivation in LFE. In the following, we discuss the concrete languages and respective realisation artefacts that are important for the variability examples introduced in the previous section.

Realisation Artefacts for Language Features

The implementation of language features at the level of expressions typically affects the abstract syntax, the concrete syntax and the semantics of a language family. Consequently, languages for specifying abstract syntax (e.g., MOF, Ecore) for defining textual (e.g., grammar specifications) or graphical concrete syntax (e.g., graph grammars, GMF), and for implementing semantics (e.g., ASMs) need to be supported.

Implementation of Languages in Software Development

The example on language families used in software development may be implemented with realisation artefacts of different structure. They could be mapped to components of a modelling tool that implement tooling for a particular language. They may be mapped to text documents that describe a particular project process, or they may be mapped to
Chapter 3. Feature-Oriented Language Family Engineering

a specification that allocates developers and resources in accordance to the required and provided skills.

Implementation of Language Tooling

In the respective example, we introduced various code generation options that impact the language tooling derived with EMFText. Consequently, the generation options that are part of the EMFText concrete syntax specifications contribute a realisation artefact. Other generators may provide other ways for parametrisation like configuration files or command line options that need to be supported in the solution space of feature-oriented LFE.

Implementation of Metamodelling Languages and Tools

The different metamodelling languages and tools used in our example are deployed as components (so-called plug-ins) for the Eclipse IDE. Such components are represented as a number of realisation artefacts (.jar archives) in the file system. Different file-types and realisation artefacts may be used in other metamodelling platforms.

The diversity of modelling languages required to implement language families and the according tools requires a versatile approach for mapping features to realisation artefacts. As discussed in Section 3.1, the FeatureMapper provides a generic approach to map features to elements of models. Given that flexibility, the various realisation artefacts can be addressed. However, technically the FeatureMapper still requires an EMF-based representation of the realisation artefacts. Providing such a representation ends up in the task to find or develop appropriate languages. Our model-driven language development tool EMFText plays an important role in this regard. First, it comes with a Zoo\(^2\) that can be used to find existing languages. Second, EMFText significantly increases the productivity in developing new languages if needed [Heidenreich 2009a]. In Table 3.3, we introduce a selection of languages from the EMFText Zoo that are applicable for a mapping of features to the respective variability dimensions.

Given the plethora of available languages and the ease of developing new languages on demand, this combination of FeatureMapper and EMFText leverages variability mapping and variant derivation for different abstraction and metamodelling levels. As depicted in Fig. 3.9, variability mapping starts with a so-called Variant-Independent Model (VIM), i.e., a solution space model that contains all realisation artefacts needed to implement the features of the complete language family. If the model is given in a textual notation (for instance an abstract syntax specification in TextEcore), (1) EMFText is used to transform the textual specification to a EMF-based model. This model, in combination with the feature model defined during variability specification, is fed to the FeatureMapper. In a (2) interactive mapping process, a developer assigns features or boolean feature expressions to elements of the VIM. The mapping information is stored in an external

\(^2\)http://www.emftext.org/zoo/
Table 3.3.: Selection of languages from EMFText zoo applicable for variability mapping.

<table>
<thead>
<tr>
<th>Language</th>
<th>Description</th>
<th>Mapping of</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMFText .cs</td>
<td>Metamodelling language for the specification of concrete syntax and the configuration of EMFText code generation options</td>
<td>language features, language tooling, metamodelling languages and tools</td>
</tr>
<tr>
<td>Java 5</td>
<td>EMFText-based syntax for complete Java 5 specification</td>
<td>languages in software development</td>
</tr>
<tr>
<td>Java Property Files</td>
<td>Language to parse Java property files that are often used in tool configuration</td>
<td>language tooling</td>
</tr>
<tr>
<td>KM3</td>
<td>Textual syntax for the KM3 abstract syntax metamodelling language</td>
<td>language features, metamodelling languages and tools</td>
</tr>
<tr>
<td>MinEcore</td>
<td>Very concise textual syntax for the Ecore abstract syntax metamodelling language</td>
<td>language features, metamodelling languages and tools</td>
</tr>
<tr>
<td>OCL</td>
<td>Textual syntax for the OCL</td>
<td>metamodelling languages and tools</td>
</tr>
<tr>
<td>OSGI Manifest Files</td>
<td>Language to parse OSGI manifest files that are used to configure Eclipse plug-ins</td>
<td>language tooling</td>
</tr>
<tr>
<td>OWL 2</td>
<td>Textual syntax for OWL 2 Manchester syntax</td>
<td>metamodelling languages and tools</td>
</tr>
<tr>
<td>Plain Text</td>
<td>Language to parse plain text files that can be used in documentation</td>
<td>languages in software development</td>
</tr>
<tr>
<td>QuickUML</td>
<td>Concise textual syntax to describe UML class diagrams that can be used in software design</td>
<td>languages in software development</td>
</tr>
<tr>
<td>TextEcore</td>
<td>Textual syntax for Ecore metamodelling language</td>
<td>language features, metamodelling languages and tools</td>
</tr>
<tr>
<td>UML Statemachines</td>
<td>Textual syntax to describe UML statemachines that can be used in software design</td>
<td>languages in software development</td>
</tr>
<tr>
<td>XML</td>
<td>Language to parse XML files that are often used in tool configuration</td>
<td>language tooling</td>
</tr>
</tbody>
</table>
mapping model. Given a concrete variant specification, the FeatureMapper can now interpret the mapping model and reduce the VIM wrt. selected features and the realisation artefacts they are mapped to. This (3) automatic variant derivation is implemented as a generic model transformation that again works with arbitrary EMF-based models and results in a VSM. Finally, the (4) EMFText printer for the respective solution space language is used to derive a textual representation of the VSM that can be (5) further processed to derive an implementation for the language variant.

The discussed combination of EMFText and FeatureMapper is just one example to access languages for LFE with the FeatureMapper. We also investigated modelling languages that use the standard tree-based representation generated by the EMF framework and graphical modelling languages implemented with the GMF framework. In these cases, different techniques to bridge the concrete and the EMF-based model representations are involved. However, the mapping and derivation steps are not affected.
3.3. Case Study: Scalability in Ontology Specification, Evaluation and Application

In the previous section, we motivated feature-oriented LFE and suggested its application at different abstraction and metamodelling levels. The goal of this section is to evaluate this applicability for a language family based on the OWL standard [OWL 2009a]. This is meant to evaluate our qualitative contributions compared to state-of-the-art and indicate the practical relevance of our work.

OWL 2 is a very expressive language for conceptual modelling which makes reasoning on OWL ontologies that use the full language expressiveness a non-trivial problem. For OWL 2 with direct semantics, the worst-case computational complexity for problems like consistency or satisfiability checking is 2NExpTime-complete [OWL 2009c].

We argue that efficient reasoning can be achieved by a custom configuration of languages, reasoners, and a sensible application of ontology-based technology. Scalable ontology specification, evaluation and application can, thus, be understood as a problem of customising an appropriate variant from a family of ontology languages and the respective tools.

In the following, we investigate the application of feature-oriented LFE to realise a systematic approach for OWL customisation. In Section 3.3.1, we evaluate the historical development of OWL 2 in terms of the introduced three sufficient phenomena for language families to further substantiate its reception as a language family. In Section 3.3.2, we discuss how the application of feature-oriented LFE at different abstraction levels helps the realisation of efficient reasoning. Finally, in Section 3.4, we evaluate the qualitative impact of feature-oriented LFE in comparison to previous work and with regard to the requirements on LFE processes discussed in Section 2.1.

3.3.1. Review of Evolution, Customisation and Combination in the OWL Language Family

The goal of this section is to support our assumption to consider OWL 2 a characteristic example for language families. Therefore, we discuss the sufficient phenomena for founding language families in the context of OWL. This is a vital foundation to be able to expand the results for this case study to feature-oriented LFE in general.

**OWL Evolution** The development of OWL started from a merge of the ontology languages DARPA Agent Markup Language (DAML) and Ontology Inference Layer (OIL). DAML was an XML-based language introduced by Hendler and McGuinness [Hendler 2000]. It was meant to represent the semantic information on web pages in a machine-readable format with a formal semantics. Later it was merged with OIL [Fensel 2001] to DAML-OIL [Connolly 2001]. The development of the OWL language family started from a revision of DAML-OIL.

The first official version of OWL became a W3C recommendation in 2004. In 2006, version 1.1 of OWL was submitted. It contributed extensions to OWL DL like qualified cardinality restrictions, disjoint properties, property chains, or custom datatypes. Later
Chapter 3. Feature-Oriented Language Family Engineering

this version was renamed to OWL 2 to indicate the major evolution compared to OWL. OWL 2 became a W3C recommendation in 2009.

This short review of the development of OWL indicates a steady evolution of the language that is triggered by progress in the underlying formalisms and the expressiveness required by language users. Although the evolution can be expressed by means of a revision and extension of language features, we are not aware of previous work that investigated feature-oriented techniques for OWL evolution.

**OWL Customisation** As discussed in [Baader 2003], all OWL profiles are based on the formalism of DL. That means, they share a common approach of representing knowledge. They are based on hierarchies of unary atomic concepts that are augmented with binary logical operators or roles to describe concept relationships. As expressiveness and reasoning efficiency is directly determined by the operators a language variant provides [Donini 1997], customisation can be used to configure the trade-off between performance and reasoning capabilities.

The original OWL standard consisted of the three sub-languages OWL Lite, OWL DL, and OWL Full that contribute custom expressiveness and reasoning capabilities. In OWL 2, a profile mechanism was introduced that enables a more detailed customisation of language variants. OWL 2 also introduced three standard profiles: OWL 2 EL, OWL 2 QL, and OWL 2 RL.

The customisation of ontology languages is typically expressed in a syntactic reduction of the full OWL expressiveness. Consequently, it also impacts language tooling like semantics reasoners, OWL parsers, (textual) ontology editors, or approximation approaches (e.g., [Groot 2005, Hitzler 2005, Pan 2007b, Pan 2006]). As we discussed in [Wende 2009b], addressing these various realisation artefacts during OWL customisation is a challenge that could heavily benefit from feature-oriented LFE.

**OWL Combination** OWL is just one language in a set of languages applied in the development of ontology-based systems. There are several related W3C standards. For instance, the language SWRL [SWR 2004] combines OWL with Rule Markup Language (RuleML). It enables the definition of a rule consisting of an implication between an antecedent and a consequent. The meaning of the rule is that if the antecedent is evaluated to "true", then the consequent must also hold. SWRL rules are posed against the concepts and roles defined in an OWL ontology which motivates a tight integration of both languages. Semantically, SWRL is also based on the formal foundations of DL, but contributes more expressiveness in reasoning on relations between individuals or with data values.

SPARQL is another query language that can be used in combination with OWL. It was originally developed to query data in the Resource Description Framework (RDF) format. Queries in SPARQL use triple patterns that contain variables and are matched against a triple-based representation of RDF models. In [Sirin 2007a], Sirin and Parsia discuss a sophisticated language customisation to derive a SPARQL variant for OWL-DL ontologies.
Furthermore, there are various approaches to combine OWL with languages for metamodeling or modelling. Their primary aim is to employ capabilities of semantic reasoning for providing a semantic foundation for metamodeling and to enable consistency checking or validation during modelling. Such combination typically relies on integration or transformation bridges. In [Walter 2010], we discussed an integration of the Ecore metamodeling language and OWL. The OWLText approach [OTe 2011] refines this integration and contributes the language OWL-CL—a merge of OWL and Ecore to enable a convenient specification of semantic constraints for metamodels. Both integration approaches are generic which means that they can be applied to arbitrary modeling languages that are built using the respective approach. Another way of integrating ontologies and modelling languages is to use custom transformation bridges that translate models of a specific language to a specific ontology-based representation (e.g., [Miksa 2010, Lemcke 2010]). These transformation bridges require more effort in implementation as they need to be built from scratch for every concrete language. On the other hand, they can employ more expressive and optimised translation patterns leading to enhanced reasoning performance.

These heterogeneous examples of combination and integration demonstrate the broad variability found for OWL wrt. language combination. The goal of achieving an efficient and reuse-oriented implementation and application of these approaches again motivates an explicit specification and categorisation of dependencies, relationships and bridges among the involved languages.

### 3.3.2. Application of Feature-Oriented Language Family Engineering for OWL

In the previous section, we motivated the application of feature-oriented LFE for managing variability found for the OWL language family. Next, we discuss how feature-oriented management and configuration of such variability helps the realisation of efficient reasoning. Efficient reasoning can be achieved by customisation on different abstraction levels. In this thesis, we focus on the following three:

**Ontology Specification** When OWL is used for knowledge specification, there is a trade-off between language expressiveness and reasoning efficiency. Efficiency of reasoning directly depends on the language features (expressions) available in an ontology language. The need for efficient reasoning has led to a manifold alternative ontology languages with specific reasoning characteristics (e.g., [Calvanese 2007, Berardi 2003, McGuinness 2002, Pan 2007a]).

In contrast to having a number of proprietary languages, OWL 2 introduces the profiles mechanism that enables the definition of custom OWL 2 sub-languages that come with a custom expressiveness and reasoning efficiency. Removing costly expressions results in strongly enhanced reasoning performance.

**Ontology Evaluation** A second branch of work deals with the development of more efficient reasoners or the enhancement of existing reasoning techniques. This led to a
number of highly-optimised, native ontology reasoners (e.g., [Haarslev 2001, Sirin 2007b, Tsarkov 2006]) that perform well even for expressive ontology languages, but only for reasonable sized ontologies.

Larger ontologies often result in poor response times that impede applicability in practice [Keet 2007]. Approaches presented in [Broekstra 2002, Fang 2008, Zhou 2006] store ontologies in relational databases to use the optimised database query engines for ontology reasoning. As discussed in [Ma 2006], this leads to increased load-time but more efficient reasoning compared to native ontology systems. In the approach introduced in [Hustadt 2004], ontologies are represented in disjunctive Datalog programs. Additional algorithmic optimisation can be applied on the Datalog facts to enhance reasoning efficiency.

Other approaches [Groot 2005, Hitzler 2005, Pan 2007b, Pan 2006] enhance reasoning efficiency by approximating more expressive ontology languages to less expressive ones. The reduction of complexity leads to better reasoning performance while preserving the completeness and soundness of the reasoning results. Reasoners and approximation approaches are designed for a very specific subset of DL features. Using a generic reasoning infrastructure like TrOWL [Thomas 2010] that allows for exchanging the reasoning backend they can be combined with our feature-based ontology language configuration. Thus, we can provide appropriate (semantic) reasoning infrastructure wrt. a specific language variant.

Ontology Application  A third direction is to enhance the way how ontology languages are applied to address practical problems. Reasoning efficiency can not be considered an absolute characteristic, but heavily depends on the application context. In an interactive application, other response times are required than in an analysis tool that validates a huge knowledge base. Selecting a concrete ontology language that satisfies the requirements of a specific use case, is a vital aspect of ontology application. In [Keet 2007], the authors provide a comprehensive comparison of nine DL-based ontology languages wrt. their syntactic features and reasoning efficiency. The results of this survey are envisaged as guideline for matching ontology languages to use cases.

A second aspect that influences reasoning efficiency is how knowledge is represented in ontologies. The involved transformation may employ different patterns that lead to more or less efficient reasoning performance. Both aspects can also be combined. Then alternative transformation patterns may be used depending on the evaluation context and its requirements.

In the following, we investigate the application of feature-oriented LFE in these dimensions to implement a scalable approach for ontology specification, evaluation and application. For each dimension, we discuss the characteristics of specifying the respective problem and solution space.
3.3. Case Study: Scalability in Ontology Specification, Evaluation and Application

Variability in Ontology Specification

Specifying and implementing variability wrt. language features used for specifying ontologies with OWL first requires a sensible analysis of the language design. The goal of this analysis is to come up with an understanding how the expressiveness of a language can be described by nameable features, how these features interrelate, and how they manifest in the implementation of the language.

Considering the identification of features for OWL, we already have a very good starting point. In [Baader 2003], Baader et al. introduced a classification approach for ontology languages that is based on atomic concept and role constructors. These constructors contribute distinct parts of syntax and semantics and can be combined to form an ontology language of a particular expressiveness. Furthermore, a given combination can be used to estimate the efficiency of reasoning for the respective language. In Table 3.4, we conclude these constructors and exemplify their textual representation in OWL using Manchester syntax.

Table 3.4.: Representation of DL constructs in OWL2 Manchester syntax.

<table>
<thead>
<tr>
<th>DL construct</th>
<th>OWL2 (Manchester Syntax)</th>
</tr>
</thead>
<tbody>
<tr>
<td>where</td>
<td></td>
</tr>
<tr>
<td>classA and classB denote classes</td>
<td></td>
</tr>
<tr>
<td>i1 and i2 denote individuals</td>
<td></td>
</tr>
<tr>
<td>op, op1, and op2 denote object properties</td>
<td></td>
</tr>
<tr>
<td>dp and dp1 denote data properties</td>
<td></td>
</tr>
<tr>
<td>Dt denote a data type</td>
<td></td>
</tr>
<tr>
<td>Atomic Concept</td>
<td>Class: classA</td>
</tr>
<tr>
<td>Bottom</td>
<td>owl:Nothing</td>
</tr>
<tr>
<td>Top</td>
<td>owl:Thing</td>
</tr>
<tr>
<td>Union</td>
<td>classA or classB</td>
</tr>
<tr>
<td>Intersection</td>
<td>classA and classB</td>
</tr>
<tr>
<td>Atomic Negation</td>
<td>not classA</td>
</tr>
<tr>
<td>Value Restriction</td>
<td>op only classA, dp only Dt</td>
</tr>
<tr>
<td>Limited Existential Qualification</td>
<td>op some classA, dp some Dt</td>
</tr>
<tr>
<td>Nominals</td>
<td>op only {i1, i2}, dp only {1, 2}</td>
</tr>
<tr>
<td>Functionality</td>
<td>ObjectProperty: op</td>
</tr>
<tr>
<td></td>
<td>Characteristics: Functional, DataProperty: dp</td>
</tr>
<tr>
<td></td>
<td>Characteristics: Functional</td>
</tr>
</tbody>
</table>

Continued on next page
Chapter 3. Feature-Oriented Language Family Engineering

<table>
<thead>
<tr>
<th>DL construct</th>
<th>OWL 2 (Manchester Syntax)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unqualified Number Restriction</td>
<td>op min 1,</td>
</tr>
<tr>
<td></td>
<td>op max 1,</td>
</tr>
<tr>
<td></td>
<td>op exactly 1,</td>
</tr>
<tr>
<td></td>
<td>dp min 1,</td>
</tr>
<tr>
<td></td>
<td>dp max 1,</td>
</tr>
<tr>
<td></td>
<td>dp exactly 1</td>
</tr>
<tr>
<td>Qualified Number Restriction</td>
<td>op min 1 classA,</td>
</tr>
<tr>
<td></td>
<td>op max 1 classA,</td>
</tr>
<tr>
<td></td>
<td>op exactly 1 classA,</td>
</tr>
<tr>
<td></td>
<td>dp min 1 Dt,</td>
</tr>
<tr>
<td></td>
<td>dp max 1 Dt,</td>
</tr>
<tr>
<td></td>
<td>dp exactly 1 Dt</td>
</tr>
<tr>
<td>Roles</td>
<td>ObjectProperty: op,</td>
</tr>
<tr>
<td></td>
<td>DataProperty: dp</td>
</tr>
<tr>
<td>Concept Identity</td>
<td>HasKey: op</td>
</tr>
<tr>
<td>Inverse Roles</td>
<td>ObjectProperty: op</td>
</tr>
<tr>
<td></td>
<td>InverseOf: op1</td>
</tr>
<tr>
<td>Transitive Roles</td>
<td>ObjectProperty: op</td>
</tr>
<tr>
<td></td>
<td>Characteristics: Transitive</td>
</tr>
<tr>
<td>Role Hierarchies</td>
<td>ObjectProperty: op</td>
</tr>
<tr>
<td></td>
<td>SubPropertyOf: op1,</td>
</tr>
<tr>
<td></td>
<td>DataProperty: dp</td>
</tr>
<tr>
<td></td>
<td>SubPropertyOf: dp1</td>
</tr>
<tr>
<td>Complex Inclusion</td>
<td>ObjectProperty: op</td>
</tr>
<tr>
<td></td>
<td>SubPropertyOf: op1 o op2,</td>
</tr>
<tr>
<td>DataTypes</td>
<td>Datatype: Dt</td>
</tr>
</tbody>
</table>

This mapping prepares the realisation of a feature-oriented approach for customisation of language expressiveness. In the following, we go through the five steps of the feature-oriented process for LFE (cf. Fig. 3.3) and explain their application.

(S1) Variability Specification For variability specification, we took the DL constructors introduced in Table 3.4 and derived a feature model that specifies their variability and interdependencies. It was specified using the tool FeatureMapper. The resulting feature model was already introduced in Fig. 3.4.

(S2) Product-line Realisation For the realisation of the language features, we used the model-based language engineering tools EMF and EMFText. The OWL metamodel was defined using EMF. It is optimised for a frame-based representation of OWL 2 ontologies as used in OWL Manchester syntax. An excerpt is given in Listing 3.1. As illustrated there, the metamodel covers the complete set of OWL 2 language concepts ranging from Ontology and frames (e.g., Class, ObjectProperty) to primitive data literals.
(e.g., AbbreviatedXSDStringLiteral, AbbreviatedRDFTextLiteral IntegerLiteral, BooleanLiteral). The complete metamodel consists of 53 classes, 13 abstract classes, 21 attributes, 27 containment, and 49 non-containment references.

As textual concrete syntax for OWL 2, we decided to use OWL 2 Manchester Syntax [OWL 2009b]. It provides a compact, frame-based representation of ontologies that is easy to read and comprehend for humans. The syntax was defined using EMFText. We implemented a syntax definition that contains syntax rules for all concrete classes of the OWL 2 metamodel. An excerpt is given in Listing 3.2. It shows the declaration of a textual syntax for the metamodel classes introduced in Listing 3.1. The head of an EMFText syntax rule refers to a concrete class from the language metamodel. Rule bodies can refer to attributes and references of this class. The rule for the class Ontology (Lines 11 to 14) exemplifies the application of EMFText. It defines that the textual representation of an Ontology instance is started with the keyword “Ontology”. An optional ontology uri and versionUri follows. Afterwards, an arbitrary number of import statements, represented by the keyword “Import” and the uri of the ontology to be imported, can be specified.

The generators for EMF and EMFText can be used to derive an implementation of the full OWL 2 language when given the complete metamodel and syntax specification. In the following, we discuss how the feature model and the model-based language specifications are combined to enable the generation of custom OWL 2 language variants.

(S3) Variability Mapping To realise a mapping of variability defined by features in the feature model to (parts of) realisation artefacts, we again applied the tool FeatureMapper. As discussed in Section 3.1.2, the FeatureMapper provides tools to interactively map feature models to arbitrary EMF-based languages. Both the OWL 2 metamodel and the concrete syntax specification for OWL 2 are based on EMF-based languages.

Fig. 3.10 illustrates the mapping process. It shows a screenshot of the FeatureMapper. The MappingView on the left consists of four parts.

- The tool bar (1) for loading and saving mappings for feature models and for controlling different options to visualise feature mappings.
- The feature model compartment (2) that contains the feature model which is currently mapped.
- The current expression compartment (3) that contains the feature or feature expression that is currently active. This feature or feature expression can either be changed by selecting features in the feature model or by using a context menu in this compartment. Using this context menu boolean feature expressions can be built.
- The assigned feature expression compartment (4) that contains the feature or feature expression that has already been applied to the currently selected elements of the solution model.
package owl owl "http://org.emftext/owl.ecore" {

abstract class URIIdentified {
    attribute EString uri (1..1);
}

class OntologyDocument {
    containment reference Ontology ontology (1..1);
    containment reference Namespace namespace (0..-1);
}

class Ontology extends URIIdentified, Annotateable {
    attribute EString versionIRI (0..1);
    reference Ontology imports (0..-1);
    containment reference Frame frames (0..-1);
}

class Class extends Frame, Annotateable {
    containment reference Description superClassesDescriptions (0..-1);
    containment reference Description equivalentClassesDescriptions (0..-1);
    containment reference Description disjointWithClassesDescriptions (0..-1);
    containment reference Description disjointUnionWithClassesDescriptions (0..-1);
}

class ObjectProperty extends Feature, Annotateable {
    containment reference Description propertyRange (0..-1);
    attribute Characteristic characteristics (0..-1);
    containment reference ObjectPropertyReference superProperties (0..-1);
    containment reference ObjectPropertyReference equivalentProperties (0..-1);
    containment reference ObjectPropertyReference disjointProperties (0..-1);
    containment reference ObjectPropertyReference inverseProperties (0..-1);
    containment reference ObjectPropertyReference subPropertyChains (0..-1);
}

class AbbreviatedXSDStringLiteral extends Literal {
    attribute EString value (1..1);
}

class IntegerLiteral extends Literal {
    attribute EInt value (1..1);
}

class BooleanLiteral extends Literal {
    attribute EBoolean value (1..1);
}

Listing 3.1: Excerpt of OWL2 metamodel definition in TextEcore syntax.
3.3. Case Study: Scalability in Ontology Specification, Evaluation and Application

```plaintext
SYNTAXDEF owl
FOR <http://org.emftext/owl.ecore> <owl.genmodel>
START OntologyDocument
OPTIONS {...}
TOKENS {...}
RULES{
OntologyDocument ::= namespace* ontology;
Ontology ::= "Ontology:" (uri[IRI] versionIRI[IRI] ? !1)?
{ ("Import:" imports[IRI] !1)*
(annotations !1)*
(!! frames !1)*
...}
Class ::= "Class:" iri[IRI] !1 ( (annotations !1) |
("SubClassOf:" superClassesDescriptions ("," superClassesDescriptions )* !1)
| ("EquivalentTo:" equivalentClassesDescriptions ("," equivalentClassesDescriptions)* !1)
| ("DisjointWith:" disjointWithClassesDescriptions ("," disjointWithClassesDescriptions )* !1)
| ("DisjointUnionOf:" disjointUnionOfClassesDescriptions ("," disjointUnionOfClassesDescriptions)* !1)
)*;
ObjectProperty ::= "ObjectProperty:" iri[IRI] !1 ((annotations !1)
| ( "Domain:" domain ("," domain)* !1)
| ( "Range:" propertyRange ("," propertyRange)* !1)
| ( "Characteristics:" characteristics[CHARACTERISTICS] ("," characteristics[CHARACTERISTICS])* !1)
| ( "SubPropertyOf:" superProperties ("," superProperties)* !1)
| ( "EquivalentTo:" equivalentProperties ("," equivalentProperties)* !1)
| ( "DisjointWith:" disjointProperties ("," disjointProperties)* !1)
| ( "InverseOf:" inverseProperties ("," inverseProperties)* !1)
| ( "SubPropertyChain:" subPropertyChains ("o" subPropertyChains)+ !1)
)*;
AbbreviatedXSDStringLiteral ::= value[STRING_LITERAL];
IntegerLiteral ::= value[INT];
BooleanLiteral ::= value["true": "false"];}
```

Listing 3.2: Excerpt of OWL2 syntax definition in EMFText syntax.
Figure 3.10.: Application of FeatureMapper for mapping variability to metamodel elements in LFE.
3.3. Case Study: Scalability in Ontology Specification, Evaluation and Application

The solution space model is depicted in the editor on the left (5). The concrete editor used depends on the concrete language used to specify the realisation artefacts.

In the example, the feature model compartment (2) shows the features describing DL constructs for the OWL 2 language family. The feature InverseRoles is selected as current expression (3). It can be applied to the fragments (Line 86) of the OWL 2 metamodel selected in the editor (5) using the down-arrow toolbar button below the current expression compartment. As a result, a new mapping element in the mapping model is created which specifies a link between the feature expression and the model elements that are selected in the editor. This is later used for visualisation [Heidenreich 2008a] or feature-oriented variant derivation. Compartment (4) contains the feature or feature expression that has already been applied to currently selected model elements of the solution model (5). Again the feature InverseRoles is given, telling the developer that the selected metamodel fragment is already mapped to this feature. Furthermore, the screenshot exemplifies the colouring view provided by the FeatureMapper. Features and the realisation artefacts they are mapped to are rendered in the same colour. This is meant to support developers in understanding feature mappings and their interrelations in the solution space.

(S4) Variant Specification Deriving a concrete OWL 2 variant with a particular expressiveness starts with the specification of a variant model. This variant model has to conform to the variability constraints defined in the feature model. To model concrete variants, the FeatureMapper provides the so-called variant editor. Fig. 3.11 demonstrates its application. The variant editor shows the complete feature model with check-boxes for each optional feature. Enabling a check box means the inclusion of the respective feature in the variant model. The depicted variant model consists of the features Concepts, Roles, Concept Constructors, Minimal (and all sub-features), Restrictions, Role Constructors, ConceptIdentity, InverseRoles, and ComplexInclusion.

The variant editor provides means for an automated validation of the variant model against the feature model. Therefore, we translate both the variant model and the feature model to an ontology-based representation and employ satisfiability checking to detect inconsistencies and classification to suggest variant completions. The approach is described in detail in [Zhao 2011]. Found inconsistencies and suggestions are annotated to the variant model. The example in Fig. 3.11 contains two inconsistencies wrt. the feature model defined for the OWL 2 language family (cf. Fig. 3.4). First, the feature Restrictions is included, but none of its alternative sub-features is selected. Second, the feature ComplexInclusion was selected without selecting the implied features TransitiveRoles and RoleHierarchies. Both inconsistencies are reported to the user. As validation and inconsistency annotation is triggered by every change in the variant model, the variant editor provides guidance during a stepwise refinement of a variant with the required expressiveness.

(S5) Product Derivation Given a variant model, the FeatureMapper can be used to automate the derivation of a custom OWL 2 variant. Therefore, it evaluates the variant
Figure 3.11.: Application of variant editor in variant specification for OWL2 language features.

model, the mapping model and the corresponding solution space models (OWL 2 metamodel, OWL 2 syntax specification). The derivation process removes all artefacts from solution space models that are mapped to features that were excluded from the variant specification. This results in a set of model-based language specifications that only consist of specification artefacts belonging to the language core and to features selected for the variant.

These specifications can then be fed to the generators provided in EMF and EMFText to derive a language implementation. The EMF generator derives a Java-based metamodel implementation that provides an API to load, store and manage model instances. EMFText generates a Java-based parser, a printer and a sophisticated editor compatible to the metamodel implementation to edit models. These implementations are customised to the specific OWL 2 variant, i.e., the editor and tooling only supports the language features enabled.

Fig. 3.12 depicts an exemplary editor for an OWL variant that does not contain the feature Datatypes. The generated editor shows two errors that result from the removal
3.3. Case Study: Scalability in Ontology Specification, Evaluation and Application

Figure 3.12.: Application of custom editor derived for OWL2 language variant.

of this feature. It let to a reduction of the concrete and abstract syntax artefacts that were mapped to Datatypes. The resulting language and language tooling prohibits the definition of custom datatypes.

The contribution of the presented approach is twofold: First, we transferred the existing DL-based classification of ontology languages to the paradigm of feature modelling. This helps to capture interdependencies between OWL 2 language features that were not explicitly defined before. Feature-Oriented variant specification provides enhanced guidance in customisation OWL 2 wrt. a specific use case. Second, the approach contributes a fully automated process to derive a custom parser, printer, and a sophisticated editor from a given variant specification. This infrastructure is crucial for the practical application of the language variant in knowledge modelling.

Variability in Ontology Evaluation

The approach presented in the previous section tackled syntactic language variations. To further advance the impact on reasoning efficiency, the approach should be extended to support a customisation of the reasoning infrastructure. For capturing the variability
found in reasoning infrastructures, a number of factors are important. In the following, we discuss these:

**Language** As discussed in the previous section, the ontology language or language variant used for knowledge specification is a determining factor for the efficiency of a reasoning. As particular reasoners support a specific set of languages, the customisation of languages also influences the applicability of a particular reasoner.

**World Assumption** Reasoning can be done with different assumptions regarding knowledge that is not explicitly given in the ontology. Literature distinguishes OWA, CWA, and sometimes also Closed Domain Assumption (CDA). In CWA, the knowledge given is considered closed, i.e., all statements in the knowledge base are considered true and things not specified are considered false. Ontology reasoning typically uses the OWA. It considers a given knowledge base as potentially incomplete part of an open world. Consequently, a knowledge base can be extended by additional statements during reasoning. CDA is a mixture of both CWA and OWA. Here, only certain kinds of statements are considered closed all others other open.

**Reasoner** Different reasoners (e.g., [Haarslev 2001, Sirin 2007b, Tsarkov 2006]) for ontologies are available. The configuration of a concrete reasoner strongly interrelates with other factors like the language it supports or the applied world assumption.

**Approximation** Approximation techniques (e.g., [Groot 2005, Hitzler 2005, Pan 2007b, Pan 2006]) transform knowledge given in a specific language to knowledge represented in a less expressive and, thus, more efficient language. They aim at enhancing the reasoning efficiency, while preserving specific quality characteristics for the reasoning results.

**Database** For some ontology languages, the application of database systems and their highly optimised query engines can help to enhance reasoning performance.

**Ontology Size** The size of the knowledge base is another determining factor for reasoning efficiency.

**Reasoning Services** Reasoning efficiency is relative to the concrete reasoning service used. The following services are typically available:

- **Consistency Checking** Checks if the given ontology O is consistent, i.e., if there exists a model (a model-theoretic instance) for O.
- **Classification** Checks if the given individual i is an instance of concept A in the ontology O.
- **Satisfiability Checking** Finds all unsatisfiable concepts in a given ontology O. A concept in an ontology is unsatisfiable if it is an empty set.
- **Subsumption Checking** Checks whether the interpretation of A is a subset of the interpretation of B in the given ontology O.
3.3. Case Study: Scalability in Ontology Specification, Evaluation and Application

**Explanation** Retrieve the set of axioms that entail axiom Ax in the given ontology O.

**Query Answering** Returns an answer set for a query q to ontology O.

This overview of factors that determine the performance of ontology evaluation can be used to extend the feature-oriented customisation for the OWL language family to also support the corresponding reasoning infrastructure. Next, we discuss how we applied the steps of feature-oriented LFE for enabling the customisation of OWL 2 reasoners.

**(S1) Variability Specification** First, we derived a feature model from the factors that influence reasoning efficiency. The feature model depicted in Fig. 3.13 categorises variability along the introduced factors. Each factor is represented by a top level feature. The selection of the OWL2 Language profile, the World Assumption, the Reasoner and the required Reasoning Services is mandatory. A configuration of Approximation options, a Database, the Ontology Size, and the specification of the required efficiency for reasoning services in terms of complexity classes is optional. The configuration features under top level features are typically exclusive alternatives, i.e., exactly one feature has to be selected.

To capture the impact of concrete configurations on reasoning performance for particular reasoning services, feature constraints are used. The first constraint exemplified in Fig. 3.13 means that to achieve a complexity of NLogSpace for consistency checking (CC NLOGSPACE) either the language OWL QL has to be selected or OWL DL and the Approximation DL to QL. Similar constraints are defined for other reasoning services and complexity classes. The second example constraint expresses that the reasoner Quill requires the configuration of a Database.

**(S2) Product-Line Realisation** For the realisation of the specified variability, we applied the reasoning infrastructure TrOWL\(^3\). TrOWL is a highly configurable reasoning infrastructure that supports configuration wrt. the introduced factors [Thomas 2010].

TrOWL can be configured in three ways.

1. In case of **embedded configuration**, TrOWL tries to derive an adequate configuration automatically. Therefore, TrOWL analyses a given ontology for its expressiveness. Afterwards, it employs approximations, if available, for the given language and, finally, selects an applicable reasoner.

2. In case of an **external configuration with Java property files**, the automatic configuration process can be overridden when TrOWL is deployed for a concrete application.

3. In case of an **external configuration with Java system properties**, the automatic configuration process can be overridden during the runtime of TrOWL.

\(^3\)http://trowl.eu/
For supporting feature-oriented customisation, the external configuration with Java property files during deployment of TrOWL is most suitable. It allows for an explicit and dedicated configuration of a reasoning infrastructure that meets the requirements of a specific OWL 2 application.

(S3) Variability Mapping For connecting the features in the variability specification to their realisation in the TrOWL infrastructure, we require an EMF-based representation of Java property files. Therefore, the used the syntax definition for Java Property Files provided in the EMFText Zoo. Furthermore, we prepared a variant-independent Java property file, i.e., a TrOWL configuration file containing all possible configuration options and values for TrOWL. Finally, we used the FeatureMapper to interactively map features and feature expressions to key-value pairs in the variant-independent property file. The screenshot depicted in Fig. 3.14 exemplifies the mapping procedure. It shows how the feature expression OWL DL and Pellet is mapped to the key-value pair ReasonerFactory.DL = com.clarkparsia.pellet.owlapi.PelletReasonerFactory that initialises TrOWL for the application of the Pellet reasoner for OWL 2 DL ontologies.
3.3. Case Study: Scalability in Ontology Specification, Evaluation and Application

Figure 3.14.: Application of FeatureMapper for mapping variability to TrOWL configuration files in LFE.

(S4) Variant Specification For the specification of a concrete reasoning infrastructure that matches the requirements of a concrete application, we can again use the variant editor. The variability constraints defined in the feature model ensure that a valid and complete configuration is derived. The configuration process can be driven by different kinds of requirements. One can, for example, start by giving the efficiency required for a concrete reasoning service and then configure a matching language and reasoner. If the specific situation requires the application of a particular reasoner, the customisation process can also run in the opposite direction. In either case, variant specification is guided by the variability knowledge encoded in the initial feature model.

(S5) Product Derivation The derivation of a concrete reasoning infrastructure can again be automated for a given variant specification. The derivation process uses the defined feature mapping to reduce the variant-independent Java property file by all key-value pairs not mapped to features that are included in the variant model. Finally, it deploys the resulting configuration with the TrOWL infrastructure. The customised infrastructure can then be applied for ontology reasoning.

The presented customisation approach provides the following benefits not found for current approaches to configure TrOWL or other reasoning infrastructures. Feature models support a comprehensible analysis and a rich documentation of variability in reasoning infrastructures. They also enable an explicit representation of variability and interdependencies between configuration options. This prevents inconsistent configurations that may otherwise only be detected during runtime. The rich variability specification captures expert knowledge needed to configure TrOWL wrt. efficiency or technical constraints. Consequently, such knowledge can be exploited during variant specification to guide non-experts.
Chapter 3. Feature-Oriented Language Family Engineering

Variability in Ontology Application

The concrete application of OWL 2 language variants requires tooling beyond custom reasoning infrastructures as introduced in the previous section. To fully exploit their potential, domain-specific tooling is required. In this section, we investigate how feature-oriented LFE can be used to customise OWL application for ontology-based MDSD.

The idea of ontology-based MDSD was developed in the MOST project that aims at leveraging software engineering by exploiting ontologies and the corresponding reasoning services [Wende 2009a]. Ontology-based MDSD founds on an integration of metamodelling and ontology technology. The application of the resulting technology requires tool environments [Charette 1986] equipped with reasoning technology, metamodelling technology and bridging infrastructure. Furthermore, such tool environments need to be customised to the concrete domain that ontology-based MDSD is meant to be applied in.

In this section, we demonstrate the application of feature-oriented techniques to systematise such customisation. The case study covers the realisation of the MOST TOol Product Family (MOST TOPF), a product line of tool environments for ontology-based MDSD developed to enable an efficient and reuse-oriented realisation of demonstrators in the MOST project.

In general, a tool environment is considered a selection of integrated tools to support a concrete development scenario. In accordance to Isazadeh and Lamb [Isazadeh 1997], we consider individual tools falling into one of the following categories [Wende 2011]:

**Software Process Support** A software process defines the temporal and structural order of individual development phases [Lamb 1988]. Tools for supporting the software process should guide developers through such phases to help the effective execution [Bruckhaus 1996] of a complex software process. This could be achieved by automatically updated task lists suggesting the next development steps. Besides task-oriented guidance tools, we also consider bug- or issue trackers to fall into this category. They help a group of collaborating developers to collaborate in accordance to concrete rules of the bug- or issue tracking process.

**Development Methods Support** A development method is a specific approach applied in (a concrete phase of) software development, e.g., object-oriented modelling [Booch 2007] for the software design. Methods aim at producing artefacts that specify a particular part of the software product, e.g., a requirements document in the analysis phase or an architecture model in the design phase. Here, tool environments need to provide means for creating, editing, and browsing specification artefacts in method specific languages.

**Repetitive Tasks Automation** Finally, tool environments are required to automate repetitive tasks. Such tasks typically involve the validation of specification artefacts or their transformation when entering a new development phase [Stinson 1989], e.g., the generation of implementation code from a design model.
The MOST TOPF is meant to enable the feature-oriented customisation of ontology-based MDSD tool environments for three MDSD case studies provided by the MOST industry partners Comarch and SAP:

**C1 Modelling Physical Network Devices** The case study targets the specification and management of physical network configurations [Miksa 2009]. For that purpose, the case study provider Comarch developed a model-driven method for network specification. They use a specific language called Physical Device DSL (PDDSL) to describe standard configurations of physical devices. For the specification of concrete physical networks, a second language called Physical Device Instance DSL (PDIDSL) is employed. It reuses and specialises standard configurations to specify a customer-specific network structure.

**C2 Structural Modelling in Comarch OSS** The case study covers a classical example of MDSD for the Comarch Operations Support Systems (Comarch OSS)\(^4\). The development involves different languages on different abstraction layers: The Business Entities DSL (BEDSL) is used to specify business objects, their attributes, and relations. The Managed Entities DSL (MEDSL) enables the object-oriented design of entities managed within Comarch OSS. The DataBase DSL (DBDSL) is a language used to model relational databases. The case study involves various model transformations and inter-model relationships.

**C3 Business Process Modelling and Refinement** The case study was provided by SAP and targets the specification and refinement of business processes [Ren 2009, Wende 2009a] using the Business Process Model and Notation (BPMN) [BPM 2009]. The applied business process modelling method is based on a stepwise refinement of abstract business process specifications to more concrete business processes. Usually, an abstract business process describes the core functionality of an application. Each refinement is a mapping of a more abstract business process into a more specific business process with a more detailed process behaviour. The refined business process needs to conform to the behaviour of the abstract business process. Finally, all activities in a business process need to be correctly grounded to an implementation component [Ren 2009]. This method enables the reuse of generic business processes and their adaptation for concrete applications.

In Table 3.5, we conclude the variability of these case studies wrt. above introduced tool categories. All three case studies introduce a specific software process covering design and implementation of software systems in their specific domain and employ a model-driven development method with custom languages. Automation can be achieved in domain-specific validation and transformation tasks.

One particular characteristic of the tool environments developed in MOST is their common dedication to the vision of ontology-based MDSD. Sharing the abstraction level of MDSD metamodelling languages [Happel 2006], ontologies can be easily integrated and composed with existing metamodelling approaches [Walter 2009]. Therefore, all tools developed for the MOST case studies rely on a common (reuseable) integration infrastructure to bridge the gap between the modelling and ontology technical spaces that enables the exchange of languages between technical spaces, and the transformation

\(^4\)http://www.comarch.com/telecommunications/our-offer/operations-support-systems-oss-suite
of models and derived information. Based on this infrastructure, the semantic capabilities of ontology technology can applied in the realisation of tools in all three tool categories. The concrete tools developed are out of scope for this thesis. Details can be found in [Wende 2011].

These common and variable requirements motivate the reuse and case-study specific adaptation of an extensible tool product line. While this promises a reduction of development costs for the individual tool environments, it also introduces the complex and expensive task of platform development and tool customisation. In the following, we describe the application of feature-oriented LFE for the systematic development of customised tool environments for ontology-based MDSD.

**S1 Variability Specification** The development of the MOST TOPF requires a systematic specification of the commonalities and variability of all products. For that purpose, we developed the feature model for the MOST TOPF depicted in Fig. 3.15. In accordance to the three tool categories, the features are organised in three feature groups: **Software Process Guidance**, **Development Method**, and **Automation**. The features within these groups result from the variability found for the MOST case studies (cf. Table 3.5). A fourth feature group relates to the **Ontology Technology** used to address the vision of ontology-based MDSD. This group provides features to customise different **Ontology Languages** with a specific reasoning complexity [Donini 1997] and corresponding **Reasoners**. The configuration of a concrete **Development Method** and the according **Modelling Language** is mandatory. **Automation**, **Ontology Technology** and **Software Process Guidance** are considered optional as not every MDSD environment needs to provide them. Within the groups, features are considered alternative (e.g., the supported **Modelling Language**), exclusive alternative (e.g., the applied **MDSD Process**), or optional.
There are also dependencies between features of the MOST TOPF that are more complex. As discussed in [Ren 2009], the BPMN validation developed in MOST relies on a transformation of business process models to an OWL DL ontology. Consequently, the selection of the ValidationSpecification for BPMN requires to also include the according ontology language (OWL DL). The corresponding constraint is exemplified in Fig. 3.15.

Further constraints used in the MOST TOPF are presented in [Parreiras 2009].

(S2) Feature Realisation The realisation of the MOST TOPF is based on a generic, component-based architecture. It consists of several standard components for MDSD tool environments such as graphical and textual editors, model and metamodel management tools, and a model transformation infrastructure [Karagiannis 2002, Steinberg 2008, Greenfield 2004]. In addition, an ontology-based tool environment needs to provide components that contribute and integrate ontology technology. The challenge of building such tool environment is raised by the technological clash between conventional MDSD technology and ontology technology. Heterogeneous technology spaces need to be integrated into a uniform infrastructure, in order to provide the end user uniform access to the tool environment services.

Therefore, we introduced the generic architecture for ontology-based MDSD tool environments (cf. Fig. 3.16) that abstracts from concrete implementations of tool environments and decomposes the system into layered blocks of related system components, addressing the key characteristics of ontology-based MDSD tool environments. The uppermost layer contributes various Editors to create and edit model-based system specifications and several Views providing developers information of the current development status. Below these components, we find various components that contribute Ontology-based Services for software process guidance, specification validation, explanation of validation results, and others. These services are enabled by the subjacent
Integration Infrastructure that provides integration of the Modelling Infrastructure typically found in MDSD tools and the Ontology Infrastructure that makes our tool environments ontology-based. In addition, the generic architecture contains Vertical Services like user and rights management or versioning and Persistency Services.

The MOST TOPF architecture was fully specified using UML component diagrams [UML 2009] and is considered a platform-independent blueprint for ontology-based MDSD tool environments [Zivkovic 2009].

(S3) Feature Mapping To enable a feature-oriented customisation of MOST TOPF products, we need to define how the previously identified features are mapped to concrete realisation components. Therefore, we mapped features and feature expression to elements of the MOST TOPF component diagram.

Fig. 3.17 exemplifies the application of FeatureMapper. All components in the UML component diagram that are associated to the features Specification_Validation, BPMN, or both are highlighted. This visualisation indicates a potential mapping problem, as the highlighted component BPMN Validation Services depends on the masked component OWL-DL. However, the feature constraint defined in the feature model would prohibit the selection of feature Specification_Validation without selecting the feature OWL_DL that itself triggers the inclusion of the OWL-DL component. For further details on the mappings, we refer to [Zivkovic 2009].

(S4) Variant Specification The configuration of a concrete MOST TOPF product that is customised for a specific case study starts with the specification of a case-study specific variant model. Again this step employs the variant editor provided by the FeatureMapper to guide the customisation process. Based on the feature groups and constraints defined
3.3. Case Study: Scalability in Ontology Specification, Evaluation and Application

Figure 3.17.: Application of FeatureMapper for mapping variability to components of the MOST TOPF architecture in LFE.

in the feature model, intermediate variant models are validated wrt. the feature model and inconsistencies are reported to the user. Fixing these inconsistencies provides guidance for the stepwise refinement of the intermediate variant model to a valid variant. Fig. 3.18, depicts the variant model a MOST TOPF demonstrator provided for the SAP case study.

(S5) Product Derivation If valid, the variant model is evaluated together with the specification of the MOST TOPF components and the mapping model. An automatic transformation task now removes all MOST TOPF components that are mapped to features not contained in the variant model. This results in a refined and customised MOST TOPF component model for the case-study specific requirements. Finally, the reduced UML component model is used to deploy a physical MOST TOPF instance. Each component is associated to a set of physical realisation artefacts, i.e., files of the MOST TOPF implementation. The deployment task takes all physical artefacts for in-

Figure 3.18.: Variant model for SAP MOST TOPF variant.
Figure 3.19.: Demonstrator for SAP case study derived from MOST TOPF.

included components, bundles, and deploys them in an executable MOST TOPF instance.

Fig. 3.19 shows the demonstrator derived for SAP variant model given in Fig. 3.18. It contributes tooling for each of the tree tool categories. The top compartment (1) shows the task-list view. It realises the ontology-based process guidance for SAP business process modelling by suggesting tasks a business process modeller can perform wrt. the current development context. The middle compartment (2) shows a graphical editor for the BPMN language. The lower compartment (3) contains the validation view showing the results of the automated, ontology-based validation of BPMN refinements and groundings.

3.4. Discussion

In this section, we discuss experiences made during the presented case studies and evaluate the impact of feature-orientation for LFE in general. The feature-oriented LFE process introduced in Fig. 3.3 consists of five steps (S1)-(S5) each contributing particular means for variability specification and management in LFE. Our discussion relates to these phases. Afterwards, we discuss related work, conclude the contributions of feature-oriented LFE wrt. the requirements introduced in Section 2.1, and present open issues.
3.4.1. Contributions

Impact of Feature-Orientation on Variability Specification in LFE The various ex-
amples in Section 3.1.2 and the case studies in this section demonstrated the universal
applicability and the rich expressiveness of feature models for variability specification
and analysis for various abstraction levels in LFE. Compared to no or rather unstruc-
tured management of variability found in current language engineering, feature models
enable a hierarchical specification of variability and commonalities for language families.
The means to describe variability constraints enable a precise and concise specification
of interdependencies between features.

Furthermore, we experienced a strong communicative benefit in applying feature mod-
els. Variability analysis in the case studies required a tight collaboration of experts in
language engineering, MDSD, and knowledge engineering. Although these research areas
are related in some parts, there are differences in terminology and basic principles. In
addition, there is typically a set of implicit assumptions that are shared within a given
research area but not among several ones. The explicit variability modelling triggered a
settlement on a common terminology and an explicit communication and specification of
implicit assumptions. Feature models served both as medium for communication and as
medium to record the communication results in a precise format.

Finally, we experienced that feature models also structure and guide evolution in LFE.
Evolution steps during the case studies typically involved extensions to existing feature
groups or refinements of features with sub-features or new alternatives. Feature-oriented
variability specification, thus, not only help to document, but also to refine variability.

Impact of Feature-Orientation on Product Realisation in LFE The realisation of
the discussed case studies was founded on MDSD with EMF-based languages. This
was motivated by the fact that our tool FeatureMapper requires EMF-based languages
for variability mapping. Although this may sound like a limitation for the applica-
bility of our approach, our experiences showed that EMF and EMFText provide a
sound foundation for the realisation of all kinds of languages ranging from simple
modelling languages [Heidenreich 2009a] to sophisticated programming languages like
Java [Heidenreich 2010]. The availability of various ready-to-use metamodelling and
modelling languages from the EMFText Zoo5 and the contributions in variability map-
ing counterbalance this technical constraint. Overall, product realisation in LFE ben-
efits from the general advantages of MDSD like appropriate abstraction or increased
productivity and quality [Stahl 2006].

We also experienced some impact of variability specification in language family realis-
tion. Variation points and functional units in language families could be easily identified
by the given feature model. During the design and realisation of the language families, we
could systematically prepare variation and modularise the language realisation artefacts
in accordance to the identified variability. This has a number benefits. First, it leads
to a clear and comprehensible design in the solution space of LFE. Second, a structural

5www.emftext.org/zoo
modularisation of feature realisation eases the later variability mapping. Third, it motivated and helped the realisation of extensible, generic and reusable implementations that prepare evolution and combination in LFE.

Regarding the modularity of realisation artefacts, we experienced two principle cases in product realisation. The realisation of variability in OWL 2 language features and OWL evaluation involved languages without the concept of components whereas the MOST TOPF was built using an extensible component-based infrastructure. In the first case, the realisation of different features could not be encapsulated using modularity concepts of the realisation language and it required sensible design and implementation to keep the realisation reusable and extensible. In the second case, components provided a solid and expressive foundation to encapsulate features of different granularity. The modularity concept of components enabled and induced a highly extensible and reusable feature realisation. This has two reasons. First, components enforce an explicit specification of required and provided interfaces. This decouples individual components (i.e., feature realisations), separates interface and implementation, and, thus, enhances reusability and interchangeability. Second, component-based systems are typically based on an extension infrastructure that provide means for component deployment, registration, activation, deactivation and disposal. Such infrastructure contributes a generic foundation for handling variability in product realisation. Consequently, system realisation does not need to deal with technical issues of variability realisation. Furthermore, the clean separation of feature implementations in component-based architectures reduced the threat of feature interaction [Calder 2003] that required special attention in the first two case studies.

Impact of Feature-Orientation on Variability Mapping in LFE During variability mapping, we used the tool FeatureMapper to map features or feature expression to realisation artefacts of the language family. We experienced the interactive, visual mapping approach implemented in the FeatureMapper advantageous for universal variability mapping at different abstraction levels and to various realisation languages. The approach to specify a mapping by selecting the feature in the feature model and the corresponding solution space artefacts worked well and intuitively for both graphical and textual languages. Also the visualisation of feature mappings that involve different colouring schemata for particular purposes were applicable and useful for the various languages. Colours provide an intuitive meaning but are rarely associated with a syntactic meaning in common modelling language. Consequently, mapping visualisations did not interfere with the languages syntax.

Complex feature mappings that involve boolean feature expressions turned out to be useful to handle situations, where a combination of features triggers the inclusion or invasive adaptation of particular language family realisation artefacts. The visualisation of such complex mappings is harder than the visualisation of simple mappings and typically requires more user interaction. This motivates future work to investigate means to address this challenge.

The mapping process and the views provided by the FeatureMapper provide immediate
feedback during the mapping process. For instance, feature expressions that are meant
to be mapped are checked for consistency wrt. the feature model and users are warned
if inconsistencies are found. The FeatureMapper also reports incomplete mappings that
may result from changes in the feature model or on solution space artefacts. This helped
both maintenance and evolution in LFE.

During the evolution of the case studies, we repeatedly experienced broken mappings
to realisation artefacts which were only identified by their logical position in the solution
space and not by a unique name. This was for instance the case for key-value pairs in
TrOWL property files that are stored in the model as a list of entries. Their identification
in the feature mapping was based on their position in that list. Whenever new entries
were added to that list, the position of other entries changed and mappings pointed to
missing or erroneous entries. This issue could be fixed by introducing unique names for
key-value pairs based on the key name and the associated value. Again, the component-
based design of the MOST TOPF was experienced beneficial in this case, as components
and interfaces are commonly identified by a unique name.

Impact of Feature-Orientation on Variant Specification in LFE  Variant specification
is strongly influenced by the contributions of systematic and explicit variability specification.
The feature model captures the available variability and interdependencies between
variability options. During variant specification, this knowledge can be exploited to guide
the variant specification and ensure the derivation of valid and complete variant models.

The variant editor enables an interactive, guided and stepwise refinement during vari-
ant specification. It, therefore, immediately validates changes in the variant model wrt. their consistency with the feature model. Inconsistency and incompleteness is re-
ported to the user and repair options are suggested. A particular benefit is the domain-
specific terminology and communication that is enabled by the feature models. This
advances the comprehensibility of error reports and repair suggestions and hides the
complex realisation details found in the solution space.

Finally, we experienced that guided variant specification allows for an exploration of
additional variants. This is possible as feature-oriented LFE only captures the neces-
sary variability constraints, but does not prescribe concrete variants. For instance, in
the MOST TOPF case study, we discovered variants of tool environments that were not
explicitly considered during variability specification, but still useful in practical applica-
tions.

Impact of Feature-Orientation on Product Derivation in LFE  The step of product
derivation exploits the effort invested in the previous phases to leverage reuse and cus-
tomisation in LFE. Variability and variant specification ensures the consistency of the
custom product wrt. feature interdependencies. The feature mapping enables and checks
the mapping of features in the feature model to realisation artefacts built for product
realisation. In each case study, we experienced that, based on these preparations, prod-
uct derivation enables a fully-automated instantiation of a custom product for arbitrary
abstraction levels of LFE, e.g., a concrete OWL 2 variant, a custom OWL 2 evaluation
infrastructure, or custom tooling for ontology-driven MDSD with OWL 2.

Literature distinguishes additive and subtractive approaches to variant derivation [Völter 2007]. Additive approaches extend the core of a product line with realisation artefacts that realise the features included in the variant model. Subtractive approaches remove realisation artefacts from the solution space that are connected to excluded features. The discussed case studies demonstrated the applicability our approach for subtractive variant derivation. We modelled the implementation of all realisation artefacts of the language family in a VIM. Variant derivation subtractively reduced this VIM by removing realisation artefacts that were mapped to features not included in the variant model. As discussed in [Heidenreich 2007], the FeatureMapper also supports additive variant derivation when using a composition system in the solution space. Therefore, we mapped features to additive composition steps. During variant derivation, only composition steps that are mapped to included features are evaluated and the respective realisation artefacts are added to the solution space.

3.4.2. Related Work

We are aware of a number of publications that discuss means to leverage the state-of-the-art in language engineering processes. A first set of publications suggest and evaluate novel processes for language engineering:

- In [Visser 2008], E. Visser describes the development of a family of DSLs for the design of web applications. Visser proposes an inductive and iterative processes that implements the language engineering phases introduced in [Mernik 2005] (cf. Section 2.1). Visser describes the benefits of an inductive approach to language analysis, i.e., the derivation of required language concepts from common programming patterns found in the domain the language is built for. Furthermore, he suggests an iterative development of a family of orthogonal DSLs for particular subdomains instead of strict top-down engineering of a single all-embracing language. For design and implementation, Visser employs SDF [Heering 1989] and Stratego/XT [Bravenboer 2008]. The inductive approach suggested by Visser requires the availability of programming patterns for the domain the language family is designed for. This is the case in domains like web engineering, where a plethora of applications was already built using GPLs.

When designing language families in other domains, deductive analysis methods are more applicable. Feature-oriented LFE can be used in both ways. In a deductive way of application, the features for a language family are specified top-down before its implementation. In an inductive way of application, the variability specification for the language family is derived from a set of initial realisation artefacts and successively refined.

- Kleppe [Kleppe 2009] suggests three extensions for the classical process of implementing the abstract syntax, the concrete syntax, semantics, and language tooling. First, she emphasises the need for revision and reiteration in language engineering.
Second, she emphasises the potential of model-driven formalisms in language engineering. Third, she mentions the need to handle different versions of a language for different language users. Kleppe suggests a reactive approach to variability management in language engineering. After shipping the first version of a language, language engineers should prepare to come up with different versions and variants for different language users. Within LFE, we suggest a proactive approach, since variability is considered natural for language families that evolve over time, are customised are combined.

- In [Alves 2009], Alves and J. Visser suggest the application of an iterative, grammar-centered process for the development of languages and language tools. They emphasise the importance of testing, metrics, and coverage analysis in controlling the language engineering process. Their evaluation indicates how metrics can be used to quantify the process progress and how tests help grammar disambiguation and refactoring. We use a qualitative, requirements-driven approach to process monitoring and guidance. Feature-driven LFE is not restricted to language grammars, but can handle various artefacts in language family implementation. In future work, it would be interesting to investigate the integration of quantitative, artefact-specific and qualitative means for process guidance.

- In [Gargantini 2010], Gargantini et al. introduce LEMP—a model-driven language engineering process. The authors emphasise the central role of language meta-models in language design. LEMP introduces an extended language engineering process that uses model-driven techniques for the specification of language syntax, constraints, and semantics. The application of the process is demonstrated using an illustrative example. The LEMP process is concerned with the development of a single language, while we focus language families. Furthermore, requirements are handled in a less explicit way than in our approach. In feature-oriented LFE, requirements for language families and single languages are analysed, formally specified and used for process guidance.

A second set of publications introduces approaches concerned with the recognition and management of variability in language analysis, design and implementation:

- In [Thibault 1999], Thibault et al. suggest the application of methods for variability analysis in the design of DSLs. They argue, that a DSLs can be derived from families of programs written for a common domain. They employ the FAST method [Gupta 1997] for variability analysis in such program families. The analysis phase results in a specification of domain terminology, commonalities and variations among the programs in the investigated family.

  Compared to feature-oriented LFE Thibault et al. focus on the development of a single DSL instead of a language family. Their analysis does investigate variability in the domain the language is applied to, but not between different variants or versions of the language. Furthermore, the results of variability analysis are specified in a less formal and explicit way than in feature-oriented LFE.
• In [van Deursen 2002], Deursen and Klint suggest the application of feature specifications to explicitly specify the results of variability analysis in language engineering. They also provide a transformation to generate a class diagram that prepares the implementation of the language from a given feature model. This transformation implements a fixed mapping between features and realisation artefacts that is less flexible than in feature-driven LFE. Deursen and Klint recognise the need for further refinement of the initial class diagram. However, this would break the initial feature mapping. Furthermore, their approach is only concerned with designing a single language not a language family.

• In [Mernik 2005], Mernik et al. review different approaches for requirements and variability analysis in language engineering. They distinguish informal methods, formal methods and methods that extract languages from given code. Mernik et al. criticise the lack of a systematic approach for integrating the language analysis and subsequent phases of language engineering.

• In [Völter 2008], Völter is the first to apply feature modelling in the realisation of a family of languages. He argues that the diversity of software architectures found in practice motivates customisable Architecture Description Languages (ADLs) instead of a generic, standardised language like UML. To still enable the reuse of concepts found in all ADLs, he introduces a language family. Völter’s approach shares our intention for explicit variability modelling and mapping in LFE. He uses the tool pure::variants [Beuche 2003] for mapping ADL features to Xtext grammars [Efftinge 2006]. Völter only investigates language grammars. Language metamodels, semantics, or tooling are not covered.

Compared to the model-driven mapping approach used in the FeatureMapper, pure::variants requires a special markup of grammar fragments to enable a mapping to features. Consequently, the approach is less flexible wrt. the metamodelling languages applicable in language family realisation.

• In [Zschaler 2009], Zschaler et al. introduce a generative approach for realising a family of variability modelling languages. Motivated by the idea to have custom languages for describing the mapping of features to a SPL, they introduce a core variability modelling language that is meant to be customised for specific solution space languages. To specify such customisation, the authors introduce a dedicated DSL. Given a customisation specification, a generator extends the core variability language and derives a custom variability modelling language. The approach is evaluated by deriving a number of concrete language variants.

Compared to feature-driven LFE the presented approach does not rely on a common formalism for variant specification, but introduces a dedicated DSL and the according generative infrastructure to derive language variant. Its questionable whether the effort invested for the setup of such a language family pays of in the general case.
3.4.3. Conclusion

In Table 3.6, we conclude the contributions of the introduced feature-oriented process in relation to the requirements for LFE processes introduced in Sect 2.1. In general, we see the requirements for LFE addressed. We discussed benefits like enhanced documentation, consistency validation, language-agnostic mapping, guidance, and automated instantiation of variants in LFE that support systematic and iterative language evolution, customisation and combination. Although some authors suggested an application of domain analysis techniques for language engineering, we are not aware of another approach that exploits features for LFE to the same degree as our approach.

Table 3.6.: Contribution of feature-oriented LFE wrt. requirements for the development process in LFE introduced in Section 2.1.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP 10: Explicit Variability Analysis and Specification</td>
<td>• feature models with constraints enable feature-oriented variability analysis and specification for language families</td>
</tr>
<tr>
<td></td>
<td>• features document customisation options and guide language evolution and combination</td>
</tr>
<tr>
<td>DP 20: Different Abstraction and Granularity</td>
<td>• features enable abstraction- and granularity-agnostic variability specification in language families</td>
</tr>
<tr>
<td></td>
<td>• features support a terminology that matches the respective abstraction and granularity level</td>
</tr>
<tr>
<td></td>
<td>• FeatureMapper enables abstraction- and granularity-agnostic variability mapping</td>
</tr>
<tr>
<td>DP 30: Continuous Process Application</td>
<td>• explicit mapping of features to realisation artefacts interconnects analysis, design and implementation of language families</td>
</tr>
<tr>
<td></td>
<td>• the process phases are not strictly sequential, but can be executed concurrently and iteratively</td>
</tr>
<tr>
<td>DP 40: Technology Agnostic</td>
<td>• FeatureMapper provides a universal approach for variability mapping to arbitrary metamodelling and modelling languages involved in LFE</td>
</tr>
</tbody>
</table>

Continued on next page
### Chapter 3. Feature-Oriented Language Family Engineering

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DP 50: Guided Language Customisation</strong></td>
<td>• interactive, feature-oriented variant validation and completion guides customisation of language variants</td>
</tr>
</tbody>
</table>
| **DP 60: Automated Variant Derivation** | • FeatureMapper supports automatic derivation of language variants  
• derivation process can be additive or subtractive depending on the language composition system used in the solution space |
| **DP 70: Iterative Refinement** | • structural and semantic validation of variability and variant specification helps consistent refinement of variability in language families  
• structural validation of variability mapping helps consistent refinement of variability mapping in language families  
• interactive reporting of inconsistencies and repair suggestions helps addressing inconsistencies when refining mappings or variants |

Besides these advantages, we see some open issues for feature-oriented LFE. First, explicit variability analysis and management introduces some additional effort in LFE. Our case studies showed that the effort grows with the complexity of variation in the language family. In case of a smaller language family, the investment is comparable low, but might be unnecessary. For the development of complex language families, we regard this investment higher but inevitable. Further empirical evaluations on feature-oriented LFE should be conducted to identify the sweet spot and limits of its application.

Second, feature mapping introduces variability in the realisation artefacts of language families. This variability can interfere with the validation of such artefacts (e.g., ambiguity checks for language grammars). Ensuring the validity of all potential products in a product line is a well-known issue in SPLE [Pohl 2005]. Future work should investigate how approaches to address this issue [Czarnecki 2006, Thaker 2007, Gröner 2011] can be transferred from SPLE to LFE.

Third, we already discussed some issues experienced in the phases related to the problem space of LFE. They were caused by missing or inadequate modularity concepts at the granularity of language family features in current techniques for language realisation. In the next section, we discuss means to address these modularity issues.
Our experiences in applying feature-oriented LFE indicate that a component-based language implementation at the granularity of single language features eases the task of feature mapping, reduces the threat of unexpected feature interaction, and helps language evolution, customisation and combination. Therefore, this chapter introduces a language composition system. Due to its pivotal role in representing language expressions derived from concrete syntax and evaluated by language semantics, the abstract syntax metamodel is considered the essential artefact in such a composition system.

In search for a natural extension to object-oriented metamodelling that alleviates the modularity issues identified in Section 2.2.1, we discovered role modelling [Reenskaug 1996, Steimann 2000b]. Role models decompose object-oriented software specifications into groups of collaborating objects and enable their flexible composition to an integrated system. In a survey on the application of roles in modelling [Steimann 2000b], Steimann concludes 15 features for roles. As depicted in Table 4.1, some of these features have the potential to enhance modularity properties criticised for existing language modularisation approaches. This motivates the introduction of a role-based language composition system. For an in-depth discussion of this motivation, we refer to Section 4.1.

The remainder of this chapter is structured as follows. In Section 4.1, we provide the foundations of role-based modelling and discuss our motivation to employ roles in language engineering. In Section 4.2, we introduce LanGems, a language composition system based on roles and role-based composition. In Section 4.3, we discuss the realisation of the OCL language family using LanGems. Finally, in Section 4.4, we evaluate the case study, conclude benefits, and discuss open issues for role-based language composition.

4.1. Foundations of Role-Based Modelling

The concept of a role is experienced quite natural in human science, theatre, or reality. Here, a role is considered a particular pattern of behaviour that a person is expected
to comply to, when he acts in a given context. It is important to notice that roles are typically related to other roles, e.g., doctor and patient, or mother, father and child.

In computer science, roles are applied in various disciplines ranging from data modelling (e.g., [Bachman 1977]), to conceptual modelling (e.g., [Halpin 1995]), object-oriented design (e.g., [Reenskaug 1996, Riehle 1998]), and programming languages (e.g., [Smaragdakis 2002, Mezini 2003, Herrmann 2007]). For a detailed review of the history of role modelling, we refer to [Steimann 2000a, Steimann 2000b].

In [Reenskaug 1996], Reenskaug et al. introduce the concept of role models to leverage the design of complex software systems. Role models are used to separate and capture different areas of concern in a software system. Individual roles represent participants in the role model. They specify special place-holders and also an interface that needs to be implemented by role players. Thus, roles can be considered a special kind of type besides, for example, classes as found in object-oriented modelling.

Guarino [Guarino 1992] defines two properties to distinguish role types and natural types (i.e., classes in object-oriented modelling): rigidity and foundedness. A type is founded if its instances need to be related to instances of another type to exist. A type is rigid if it assigns an identity to it’s instances. Role types are considered founded and non-rigid, whereas natural types are considered independent (not founded) and rigid.

In [Andersen 1997], Andersen introduces the technique of role composition to integrate several role models. Role composition takes several role models and a specification of role bindings that interconnect roles with naturals or other roles to derive an integrated system implementation.

The separation of concerns in role models and the synthesis of system implementations by role composition provide a foundation to enable modularisation and integrative composition in a role-based language composition system, respectively. In the following, we discuss how the characteristics of role models and role composition can be exploited to address the lacking support for modularity properties (cf. Section 2.2.1) criticised for current language modularisation approaches. The 15 features for roles identified by Steimann in [Steimann 2000b] can be considered a comprehensive selection of role characteristics found in literature. Thus, our discussion refers to these features (cf. Table 4.1).

4.1.1. Information Hiding and Interface Specification in Role Models

Definitions of roles commonly state that a role always depends on other roles (cf. [Steimann 2000b]). Reenskaug expresses this in defining role models as a pattern of collaborating objects. This characteristic is also reflected by Guarino’s [Guarino 1992] definition of roles as founded types, i.e., types that can only exists in relation to other types. In [Steimann 2000b], this characteristic is described by the role feature (#2) Roles depend on relationships. Each collection of related roles is contained in a so-called context. Each context consists of a self-contained and comprehensive set of roles to specify the implementation of a system from a particular perspective. This implementation needs to be solely defined using the roles that take part in the context. This characteristic is partly reflected by Steimann in feature (#1) A role comes with its own properties and behaviour. As a consequence of this characteristic, direct dependencies between several
4.1. Foundations of Role-Based Modelling

<table>
<thead>
<tr>
<th>Desired Modularity Property</th>
<th>Enabling Feature of Roles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information hiding</td>
<td>(#1) A role comes with its own properties and behaviour; (#2) Roles depend on relationships; (#12) Roles restrict access</td>
</tr>
<tr>
<td>Explicit interface specification</td>
<td>(#12) Roles restrict access</td>
</tr>
<tr>
<td>Loose coupling</td>
<td>(#3) An object may play different roles simultaneously; (#7) Objects of unrelated types can play the same role; (#8) Roles can play roles</td>
</tr>
<tr>
<td>Flexible integration</td>
<td>(#11) Features of an object can be role-specific, (#13) Different roles may share structure and behaviour</td>
</tr>
</tbody>
</table>

Table 4.1.: Mapping of Steimann’s role features (feature numbers refer to [Steimann 2000b]) to modularity properties desired for language modularisation.

role models are not supported. Otherwise the involved types would belong to the same context by definition. Consequently, the implementation of role models does not need to be exposed to the outside which enables information hiding in role models.

As described by Steimann’s feature (#12) Roles restrict access, the roles in a role model can be understood as its sole interface to the outside. They define a contract for potential role players which are bound later during role composition. At the same time, roles restrict the access to the role player. This enables an explicit specification of required and provided interfaces for role models and further contributes to information hiding.

4.1.2. Loose Coupling and Flexible Integration in Role Composition

The technique of role model composition describes the synthesis of an integrated system specification by a superimposition of several role models. In accordance to [Andersen 1997], role composition is realised by composing classes and roles from different role models. The composition of a class and the roles it plays is typically specified using the so-called played-by relation [Steimann 2000b, Herrmann 2007]. This relation connects roles with types from different role models. The Steimann features (#3) An object may play different roles simultaneously, (#7) Objects of unrelated types can play the same role, and (#8) Roles can play roles document the flexible options to specify played-by relations.

Making role model composition an explicit step, reduces the coupling of the individual role models. They more independent of each other and can, thus, be reused more easily.
All their requirements are defined in terms of role-based interfaces that are bound not until role composition.

Furthermore, the composition specifications found in current role-based programming languages provide sophisticated composition operators for invasive integration of roles and role players during role binding. Mezini and Ostermann [Mezini 2003] distinguish the composition operators binding and pointcut as means for integration. Bindings are used to specify the implementation of a role’s required interface by the role player. Pointcuts are used to invasively augment the control flow of a role player. In [Herrmann 2007], Herrmann defines a comparable pair of composition operators: callin and callout binding. The callout binding forwards calls to the role interface to implementations in the role player. Using callin bindings the role is able to intercept and redirect the control flow of the role player.

Steimann’s role features also reflect both composition operators. First, feature (13) Different roles may share structure and behaviour states that the realisation of roles may depend or delegate to attributes and behaviour of the role player. Second, feature (11) Features of an object can be role-specific states that the attributes and behaviour of the role player can be overloaded by the role.

Our analysis of role features and their potential impact on modularity properties in abstract syntax metamodeling is concluded in Table 4.1.

4.2. The LanGems Language Composition System

In this section, we present the fundamental constituents of our role-based language composition system LanGems and its model-driven process for language composition in different technical spaces of language implementation.

Fig. 4.1 depicts the central elements and coarse triple structure of LanGems. Individual language components are specified based on the Language Component Specification Language (LCSL) using the concepts of LanGems component model. Every language component specifies its abstract syntax, concrete syntax and semantics. The composition of several language components is specified in a composition program formulated in the Language Composition Language (LCL). This program is evaluated by a composition workflow that implements the language composition technique and generates an integrated language. The integrated language consists of a combined abstract syntax, a composed concrete syntax, and a composed semantics.

4.2.1. The Language Component Specification Language

The central artefact of every language component is its abstract syntax metamodel. The aim of this metamodel in language implementation is twofold. First, it provides a data structure to syntactically represent language expressions parsed from their representation.
4.2. The LanGems Language Composition System

in concrete syntax. Second, it needs to expose this representation to enable a static and dynamic evaluation of language semantics [Selic 2010].

To realise these requirements, metamodelling languages contribute concepts to specify the syntactic and semantic interface of a language metamodel (cf. Fig. 4.2). When transforming concrete syntax to abstract syntax, the respective tool (i.e., parser) writes against the syntactic metamodel interface. A semantics analyser reads from the syntactic interface to implement the semantic interface of the language metamodel. Furthermore, Within a language component it is, thus, necessary to expose the full semantic and syntactic interface.

When composing several language metamodels to evolve, customise, or combine existing languages, the syntactic and semantic interfaces of the involved languages need to be integrated. As discussed in Section 2.2.1, existing techniques to metamodel modularisation and composition do not provide means to restrict access on these interfaces and, thus, break the principle of information hiding between language modules. In the previous section, we identified a number of features for roles that suggest their potential to alleviate these drawbacks. For the purpose of exploiting these features for a definition of self-contained, extensible language components, we introduces the LCSL—a role-based metamodelling language in this section. Fig. 4.3 illustrates the contribution of roles for introducing explicit provided and required interfaces between language components. Current metamodelling approaches allow for arbitrary ways of overwriting and delegation.

Figure 4.1.: Overview of the LanGems language composition system.

Figure 4.2.: Syntactic and semantic interface of the language metamodel and their relation to concrete syntax and semantics.
when combining language modules. In contrast, role-based metamodels enforce language engineers to use explicit interfaces during language composition.

LCSL extends the object-oriented metamodeling language Ecore provided by EMF [Steinberg 2008]. We selected Ecore as foundation to evaluate the feasibility of implementing LCSL since Ecore is widely applied in industry and research and comes with an exhaustive infrastructure of tools for model transformation and validation. However, the results of our feasibility study can be mapped to other metamodeling languages, as Ecore shares the expressiveness of common metamodeling approaches (cf. Section 2.2.1). In the following, we sketch the path from object-oriented metamodeling to role-based metamodeling. Therefore, we discuss means to define syntactic and semantic interfaces in existing metamodeling approaches and describe the introduction of provided and required interfaces with role-based metamodeling.

**EPackages, EClassifiers, and EFeatures in Object-oriented Metamodelling**

Fig. 4.4 depicts a detailed excerpt of the Ecore metamodel. It contributes the metamodeling concepts typically found in current object-oriented metamodeling approaches.

Each language metamodel built with Ecore consists of an EPackage describing a module that contains all (eClassifiers) types (called EClassifier in Ecore) of that language. Ecore distinguishes different kinds of types (EClassifier). Primitive datatypes can be introduced using EDataType, instantiable concepts are specified using EClasses. An EClass can define a number of structural features (EStructuralFeature) and operations (EOperation).

The structural features describe the syntactic interface of an EClass, i.e., properties
4.2. The LanGems Language Composition System

 statically specified by a language user when writing language expressions in concrete syntax. Ecore distinguishes features with a primitive type (EAttributes) and features that allow for defining references (EReference) among EClasses. Both EAttributes and EReferences can define a lowerBound and a upperBound attribute to specify their cardinality (inherited from (ETypedElement)). The source type of an EReference is defined by the containing EClass, the target type by the EClass given as eReferenceType (derived from eType of ETypedElement). Ecore uses a boolean feature (containment) to distinguish containment references that have a part-of semantics and non-containment references.

EOperations define the semantic interface of an EClass, i.e., behaviour or semantics of the language that is dynamically derived from the static properties. The specified semantic interface is expected to be implemented using a concrete semantics formalism. Each EOperation is an ETypedElement. The associated eType specifies the respective return type. Furthermore, operations can specify a number of typed EParameters.
Naturals, Roles, Role Features, Role Operations and Augmentations in Role-Based Metamodelling

Fig. 4.5 shows the metamodel of the LCSL. It describes an extension of the Ecore language to enable role-based metamodelling.

The Component concept is introduced as a refinement of EPackage. Each Component can again contain a set of eClassifiers describing the concepts of the respective language component. LCSL introduces the concept Type as subclass of EClass. It is used as common super class to distinguish Natural types and Role types in a language component.

Natural types inherit the attributes and features of EClasses and are used to represent rigid, non-founded concepts as conventionally used in object-oriented metamodelling. Their behaviour and structure is completely defined within the language component. Naturals contribute to the provided syntactic and semantic interface of the language component, i.e., the types that can be accessed and used as role players during language composition.

Roles also inherit attributes of EClass from the ecore metamodel. They use role features (eStructuralFeatures inherited from EClass) to define the parts of the role’s syntactic interface to be provided by the role player. Furthermore, they contribute means to specify roleOperations and augmentations. Both RoleOperations and Augmentations can be used to declare parts of a role’s behaviour to be provided by the role player. Roles, thus, define the required syntactic and semantic interface of the
4.2. The LanGems Language Composition System

Figure 4.6.: Control flow between integrated language components for role operation and augmentation bindings.

language component, i.e., the types that need to be bound to naturals of other language components to complete the roles behaviour.

A RoleOperation prepares a callout binding where the control flow is delegated from the role to the role player. RoleOperations contribute to the role’s interface. First, they can be accessed internally within the language component to define its semantics. Second, the role operations introduce a contract that has to be realised by each role player.

An Augmentation prepares a callin binding during language composition. In a callin binding, the control flow of the role player is intercepted by the role and augmented with additional behaviour. Both RoleOperations and Augimations are ETypedElements and define a number of parameters. Thus, they can specify their return type and a number of input parameters, respectively.

The difference between the callout binding and the callin binding is illustrated in Fig. 4.6. For the callout binding using RoleOperations, the role delegates the control flow to the role player. In the callin binding, the control flow of the role player is intercepted and redirected to the role.
Example of LCSL Application

Listing 4.1 demonstrates the application of LCSL for the specification of an exemplary language component. It uses a textual concrete syntax for LCSL built with EMFText. The listing gives a role-based specification of the abstract syntax for a language component to define statecharts.

A **Statechart** provide means to define the behaviour of a system in terms of **States** and state **Transitions**. Each state has a **stateName**. To define the **States** behaviour, an **entry Action**, an **exit Action**, and a **doActivity** can be defined. **Actions** describe atomic behavioural units that cannot be interrupted and do not affect the statecharts overall behaviour. **Activitys** are long running behavioural units. Their execution returns a result in terms of a **Trigger** that further influences the statecharts evaluation. Therefore, each **State** can have a number of outgoing transitions (**out**). **Transitions** describe changes between two states (**source, target**) that are activated by a concrete **trigger**.

The types **Statechart**, **Element**, **State**, and **Transition** are specified as **naturals**. This means, that they contribute to the provided syntactic and semantic interface of the statechart language component. In contrast, **Action**, **Activity**, and **Trigger** are **role** types. They define the required interface of the language component and need to be bound during language composition. They refine the structural and semantic requirements for role players using role features and operations. Each player of the role **Action** needs to bind the role operation **run()**. Each **Trigger** player needs to provide a name (**getTriggerName()**). **Activity** players need to provide a name (**getActivityName()**), a set of triggers its evaluation may return (**triggers**) and an evaluation behaviour (**evaluate**) that returns a concrete **Trigger**.

4.2.2. The Language Composition Language

To enable the composition of several language components to an integrated language, LanGems introduces the composition language LCL. In this section, we first introduce the design of LCL and then demonstrate its application.

Role, Role Feature, Role Operation, and Augmentation Bindings in Role-Based Language Composition

Fig. 4.7 depicts the LCL metamodel. It is connected with the metamodel for the LCSL to describe the combination of several language components using the concepts defined in their specification.

Every composition program defines a **Composer**. The **Composer** declares a concrete language component as integrating component. This will be the component that provides the root element of the integrated language metamodel. The **Composer** consists of a number of **compositions**.
4.2. The LanGems Language Composition System

```java
component statechart {
    natural Statechart {
        attribute EString chartname (1..1);
        containment reference Element elements (0..-1) opposite container;
        reference State init (1..1);
        reference State end (1..-1);
    }
    abstract natural Element {
        reference Statechart container (1..1) opposite elements;
    }
    natural State extends Element {
        attribute EString stateName (1..1);
        reference Transition out (0..-1) opposite source;
        reference Transition in (0..-1) opposite target;
        containment reference Action entry (0..1);
        containment reference Action exit (0..1);
        containment reference Activity doActivity (0..1);
    }
    natural Transition extends Element {
        reference State source (1..1) opposite out;
        reference State target (1..1) opposite in;
        containment reference Action action (0..1);
        operation void (0..1) evaluate ();
    }
    role Action {
        roleOperation void (0..1) run();
    }
    role Activity {
        roleOperation EString (1..1) getActivityName();
        roleOperation void setName(EString name);
        roleOperation Trigger (0..-1) getTriggers();
        roleOperation Trigger (1..1) evaluate();
    }
    role Trigger {
        roleOperation EString (1..1) getTriggerName();
    }
}
```

Listing 4.1: Role-based specification of statechart language component in LCSL.
Chapter 4. Integrative, Role-Based Composition for Language Family Engineering

Figure 4.7.: Metamodel of the LanGems LCL.
4.2. The LanGems Language Composition System

Each Composition describes the integration of two language components. The role types of the extendedComponent define the required interface of the composition and the types of the extendingComponent describe the provided interface. Both interfaces have to adapted for integration. Therefore, the LCL provides roleBindings.

Each RoleBinding imposes a played-by relation between a player Type of the extending component and a role of the extended component. The Type given as player can either be a Natural or a Role. This allows for roles playing roles. RoleBindings can define an optional Restriction that checks role players at runtime for their ability for playing the role they are bound to. Means to define such restrictions depend on the semantics formalism used in the SemanticBinding. For a more detailed discussion, we refer to Section 4.2.3. In a role binding, augmentationBindings for the augmentations and roleElementBindings for role elements defined in the bound role can be specified.

For RoleElementBindings, the LCL distinguishes RoleOperationBindings and RoleFeatureBindings. In a RoleOperationBinding, we specify how a RoleOperation is realised using the syntactic and semantic interface of the role player. The means used for specifying this integration depend on the semantics formalism used in the SemanticBinding. For a more detailed discussion, we refer to Section 4.2.3. RoleFeatureBindings describe the implementation of the features contained in the roles syntactic interface using the role player. First, RoleFeatureBindings can simply delegate to a feature of the role player (NaturalFeatureDelegation). Second, a so-called GetSetBinding can be used, where the get and set semantics of the bound feature are specified using a SemanticBinding.

A NaturalAugmentation describes how a role intercepts the control flow of the role player. An augmentation can be considered an aspect advice [Filman 2004] that intercepts the control flow of the role player using its syntactic or semantic interface. To specify the concrete join point in the control flow to be augmented, each NaturalAugmentation refers to an EOperation of the role player.

We distinguish before and after bindings. As depicted in Fig. 4.6, a before binding is evaluated before the augmented EOperation of the role player is executed. Before bindings allow for inspecting and adapting the input data of the augmented operation. In addition, they can access the syntactic and semantic interface of the role player. Using these interfaces AugmentationParameterBindings specify the binding of the input parameters of the Augmentation. On the other hand, after bindings are evaluated after the augmented EOperation is evaluated. They can access the input data (AugmentationParameterBindings) and adapt the result data before the augmented operation call is returned to the caller. Means to describe the concrete binding semantics (SemanticBinding) for input and output data are discussed in Section 4.2.3.

Example of LCL Application

In Listing 4.2, we introduce a second language component called form. It is meant to be composed with the statechart component introduced in Listing 4.1. Both are combined to a simple composed language to specify form-based wizard dialogues. The basic idea is to use statechart to describe the transition between different pages of the dialogue.
Chapter 4. Integrative, Role-Based Composition for Language Family Engineering

and to use form to describe the outline of an individual page.

The design of the form component in Listing 4.2 is very simple. A Form consists of a number of Fields to enter data. Each field has a name and can be of a special kind, e.g., TextField or Selection. Furthermore, a Form declares a number of Buttons. The number of available fields or the means for layouting forms could easily extended for the form language component, but is considered out of scope for our illustrative example.

Listing 4.3 shows the composition program that integrates the form and the statechart components. It first imports the required language components and assigns them short, symbolic names within the composition program (Lines 3-4). Line 6 declares the statechart component as the integrating one. This means that statecharts are considered to contribute the core for the integrated language. Next, the composition of both components is defined (Lines 8-20). The form component refines the statechart component by binding two of its roles. First, the natural Form is bound to the role Activity. Intuitively, this means the activity that is executed within a State of the statechart can be specified by describing a form which is meant to be rendered. The second role binding assigns Button to the Trigger role. This means that a Button specified within a form controls the transition between two states of the formFlow statechart. Again this realises
4.2. The LanGems Language Composition System

```java
composer

import <platform:/resource/de.tudresden.lm.form/model/form.mdl> as <form>
import <platform:/resource/de.tudresden.lm.statechart/model/statechart.mdl> as <statechart>

{ integrating: <statechart>
  
  <form> refines <statechart> {
    Form plays Activity {
      getActivityName() : «player.getHeading()»;
      evaluate() : «player.open()»;
      setActivityName(EString name) : «player.setHeading(name)»;
      triggers : get: «player.getButtons()»;
    }
    
    Button plays Trigger
    when «player.getButtonName().length() > 0» {
      getTriggerName() : «player.getButtonName()»;
    }
  }

Listing 4.3: Role-based composition of formFlow language in LCL.
```

the intuitive semantics of form-based wizard dialogues.

Both role bindings employ role feature and role operation bindings to describe the implementation of the role interface by the respective role player. In our example, we provided a Java-based implementation of the operational semantics for both language components. Their semantic integration relies on fragments of Java code that describe the SemanticBinding. To be able to describe such bindings, we derived a dialect of the LCL that contributes JavaBindings as subclass of SemanticBinding (cf. Fig. 4.7).

A JavaBinding holds a fragment of Java code that specifies glue code to realise role operations or features in the context of the role player. To enable role bindings using a different semantics formalism, custom subclasses of SemanticBinding and custom composition operators need to be contributed (cf. Section 4.2.3).

For an example of Java-based semantics binding see the Lines 10-13 in Listing 4.3. The role operations for getting and setting the name of the role Activity are realised by delegating to the getter and setter for the heading of the role playing Form. The triggers an Activity needs to declare, are given by returning all Buttons defined in the respective Form. Finally, the evaluate() operation is bound by referring to the method open that actually renders the form.

An example for an augmentation binding is presented in Listing 4.4. It integrates the statechart component with the logger component introduced in Listing 4.5. The logger component defines a role LoggingArtifact that can be assigned to language concepts that are meant to be augmented by some simple logging behaviour. In our example, such logging behaviour is specified in the augmentation doLog(EString message) that prints a simple log message. An exemplary role binding for LoggingArtifact to the State of a statechart is given in Lines 7-13 of Listing 4.4. The natural augmentation in this binding describes a before binding of the augmentation wrt. the oper-
Chapter 4. Integrative, Role-Based Composition for Language Family Engineering

Listing 4.4: Role-based composition of logging for formFlow language in LCL.

```java
import <platform:/resource/de.tudresden.lm.form/model/form.mdl> as <form>
import <platform:/resource/de.tudresden.lm.statechart/model/logger.mdl> as <logger>

{ <statechart> refines <logger> {
    State plays LoggingArtifact {
        doLog(EString) before evaluate() 
        augmentation parameters {
            message : "evaluating " + player.getStateName();
        }
    }
}
```

Listing 4.5: Role-based specification of logger language component in LCSL.

```java
module logger {
    role LoggingArtifact {
        augmentation void doLog(EString message);
    }
}
```

ation `evaluate()` defined for the role player. This means, before every execution of `evaluate()` the augmentation `doLog()` is evaluated. In the augmentation parameter binding for the augmentation parameter `message`, we specify that the log message is a concatenation of the String “`evaluating `” and the `stateName` of the role playing `State`.

In our illustrative examples, the semantics bindings are quite simple. In practice, more complex bindings might be necessary. As this chapter introduces an approach to use Java, a full-fledged GPL, for specification of semantics bindings even such complex integrations can be supported.

4.2.3. Techniques of Language Composition

Given the languages LCSL and LCL, the last missing piece of a language composition system is a composition engine. This composition engine is meant to derive an integrated language for a set of role-based language components and a composition program.

Language composition has to deal with the different language specification artefacts. First, the role-based abstract syntax metamodels have to be integrated. Second, concrete syntax specifications and semantics specifications have to be composed, if given for the language components. All compositions are driven by the role-based composition program. However, the different specification formalisms and their respective composition rules require specialised composition techniques.

In the following, we discuss a model-driven language composition system that provides an extensible infrastructure for implementing custom composition techniques for specific language specification artefacts. This is meant to prove the feasibility of role-based com-

110
4.2. The LanGems Language Composition System

position for abstract syntax, concrete syntax and semantics composition. The selection of concrete specification formalisms for this feasibility study is based on our review of state-of-the-art in language engineering in Chapter 2.

LanGems Language Composition Workflow

In the language composition system LanGems, the derivation of an integrated language from a set of language components is considered to be a workflow consisting of several composition processes. Each process automates a particular part of the composition technique, deals with a particular set of artefacts, and may be optional or mandatory. In the following, we use the BPMN [BPM 2009] to describe the overall workflow and individual workflow tasks.

Fig. 4.8 gives an overview of the general LanGems language composition process. It is implemented by the LanGems Composition Workflow that is depicted as complex BPMN process in the lower part of Fig. 4.8. The upper part of the figure shows a repository with a set of role-based language components (language component₁-language componentₙ) and a Role-Based Composition Program that describes their integration. The composition of the language components is realised in a dedicated «Specification» Composition Process for each kind of language specification and specification formalism used. That means, concrete syntax specified in the EMFText formalism is composed in a dedicated EMFText Concrete Syntax Composition Process, abstract syntax that is realised on an EMF foundation in a dedicated EMF Abstract Syntax Composition Process, and so on.

The «Specification» Composition Process presented in Fig. 4.8 describes a general pattern for such concrete processes. Each composition process consists of at least two sub-tasks. A «Specification» Composition Task that realises the actual composition technique and a «Specification» Generation Task that maps a composed language specification to an executable Specification Implementation based on a specific Implementation Platform. The «Specification» Composition Task is configured with the Role-based Composition Program and a set of language specifications (language specification₁-language specificationₙ) that are extracted from the language components. It implements a composition pattern that describes how the component integrations described in the composition program are realised for the respective language specification formalism. As depicted in Fig. 4.8, the application of this pattern results in specification glue that composes the individual language specifications. Next, the composed language specifications are sent to the «Specification» Generation Task. This task realises a platform mapping that describes how the specifications are transformed to an executable specification.

Fig. 4.9 depicts an exemplary LanGems Composition Workflow with three instances of the «Specification» Composition Process for composing abstract syntax using EMF, concrete syntax using EMFText, and language semantics using Java operational semantics, respectively. When the «Specification» Composition Process is instantiated for a concrete kind of language specification, e.g., EMFText concrete syntax specifications, the «Specification» Generation Task typically calls the generators pro-
Chapter 4. Integrative, Role-Based Composition for Language Family Engineering

Figure 4.8.: Overview of the LanGems composition process.
4.2. The LanGems Language Composition System

Figure 4.9.: Exemplary instance of LanGems composition process.
Chapter 4. Integrative, Role-Based Composition for Language Family Engineering

Figure 4.10.: Process for EMF abstract syntax composition in LanGems.

provided for the respective kind of specification, e.g., the EMFText Parser generator. In an instance of the presented composition process pattern the two task can also be further refined by a set of sub-task (cf. Fig. 4.10). Furthermore, there might be interdependencies between concrete «Specification» Composition Processes. For instance, the EMFText Concrete Syntax Composition Process depends on the EMF Abstract Syntax Composition Process, as parser generation requires a composed abstract syntax metamodel.

To evaluate the feasibility of implementing the envisioned role-based language composition system, we implemented a set of concrete composition processes for selected language specification formalisms in LanGems. These will be discussed in the following. In general, LanGems is not limited to these specification formalisms, but can be extended with additional ones by contributing a new custom «Specification» Generation Task.

Composition of Abstract Syntax

The first vital part of the LanGems language composition workflow is the derivation of an integrated abstract syntax metamodel from a set of role-based language components. As depicted in Fig. 4.10, this is done by three mandatory tasks. In the following, we discuss the objective and realisation of each task.

Role Normalisation During Role Normalisation, the role-based abstract syntax metamodel for each language component is transformed to a normalised object-oriented metamodel specified in Ecore. Naturals, their features and operations are transformed to conventional EClasses. All Roles are transformed to abstract EClasses. Their RoleOperations are transformed to conventional EOperations with empty bodies. Implementing this body is done during role binding when the semantics of RoleOperations is realised by the role player. Augmentations are also transformed to
4.2. The LanGems Language Composition System

empty EOperations. These operations are meant to already be implemented during the
design of a concrete language component. As discussed previously, the binding of an aug-
mentation to the control flow of the role player is done not until role binding. The role
normalisation results in a normalised Ecore metamodel for every language component
that was fed to the composition workflow.

**Metamodel Integration** During *Metamodel Integration* the normalised metamodels
are integrated to a combined metamodel. The language component that is marked as
integrating component in the composition program (cf. Fig. 4.5) contributes the root
element for the integrated metamodel.

Each role binding is mapped to a subclass relationship from the EClass that represents
the natural to the EClass that represents the role. The result of metamodel integration is
a metamodel consisting of the individual metamodels derived for all language components
and connected by the generated subclass relationships.

**EMF Implementation Generation** For the generation of a Java-based implemen-
tation from the integrated metamodel, we employ the code generator provided by the
EMF. It evaluates a given Ecore-based metamodel specification and derives a set of Java
interfaces and classes that implement an API to create, represent, access, serialise, and
de-serialise concrete model instances. EClasses are mapped to a Java interface and a
Java classes. The Java interface provides a declaration of the syntactic and semantic
interface of the EClass. The Java class provides the implementation of this interface.
EStructuralFeatures are represented by Java fields with getter and setter methods.
EOperations are mapped to Java methods with the respective parameters and return
type. Their method body is left empty and throws an UnsupportedOperationException,
if called. The conventional way of providing semantics for Ecore-based languages is to
implement these method bodies and mark them as not generated [Steinberg 2008]. This
ensures that the method body is not overwritten by following runs of the EMF code
generator. In Section 4.2.3, we discuss how this approach is extended for language com-
ponents.

Fig. 4.11 depicts a pattern to illustrate the effect of the discussed EMF composition
steps. The pattern shows the classes, interfaces and operations generated from the nor-
malised metamodels and the subtype relationship introduced during metamodel integra-
tion. This pattern also illustrates that we require an invasive integration technique, as the
type hierarchy and the features of the involved classes are changed during composition.

Using the presented pattern the process of *role composition* derives an object-oriented
implementation of the composed metamodel that statically realises the role bindings
specified in the composition program. Fig. 4.12 depicts the EMF-based metamodel de-
rived from the formFlow language described in Listing 4.3. It consists of an individual
package for each normalised language component. In addition, it shows the generated
subclass relationships (e.g., Button -> Trigger, Form -> Activity).

**Composition of Concrete Syntax**

Next, we discuss the composition of concrete syntax. To demonstrate the universality of
role-based language composition and to continue our model-driven approach to language
Figure 4.11.: Integrative composition pattern for EMF to implement statics of role composition.

Composing Textual Concrete Syntax  
Our implementation of a composition technique for concrete syntax specifications in LFE is based on EMFText. As discussed in detail in Sect. 2.2, we selected EMFText due to its tight integration with EMF and the sophisticated means to generate advanced language tooling for Eclipse. However, the presented approach can also be transferred to other concrete syntax formalisms if needed.

As depicted in Fig. 4.13, we extended the model-driven parser generation process of EMFText with additional tasks that implement an integrative concrete syntax composition technique. In the following, we discuss each task in detail.

Core Syntax Initialisation  
The task of core syntax initialisation creates a new model-based EMFText syntax specification for the integrated metamodel that was derived during abstract syntax composition. The start symbol for the core syntax specification is derived from the integrating language component specified in the composition program.

Syntax Weaving  
The task of syntax weaving evaluates the individual syntax specifications given for each language component and invasively integrates their token defini-
4.2. The LanGems Language Composition System

During the integration of token definitions, they are checked for overlaps. When two token definitions are equal, they are unified. If one token definition is more specific than the other, the more specific token definition is prioritised, i.e., the scanner will try the more specific token definition first. This avoids unreachable token definitions. As such overlaps restrict the tokens that are matched to the less specific token definition, a warning is provided that prepares a manual overlap treatment. If two token definitions partially overlap, but none is completely included in the other, no automatic ordering is possible. In such cases, an error is annotated that forces language developers to manually resolve the conflicts.

For the integration of production rules, the rule sets of all languages are combined. Role bindings that are mapped to subclass relationships in the integrated metamodel result in alternative rules in the generated grammar. ANTLR employs a LL(*) parsing algorithm with backtracking and memoisation, and prioritises alternatives in the order of their
Chapter 4. Integrative, Role-Based Composition for Language Family Engineering

Figure 4.13.: Process for EMFText textual concrete syntax composition in LanGems.

definition. This reduces the threat of ambiguity when combining the sets of production rules. Furthermore, EMFText provides a set of checks that detect ambiguities or left-recursive rules and generates warning and error messages that help language engineers to detect and resolve such issues (cf. Section 2.2.2).

EMFText Parser Generation For generating a parser and editor implementation from the integrated language, the EMFText parser generation task first derives an ANTLR grammar from the composed syntax specification. This grammar is enriched with semantic actions that construct instances of the integrated metamodel during the parsing process. The complete grammar is fed to the ANTLR parser generator that derives a Java-based parser implementation.

EMFText also generates a plethora of Java classes that implement a sophisticated editor implementation for the integrated language. The editor comes with syntax checking, syntax colouring, code completion, text hovers, and an outline view. Furthermore, a printer is generated that transforms a given metamodel instance to its textual representation. To further enhance the language tooling, the generated implementation provides stub classes and extension points that enable the contribution of custom semantics checkers, quick fixes, interpreters, and debuggers.
4.2. The LanGems Language Composition System

The discussed tasks are again implemented using model transformation and code generation techniques. They are added as optional compound process to the LanGems composition workflow. Fig. 4.14 depicts the Eclipse editor derived for the composed formFlow syntax. A wizard specification in formFlow is based on a statechart that describes the general page flow. In the states of this statechart, individual forms are described. Transitions refer to buttons of these forms to specify changes between the wizard’s pages depending on the pressed button. In the example, two states (init and data) are declared. Both contain a form specification. The transition in Line 16 is triggered when the Login button in the init form is pressed. It changes the wizard page to data. The editor helps editing formFlow specifications by sophisticated functions like code completion and occurrence highlighting. These are fully generated from the composed metamodel and syntax specification.

Composing Graphical Concrete Syntax  As technical foundation to implement a composition technique for graphical concrete syntax, we selected the GMF. The GMF implements a model-driven approach for the generation of graphical editors for EMF models. As indicated in Section 2.2.2, it contributes three languages to specify different aspects of the editor implementation. Fig. 4.15 depict show concepts of these languages map to elements of an exemplary editor for CPNs. In the following, we introduce the three GMF
specification languages and their concepts in detail:

**GMFgraph Language** The GMFgraph language enables the definition of an alphabet of graphical notations that can be used to visualise model elements. In general, it is possible to define node and connection figures. Node figures are arranged on the editor canvas and represent a single model element. They can be of different shapes, colours and styles. Furthermore, it is possible to add labels to node figures or nest node figures using compartments. Connection figures define links between node figures and/or other connection figures. They also support different styles and decorations (e.g., arrows or labels).

**GMFtool Language** The GMFtool language can be used to specify the editor’s tool palette. So-called creation tools create new graphical elements on the editor canvas and instantiate the respective model elements. For each creation tool a name, description and icon can specified. In addition, creation tools can be arranged in tool groups.

**GMFmap Language** The GMFmap language provides means to interconnect the previous specifications and the language metamodel. Metamodel elements can be mapped to creation tools and graphical notations defined in the GMFgraph language.

Each GMFmap model defines a canvas mapping that refers to the canvas specified in a GMFgraph model and the EClass defining the root of a model instance.

For mapping classes to nodes, node references are used. The GMFmap language distinguishes top node references and child references. Top node references define nodes which can be placed directly on the canvas. Child references describe nodes contained in compartments of other nodes. Each node reference contains a node mapping that refers to the EClass of the metamodel that is visualised by the respective node. Such node mappings can contain feature label mappings that describe how EStructuralFeatures of the visualised class are used to provide node labels.

For mapping connections to metamodel classes and their features, link mappings are used. Each link mapping refers to a connection defined in the GMFgraph model. Connections can visualise both EClasses or EStructuralFeatures. When visualising EClasses, a containment EStructuralFeature that refers to the respective EClass needs to be given. Then, the EStructuralFeatures of the EClass that define the source and target of the connection need to be specified. When visualising a concrete EStructuralFeature, it is sufficient to refer to the feature that is meant to be visualised. The connection’s source element will be the containing class and the target element will be the value of the feature. Link mappings can also contain label mappings.

As discussed in Section 2.2, similar concepts like the definition of a visual alphabet (GMFgraph model) and its mapping to a domain model (GMFmap) model can be found
4.2. The LanGems Language Composition System

Figure 4.15.: Concepts of GMF specification languages mapped to an exemplary GMF editor.
in most approaches for graphical syntax specification. The technique presented here, can, thus, also be generalised to different formalism.

The GMF composition process (cf. Fig. 4.16) contributes four tasks. These implement integrative, role-based composition of GMF specifications in language families. In the following, we discuss details of each task.

**GMFgraph Weaver** The GMFgraph weaver task evaluates the composition program to retrieve all involved language components. Then, it searches for their respective GMFgraph specification and copies all graphical notations of the involved language components to a new, combined GMFgraph model.

Role types define place-holders that are bound during language composition. That means their graphical representation is defined not until language composition. However,
within the context of a language component, roles refer to other types. Such references require a syntactic representation. With respect to the notations available for GMF diagrams two different ways for representing such references are possible: connections and compartments. The decision which kind of visualisation is used to represent references to role types is considered part of the syntax definition of the language component defining the role. Consequently, a role-based language component needs to define a connection or a compartment placeholder that prescribe the representation of references to the role (and role player). These placeholders are also copied to the integrated GMFgraph model. As the mapping of graphical notations to metamodel concepts is specified in the GMFmap model, the further handling of such placeholders is discussed in the GMFmap weaver task.

**GMFtool Weaver** The GMFtool weaver task locates the GMFtool specifications for all language components involved in the composition program. It then creates a new GMFtool specification that contains a separate tool group for each language component and is filled with copies of the respective creation tools. The connection of creation tools to metamodel classes is handled in the GMFmap weaver task.

**GMFmap Weaver** The GMFmap weaver has to realise the combination of the integrated language metamodel, the GMFmap models of all involved language components, the composed GMFgraph specification, and the composed GMFtool specification. Therefore, it first locates the GMFmap models of all language components referenced in the composition program and copies their contents into a newly created GMFmap model. References to metamodel elements of the individual language components are replaced by the corresponding references to the integrated metamodel. Also references to the original GMFgraph and GMFtool specifications are replaced by references to the composed GMFgraph and GMFtool specifications.

Finally, the role bindings specified in the composition program need to be reflected in the composed GMFmap model. With respect to the two options for expressing references to role types, two different approaches are used to implement role bindings. For role bindings that refer to roles with a compartment placeholder, a new child node mapping for the role player is added to the respective compartment placeholder. For role types with a connection placeholder, a new top node reference is created that enables the creation of instances for the role player on the editor canvas. The subtype-relationship between the role and the role player in the integrated metamodel ensures that the defined connection placeholder is applicable as visualisation for references to role players.

**GMF Editor Generation** The composed GMFgraph, GMFtool, and GMFmap specifications are evaluated by the GMF editor generation task. It derives the implementation of an integrated and feature-rich graphical editor for the integrated language.

All introduced weaving and generation tasks are implemented as transformations of the model-based concrete syntax specifications used in GMF. They are added as optional compound process to the LanGems composition workflow.

Fig. 4.17 depicts a graphical Eclipse editor for CPNs [Jensen 1987, Jensen 2007] composed from three language components (cf. Listing 4.6). The first language component contributes a language to declare and define functions. The second language component defines a basic syntax for simple petri nets consisting of places, transitions and arcs.
Figure 4.17.: Example of GMF graphical editor composition for CPNs.
4.2. The LanGems Language Composition System

The third language component provides means to define types (petri net colours) using a graphical notation.

The composition program depicted in Listing 4.6 describes the required role bindings. First, it imports all three language components. The `petrinet` is declared as integrating component as it contributes the central graphical concepts for the composed language. Next, the natural `TypedElement` from the respective language component is bound to the role `Token` defined in the `petrinet` component. This binding enables a definition of coloured (typed) places in the composed language. Finally, the natural `Function` is bound to the role `ArcAnnotation` which enables the annotation of petri nets arcs with function definitions. The graphical notation for the natural types is reused for the composed language. The way these notations are embedded (as compartment or connection) is derived from the type of embedding defined for the respective role type.

Composition of Semantics

In this section, we investigate the composition of language semantics. We discuss the realisation of a model-driven composition technique for Java operational semantics to demonstrate the universality of role-based language composition.

The conventional, straight-forward approach for implementing semantics for Ecore-based languages is their operational implementation using Java. Therefore, the Java classes that are generated from the normalised metamodel of a language component are extended by operations that pragmatically implement semantics evaluation. As both role operations and augmentations are transformed to plain operations, they seamlessly integrate in this way of semantics implementation. Fig. 4.18 depicts the tasks for Java operational semantics composition. Their details are discussed in the following:

**Component Semantics Propagation** The first subtask deals with the combination of the operational semantics specified for the individual language components. As discussed above, we assume that all language semantics are specified in operations of the
Java classes implementing the individual language components. To prepare the integration of these semantics, these classes have to be physically merged with the classes of the integrated metamodel implementation. To do so, we just copy the Java classes of the individual language components to a common directory and generate the integrated metamodel implementation to the same directory. The merging facilities of the EMF code generator ensure, that all manually implemented operations are preserved during regeneration. The result of this task is a combined component semantics implementation where the individual semantics implementations are placed side-by-side, but are not yet integrated.

**Semantics Integration** The objective of the semantics integration task is to integrate the combined component semantics as specified by the role, role feature, role operation, and augmentation bindings in the language composition program. The dynamics of role binding require an invasive extension of the role and the role players with fields and operations to manage their dynamic association. For role operation bindings, the class of the role player needs to be extended with the respective binding semantics. For augmentation bindings, the control flow of the role player needs to be intercepted and redirected to the augmentation. All these extensions require an invasive, crosscutting adaptation of the existing Java classes. Therefore, we employ AspectJ [Kiczales 2001] as implementation technology.

The application of Aspect-Oriented Programming (AOP) [Filman 2004] as programming paradigm is discussed quite controversially by some authors [Steimann 2006, Apel 2008]. First, there is only a small number of motivating applications for aspects (cf. [Steimann 2006, Steimann 2005]). Second, languages like AspectJ can be used to break encapsulation and, thus, information hiding between modules. In addition, they can obfuscate the implementation of a program by using implicit invocation mechanisms. As we use AspectJ just as an implementation technology, we argue, that these issues can be disregarded. First, the application of AspectJ is well defined. We use aspects to augment and integrate the control flow of role-based language components. Second, the problems of broken module encapsulation are reduced by the restrictions imposed
by the component and composition languages used in our approach. They operate on well-defined component interfaces and aspects are only used as technique for the implementation of the invasive composition semantics defined for role superimposition. Third, the problem of obfuscating the implementation is alleviated, as the application of AspectJ is limited to the technical implementation of role composition. The aspects are generated and affect generated metamodel code. In the following, we discuss the AspectJ glue patterns used to implement a composition technique for Java operational semantics.

Listing 4.7 shows the pattern used to generate an aspect declaration for a given role binding. The listing uses the place holders «role» and «rolePlayer» to refer to the respective concepts of the role binding. The aspect is declared in an integration package whose name is composed from the package name of the role player and the role (Line 1). The aspect declaration imports the packages containing the metamodel interfaces and the metamodel implementation code for both the role player's and the role's language component (Lines 3-6). To manage the role instance, the aspect declares a private field in the role player’s implementation class (Line 11). A corresponding getter enables access to the role (Lines 12-19). Before returning the role instance, it uses (Line 13) the method `canPlay «Role»()` (Lines 29-32) to check the Restriction, if defined in the role binding. To manage access to the role player a second method is generated (Lines 21-28). It again checks the restriction defined for the role binding. If the restriction holds, it returns the instance object (this) of the role playing class. The role player contributes rigidity (identity) for the role.

Listing 4.8 shows the pattern used to implement role operation bindings. It contributes a method implementing the respective role method in the Java class generated for the role player. Within that method, the syntactic and semantic interface of the role player can be accessed to describe the semantic binding of the role operation. The means provided for such description depend on the semantics formalism used in language implementation. In case of Java-based semantics binding, a fragment of Java code is used. To implement the binding, this code is inserted (Line 4). Finally, the types of the semantic binding might be adapted to the types expected for the role operation. This is done using a helper class (Line 5). It casts objects in accordance to the subtype relationships specified by the language components and role bindings. Role feature bindings are implemented by contributing the respective getter and setter operations using a similar pattern.

Listing 4.9 shows the pattern to implement before bindings for augmentations. It uses an around advice that intercepts the augmented operation before it is executed (Lines 3-4). The target object is stored in the variable player (Line 5), all arguments to the augmented operation in respective variables (Line 6). These variables can be used to define a binding of parameters for the augmentation (Lines 7-8). In case of Java operational semantics, this binding is again defined using fragments of Java code. In before bindings, the call to the augmentation (Lines 10-11) precedes the call to the augmented operation (Lines 12-13).

Listing 4.10 shows the pattern to implement after bindings for augmentations. It also uses an around advice that intercepts the execution of the augmented operation (Lines 3-4). The advice again binds the player and the call arguments to variables (Lines 5-6). In contrast to the before binding, the augmentation is called (Lines 11-12) after the
package module.<rolePlayerPackage>..<rolePackage>.integration;
import <rolePlayerPackage>.*;
import <rolePlayerPackage>.impl.*;
import <rolePackage>.*;
import <rolePackage>.impl.*;
import java.util.*;

public aspect <rolePlayer>..<role> {
    // management and access for role
    private <role> <rolePlayer>Impl.<role> = new <role>Impl() {};
    public <role> <rolePlayer>Impl.<role>Role() {
        if (canPlay<role>()) {
            return (<role> <role>);
        } else {
            throw new IllegalArgumentException("Can't play role!");
        }
    }
    // management and access for role player
    public <rolePlayer> <rolePlayer>Impl.<role>Player() {
        if (canPlay<role>()) {
            return (<rolePlayer> this);
        } else {
            throw new IllegalArgumentException("Can't play role!");
        }
    }
    public boolean <rolePlayer>Impl.canPlay<role>() {
        // check role restrictions
        return true;
    }
}

Listing 4.7: Pattern to generate AspectJ aspect for role binding.

public <roleOperation.eType> <rolePlayer>Impl.<roleOperation.name> (<roleOperation.parameters>, ...) {
    Object result = null;
    result = <semanticBinding.source>
    return TypeAdaptationHelper.adaptToType(<roleOperation.eType>, result);
}

Listing 4.8: Pattern to generate AspectJ pointcut-advice for role operation binding binding.
4.2. The LanGems Language Composition System

Listing 4.9: Pattern to generate AspectJ pointcut-advice for before binding of augmentation.

Listing 4.10: Pattern to generate AspectJ pointcut-advice for after binding of augmentation.
augmented operation (Line 7). Again, the parameters of the augmentation can be bound using the defined variables and Java code fragments (Lines 8-9). In addition, the result of the augmented operation can be accessed. After the augmentation is executed the resultBinding can be used to adapt (Line 13) the result that is returned by the augmented operation (Line 14).

The application of the presented patterns for all role, role feature, role operation and augmentation bindings results in an integrated semantics implementation for the composed language. Listing 4.11 exemplifies the application of the introduced patterns for the formFlow language. It shows an excerpt of the aspect generated for the binding of Form to the Activity role. Lines 3-26 show the code generated to manage the role binding and to access the role or role player. Lines 28-51 implement the binding of Form to the interface of the Activity role as specified in the composition program (cf. Listing 4.3).

An excerpt of the Aspect generated for the role binding in Listing 4.4 is given in Listing 4.12. Lines 12-20 present the pointcut-advice for the before binding of the augmentation doLog(). It intercepts the execution of the evaluate() method declared in State (Line 13). Within the advice, the augmentation parameter binding is evaluated and bound to the variable message (Lines 16-17). This variable is passed as argument to the doLog() augmentation (Line 18). Finally, the execution of the method evaluate() is continued (Line 19).

The result of semantics composition is a complete implementation of Java operational semantics for formFlow. It provides an integrated evaluation of statechart and form semantics that changes wizard pages and renders the respective forms as needed. Fig. 4.19 illustrates the pages and the page flow in a wizard dialogue for managing items in a stock that was derived from the formFlow specification depicted in Fig. 4.14. It shows the forms for the init, the data, and two additional states and visualises the page transitions triggered by pressing form buttons.

```java
package module.statechartevalLogging.integration;

import module.logger.impl.*;
import module.logger.*;
import module.statechart.impl.*;
import module.statechart.*;
import java.util.*;

public aspect StateLoggingArtifact {
    void around (StateImpl loggingArtifactPlayer) :
        execution(void State.evaluate () )
        && target(loggingArtifactPlayer)
        && args () {
            java.lang.String message = "evaluating " + loggingArtifactPlayer
                .getLoggingArtifactPlayer().getStateName();
            loggingArtifactPlayer.getLoggingArtifactRole().doLog(message);
            proceed(loggingArtifactPlayer);
        }
}
```

Listing 4.12: AspectJ aspect generated for role binding of natural State to role LoggingArtifact defined in Listing 4.4.
4.2. The LanGems Language Composition System

```java
public aspect FormActivity {

    // management and access for role
    private Activity FormImpl.activity = new ActivityImpl();

    public Activity FormImpl.getActivityRole() {
        if (canPlayActivity()) {
            return (Activity) activity;
        } else {
            throw new IllegalArgumentException("Can’t play role!");
        }
    }

    FormImpl FormImpl.activityPlayer = this;

    public Form FormImpl.getActivityPlayer() {
        if (canPlayActivity()) {
            return (Form) activityPlayer;
        } else {
            throw new IllegalArgumentException("Can’t play role!");
        }
    }

    public boolean FormImpl.canPlayActivity() {
        return true;
    }

    public java.lang.String FormImpl.getActivityName() {
        java.lang.Object result = null;
        result = getActivityPlayer().getHeading();
        return de.tudresden.emf.utils.TypeAdaptationHelper.adaptToRoletype()
               java.lang.String.class, result);
    }

    public Trigger FormImpl.evaluate() {
        java.lang.Object result = null;
        result = getActivityPlayer().open();
        return de.tudresden.emf.utils.TypeAdaptationHelper.adaptToRoletype(
                        Trigger.class, result);
    }

    public void FormImpl.setActivityName(java.lang.String name) {
        getActivityPlayer().setHeading(name);
    }

    public List<Trigger> FormImpl.getTriggers() {
        Collection<?> result = null;
        result = getActivityPlayer().getButtons();
        return de.tudresden.emf.utils.TypeAdaptationHelper.adaptToRoletype(
                        Trigger.class, result);
    }
}
```

Listing 4.11: AspectJ aspect generated for role binding of natural Form to role Activity defined in Listing 4.3.
4.3. Case Study: Component-based OCL

The goal of this section is to demonstrate the applicability of role-based language composition for the development of the real-world language family OCL. This is meant to show our qualitative contributions compared to state-of-the-art and indicate the practical relevance of our work.

The OCL was released in 2000 as a constraint language used for models defined with the UML. Its abilities to describe static and dynamic constraints for object-oriented systems and its standardisation helped a wide adoption of OCL in academia and industry. Successively, the application of OCL extended to a constraint language for MDSD in general. This includes applications for modelling [Dang 2008, Debnath 2007, Stölzel 2006, Demuth 2001, Akehurst 2004a, Bräuer 2007] and metamodelling scenarios [Loecher 2004, MDT 2011, Steinberg 2008].

The OCL language family represents a prime example of language evolution, customisation and combination. In the following, we discuss the appearance of each phenomenon in detail.
4.3. Case Study: Component-based OCL

**OCL Evolution** The OCL standard evolved from a constraint language defined in the context of UML, to a self-contained specification that is regularly updated and extended. The evolution of OCL involved the introduction of additional language features, bug fixes in the language specification, and the deprecation of language features considered erroneous or superfluous [Chiorean 2005]. Over the years, a number of OCL versions were released as part of the UML specification [UML 2000, UML 2001, UML 2003] and later as stand-alone standard [OCL 2006, OCL 2010a, OCL 2010b].

**OCL Customisation** A large number of extensions to OCL have been and still are proposed. These extensions customise the expressiveness of OCL for particular domains. Among them, we find OCL extensions to describe temporal constraints [Ziemann 2003, Bradfield 2002], for the definition of real-time constraints [Flake 2002], for the definition of transformation contracts [Cariou 2004], for designing constraints in geographic data [Kang 2004], to add relations to OCL [Akehurst 2004b], for the definition of invariability clauses [Kosiuczenko 2006], to add syntactic sugar [Süß 2006], for business process modelling [Takemura 2006], for the definition of transformation and query languages, e.g., [Akehurst 2001, Siikarla 2003, ATL [Jouault 2008], MQL [Hearnden 2003], ETL [Kolovos 2008b], or YATL [Patrascoiu 2004].

**OCL Combination** In recent years, OCL advanced from a constraint language for the UML to a constraint language widely applied in MDSD. It was combined and integrated with various languages. This includes applications with modelling languages [Demuth 2001, Akehurst 2004a, Stölzel 2006, Bräuer 2007, Dang 2008, Debnath 2007, Wilke 2010], with metamodelling languages [Löcher 2004, MDT 2011, Steinberg 2008], and with programming languages [Vajk 2010, Wilke 2011].

Although not planned from the beginning, the discussed evolution and the experienced demand for customisation and integration makes the OCL a language family by accident. Its numerous members are depicted in Fig. 4.20.

The aim for reuse in implementing the OCL language family motivated a number of extensible and adaptive approaches for implementing OCL [Akehurst 2004a, Bräuer 2007, MDT 2011, Kolovos 2008a, Wende 2010, Wilke 2010]. Modularisation is considered beneficial for OCL evolution as modules are considered to reduce complexity in language evolution [Akehurst 2004a]. A systematic modularisation of OCL makes it possible to integrate new language extensions in a systematic manner, in contrast to rewrite the monolithic standard specification [Akehurst 2004a, Wende 2010]. Furthermore, modularisation helps constructing customised OCL variants based on a selection of core language features and extensions. Such languages use a custom subset of the full OCL standard and are better suited for a particular domain. The combination of OCL with
other languages is again a language modularisation problem.

The primary intent of the OCL language family is to adapt the expressiveness, syntax and semantics of OCL to the various application scenarios. The existing approaches for OCL modularisation rely on state-of-the-art techniques for grammar inheritance and metamodel integration. They to enable reuse at the level of groups of OCL expressions.

In this section, we exemplify our approach for role-based composition for the OCL language family [Wende 2010]. This is naturally split into two parts. In Section 4.3.1, we discuss a modularisation of OCL using the role-based language components. In Section 4.3.2, we describe the role-based composition of OCL with LanGems.

4.3.1. Role-Based OCL Modularisation

The first step in modularising OCL is to identify potential language modules or components. In [Akehurst 2005] and [Akehurst 2007], Akehurst et al. suggest a modularisation of the OCL concrete syntax into 13 subgrammars.

As depicted in Fig. 4.21, these grammars successively contribute core concepts, logics expressions, numbers expressions, types, primitives, invariants, operation constraints, and others. The subgrammars are combined and derived from each other using grammar inheritance. This introduces fixed dependencies of the inheriting subgrammars to the inherited ones, e.g., from ocl::primitives to ocl::numbers and ocl::logic. The presented approach modularises the monolithic OCL standard to an extensible language family, but still suffers from the drawbacks discussed for state-of-the-art approaches for modular language engineering (cf. Section 2.2).
4.3. Case Study: Component-based OCL

Figure 4.21.: Modularisation of OCL grammar using grammar inheritance [Akehurst 2007].

To address this, we revised the suggested modularisation and defined 11 basic language components that contribute orthogonal features to OCL [Wende 2010]. We followed a different modularisation approach. Instead of grouping concepts of a similar kind, e.g., all types, literals, or expressions, we grouped the types, literals and expressions belonging to the same language feature. For instance, our logic component contains logic types, logic literals and logic expressions.

In the following, we discuss each component in detail. The role-based metamodels and syntax specifications for each language component can be found in Appendix A.

core Component (c) The component core introduces all foundational concepts of OCL. It contributes expressions to describe navigation on object-oriented data structures, basic expressions, literals and types. The role-based metamodel of the core component is depicted in Fig. 4.22. A complete role-based abstract syntax specification is also given in Appendix A.1.

The central class of the core component is ExpressionInCore. It comprises different kinds of fundamental OCL expressions. This class is subtyped by the role class C_ExpressionRole. This prepares a later contribution of additional OCL expressions without breaking information hiding (cf. subtyping semantics discussed in Table 2.3). The following expressions are defined in core:

- LetExpressions enable the definition of local variables (vars) that can be used in expressions (exp). Variables are typed and initialised by an init expression. For declaring the type of a TypedDeclaration, the core component contributes two basic OCL types (InvalidType and UndefinedType). The introduction of further types is prepared by the role C_TypeRole that is subtype of Type.
• **CoreNested** expressions enable the grouping of expressions which is necessary when working with operators with different priorities.

• **Literal** expressions introduce two basic OCL literals to denote invalid (**InvalidLiteral**) and undefined (**UndefinedLiteral**) values.

• **VariableCall** expressions allow for referring to declared variables from within an OCL expression.

• **NavigationCall** expressions allow for navigating object-oriented structures. A **NavigationCall** has a left argument that can be any **ExpressionInCore**. The right argument is a **CallAtom**. The core component introduces two kinds of **CallAtoms**: **OperationCallAtoms** are used to navigate object structures using operations defined among them. They can be parametrised by OCL expressions (**ExpressionInCore**). **PropertyCallAtoms** can be used to navigate object structures using properties (attributes or references) defined for the objects.

Besides **C_TypeRole** and **C_ExpressionRole**, two other role types are defined in core. The role **CallModifier** allows for an extension of **VariableCalls** or **NavigationCalls** with additional modifiers. This is, for instance, used to contribute the **&pre** modifier (cf. **operationContext** component). The **CallAtomRole** can be used to contribute additional atoms as the right argument of **NavigationCalls**. This is, for instance, used to add calls to OCL collection expressions (cf. **collections** component).

Next, the **core** component contributes an EMFText-based syntax specification. It contributes production rules for all non-abstract natural types. Its complete specification is given in Appendix A.1.

**logic Component (l)** The component **logic** defines the concrete and abstract syntax for common logic expressions (**Implies**, **AndOrXor**, **Not**). In addition, different kinds of logic **Comparisons** and **IfThenElse** expressions are contributed. Finally, a literal (**BooleanLiteral**) to express the boolean values **true** and **false**, the **BooleanType**, and the role **LogicLiteralRole** to prepare a contribution of additional logic literals are included in **logic**. For the complete specification of the **logic** component, we refer to Appendix A.2.

**math Component (m)** The component **math** defines the concrete and abstract syntax for common mathematical expressions (**Additive**, **Multiplicative**, **Unary**). In addition, literals for integer (**IntegerLiteral**) and real numbers and their corresponding types (**IntegerType**, **RealType**) are introduced. Finally, the **math** component defines the role **MathLiteralRole** to prepare a contribution of additional mathematical literals.

The operators used in **math** expressions have different priorities that need to be considered in the concrete syntax specification. As EMFText generates an LL(*) parser with ANTLR, a special pattern is used to avoid left recursion and implement operator precedence.
The pattern is illustrated for additive and multiplicative expressions in Listing 4.13. Line 1 shows the production rule for Additive expressions. It has the lowest priority and is, thus, used as entry point for the parsing process. To avoid left recursion and to reflect operator priorities, Multiplicative expressions are declared as the expected left argument of an Additive expression. During the parsing process, this production structure ensures that expressions with a higher priority are parsed, before expressions with a lower priority are tried to match. The parser first descends into the productions given as left argument, before matching the ADDITIVE_OPERATOR and the right arguments. The pattern is continued for the Multiplicative (Line 3) expression. UnaryMath (Line 5) expressions, literals (Line 7-9), and the MathLiteralRole, have the highest priority in the math component. The concrete syntax for expressions in other language components, e.g., logic, core, or temporal, is implemented in a similar way. For the complete specification of the math component, we refer to Appendix A.3.
collection Component (col) The component collection contributes the concrete and abstract syntax of expressions to describe and process collections in OCL. It includes a literal (CollectionLiteral) to define a collection by enumerating all its items (CollectionItem) or by using ranges (CollectionRange). Ranges and items can be defined using OCL expressions bound to the role Col_ExpressionRole. A special natural is introduced to declare collection types (CollectionTypeSpecifier).

Furthermore, the collection component introduces expressions to represent calls to collection operations. In OCL, collection operations are called using the arrow-operator (\(-\rightarrow\)) instead of the dot-operator (\(\cdot\)) used for conventional operations. Two kinds of collection operations are distinguished. Basic collection operations, e.g., \(\text{size}()\) or \(\text{first}()\) are represented by the natural CollectionOperation. In addition, OCL has special Iterator expressions, e.g., \(\text{forall}\) or \(\text{exists}\), that allow for evaluating a given OCL expression over a collection of items. For the complete specification of the collection component, we refer to Appendix A.4.

messages Component (me) The component messages contributes the concrete and abstract syntax for OCL message expressions. It contributes two new operators to query message calls in object structures. Both are meant to observe calls to a given message, but differ in their return type. The hat-operator (\(^\hat{}\)) returns a boolean value indicating whether the respective message was called or not. The double-hat-operator (\(^{\hat{\hat{}}}\)) returns the set of messages that was actually called. Message queries can be restricted by expressions (Mes_ExpressionRole) that restrict their parameter values and types (Mes_TypeRole). In addition to message expressions, a special MessageType is contributed. For the complete specification of the messages component, we refer to Appendix A.5.

tuple Component (t) The component tuple contributes the concrete and abstract syntax of expressions to define tuples in OCL. A tuple (TupleLiteral) consists of a number of named elements. These TupleLiteralElements can be defined using OCL expressions that are bound to the role T_ExpressionRole. Furthermore, the natural TupleType is introduced to define types for OCL tuples and their elements T_TypeRole. For the complete specification of the tuple component, we refer to Appendix A.6.


```text
Additive ::= left:Multiplicative (operator[ADDITIVE_Operator] right:Additive)*;
Multiplicative ::= left:UnaryMath (operator[MULTIPLICATIVE_Operator] right:Additive)*;
UnaryMath ::= (operator[ADDITIVE_Operator])? atom:Literal;
IntegerLiteral ::= integerSymbol[INTEGER_LITERAL];
RealLiteral ::= realSymbol[REAL_LITERAL];
```
4.3. Case Study: Component-based OCL

**string Component (s)**  The component string only contributes the concrete and abstract syntax of an expression to represent StringLiterals and the String type. For the specification of the string component, we refer to Appendix A.7.

**classifiercontext Component (cc)**  The component classifiercontext contributes the concrete and abstract syntax to define OCL constraints for classifiers in an object-oriented model (e.g., a UML class diagram) or metamodel (e.g., an Ecore-based language metamodel). In a ClassifierContext, different kinds of ClassifierConstraints can be defined. **Invariant**s introduce constraints that must be true for all instances of the classifier at any time. **AttributeDefinitions** use OCL constraints to declare new attributes and their computation. **OperationDefinitions** are used to declare new operations and specify their realisation using OCL constraints. The component prepares a contribution of concrete constraint expressions by introducing the role CC_ExpressionRole. For the complete specification of the classifiercontext component, we refer to Appendix A.8.

**attributecontext Component (ac)**  The component attributecontext contributes the concrete and abstract syntax to define OCL constraints for attributes declared in object-oriented structures. It includes **init** and **derive AttributeConstraints** that use OCL constraints (AC_ExpressionRole) to describe an one-time initialisation or the repeated derivation of the constrained attribute, respectively. For the complete specification of the attributecontext component, we refer to Appendix A.9.

**operationcontext Component (oc)**  The component operationcontext contributes the concrete and abstract syntax to define OCL constraints for operations declared in object-oriented structures. It includes three kinds of OperationConstraint: **body** constraints use OCL expressions to define the realisation of query operations declared in the object-oriented structure, **pre** constraints define pre-conditions that need to hold whenever the respective operation is called, and **post** constraints define conditions that need to be satisfied after every operation execution. The operationcontext component prepares a contribution of concrete constraint expressions by introducing the role OC_ExpressionRole. For the complete specification of the operationcontext component, we refer to Appendix A.10.

**package Component (p)**  The component package introduces the concrete and abstract syntax to declare a package that contains different kinds of OCL constraint contexts and their respective constraints. It only consists of two types: the natural OCLPackage describing the declared package and the role ConstraintContext that enables a contribution of concrete constraint contexts from other language components. For the complete specification of the package component, we refer to Appendix A.11.

We defined two additional language components to enhance the usability of OCL constraints and to evaluate the extensibility of our OCL implementation.
Chapter 4. Integrative, Role-Based Composition for Language Family Engineering

initial Component (i) The component initial contributes a simple syntactic extension to OCL that enables the definition of the metamodel and optionally a model which a package of OCL constraints refers to. Both is useful when using OCL to define well-formedness rules for concrete languages and to easily test a set of given constraints for concrete model instances. The component also includes the role OCLPackageDeclaration to prepare the integration of OCL package declarations. For the specification of the initial component, we refer to Appendix A.12.

temporal Component (te) The component temporal realizes an extension of OCL with means to define temporal constraints as motivated in [Ziemann 2003]. The authors suggest the introduction of temporal invariants, pre-, and postconditions. Such constraints allow for a more fine-grained specification of restrictions on the sequence of allowed states in the life cycle of an object-oriented system.

The temporal component contributes expressions to refer to the previous (Previous) or next state (Next) and to define restrictions on some or all next (Always, Sometime) or previous states (AlwaysPast, SometimePast) of an object’s lifecycle. In addition, temporal property calls (TempPropertyAtPre, TempPropertyAtNext) and temporal operation calls (TempOperationAtPre, TempOperationAtNext) are introduced. For the complete specification of the temporal component, we refer to Appendix A.13.

4.3.2. Role-Based OCL Composition

To combine the introduced OCL components to an integrated language family, a composition program is defined on their natural and role types. Some exemplary bindings and their meaning are described in Table 4.2. The complete composition program can be found in Appendix A.14.

Table 4.2.: Exemplary role bindings in OCL composition program.

<table>
<thead>
<tr>
<th>Player</th>
<th>Role</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc::ClassifierContext</td>
<td>p::ConstraintContext</td>
<td>binds different constraint contexts to the constraint context role, this enables a definition of different kinds of constraints in an OCL package</td>
</tr>
<tr>
<td>ac::AttributeContext</td>
<td></td>
<td></td>
</tr>
<tr>
<td>oc::OperationContext</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
4.3. Case Study: Component-based OCL

<table>
<thead>
<tr>
<th>Player</th>
<th>Role</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>l::BooleanType</td>
<td>c::C_TypeRole</td>
<td>binds types introduced in different language components to the type role of the core component, this enables their application in defining typed elements, e.g., variables or parameters</td>
</tr>
<tr>
<td>m::IntegerType</td>
<td></td>
<td></td>
</tr>
<tr>
<td>m::RealType</td>
<td></td>
<td></td>
</tr>
<tr>
<td>col::CollectionType</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t::TupleType</td>
<td></td>
<td></td>
</tr>
<tr>
<td>me::MessageType</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s::StringType</td>
<td></td>
<td></td>
</tr>
<tr>
<td>l::Implies</td>
<td>cc::CC_ExpressionRole</td>
<td>binds the logic natural Implies to expression roles defined in different language components, the implies operator has the lowest priority in OCL and is, thus, the standard entry point for parsing OCL expressions</td>
</tr>
<tr>
<td></td>
<td>ac::AC_ExpressionRole</td>
<td></td>
</tr>
<tr>
<td></td>
<td>oc::OC_ExpressionRole</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c::C_ExpressionRole</td>
<td></td>
</tr>
<tr>
<td></td>
<td>t::T_ExpressionRole</td>
<td></td>
</tr>
<tr>
<td></td>
<td>me::Mes_ExpRole</td>
<td></td>
</tr>
<tr>
<td></td>
<td>te::Temp_ExpressionRole</td>
<td></td>
</tr>
<tr>
<td>m::Additive</td>
<td>l::LogicLiteralRole</td>
<td>binds the literals and expression of different language components to the logic component, the expressions either represent literals or expressions with a higher priority than logic expressions, thus, they are bound to the LogicLiteralRole</td>
</tr>
<tr>
<td>s::StringLiteral</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t::TupleLiteral</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c::CoreExpression</td>
<td>m::MathLiteralRole</td>
<td>binds the literals and expression of different language components to the math component, they have a higher priority than math expressions, thus, they are bound to the MathLiteralRole</td>
</tr>
<tr>
<td>te::TemporalExpression</td>
<td></td>
<td></td>
</tr>
<tr>
<td>col::CollectionPart</td>
<td>c::CallAtomRole</td>
<td>binds call expressions of the collection and temporal component to core, this extends the call operators available in OCL navigation expressions</td>
</tr>
<tr>
<td>te::TemporalPart</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Evaluating the composition program results in a sophisticated OCL editor that supports all language constructs included. Fig. 4.23 shows the editor generated for OCL including the temporal OCL extensions. The exemplary temporal constraints are taken
Chapter 4. Integrative, Role-Based Composition for Language Family Engineering

Figure 4.23.: Editor generated for OCL with temporal and initial extensions.

from [Ziemann 2003]. The editor could easily be derived from the OCL standard components by adding one additional language component and extending the composition program with three additional bindings. This illustrates the potential, ease of extension and amount of reuse achieved with role-based language composition.

4.4. Discussion

In the previous sections, we described the application of role-based language engineering for the realisation of the OCL language family and the two smaller language families formFlow and CPNs. In this section, we discuss experiences made during this application and evaluate the impact of roles for LFE in general.

In Section 4.4.1, we discuss the contributions of role-based language composition. Our discussion relates to the three parts every composition system consists of and discuss the impact of roles on each. In Section 4.4.2, we review related publications that aim at leveraging state-of-the-art in language engineering and composition. Finally, in Section 4.4.3, we conclude the contributions of role-based LFE wrt. the technical requirements induced by language evolution, customisation and combination and discuss open challenges.

4.4.1. Contributions

Impact of Role-Based Component Specification In [Parnas 1972], Parnas introduced criteria to evaluate a given modularisation for software implementations: module independence, information hiding, module exchangeability, module decomposition, and module comprehensibility. We argue that these criteria are also of central importance for
4.4. Discussion

LFE. The contributions of our role-based metamodelling language are, thus, discussed in relation to these criteria.

In our exemplary language families and during the OCL case study, roles enable a natural specification of self-contained language components that define explicit interfaces for communication and integration with other components. This leads to individual, decoupled components that can, thus, be developed and reused independently. The clear interfaces enable information hiding for the inner working of each language component and makes components exchangeable.

Furthermore, we could stepwise realise and extend the OCL language family with new components without interfering with existing ones. In the other examples, we found language components, e.g., statechart or forms, that encapsulate general formalisms for particular system engineering tasks. We see a lot of potential for reusing such components for different scenarios. The components are easier to customise than comprehensible standard languages and already prepared for systematic integration with other, scenario-specific languages.

We also experienced a reduced implementation complexity by decomposing a complex monolithic language like OCL into several comprehensible language components. Each language component covers only a particular, closed part of OCL’s syntax and hides the complexity of associated components behind well-defined role types. In addition, role-based language components can smoothly integrate specifications for concrete syntax and semantics completing the implementation of a language component.

Impact of Role-Based Composition Specification A composition specification describes the integration of a number of components to an integrated system. Therefore, it has to serve two basic functions. First, the binding of components via their interfaces needs to be specified. Second, components need to be adapted for interoperation.

In the introduced language composition system, role model composition founds the base for component integration. We experienced the binding of roles to role players a natural and comprehensible way for specifying such integration. The invasive integration semantics of the played-by relation in language composition matches human’s common understanding of role playing. The role player acts in place of the role and is required to fulfill particular expectations that may refine his natural behaviour.

Our composition language LCL introduces flexible, advanced, and invasive means to specify the integration of components for interoperability. This helps the integration of components even if they were not planned for interoperation and further contributes to component reuse. In our case studies, role bindings were commonly used for integrating the semantics of language components. Only in the formFlow example also augmentations were used to log the evaluation of wizard dialogs. This observation corresponds to the general recognition that the application of aspects that intercept the control flow for a given program are limited to some special use cases like logging or debugging (cf. [Steimann 2006, Steimann 2005]). Future work should investigate, whether augmentations can help to better modularise highly crosscutting aspects of language implementation like error reporting or type analysis.
Chapter 4. Integrative, Role-Based Composition for Language Family Engineering

In contrast to modular language engineering techniques that use modules to describe both fragments of a language implementation (language components) and their integration (component composition), these issues are clearly distinguished in our approach. The requirements for component specification and composition specification strongly differ. Our approach enables the application of customised languages for both. By the separation of these concerns, we experienced an enhanced comprehensibility of component interconnection and reduced complexity during component specification. The composition program can be analysed for completeness and well-formedness: Are all roles bound? Do role players conform to the described composition interface?

Finally, component and composition specification are decoupled structurally and temporarily. Language components can be built by one party and integrated by another. Language composition can take a set of previously defined components and later bind them to an integrated language. Existing languages can, thus, be extended, customised and evolved continuously and more easily.

Impact of Role-Based Composition Technique  We introduced a number of composition techniques which are employed in LanGems to integrate different artefacts of language specifications. The design of the composition system contributes an extensible, model-driven approach to language composition. This is beneficial for the following reasons.

LanGems builds on the foundations of the EMF infrastructure that provides and industry-strength, reliable and sustainable framework for implementing metamodelling languages and composition techniques. We discussed metamodelling languages for concrete syntax (EMFText, GMF) that were built with EMF. This enables an exploitation of model-to-model transformations for a comprehensive and efficient specification of composition techniques for metamodelled language specifications. Thus, existing techniques like grammar inheritance could be easily integrated and customised for role-based language composition.

EMF smoothly integrates Java for a straight-forward implementation of language semantics. We again benefited from standardised means for model-to-text transformation to realise composition techniques for non-metamodelled language specifications. We generated AspectJ code for integrating semantics of different language components. In analogy to concrete syntax specifications, metamodelling could also be employed for specifying semantics (e.g., [Efftinge 2006, Di Ruscio 2006, Wachsmuth 2008]). This would enable a more comprehensible and formal specification of language semantics and ease analysis of semantics specification and composition.

We experienced the application of model-driven tooling as a very flexible and productive way of implementing a composition technique. The specified model transformations and code generation templates naturally extend the generative language implementation approach of EMF. They provide a declarative and comprehensible representation of the composition algorithm that is free of platform-specifics and could easily be transferred to technical spaces different from EMF.

We implemented composition operators for prominent language engineering formalisms
and showed their applicability for realising sophisticated languages like OCL. However, we are aware of the plethora of formalisms and approaches for language realisation (cf. Section 2.2). They all come with their specific benefits and drawbacks. It strongly depends on the scenario which approach is the best solution. In one case, language engineers might be trained for or prefer a specific formalism. In other cases, the formal foundations of another approach might be indispensable. We experienced a Java-based semantics implementation beneficial, as no additional effort was needed to understand a dedicated semantics formalism and semantics realisation could benefits from the expressiveness of Java, its libraries, and the extensive tool support. The means needed for realising semantics composition strongly depend on the semantics formalism used. In case of Java-based operational semantics, semantics bindings were specified using Java code fragments and AspectJ was employed to implement the respective composition patterns. As the application of AspectJ aspects is limited to the component interfaces, our approach represents a grey-box composition technique. For other semantics formalisms, different means for specifying and realising grey-box composition are required. Given our extensible composition language and flexible but systematic means for realising composition operators, LanGems provides a good foundation for integrating additional language specification techniques.

4.4.2. Related Work

We are aware of a number of recent publications that share our interest in reuse and composition in language engineering. In the following, we shortly discuss approaches for language composition their contributions and relation to our approach.

- In [Weisemöller 2007], Weisemöller and Schürr compare three state-of-the-art approaches for building DSLs: (1) using UML profiles, (2) using inheritance to extend the UML metamodel, and (3) designing a new DSL from scratch. They criticise UML for not being prepared for DSL design by extension and conclude that the third approach provides most flexibility but also the fewest amount of reuse. Their observations match our experiences that existing approaches for language extension are to limited to realise complex language families by integration and extension of language modules.

As indicated by Weisemöller, A. Schürr and others (e.g., [Fowler 2005]), two general ways for reuse and composition in language engineering can be distinguished. In case of internal DSLs, reuse is achieved by extending an existing GPL to reuse its syntax and tooling. In case of external DSLs, reuse can be achieved by building new DSLs from reusable modules or components. We consider the second case the more general one, as it subsumes the special case of composing a GPL and DSL components. In the following, we review related work for both cases, starting with approaches to enhance embedding of internal DSLs.

- In [Bravenboer 2004], Bravenboer et al. introduce MetaBorg, an approach for embedding internal DSLs in Java. Language embedding is done in two steps:
(1) composing the DSL and the host language syntactically and (2) specifying the compilation of DSL concepts to concepts of the host language. The first step is realised using grammar inheritance where an integrating grammar describes the composition of host language and DSL syntax. Compilation is described by term rewriting rules specified using Stratego/XT [Bravenboer 2008].

Compared to our approach, MetaBorg is focusing on internal DSLs. The realisation of stand-alone external DSLs has different methodical and technical requirements. It does not distinguish means for specifying modules and module composition, but uses grammars for both cases. Furthermore, language engineering is strongly bound to ASF+SDF and Stratego/XT and less extensible than our language composition system.

- In [Atkinson 2007], Atkinson and Kühne provide a review of different customisation techniques for the UML language family. They motivate an application of the orthogonal classification architecture (OCA) that distinguishes conventional, linguistic metatypes and ontological metatypes. Ontological metatypes introduce further means to ease language customisation scenarios that aim at refining a given language with ontological information for a particular domain. They reside in the same metalevel as their instances and introduce a second classification dimension that corresponds to the classification hierarchy found in the respective domain. Furthermore, the authors suggest deep instantiation [Atkinson 2001] to enable an efficient representation of multiple classification levels in the domain.

In our approach, we implement language customisation at the level of abstract syntax metamodels. Ontological metatypes contribute expressive means for language customisation at the model level. The actual language specification is kept minimal and custom expressiveness can be contributed dynamically. This seems to enable very agile language customisation. It would be interesting to investigate a conceptual transfer of our ideas to an OCA. As we make make heavy use of MDSD tooling, a technical transfer is yet complicated by the few and immature tooling for OCA [Atkinson 2009].

- In [Renggli 2010], Renggli et al. present a compositional approach for embedding DSLs into host languages. The authors introduce language boxes to implement single language features in a modular way. Additional concrete syntax can be introduced using so-called language changes that allow for an adaptation of the host language grammar. Changes that are supported are replacement and insertion at the beginning or end of sequences or choices.

Furthermore, a language box needs to specify concerns like compilation of extended concepts to the host language and the scoping rules for the embedded DSLs wrt. the modularisation concepts of the host language. Language boxes are limited to the development of internal DSLs which is only one scenario for LFE. Internal DSLs are a very efficient way for DSL engineering as a reuse of features and tooling of host language reduces the development costs [Fowler 2005]. On the other hand, the binding to the host language makes internal DSLs less portable and flexible.
4.4. Discussion

One can not easily switch the host language, because the DSL syntax as well as
the structure must comply with the host language.

- In [Erdweg 2011], SugarJ, another approach for realising internal DSLs for Java
is presented. Language extensions are implemented in Java libraries that can be
dynamically imported in Java source files. SugarJ is based on SDF and Stratego
to describe syntactic embedding and compilation of the internal DSL. In contrast
to MetaBorg, SugarJ is focusing on the Java host language and meant to support
a more dynamic activation and deactivation of internal DSLs.

Approaches that enable the definition of external DSLs from modules and/or by com-
position are discussed in the following.

- In [Krahn 2008], Krahn et al. introduce a modular approach for developing new
DSLs more efficiently. They recognise the need for independence of individual lan-
guage modules for efficient reuse. In addition to conventional language integration
using inheritance, they also introduce a custom mechanism for language embed-
ding. Therefore, their tool MontiCore provides a unified metamodelling language
to specify both abstract syntax and concrete syntax in one specification. In such
specifications, abstract syntax concepts can be marked external to indicate that
their syntax is meant to be provided by another language module. External con-
cepts can, thus, be compared to roles in our approach. In addition, MontiCore
comes with a dedicated language to define the binding of external concepts. How-
ever, the tight binding of concrete and abstract syntax and the fixed composition
technique makes language embedding in MontiCore less flexible than role-based
language composition. Furthermore, our approach provide more expressive means
to define component interfaces and invasive component integration.

- In [Cazzola 2009, Cazzola 2010], the authors propose sectional DSLs. They are
constructed from slices, modules, and roles. A slice describes a language feature by
a collection of modules and can be compared to a component in our terminology.
Each module realises a part of that feature. The concept of a role also differs from
our terminology. In sectional DSLs, roles provide a realisation of a specific language
concern like concrete syntax, abstract syntax or semantics. We consider these con-
cerns inherent parts for each language component as the reuse of concrete syntax
and semantics is strongly coupled to the component’s abstract syntax. While this
can be viewed as a limitation, we are convinced that it does not reduce the appli-
cability of our approach. It also has the benefit of simplifying module specification
substantially. The composition technique employed for sectional DSLs is not de-
scribed in detail, but seems to rely on conventions and implicit interdependencies
between the language modules and, thus, is less comprehensible and flexible than
in our approach.

- In [Dmitriev 2005], projectional workbenches are introduced. They avoid parsing
by letting language users directly edit the abstract syntax using graphical shapes.
Based on a special tabular, cell-based layout, working with projectional editors is quite similar to using conventional text editors. As discussed in [Völter 2011], avoiding a parser is beneficial for language composition since it prevents ambiguities or conflicts in token definitions and grammar rules. Furthermore, projectional workbenches provide advanced editing features that are tightly coupled to language implementation and can easily be co-composed. Language composition is based on inheritance between language modules. In his evaluation, Völter emphasises the need for a more systematic approach to language extension. He suggests to build languages from parametrisable components. Our role-based approach to LFE can be considered an realisation of this idea.

4.4.3. Conclusion

In Table 4.3, we conclude the contributions of role-based language composition wrt. the requirements for a language family composition technique introduced in Sect 2.2. In general, we see the technical requirements for component-based LFE addressed. We discussed the contributions of explicit interfaces and information hiding for language components as well as loose coupling and flexible, invasive component integration to enhance language evolution, customisation and combination. Although, others discussed language composition approaches for internal and external DSLs, we are not aware of approaches that introduce a grey-box language composition system as comprehensible, flexible, and extensible as our approach.

Table 4.3.: Contribution of role-based language composition wrt. technological requirements for LFE introduced in Section 2.2.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Contributions</th>
</tr>
</thead>
</table>
| CM 10: Self-Contained Language Components | • role-based language components focus on role collaborations in a given context making them self-contained by definition  
• our role-based component model does not allow for interdependencies at the level of language components making language components independent |
| CM 20: Explicit Component Interfaces | • role types describe the required interface of role-based language components  
• natural types describe the provided interface of role based language components |

Continued on next page
<table>
<thead>
<tr>
<th>Requirement</th>
<th>Contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM 30: Information Hiding</td>
<td>• role and natural types restrict access to the language components, they declare syntactic and semantic interfaces for the language components, but hide their implementation</td>
</tr>
<tr>
<td>CM 40: Comprehensible Component Specification</td>
<td>• role-based language components can be complemented by specifications of concrete syntax and semantics to realise complete implementations of individual language features</td>
</tr>
</tbody>
</table>
| CL 10: Flexible Component Binding | • component integration is decoupled from component specification to enable more flexible binding  
• a natural type can play many role types  
• a role type can be played by many natural types  
• role composition contributes flexible means for integrating several language components by binding role types and naturals |
| CL 20: Component Integration | • role bindings can contain a number of role operation or augmentation bindings to specify the invasive integration of role player and role |
| CL 30: Comprehensible Composition Specification | • by customising the composition language, role operation and augmentation bindings can be specified using different semantics formalisms |

Continued on next page
## Requirement Contributions

### CT 10: Extensibility
- Language composition is described in terms of role-based abstract syntax of language components, the composition technique is not fixed to a specific approach for syntax or semantics specification.
- The techniques employed for role-based language composition are managed by an extensible composition workflow.
- To extend role-based language composition with new composition operators for custom specification approaches for concrete syntax, additional workflow steps can be contributed.
- To contribute custom semantics specification approaches the composition language and workflow can be extended.

### CT 20: Universality
- Component and composition specification are provided as models, this enables the application of universal and standardised model-driven tooling.
- For metamodelling specifications of concrete syntax or semantics model-to-model transformations provide a universal technique for implementing composition operators.
- For proprietary non-metamodelling specifications of concrete syntax or semantics model-to-text transformations provide a universal technique for implementing composition operators.

### CT 30: Sustainability
- The EMF infrastructure provides a standardised, widely-used, stable, and sustainable foundation for the introduced language composition system.

Besides the presented benefits, we also see a number of challenges for role-based language composition. First, role-based language composition introduces a novel, top-down approach to language engineering. Language components need to be build from scratch and designed for extensibility which implies an initial effort for porting existing languages or language families.

Second, we experienced some well-known issues of composing scanner-based parsers.
for context-free grammars. Tokens of different language components need to be free of conflicts. This means they should not contain overlapping token definitions which requires a global alignment of token definitions. Such alignment interferes with component independence. Alternatively, token conflicts could be avoided by extending our language composition system for scannerless parsing. In addition, grammar composition might result in ambiguities. This can to a certain degree be avoided by foresightful design of language components again reducing component independence. Alternatively, parser generators that can deal with ambiguous grammars could be used (cf. Section 2.2).

However, this introduces the need for dealing with additional disambiguation rules.

Third, for bigger language families, the composition program can become quite complex. This makes it harder to find errors and evolve the composition when components are updated. We already implemented basic well-formedness rules to evaluate the completeness of composition specifications: Are all roles bound? Is the interface of a role type completely implemented? Are there cycles in the binding specification? However, when it comes to validating the semantics bindings, things get more complicated. Currently, we employ a pragmatic approach. The composition program is executed and we use existing Java compilers to find typing errors in the generated language implementation. To implement type checking for the composition program, we would need to integrate and extend static semantics analysis for Java. An alternative would be to contribute a more formal semantics formalism that provides means for a systematic analysis of semantics specifications.

Fourth, this chapter provided an in-depth discussion of the relation between role modelling and current techniques for abstract syntax metamodelling. We showed how roles can be introduced as extension of existing metamodelling languages and presented techniques for the integrative composition of language syntax (cf. Fig. 4.11). This explicitly describes the statics of roles and the static impact of role composition and enables a transfer of our conceptual approach to other metamodelling formalisms. On the other hand, the AspectJ aspects given for the composition of Java operational semantics provide a technical realisation of the dynamics of role composition. However, we argue that a more explicit approach of describing the impact of roles in semantics composition is required to enable a better understanding and a transfer of semantics composition to other semantics formalisms.

Fifth, although the language components are technical independent, we experienced practical constraints on combining the language components. For the OCL language family, the core component is of fundamental importance. From a practical point of view, it makes no sense to build an OCL variant without this component. Furthermore, at least a component that declares a context for OCL constraints is required. Finally, there might also be technical constraints resulting from component realisations in the solution space or the modularity properties of the used language specification formalisms. In our case study we experienced constraints for integrating the components core, logic and math that resulted from operator priorities in the OCL standard. They are also a result of using a LL(*) parser that does not support left-recursion. Although different parsers might not have problems with left-recursion and enable a more sophisticated handling of operator priorities, the discussed examples show the need for managing technical con-
Chapter 4. Integrative, Role-Based Composition for Language Family Engineering

Constraints during language composition. Especially, when role-based language composition is employed to develop families of languages, these constraints need to be considered. This observation brings us back to variability management in LFE and motivates a combined application of feature-oriented LFE and role-based language composition. In the next chapter, we, thus, describe and evaluate their combined application in an integrated case study.
The two previous chapters closed with open challenges for feature-oriented LFE and role-based language composition, respectively. In this chapter, we demonstrate how a combined application of feature-driven variability management and role-based language composition in LFE addresses two of these challenges. First, we show that role-based language composition contributes means to modularise the solution space of language families in accordance to a feature-oriented modularisation of the problem space. Second, we illustrate how feature-oriented variability management contributes means to manage technical and practical constraints on component combination and, thus, enhances guidance in evolving and customising role-based language families.

In addition, we contribute a formalism for the specification and composition of dynamic semantics of language components based on CPNs. This is meant to provide a more explicit and formal foundation of the dynamics of role composition. Furthermore, we aim at alleviating the modularity issues identified for current semantics approaches and at addressing the lack of integration for metamodels and semantics specifications. In particular, we will introduce a novel integrative semantics composition technique that is based on the unique modularity and composition capabilities provided by CPNs.

This chapter is structured as follows. In Section 5.1, we introduce an integrated process for LFE with features and roles. In Section 5.2, we illustrate the application of this process for the development of an exemplary language family. This is also meant to provide a tutorial-like introduction on applying our comprehensive approach to LFE. In addition, we introduce CPNs for semantics specification and discuss the required extensions in our language composition system. In Section 5.3, we evaluate the case study and conclude our findings.
Chapter 5. LFE with Integrative, Role-Based Syntax and Semantics Composition

5.1. Integrating Features and Roles

Feature-oriented variability management and role-based language composition serve complementary aspects in LFE. While the first is meant to provide systematic means for designing language families and concrete family variants, the latter contributes a modularisation technique required for implementing language families by integrative, composition of the implementation of their features.

Fig. 5.1 depicts a version of the feature-oriented process for LFE that integrates role-based language composition. It was revised wrt. the steps (S2), (S3) and (S5) that are related to the solution space of LFE. During (S2) Language Family Realisation, the process employs the previously introduced languages LCSL and LCL to describe the realisation of individual features of a language family and their composition, respectively. This eases step (S3) where features now can be mapped directly to components in the language family. Finally, in (S5) feature-driven derivation eliminates language components that are mapped to features not included in the language variant specification and role-based composition integrates the remaining language components to a specific language variant.

5.2. SumUp Case Study

In this section, we demonstrate the application of the integrated process for an exemplary language family. This language family is called SumUp and comprises a family of languages to formulate mathematical equations.
5.2. Motivation

The development of the family is motivated by a scenario that involves three customers that require different kinds of mathematical languages in different contexts. In the following, we introduce each customer individually.

Christian Christian is a student in computer science. He is interested in a simple language to describe common mathematical operations like addition or multiplication. Furthermore, he wants to describe more complex computations that involve a number of equations and can exchange values using variables.

Susi Susi is a physics teacher in primary school. She regularly goes on excursions with her class and wants a language to describe and compute the costs for these excursions. For didactic reasons, she requires a language that enables a definition of units in computations to teach her pupils canceling rules.

Martin Martin owns a web shop and requires a language for computing prices and shipping costs for his products. Since such equations depend on different conditions, e.g., which delivery service is used or whether the customer requests a shipping insurance, he requires means to describe and evaluate such conditions in his equations.

Given the respective application context, each customer requests a slightly different language for mathematical equations. While it might be possible to develop a single language that satisfies all their needs, there are reasons to build custom language variants in a language family. First, from an economic perspective, it might be beneficial to sell different language variants to different customers. Second, for more complex language families, language variants that are customised to the needs of specific users are more comprehensible and easier to use. Third, depending on the application scenario, it might be useful to provide different users with different expressiveness, e.g., for security reasons.

5.2.2. Feature-Oriented Variability and Variant Specification

The first step in applying the process introduce above for the development of the SumUp language family is to develop a feature-based variability specification. Fig. 5.2 depicts

Figure 5.2.: Feature model for SumUp variability specification.
the initial feature model resulting from this step. As all SumUp applications deal with mathematical equations, each language variant needs to provide Math expressions. Furthermore, the optional features Conditional expressions, Variables and Units can be included.

Given this variability specification, the requirements of our exemplary customers can easily be formalised. The corresponding variant specifications are depicted in Fig. 5.3. Christian requires a SumUp variant that only includes the mandatory Math expressions and Variables. Susi requires a SumUp variant with Math expressions, Variables and Units. Martin additionally requires Conditional expressions, but excludes Units as all his equations are made in Euro.

5.2.3. Role-Based Component Realisation

The next step in realising the SumUp language family is concerned with implementing the identified features using role-based language components. For each leaf feature of the introduced feature model, an individual language component is created. In addition, a component sheet is introduced that specifies the core structure for the SumUp language as a plain sheet for equations. In the following, we introduce each component. Their general structure is depicted in Fig. 5.4. The complete specifications of their abstract syntax can be found in Appendix B.

For modelling SumUp, we use the export feature of the LCSL. This enables a further restriction of the provided interface for language components to a subset of its naturals. Instead of exposing all natural types, only natural types that are marked as exported in the component specification are available for role bindings during language composition. In Fig. 5.4, exported naturals are depicted as rectangles that cross the border of a language component. Using the export features allows for a more fine-grained definition of component interfaces and further enhances information hiding. On the other hand, it reduces the flexibility in component binding. When language components are designed for a single language family, like SumUp, this is not an issue. However, it may reduce

Figure 5.3.: Variant specifications for custom SumUp variants.
Figure 5.4.: SumUp language components.
the potential of reuse of language components among several languages and language families. Further research is required to investigate the trade-off between information hiding and flexibility.

**sheet Component**  The component sheet contributes the general concepts to describe an equation sheet. Each equation sheet is represented by a single Sheet element that consists of a set of statements. The role type Statement prepares a contribution of concrete statements by other language components.

**math Component**  The component math contributes common kinds of mathematical expressions (e.g., Additive, Multiplicative, Nested, NumberLiteral). Furthermore, it introduces the role type MathPrimitiveRole that enables other language components to contribute constructs that can be used as primitives in mathematical expressions.

**conditional Component**  The component conditional contributes special conditional expressions as required by Martin to describe the business rules for his web shop. Each ConditionalExpression defines a boolean condition (BooleanExpression). The component enables the application of all common CompareOperators to define such conditions. The arguments to Comparisons can be simple BooleanLiterals or types of other language components that are later bound to the role type Argument. If a ConditionalExpression evaluates true, the corresponding Consequence is executed. As Consequence is a role type, concrete consequences need to be provided by other language components.

**units Component**  The component units contributes special numbers (NumberWithUnit) that are annotated with Units as required by Susi. Units need to be declared (UnitDeclaration) before they can be used in equations.

**variables Component**  The component variables contributes means to declare and use variables in equations. Variables are declared using a VariableDeclaration. To assign values to a variable the Assignment can be used. Concrete expressions to compute values for variables can be contributed using the role type AssignmentExpression. Finally, equations can refer to variables using the natural VariableReference.

**string Component**  The component String introduces means to work with strings in SumUp. It only contributes a simple StringLiteral to define basic strings.

### 5.2.4. Feature-Oriented Variability and Variant Evolution

The string component was not planned for the SumUp language family, but is required to define business rules that refer to String-based conditions. It exemplifies an evolution for the initial language family to enable an easier application for a specific scenario. This motivates an extension of the initial feature model as depicted in Fig. 5.5. The
5.2. SumUp Case Study

The refined feature model introduces an additional feature for the string component. Furthermore, it uses a number of cross-tree relationships to indicate that an inclusion of the Conditional feature implies the selection of the features String, Variables, and Math. These dependencies originate from pragmatics in applying the SumUp language family. First, a useful definition of conditional expressions requires the availability of string expressions and variables. Second, the component math provides the only player for the Consequence role.

Propagating such practical and technical constraints to the problem space makes them more explicit and enables guidance in building valid language variants. Fig. 5.6 exemplifies the benefits of guided variant refinement for the SumUp variant built for Martin. On the left it shows the initial variant model for Martin in the Variant View of FeatureMapper. Due to the newly introduced feature String and its dependency to Conditional, the variant specification is marked inconsistent. Furthermore, the VariantView gives comprehensible feedback for the cause of the inconsistency and how it could be alleviated. Following the suggestion to include the feature String, results in a refined variant specification for Martin that is shown on the right. It is now consistent wrt. the variability constraints.

5.2.5. Model-driven Concrete Syntax Realisation

Next, the concrete syntax for the individual SumUp components can be implemented. As we already discussed the model-driven realisation of textual concrete syntax with EMFText, this step is not explained in detail here. The complete syntax specification for each SumUp component can be found in Appendix B.

5.2.6. Model-driven Semantics Realisation

Finally, the semantics of each language component needs to be specified. In contrast to defining semantics operationally in Java, as done for the formFlow language, we aim at a more formal approach. In our review of state-of-the-art in semantics specification (cf. Section 2.2), we already identified the lack of tight integration with current metamodelling approaches a significant drawback for common semantics approaches. Furthermore,
Chapter 5. LFE with Integrative, Role-Based Syntax and Semantics Composition

Inconsistent Variant Specification

Refined Variant Specification

The selection of feature Conditional requires the selection of String. You may either exclude Conditional or include String.

Figure 5.6.: Guided refinement of SumUp variant specification for Martin.

existing model-driven approaches for a formal specification of semantics disregard important properties for realising independent language components like component interfaces, information hiding, or rich and dedicated means for specifying component integration. CPNs provide rich concepts for the specification of semantics interfaces (port places and substitution transitions) and means for integrative semantics composition (port place assignment and substitution transition assignment). In this section, we demonstrate the feasibility and benefits of semantics specification and integrative semantics composition with CPNs.

Foundations of CPNs

CPNs [Jensen 1987, Jensen 2007] define a formal language for the specification and validation of concurrent, asynchronous systems. They extend basic petri nets [Petri 1962, Murata 1989] with a notion of data types and a functional language for describing data manipulation on such data types. A CPN can be used to describe, simulate, or generate the behaviour of a complex system. In addition, state space analysis can be applied to verify the liveness of the petri net or the absence of deadlocks and unreachable states. The modelling concepts of CPNs and their graphical syntax are introduced in Table 5.1. In the following, we shortly describe their meaning. For a detailed introduction on CPNs, we refer to [Jensen 2007].

Basic petri nets provide places, transitions and arcs for describing synchronisation, concurrency, and communication in complex systems. By using tokens that dynamically
Table 5.1.: Concepts and Syntax for CPNs.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Graphical Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Place</strong> with name, colour set, and initial token marking</td>
<td>![Place Diagram]</td>
</tr>
<tr>
<td><strong>Transition</strong> with condition and functional behaviour</td>
<td>![Transition Diagram]</td>
</tr>
<tr>
<td><strong>Consuming Arc</strong> with variable to bind consumed token</td>
<td>![Consuming Arc Diagram]</td>
</tr>
<tr>
<td><strong>Producing Arc</strong> with variable to produce token and property bindings</td>
<td>![Producing Arc Diagram]</td>
</tr>
<tr>
<td><strong>Port Place</strong> with port name, colour set, and direction</td>
<td>![Port Place Diagram]</td>
</tr>
<tr>
<td><strong>Substitution Transition</strong> with transition name</td>
<td>![Substitution Transition Diagram]</td>
</tr>
</tbody>
</table>
move between places, it is possible to represent different system states in a petri net. Events that change these states are called transitions. Arcs are used to connect places to transitions or transitions to places. These arcs are directed. Based on their direction, we distinguish incoming or outgoing arcs with respect to a given place or transition. For the execution of a transition, it is necessary that all places that are connected with its incoming arcs contain a token. If this is the case, the transition can be executed. During execution, it consumes all tokens from incoming arcs and produces new tokens at the places connected with outgoing arcs.

In contrast to existing state-based semantics formalisms like ASMs, petri nets allow for modelling concurrency. Transitions can have multiple incoming arcs (consuming arc) and multiple outgoing arcs (producing arc). Thus, petri nets can simultaneously represent and synchronise a number of different states for different parts of a complex system. This is a useful property for specifying and integrating semantics for independent language components.

CPNs extend the binary tokens used in basic petri nets with so-called colours. Basically, a colour set can be considered a complex data type. The tokens that can reside in a coloured place need to match the place’s colour set. If a place has, for instance, the colour set Integer, it can only hold tokens with integer values, e.g., 1, 2, 13, or 26. As depicted in Table 5.1, places can have also an initial token marking. Complex colour sets can be built by combining basic ones. We suggest to define the colour sets using types defined in the language component’s metamodel. This enables a tight integration of CPNs and metamodeling and prepares their application in specifying the semantics of language components.

In CPNs, consuming arcs do not simply consume coloured tokens from their source places, but also bind the token value to a variable. This variable is then used in transitions as follows. First, transitions can define conditions that must be satisfied for the transition to fire. Second, transitions have behaviour specifications which manipulate the consumed tokens and compute new tokens for the target places of its producing arcs. A producing arc refers to a variable defined by consuming arcs or in the transition behaviour. For producing arcs that target places with complex data types, so-called property bindings can be used to bind properties (i.e., attributes or references) of a complex token to values stored in variables. The definition of conditions and behaviour in transitions uses a functional, side-effect-free language. Thus, transitions can be used to describe and evaluate the semantics of language constructs.

Furthermore, CPNs provide advanced modularity constructs that are useful when specifying and integrating semantics for role-based language components. A CPN can consist of a set of components. Interfaces for such components can be defined using so-called port places. Port places have a name, a colour set and a direction (IN or OUT). Port places are a good match to role types of a language component. Between an IN and an OUT port place so-called substitution transitions can be defined. They are used to abstract compound behaviour of a CPN sub-component. A sub-component can be contributed by assigning places of the sub-component to the port places around the substitution transition. In role-based language engineering, substitution transitions correspond to role operations. They can be used for abstracting the behaviour of respective role players.
5.2. SumUp Case Study

To enable an application of CPNs for semantics specification in SumUp, we implemented the CPN language in EMF. Our CPN dialect can import the types defined in a language component as colour sets and uses a functional language to query and manipulate instances of these types, i.e., concrete language expressions. This language can traverse models using the syntactic and semantic interfaces defined by the language metamodel and provides an extensible standard library that contributes functions on primitive data types and collection types. Finally, we realised a Java code generator that derives a Java class to evaluate the semantics of a given CPN.

Semantics Specification with CPNs

Fig. 5.7 depicts a CPN component that specifies the evaluation semantics for the sheet component. The execution of this net can be triggered by placing a concrete SumUp sheet at the place sheet. The place statement_execute_mutex is already initialised with the integer token 0. This enables the transition prepareStatement and binds the variables sheet and mutex to the SumUp sheet and 0, respectively. If the number of statements in the sheet is bigger than the current mutex, the variable statement is bound to the statement at the index position mutex. The producing arcs of prepareStatement move the statement token to the IN port place statement_execute_IN and the sheet token back to the place sheet.

The substitution transition execute describes a transition to the OUT port place statement_execute. It corresponds to the role operation execute() declared for the role type Statement in Fig. 5.4. By using a substitution transition, the CPN component describes an abstract placeholder for a behaviour provided by another petri net component. This reflects the semantics of role operations and prepares their binding during role composition.
Figure 5.8.: CPN component for semantics specification of math component.

If a statement arrives at the OUT port place statement_execute, this enables the transition nextStatement. The transition condition checks the position of the evaluated statement in the SumUp sheet and binds the mutex variable to the position of the next statement. The producing arcs of nextTransition move the token sheet at the place sheet and the token mutex at the place statement_execute_mutex. This enables another iteration of the evaluation cycle. If all statements of a sheet are evaluated, the net terminates. This design of the sheet CPN ensures a stepwise, ordered execution of each statement contained in the SumUp sheet.

Fig. 5.8 depicts an excerpt of the CPN component that specifies the evaluation semantics for the math component. The evaluation of a mathematical expression is initialised by placing a token of the type MathExpressionContainer at the place mathExpressionContainer. This triggers the transition initExpression and binds the token to the variable mec. The transition initialises the variable e with the expression contained in mec. Its producing arcs put the token mec to the place mathExpressionContainer_waitingForExpression and the token e to the place expressions.

The expression e represents the root of a mathematical expression tree of arbitrary complexity. As such a tree needs to be evaluated from the leaves to the root, the intermediate and leaf expressions need to be prepared for evaluation first. The transition prepare_left_right, therefore, extracts the left and right child expressions for BinaryExpressions and puts them to the place expressions to enable the fur-
ther preparation of their potential child expressions. The binary expression \( e \) is itself moved to the place \texttt{expressions\_in\_evaluate}. Similar preparation transitions are provided for \texttt{NestedExpressions}. Other, unary literal expressions are simply moved from \texttt{expressions} to \texttt{expressions\_in\_evaluate}. The respective transitions are not shown in the figure, but can be found in the complete CPN that is given in Appendix B.

Next, the prepared expressions can actually be evaluated. For each expression kind e.g., \texttt{Additive}, \texttt{Multiplicative}, \texttt{Nested}, the CPN contains a respective transition. Fig. 5.8 depicts the transition \texttt{add} to evaluate \texttt{Additives}. It consumes an expression \( e \) from the place \texttt{expressions\_in\_evaluate}. The \texttt{left} and \texttt{right} argument of this expression need to be present at the place \texttt{expressions\_evaluate} indicating that they have been evaluated previously. Furthermore, \( e \) needs to have the operator \(+\) indicating that it actually represents an addition. All these requirements are checked in the condition of the transition \texttt{add}. If they are satisfied, the transition is evaluated. It calculates the \texttt{sum} of both arguments by adding their \texttt{result} values. Finally, the evaluate expression \( e \) is moved to the place \texttt{expressions\_evaluate}. The variable \texttt{sum} is assigned to its structural property \texttt{result} (cf. Fig. 5.4). This special evaluation pattern is also used for the other kinds of expressions and ensures the correct order in evaluating the expression tree from the leaves to the root.

If all expressions are evaluated, only the root expression of the expression tree remains in the place \texttt{expressions\_evaluate}. It enables the transition \texttt{evalMec}. The transition consumes the root expression and binds it to the variable \( e \). It also retrieves the \texttt{MathExpressionContainer} waiting in the place \texttt{mathExpressionContainer\_waiting\_for\_expression} and binds it to \( \texttt{mec} \). The root expression should match the expression contained in token \texttt{mec}. When \texttt{evalMec} is evaluated, the token \texttt{mec} is moved to the place \texttt{mathExpressionContainer\_evaluate}. In addition, the \texttt{result} property of \texttt{mec} is set to the result value of expression \( e \). This finalised the evaluation of the \texttt{MathExpressionContainer} fed to the CPN.

We specified further CPNs for each language component of SumUp (cf. Appendix B). They provide a formal semantics specification that is only related to the naturals and roles defined in the respective component. This keeps the individual semantics specifications independent of each other and preserves the modularity properties of role-based language components.

**Semantics Composition with CPNs**

To integrate CPN semantics with role-based language composition, another compound composition process has to be contributed to the LanGems composition workflow. Fig. 5.9 depicts this process. It consists of three subtasks that are explained in the following.

**CPN Semantics Propagation** The first subtask realises the combination of the CPN components specified for each individual language component. Therefore, a new CPN is created. It imports all types from the integrated metamodel to enable their application
in colour sets. Next, the content of the CPNs for all language components, i.e., all arcs, places and transitions is copied to the new CPN.

The task results in a combined CPN semantics specifications where the individual CPN components of all language components are placed next to each other, but are not yet integrated.

**CPN Semantics Weaving** The objective of the subtask CPN Semantics Weaving is to actually integrate the individual CPN components using their port places and substitution transitions. Therefore, it is necessary to assign places of one CPN component to the port places of another and to replace substitution transitions by complex behaviour specified between the assigned ports. To specify such assignments, we extended the LCSL with a new subclass of SemanticBinding called PetrinetBinding. A PetrinetBinding specifies the connection of the CPN components for the language components of the role player and the role.

In Listing 5.1, we present an excerpt of the SumUp composition program showing the application of PetrinetBindings. It describes the composition of the previously introduced CPNs for the sheet and the math component. First, the role-based metamodels of the respective language components are imported (Lines 3-4). Line 7 declares the sheet component as integrating core for the SumUp language family. Lines 9-14 describe the integration of the math component. The math component refines the sheet component (Line 10). It binds the natural MathExpressionContainer declared in the math component (cf. Fig. 5.4) as role player for the role Statement declared in the sheet component. Next, the role operation execute() has to be bound. As discussed above,
execute() was mapped to a substitution transition with two port places (cf. Fig. 5.7). For the role operation binding, these port places are assigned to places in the math CPN (cf. Fig. 5.8). The IN port place is mapped to the place mathExpressionContainer and the OUT port place is mapped to the place mathExpressionContainer_evaluate (Line 12). The substitution transition is, thus, replaced by the complex CPN structure between these places that was defined in the math CPN component. The complete SumUp composition program integrating the CPNs for all language components can be found in Appendix B.7.

The presented integration exemplifies a simple case of CPN integration where two nets match perfectly. However, during previous case studies and also for SumUp, we experienced cases where the combination of two language components requires additional, invasive integration. In addition, we want to keep track of the dynamic binding between a role player and its potentially multiple roles during the execution of the integrated CPN.

Therefore, we developed an extended composition pattern for CPNs depicted in Fig. 5.10. The role binding depicted on the left describes an abstract binding between a Role with an role operation roleOp() and a Natural. The binding specifies a role operation binding for roleOp(). As discussed previously, each PetrinetBinding needs to specify an in and an out place. In addition, an optional result binding and an optional net extension can be given to implement role operations with return values and to realise invasive semantics integration, respectively.

The right part of Fig. 5.10 depicts the CPNs of the involved language components and the pattern for their composition. The CPN specified in the upper language component c1 describes the realisation of the port places and the substitution transition for roleOp(). The CPN specified in the lower language component c2 describes the behaviour of its specific Naturals.

The CPN structures in between the components describes their composition. It contains a transition connecting the IN port place of roleOp with the in place specified in the composition program. This transition consumes a role token from the IN port place.
Chapter 5. LFE with Integrative, Role-Based Syntax and Semantics Composition

Figure 5.10.: Integrative composition pattern for CPN to implement dynamics of role composition.

place and moves it to a place that keeps track of the dynamic binding between a role and its player (natural_plays_role). Furthermore, it checks whether the role token is an instance of the role-playing Natural. This is necessary to select the right semantics binding when several Naturals can play the same Role. If the instance check succeeds, the variable player is initialised with the role token and moved to the inPlace specified in the composition program. On the other hand, there is an opposite transition that connects the out place specified in the composition program with the OUT port place of roleOp. It consumes the token player from the outPlace and checks whether it matches a token role that was consumed from the place (natural_plays_role). This ensures that the current player has an active binding to the Role type. If this check succeeds, the token role is moved to the OUT port place which finalises the semantics evaluation of the roleOp().

If the role operation has a return type, it is necessary to describe the binding of the return value in the context of the role player. Therefore, a result binding can be given. The result binding is evaluated in the transition from the outPlace to the OUT port place and set in its producing arc.

Finally, it is possible to specify PetrinetBindings with or without a net extension. A net extension can be used to specify an integration net that invasively adapts the CPNs of both language components for integration. This is necessary if their behavioural descriptions can not be combined immediately or are incomplete wrt. the integration scenario. In such cases, the in and out places of the PetrinetBindings point to places in the integration net and the integration net specifies an arbitrarily complex integration semantics to glue both CPNs (cf. Fig. 5.10). An example can be found in Appendix B.7.

168
5.2. SumUp Case Study

Figure 5.11.: Exemplary application of CPN composition pattern.

(Lines 82-90). If such integration is not necessary, the in and out assignments directly point to places in the CPN of the Natural. An example was already described in the composition program in Listing 5.1. Fig. 5.11 shows the result of applying the pattern for this example. It integrates the CPNs for the components math and sheet.

The described pattern is implemented in a model transformation. During the CPN Semantics Weaving task, it is applied for each role operation binding specified in the composition program. Its execution results in an composition of the individual CPNs to an integrated semantics specification for the language family. The pattern design shows the importance of an integrative technique for composing language semantics. The behaviour of the integrated CPNs is changed invasively by their interconnection. Integration nets provide expressive means to contribute additional integration behaviour. Also the dynamics of role-binding are reflected and affect the behaviour of the integrated CPNs.

**CPN Implementation Generation** The last task of the CPN composition workflow is concerned with the generation of a Java-based implementation of the CPN behaviour. It generates data structures to manage the different places defined for the CPN and their tokens. Transitions are mapped to Java methods that implement the movement and manipulation of tokens. The generated implementation provides an API for initialising places and to trigger the evaluation of the CPN. It can, thus, be used to evaluate the semantics of a concrete SumUp equation sheet.

5.2.7. Role-Based Composition and Feature Mapping

Having a complete specification of the component syntax and semantics, it is now necessary to describe their integration using the LCL. We already dis-
Chapter 5. LFE with Integrative, Role-Based Syntax and Semantics Composition

Figure 5.12.: Exemplary role bindings in SumUp composition program.

cussed an excerpt of the SumUp composition program when we introduced CPN semantics composition. Fig. 5.12 depicts all bindings for the role Statement, i.e., the Naturals of other language components that contribute elements to describe statements in a SumUp equation sheet. A Statement could be a simple mathematical expression (MathExpressionContainer), a conditional expression (ConditionalExpression), a declaration of a unit (UnitDeclaration), a variable declaration (VariableDeclaration), or a variable assignment (Assignment). These bindings illustrate the level of abstraction and comprehensibility that is achieved by hiding the inner workings of the language components behind their explicit provided and required interfaces. The complete composition program including all role operation bindings is provided in Appendix B.7.

Given this composition program, we can map the SumUp features to their corresponding realisation artefact. As the modularisation of the solution space corresponds to the granularity of features in the problem space, this mapping is straightforward. Each component in the SumUp composition program is mapped to its respective feature in the feature model.

5.2.8. Language Variant Derivation

With a complete feature mapping, we can finally start the derivation of custom SumUp language variants. The custom variant specifications for Christian, Susi, and Martin are fed to the feature-driven derivation process of the FeatureMapper. Given a concrete feature selection, it removes all language components that are mapped to excluded features. All role bindings related to the removed components are also disabled. Fig. 5.13 depicts the reduced role bindings for Christian’s language variant. It only includes components mapped to the features SumUp, Variables, or Math.
5.3. Conclusion

The reduced composition program is evaluated by the LanGems composition system. It employs the introduced composition tasks to derive an integrated SumUp metamodel implementation, an integrated concrete syntax implementation, and an integrated semantics specification for each SumUp variant. Fig. 5.14 depicts the custom editors derived for Christian, Susi, and Martin. They illustrate the custom expressiveness of the language variants, the coherent and sophisticated editing support, and the application of SumUp variants for the individual scenarios.

5.3. Conclusion

In this chapter, we presented a continuous case study for applying feature-oriented LFE and role-based language composition in combination. We experienced both as complementing parts of a comprehensible approach to LFE. This chapter aimed at addressing one issue for feature-oriented LFE and role-based language composition, respectively. First, current approaches for language implementation lacked adequate means for modularisation at the level of language features. Second, for a development of language families, role-based language composition required a more abstract and comprehensible approach for expressing technical and practical constraints of component combination.

Regarding the modularity issue, we see a good match of feature granularity in the problem space and role-based modularisation in the problem space. This eases the mapping of both spaces. It is also expected to enhance the mapping stability, as the evolution of the language family has less impact on the feature mapping. Four general cases can be distinguished: 1) An evolution step may only affect the internals of a single language component. Here, the mapping does not need to be changed at all. 2) A change may affect the external interface of a language component and might trigger changes in related components. This, however, should also not affect the feature mapping, as the granularity of components is not changed. 3) An evolution step may motivate the introduction of a new language component. Here, a corresponding feature and a simple one-to-one mapping between component and feature needs to be added. 4) The granularity of language components may need to be changed, e.g., a component is split or a set of components are merged. This requires an adequate co-adaptation of the feature model. The last two scenarios may strongly benefit from tooling to enhance the co-evolution of problem and solution space in SPLE and motivates future work.
(a) Christian’s editor: support for plain mathematical equations with variables

(b) Susi’s editor: mathematical equations with variables and support for units

(c) Martin’s editor: mathematical equations with variables and conditional expressions

Figure 5.14.: Editors derived for custom SumUp variants.
5.3. Conclusion

Also the management and propagation of constraints from solution space to the problem space is eased by the simple one-to-one mapping. Propositional cross-tree constraints provide an expressive formalism to represent constraints on component combinations. We experienced the explicit representation of such constraints in the problem space beneficial for variant and variability management in the evolution of language families. First, it provides automated means to detect inconsistencies in variability and variant specifications. Second, it provides guidance in repairing these inconsistencies.

In addition, this chapter contributed a novel approach for a modular specification and integrative composition of language semantics using CPNs. First, this provides an explicit specification and implementation for the dynamics of role composition. Second, this demonstrates the extensibility of our composition system. Third, CPNs enable a tight integration of semantics specification and role-base metamodelling. Their support for specifying and synchronising complex, concurrent systems and their rich modularity concepts help to preserve the modularity properties of role-based language components in their semantics specifications. CPNs are typed and provide a functional language which enables expressive semantics specification. Similar to Java semantics composition, we apply type checking to ensure the well-formedness of individual and composed semantics specifications. In addition, CPNs also provide sophisticated means for state space analysis that could be exploited to analyse semantics specifications for liveness or the absence of deadlocks. This motivates some future work for implementing end evaluating existing analysis algorithms for CPNs for our CPN implementation.

Finally, this chapter provided a comprehensible documentation for applying the approaches developed in this thesis in a practical setting. This prepares their application, exploitation, and evaluation by a wider audience.
In this chapter, we conclude our achievements and provide a discussion of future work. It is structured as follows. In Section 6.1, we give an overview of the contributions of this thesis. In Section 6.2, we discuss open challenges and topics for future work.

6.1. Contributions

In Chapter 1, we motivated the importance of language families historically, conceptually, and practically. Historically, language families gained importance with the growing interest in MDSD and DSLs. This was induced by the impact of MDSD and DSLs on efficiency and quality in software engineering and the reduced costs for language engineering that results from progress in respective tooling. Practically, we showed the omnipresence of language families by a selection of existing language families and their various applications. Conceptually, we identified three sufficient phenomena of language development that found language families: language evolution, language customisation, and language combination. Given these phenomena, we identified the challenges they introduce for LFE. To address these challenges, we motivated the principles of variability and integrative composition for LFE. Based on these principles, we suggested to introduce a variability-oriented process and a grey-box composition system for.

In the following, we summarise the conceptual, technical and qualitative contributions of this thesis. We, therefore, relate to the contents of individual chapters and the hypotheses introduced in Chapter 1. For a more detailed discussion of qualitative contributions, we refer to the chapter conclusions given in Section 3.4, Section 4.4, and Section 5.3.

0. Evaluation of State-of-the-Art in Language Engineering for Applicability in LFE

In Chapter 2, we investigated state-of-the-art in language engineering for its applicability in LFE. During our comprehensive investigation of current processes and techniques for language engineering, we identified issues that aggravate an efficient realisation of
language families.

Existing processes for language engineering lack means for a systematic analysis, specification and management of variability as found in language families. As depicted in Fig. 6.1, existing processes typically consist of a sequence of loosely integrated process steps that concern the analysis, design, and implementation of a single language.

For language realisation techniques, we documented the plethora of alternative approaches and formalisms for implementing complementary language constituents like abstract syntax, concrete syntax and semantics. As depicted in Fig. 6.2, current approaches for modular language realisation use heterogeneous formalisms and tools for the different constituents. Means and granularity for the modularisation may differ for the constituents. There is no dedicated, comprehensive means for the specification of module integration. Instead, module integration employs means that are specific to the syntax and semantics approaches and their specific combination. Regarding their modularity properties, we criticised insufficient information hiding, no explicit component interfaces, no loose coupling and no flexible integration in existing approaches. Furthermore, the approaches for complementary constituents are roughly integrated and alternative approaches for the same constituent are hard to interchange. Given the identified issues, we derived an extensive collection of requirements for an enhanced LFE process and technique, respectively.

I. Novel, Feature-Oriented Process for LFE,

In Chapter 3, we introduced a feature-oriented process for LFE. As depicted in Fig. 6.1, this is a major conceptual and technical contribution which extends conventional language engineering processes for an application on LFE. The inherent variability found in all language families, requires dedicated means for the analysis and management of variability. Therefore, we customised techniques for variability management in SPLE for an application on languages. The state-of-the-art processes for language engineering were replaced by a process consisting of two integrated phases. The first phase is concerned with the development of the complete language family. The second phase enables...
a highly automated derivation of a concrete language variant.

The introduced process was employed and evaluated in an extensive case study. The case study aimed at enhancing the scalability of ontology specification, evaluation and application by realising OWL as a language family. Our evaluation revealed the following qualitative contributions of feature-oriented LFE: (1) features enable explicit variability analysis and specification in language families, (2) features can be employed at different abstraction and granularity levels of LFE, (3) the introduced process enables continuous tracing from language requirements to language implementation artefacts, (4) the mapping between features and realisation artefacts works for different metamodeling and modelling languages involved in LFE, (5) features enable guidance in specifying and completing language variants, (6) given a feature-based variant specification, the customisation of arbitrary language variants and respective tooling is fully automated, and (7) the iterative design and the explicit tracing between artefacts developed in different process steps helps a consistent evolution of language families. Our findings show the feasibility and benefits of feature-oriented LFE and, thus, prove Hypothesis 1.
II. Novel, Role-Based Approach for Integrative Language Composition

In Chapter 4, we introduced a role-based language composition system. This is a major conceptual and technical contribution which addresses the issues for modularity, integration, and interchange found in current language realisation techniques.

As depicted in Fig. 6.2, we contribute a role-based language for metamodelling the abstract syntax of language components, a role-based language for specifying the composition of language components, and a composition system to evaluate role-based language composition programs.

Using several minor and a major case study, we investigated the applicability and benefits of our composition system for different language constituents, language realisation approaches, and language families. The evaluation of our role-based language composition system revealed the following contributions: (1) role-based metamodelling enables the definition of self-contained language components, (2) roles allow for defining explicit component interfaces that enhance information hiding, (3) role-based language components can be complemented with various approaches for concrete syntax and semantics implementation, (4) role-based composition programs enable flexible component binding and invasive component integration, (5) our composition system is extensible with role-based composition techniques for various approaches for concrete syntax and semantics implementation, (6) role-based language composition enables the application of universal techniques like model-to-model and model-to-text transformations for implementing composition techniques, and (7) the composition system is implemented on a standardised and sustainable foundation. Our findings show the feasibility of realising a role-based approach for integrative language composition.

Our concrete implementation was based on the widely-used object-oriented metamodelling framework EMF. Due to the similarities of EMF and other metamodelling formalisms discussed in Chapter 2 (cf. Table 2.2), our results can be generalised. This demonstrates the feasibility and benefits of role-based, integrative language composition and, thus, proves Hypothesis II.

III. Novel, CPN-Based Approach for the Integrative Composition of Language Semantics

In Chapter 5 we introduced a CPN-based approach to specify the dynamics of role composition and as practical approach for semantics specification in language families. Therefore, we implemented a semantics metamodelling language based on CPNs that is tightly integrated with role-based language components. It can itself be considered a contribution to semantics specification in language families as (1) CPN-based semantics were seamlessly integrated with role-based metamodelling, (2) CPNs provide rich modularity concepts that preserve the modularity properties of role-based language components, and (3) CPNs enable an integrative composition of dynamic semantics. These properties are not found in existing (modular) semantics formalisms.

The approach was applied in a case study for specifying the dynamic semantics of an exemplary language family for mathematical equations. The presented extension of
### 6.1. Contributions

<table>
<thead>
<tr>
<th>Case Study</th>
<th>OWL</th>
<th>OCL</th>
<th>formFLow</th>
<th>CPN</th>
<th>SumUp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature-Oriented</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Scenario</td>
<td>redesign</td>
<td>redesign</td>
<td>new</td>
<td>redesign</td>
<td>new</td>
</tr>
<tr>
<td>Size</td>
<td>tall</td>
<td>tall</td>
<td>small</td>
<td>small</td>
<td>medium</td>
</tr>
<tr>
<td></td>
<td>(3 feature models)</td>
<td>(13 components)</td>
<td>(3 components)</td>
<td>(3 components)</td>
<td>(1 feature model, 6 components)</td>
</tr>
<tr>
<td>Abstract Syntax</td>
<td>Ecore</td>
<td>LCSL</td>
<td>LCSL</td>
<td>LCSL</td>
<td>LCSL</td>
</tr>
<tr>
<td>Concrete Syntax</td>
<td>EMFText</td>
<td>EMFText</td>
<td>EMFText</td>
<td>GMF</td>
<td>EMFText</td>
</tr>
<tr>
<td>Semantics</td>
<td>proprietary reasoners</td>
<td>operational in Java</td>
<td>-</td>
<td>-</td>
<td>CPNs</td>
</tr>
</tbody>
</table>

Table 6.1.: Overview of case studies used for qualitative evaluation of our approach.

Role-based language composition shows the feasibility and applicability of integrative, CPN-based semantics composition and, thus, proves Hypothesis III.

### IV. Qualitative Evaluation of Feature-Oriented LFE with Integrative, Role-Based Syntax and Semantics Composition

Altogether, we conducted three major case studies and two minor case studies. In Chapter 3 and Chapter 4 the individual contributions of feature-orientation and role-based composition for LFE were evaluated. In Chapter 5, we also demonstrated the application of the combined approach for feature-oriented LFE with integrative, role-based syntax and semantics composition in an integrated case study. As illustrated in Table 6.1, the design of these case studies enabled an evaluation of our approaches for different scenarios of LFE, for miscellaneous language families, and different technical settings.

The evaluation of our case study in Chapter 5 revealed the following general, qualitative contributions for combining features and role-based composition: (1) role-based language components enable a modularisation of the solution space of language engineering at the granularity of language features and (2) feature models enable an abstract, explicit representation of pragmatic and technical constraints on the combination of role-based language components enabling their flexible and consistent combination in language families. This shows benefits of the combined approach and, thus, proves Hypothesis IV.

In addition, the case studies illustrates the application of our approach for a wider audience to prepare its further exploitation, evaluation, and extension.
6.2. Outlook

During the realisation of our LFE approach and during its application in several case studies, we identified a number of open challenges and issues that require further research. In this section, we recapitulate some important challenges and indicate topics for future work.

6.2.1. Co-Evolution in Language Families

Evolution is one of the phenomena that found language families. In the application of our approach, we showed that feature-oriented LFE enhances guidance in language evolution. This was achieved by making feature-dependencies explicit and by enabling a tracing of language requirements and realisation artefacts. In addition, we discussed how information hiding in language components and their explicit interfaces reduce complexity in language evolution. These contributions provide a starting point for further improvements for evolution in language families.

Currently, feature-oriented LFE is capable of detecting inconsistencies during language family evolution and to indicate potential solutions. But language family evolution is still a mainly manual task. The idea of co-evolution in language families is to provide some automation that avoids such inconsistencies beforehand. This could be done by assisting language engineers in extending and evolving the problem and solution space of a language family simultaneously. Therefore, common evolution scenarios, e.g., the introduction of a complete new language feature, the merge of two features to one, or the split of an existing feature, could be analysed for evolution patterns. An evolution pattern consists of a set of parametrisable operations that perform a consistent evolution in the problem space, the solution space, and the mapping. For instance, a feature-introduction pattern could introduce a new language feature in the problem space, a corresponding language component in the solution space, a new entry in the composition program, and the corresponding feature mappings. Furthermore, it should trigger a review of all affected variant models. Given a set of such patterns, tooling could help language engineers to instantiate a concrete pattern and perform the corresponding changes. Compared to manual evolution, a pattern-based approach is expected to enhance the efficiency and quality of language family evolution.

6.2.2. Role-Based Tool Integration

The integration of heterogeneous tools is vital to increase the productivity gained by software development tools. Such integration can be divided into two aspects. First, tools must share data. Second, the behaviour of different tools needs to be integrated. For sharing data, metamodelling languages have introduced a standardised approach for declaring data structures. It enables tool integrators to understand the data they are dealing with and to access data of different tools in a uniform way. In [Seifert 2010], we identified two general approaches for tool integration based on metamodels. First, proactive tool integration uses existing composition techniques like inheritance and delegation.
to couple the metamodels of tools to integrate. Second, retroactive tool integration relies on model-to-model transformations to map data between given metamodels. The first approach introduces a strong coupling between the integrated metamodels and the respective tools. Consequently, the tools can not be evolved independently or exchanged with other tools. The second approach avoids such strong coupling, but introduces an overhead for synchronising data and complicates the integration of several tools.

To address these issues, we suggested to employ role models for designing families of tool metamodels and role composition for integrating them. This combines the benefits of loose coupling and tight integration. To make tool integration more flexible, we suggested an extension of the LCSL and LCL. The basic idea was to prohibit natural types in role models and to introduce rigidity by a new grounding operator in the composition program. This extension was implemented for our language composition system and applied on a simple tool integration example. The first results show the feasibility of role-based tool integration. However, a real evaluation using a complex integration scenario is still required. In addition, tool integration involves a number of stakeholders with different objectives and requirements that need to be aligned. To continue our initial work and to address the various challenges, further research is required.

6.2.3. Automatic Modularisation of Existing Language Families

The approaches introduced in this thesis prepare the realisation of a language family in a top down approach. A modernisation of existing language families is not directly supported, but requires a complete redesign of a new language family. Our OWL and OCL case studies show the feasibility of such approach. On the other hand, it meant a substantial effort to replicate an existing object-oriented metamodel and other language realisation artefacts. For our role-based composition system, we already implemented a basic transformation, that takes an Ecore metamodel and derives the corresponding role-based language component. This is possible, as the LCSL extends the expressiveness of Ecore. However, this simple transformation results in a monolithic language component that is not modularised.

The idea of automatic modularisation for existing language families is to extend the given transformation with means to derive an initial role-based modularisation. This requires a refactoring of the given language metamodel to introduce role types that act as interfaces between language components. In [Heidenreich 2009b], we presented a refactoring pattern called reference abstraction to prepare languages for extensibility. It introduces an abstract class for every EReference in the language metamodel. The type of the EReference is changed to the introduced abstract class and the previous type is added to its subclasses. The abstract class introduced in this pattern corresponds to a role type in our approach. Language components can be derived by modularising an existing metamodel along such roles. Applying the pattern to every EReference results in a huge number of role types and language components. So the challenge for automatic modularisation is to steer the application of this pattern and, thus, the size and number of language components. We think that metrics that measure and analyse the cohesion between language concepts could be a useful indicator. Alternatively, language
engineers could markup existing metamodels to indicate components. Future work should investigate which approach is feasible, or how both can be combined.

### 6.2.4. Language Component Library

The reuse of language components require their organisation and classification in a library that can be shared among language engineers. This introduces a number of issues. First, a technical approach for managing and distributing language components need to be realised. Second, language components need to be classified and annotated with metadata to enable their location in the library. Third, language components need to be prepared for sharing.

To address the first two issues a distributed language component library is needed. In [Schmidt 2010], Schmidt et al. introduce Picus, a repository for organising and searching software components using facets. Facets allow for annotating components with arbitrary metadata that can later be used to distinguish and discover existing components. Picus implements an extensible approach for defining and assigning such facets. Further work is required to identify specific facets to classify language components.

The third issue requires a revision of current means for component documentation and specification. In our case studies, we experienced that component reuse and integration requires a detailed understanding of the design of a given component. When language components are meant to be reused among language engineers, such understanding needs to be supported by an adequate documentation of language components. Furthermore, additional means to further restrict and validate language component interfaces by tests or additional constraints may be required. Including validation artefacts like tests as a vital part of software component specifications is expected to reduce the overall effort for component validation [Brenner 2007]. As discussed earlier, means to validate a composition program could be further enhanced by exploiting the capabilities of formal semantics approaches. Again this motivates an extension of our current language composition system in future work.

Altogether, we aimed at introducing methodical and technical means for LFE. The approaches introduced in this thesis provide conceptual, technical and qualitative contributions that successfully enhance state-of-the-art in language engineering. We demonstrated and evaluated their practical applicability to built different kinds of language families. In summary, we consider our novel language engineering process and composition technique a step forward for building custom language families in various domains and a foundation for the further research on LFE that was motivated above.
A.1. core Component

component core {

    abstract natural ExpressionInCore {}

    abstract role C_ExpressionRole extends ExpressionInCore {
    }

    natural LetExp extends CoreExpression {
        containment reference Variable vars ( 0 .. -1 ) ;
        containment reference ExpressionInCore exp ( 0 .. 1 ) ;
    }

    abstract natural Type {}

    abstract natural TypedDeclaration {
        containment reference Type typename ( 0 .. 1 ) ;
        attribute EString name ( 0 .. 1 ) ;
    }

    role C_TypeRole extends Type {}

    natural InvalidType extends Type {}

    natural UndefinedType extends Type {}

    natural Variable extends TypedDeclaration {
        containment reference ExpressionInCore init ( 0 .. 1 ) ;
    }

    natural Parameter extends TypedDeclaration {
    }

    abstract natural CoreExpression extends ExpressionInCore {
    }

    natural NavigationCall extends CoreExpression {
        containment reference ExpressionInCore left ( 1 .. 1 ) ;
        attribute EString operator ( 0 .. -1 ) ;
        containment reference CallAtom right ( 0 .. -1 ) ;
    }
natural VariableCall extends CoreExpression {
    attribute EString variableName (0..1);
    containment reference CallModifierRole callModifier (0..1);
}

natural CallAtom {
    containment reference CallModifierRole callModifier (0..1);
}

role CallAtomRole extends CallAtom {}

role CallModifierRole {}
A.2. logic Component

```java
component logic {
    abstract natural LogicExp {}

    abstract natural BinaryLogicExp extends LogicExp {
        attribute EString operator ( 0 .. -1 );
    }

    natural Implies extends BinaryLogicExp {
        containment reference LogicExp left ( 1 .. 1 );
        containment reference LogicExp right ( 1 .. -1 );
    }

    natural AndOrXor extends BinaryLogicExp {
        containment reference LogicExp left ( 1 .. 1 );
        containment reference LogicExp right ( 1 .. -1 );
    }

    abstract natural Comparison extends LogicExp {
        containment reference LogicExp left ( 1 .. 1 );
        containment reference LogicExp right ( 1 .. 1 );
    }

    natural LogicEqual extends Comparison {}

    natural LogicCompare extends Comparison {
        attribute EString relOperator ( 0 .. 1 );
    }

    natural IfThenElse extends LogicExp {
        containment reference LogicExp then ( 1 .. 1 );
        containment reference LogicExp elze ( 1 .. 1 );
        containment reference LogicExp condition ( 1 .. 1 );
    }
}
```

Listing A.2: Syntax specification of OCL core component.
Listing A.3: Role-based specification of OCL logic component in LCSL.

Listing A.4: Syntax specification of OCL logic component.
A.3. math Component

Listing A.5: Role-based specification of OCL math component in LCSL.
Appendix A. OCL Case Study

Listing A.6: Syntax specification of OCL math component.

A.4. collection Component

```java
component collection {
  natural CollectionExp { }
  natural Iterator extends CollectionExp {
    containment reference Col_ExpressionRole exp (1 .. 1);
    containment reference Col_ParameterRole params (0 .. -1);
    attribute EString iteratorOperation (0 .. 1);
  }
  natural Iterate extends Iterator {
    containment reference Col_VariableRole resultVar (1 .. 1);
  }
  natural CollectionOperation extends CollectionExp {
    containment reference Col_ExpressionRole params (0 .. -1);
    attribute EString operationName (0 .. 1);
  }
  natural CollectionLiteral {
    attribute EString collectionTypename (0 .. 1);
    containment reference CollectionLiteralPart items (0 .. -1);
  }
  abstract natural CollectionLiteralPart {}
  natural CollectionRange extends CollectionLiteralPart {
    containment reference Col_ExpressionRole left (1 .. 1);
    containment reference Col_ExpressionRole right (1 .. 1);
  }
```

natural CollectionItem extends CollectionLiteralPart {
    containment reference Col_ExpressionRole exp (1..1);
}

natural CollectionType {
    attribute EString typeName (0..1);
    containment reference Col_TypeRole elementType (1..1);
}

natural CollectionPart {
    containment reference CollectionExp collectionAtom (0..1);
}

role Col_ExpressionRole {}
role Col_ParameterRole {}
role Col_TypeRole {}
role Col_VariableRole {}

Serializable enum CollectionKind {
    0 : Collection = "Collection";
    1 : Sequence = "Sequence";
    2 : Bag = "Bag";
    3 : Set = "Set";
    4 : OrderedSet = "OrderedSet";
}

Listing A.7: Role-based specification of OCL collection component in LCSL.
Appendix A. OCL Case Study

Listing A.8: Syntax specification of OCL collection component.

```
CollectionLiteralExp ::= collectionTypename[COLLECTION_TYPE] "{" (items ("," items)*)? "}";
CollectionRange ::= left "." right;
CollectionItem ::= exp;
CollectionTypeSpecifier ::= typeName[COLLECTION_TYPE] "(" elementType ")";
```

A.5. messages Component

Listing A.9: Role-based specification of OCL messages component in LCSL.

```
component message {
  natural MessageExpPart {
    containment reference MessageAtom atom (0 .. 1);
    attribute EString operator (0 .. 1);
  }
  natural MessageAtom {
    attribute EString messageName (0 .. 1);
    containment reference MessageParameter params (0 .. -1);
  }
  abstract natural MessageParameter {
  }
  natural UnspecifiedValue extends MessageParameter {
    containment reference Mes_TypeRole typeName (0 .. 1);
  }
  natural MessageType {
    attribute EString typename (0 .. 1);
  }
  role Mes_ExpRole extends MessageParameter {} 
  role Mes_TypeRole {} } 
```

```
ABSTRACT SYNTAXDEF message
FOR <http://de.tudresden/message>
<../model-gen/messages.mdl.ecore.genmodel>
IMPORTS {
}
TOKENS {
  DEFINE MESSAGE_OPERATOR $'^' | '^^'$;
  DEFINE MESSAGE_TYPE $ 'OclMessage' $;
  DEFINE NAME $(a..'z'| 'A'..'Z')+ ('a'..'z'| 'A'..'Z' | '0'..'9')+ (':')
```
A.6. tuple Component

component tuple {
    natural TupleLiteral {
        containment reference TupleLiteralElement elements ( 0 .. -1 );
    }
    natural TupleLiteralElement {
        containment reference T_ExpressionRole exp ( 0 .. 1 );
        containment reference T_TypeRole typeName ( 0 .. 1 );
        attribute EString name ( 0 .. 1 );
    }
    natural TupleType {
        containment reference TypeElement elements ( 0 .. -1 );
    }
    natural TypeElement {
        attribute EString name ( 0 .. 1 );
        containment reference T_TypeRole type ( 0 .. 1 );
    }
    role T_ExpressionRole {}
    role T_TypeRole {}
}

Listing A.11: Role-based specification of OCL tuple component in LCSL.

A.6. tuple Component

Listing A.10: Syntax specification of OCL messages component.
Appendix A. OCL Case Study

Listing A.12: Syntax specification of OCL tuple component.

A.7. string Component

Listing A.13: Role-based specification of OCL string component in LCSL.

Listing A.14: Syntax specification of OCL string component.

A.8. classifiercontext Component

192
A.9. attributecontext Component

```plaintext
abstract natural ClassifierConstraint {
    containment reference CC_ExpressionRole exp ( 0 .. 1 ) ;
    attribute EString constraintName ( 0 .. 1 ) ;
}

natural Invariant extends ClassifierConstraint {
}

natural AttributeDefinition extends ClassifierConstraint {
    containment reference CC_TypeRole type ( 0 .. 1 ) ;
    attribute EString attributeName ( 0 .. 1 ) ;
}

natural OperationDefinition extends ClassifierConstraint {
    containment reference CC_ParameterRole params ( 0 .. -1 ) ;
    attribute EString operationName ( 0 .. 1 ) ;
    containment reference CC_TypeRole returnType ( 0 .. 1 ) ;
}

role CC_ExpressionRole {}
role CC_TypeRole {}
role CC_ParameterRole {}
```

Listing A.15: Role-based specification of OCL classifiercontext component in LCSL.

```plaintext
ABSTRACT SYNTAXDEF classifierContext
FOR <http://de.tudresden/classifiercontext>
<../model-gen/classifiercontext.mdl.ecore.genmodel>
START ClassifierContext
IMPORTS {
}
TOKENS {
    DEFINE NAME $('a'..'z'| 'A'..'Z')+ ('a'..'z'| 'A'..'Z' | '0'..'9')* ('::'
        ('a'..'z' | 'A' .. 'Z')+ ('a'..'z'| 'A'..'Z' | '0'..'9')* )*$;
    ('.a'..'z' | 'A' .. 'Z')+ ('a'..'z'| 'A'..'Z' | '0'..'9')* )*$;}
RULES {
    ClassifierContext ::= "context" classifierName[NAME] classifierConstraints+;
    Invariant ::= "inv" (constraintName[NAME])? ";" exp;
    AttributeDefinition ::= "def" (constraintName[NAME])? ";" attributeName[NAME] ";" type "=" exp;
    OperationDefinition ::= "def" (constraintName[NAME])? ";" operationName[NAME] 
    "(" (params ";" params)+")" ("=" returnType);
}
```

Listing A.16: Syntax specification of OCL classifiercontext component.

A.9. attributecontext Component

```plaintext
component attributecontext {
    natural AttributeContext {
```

193
Appendix A. OCL Case Study

4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
attribute EString attributeName ( 0 .. 1 );
containment reference AC_TypeRole type ( 0 .. 1 );
containment reference AttributeConstraint constraints ( 0 .. -1 );
}
natural AttributeConstraint {
attribute EString constraintSpecifier ( 0 .. 1 );
containment reference AC_ExpressionRole exp ( 0 .. 1 );
attribute EString constraintName ( 0 .. 1 );
}
role AC_ExpressionRole { }
role AC_TypeRole { }
}

Listing A.17: Role-based specification of OCL attributecontext component in LCSL.

ABSTRACT SYNTAXDEF attributeContext
FOR http://de.tudresden/attributecontext>
<../model-gen/attributecontext.mdl.ecore.genmodel>
START AttributeContext
TOKENS {
DEFINE NAME $(a'..'z'| 'A'..'Z')+ ('a'..'z'| 'A'..'Z' | '0'..'9')* ('::'
('a'..'z' | 'A' .. 'Z')+ ('a'..'z'| 'A'..'Z' | '0'..'9')+ )*$;
DEFINE ATTRIBUTE_CONSTRAINT_SPECIFIER $ 'init' | 'derive' $;
}
RULES {
AttributeContext ::= "context" attributeName[NAME] (":" type)? constraints+;
AttributeConstraint ::= constraintSpecifier[ATTRIBUTE_CONSTRAINT_SPECIFIER]
(constraintName[NAME])? "": exp;
}

Listing A.18: Syntax specification of OCL attributecontext component.

A.10. operationcontext Component

component operationContext {

natural OperationContext {
containment reference OC_TypeRole returnType ( 0 .. 1 );
containment reference OC_ParameterRole params ( 0 .. -1 );
attribute EString operationName ( 0 .. 1 );
containment reference OperationConstraint constraints ( 0 .. -1 );
}
natural OperationConstraint {
attribute EString constraintName ( 0 .. 1 );
attribute EString constraintSpecifier ( 0 .. 1 );
containment reference OC_ExpressionRole exp ( 0 .. 1 );
}
natural AtPreModifier {}

194
Listing A.19: Role-based specification of OCL operationcontext component in LCSL.

Listing A.20: Syntax specification of OCL operationcontext component.

A.11. package Component

Listing A.21: Role-based specification of OCL package component in LCSL.
Appendix A. OCL Case Study

```
OCLPackage ::= "package" packageName[NAME] contexts="endpackage";
```

Listing A.22: Syntax specification of OCL oclpackage component.

A.12. initial Component

```
component initialElement {
  natural InitialElement {
    containment reference ModelElement model (1..1);
    containment reference OCLPackageDeclaration oclPackage (1..1);
    containment reference MetamodelElement metamodel (1..1);
  }
  natural ModelElement {
    attribute EString modelName (0..1);
  }
  natural MetamodelElement {
    attribute EString metamodelName (0..1);
  }
  role OCLPackageDeclaration {};
}
```

Listing A.23: Role-based specification of OCL initial component in LCSL.

```
ABSTRACT SYNTAXDEF initial
  FOR <http://de.tudresden/initialElement>
  <../model-gen/initial.mdl.ecore.genmodel>
  START InitialElement
  OPTIONS {
  usePredefinedTokens = "false";
  }
  TOKENS {
  DEFINE SL_COMMENT $'//'(~('
'|''|'￿'))* $ ;
  DEFINE ML_COMMENT $'/*'.*'*/'$ ;
  DEFINE WHITESPACE $(' '|'	'|'')$;
  DEFINE LINEBREAKS ($('
'|''|'
')$;
  }
  RULES {
  InitialElement ::= "OCL Constraints" "{"
  model model? oclPackage ");";
  ModelElement ::= "model" modelName["",""];
  MetamodelElement ::= "metamodel" metamodelName["",""];
  }
```

Listing A.24: Syntax specification of OCL initial component.
A.13. temporal Component

```java
component temporal {
  abstract natural ExpressionInTemporal {}  
  abstract natural TemporalExpression extends ExpressionInTemporal {} 
  role Temp_ExpressionRole extends ExpressionInTemporal {} 
  natural Previous extends TemporalExpression { 
    containment reference ExpressionInTemporal exp ( 1 .. 1 ) ; 
  } 
  natural Next extends TemporalExpression { 
    containment reference ExpressionInTemporal exp ( 1 .. 1 ) ; 
  } 
  natural Sometime extends TemporalExpression{ 
    attribute EString conditionOperator (0..1); 
    containment reference ExpressionInTemporal cond ( 0 .. 1 ) ; 
    containment reference ExpressionInTemporal exp ( 1 .. 1 ) ; 
  } 
  natural SometimePast extends TemporalExpression { 
    containment reference ExpressionInTemporal exp ( 1 .. 1 ) ; 
  } 
  natural Always extends TemporalExpression { 
    attribute EString conditionOperator (0..1); 
    containment reference ExpressionInTemporal cond ( 0 .. 1 ) ; 
    containment reference ExpressionInTemporal exp ( 1 .. 1 ) ; 
  } 
  natural AlwaysPast extends TemporalExpression { 
    containment reference ExpressionInTemporal exp ( 1 .. 1 ) ; 
  } 
  abstract natural TemporalAddition{ 
    containment reference ExpressionInTemporal addParams ( 0 .. -1 ) ; 
    attribute EString name ( 1 .. 1 ) ; 
  } 
  natural TemporalPart { 
    attribute EString operatorSymbol ( 0 .. 1 ) ; 
    containment reference TemporalAddition operand ( 1 .. 1 ) ; 
  } 
  abstract natural TempProperty extends TemporalAddition { 
    containment reference ExpressionInTemporal qualifier ( 0 .. -1 ) ; 
  } 
  natural TempPropertyAtPre extends TempProperty {} 
  natural TempPropertyAtNext extends TempProperty {} 
  abstract natural TempOperation extends TemporalAddition { 
    containment reference ExpressionInTemporal params ( 0 .. -1 ) ; 
  } 
  natural TempOperationAtPre extends TempOperation {} 
  natural TempOperationAtNext extends TempOperation {} 
}
```
natural TempOperationAtNext extends TempOperation {}
import <platform:/resource/org.langems.ocl.module.operationcontext/model/operationContext.mdl> as <operationcontext>
import <platform:/resource/org.langems.ocl.module.core/model/core.mdl> as <core>
import <platform:/resource/org.langems.ocl.module.math/model/math.mdl> as <math>
import <platform:/resource/org.langems.ocl.module.logic/model/logic.mdl> as <logic>
import <platform:/resource/org.langems.ocl.module.tuples/model/tuple.mdl> as <tuple>
import <platform:/resource/org.langems.ocl.module.stringliteral/model/stringliteral.mdl> as <stringliteral>
import <platform:/resource/org.langems.ocl.module.messages/model/messages.mdl> as <messages>
import <platform:/resource/org.langems.ocl.module.collection/model/collection.mdl> as <collection>
import <platform:/resource/org.langems.ocl.module.temporal/model/temporal.mdl> as <temporal>

{ integrating: <initial>
  // include core
  <core> refines <classifierContext> { }
  Type plays CC_TypeRole { }
  <core> refines <attributecontext> { }
  Type plays AC_TypeRole { }
  <core> refines <operationcontext> { }
  Type plays OC_TypeRole { }

  // include logic
  <logic> refines <core> { }
  BooleanType plays C_TypeRole { }
  <logic> refines <classifierContext> { }
  <logic> refines <attributecontext> { }
  <logic> refines <operationcontext> { }

  // include math
  <math> refines <core> { }
  IntegerType plays C_TypeRole { }
  <math> refines <logic> { }
  Additive plays LogicLiteralRole { }
  <core> refines <math> { }
  CoreExpression plays MathLiteralRole { }
}
// include collection
<core> refines <collection> {
  Type plays Col_TypeRole { }
  Variable plays Col_VariableRole { }
  Parameter plays Col_ParameterRole { }
}

<logic> refines <collection> {
  Implies plays Col_ExpressionRole{ }
}

<collection> refines <core> {
  CollectionPart plays CallAtomRole { }
  CollectionType plays C_TypeRole {}
}

<collection> refines <math> {
  CollectionLiteral plays MathLiteralRole { }
}

// add tuple
<tuple> refines <core> {
  TupleType plays C_TypeRole { }
}

<core> refines <tuple> {
  Type plays T_TypeRole { }
}

<logic> refines <tuple> {
  Implies plays T_ExpressionRole { }
}

<tuple> refines <logic> {
  TupleLiteral plays LogicLiteralRole { }
}

// include messages
<core> refines <messages> {
  Type plays Mes_TypeRole { }
}

<logic> refines <messages> {
  Implies plays Mes_ExpRole{ }
}

<messages> refines <core> {
  MessageExpPart plays CallModifierRole { }
  MessageType plays C_TypeRole {}
}

// include string literal
<stringliteral> refines <core> {
  StringType plays C_TypeRole { }
}

<stringliteral> refines <logic> {
  StringLiteral plays LogicLiteralRole { }
}

// include classifier context
<classifierContext> refines <package> {
  ClassifierContext plays ConstraintContext { }
}

// include attribute context
<attributecontext> refines <package> {
  AttributeContext plays ConstraintContext { }
}
A.14. OCL Composition Program

Listing A.27: Composition program for complete OCL including temporal and initial extensions in LCL.
B.1. sheet Component

```lcsldoc
component sheet {
  natural Sheet {
    containment reference Statement statements (0..-1);
  }
  role Statement {
    roleOperation void execute();
  }
}
```

Listing B.1: Role-based specification of SumUp sheet component in LCSL.

```lcsldoc
ABSTRACT SYNTAXDEF sumup
  FOR <http://de.tudresden/sheet> <../model-gen/sheet.mdl.ecore.genmodel>
  START Sheet
  OPTIONS {
    usePredefinedTokens = "false";
  }
  TOKENS {
    DEFINE SL_COMMENT $'//'(~('
'|''|'￿'))*$ $;
    DEFINE ML_COMMENT $'/*'.*"*/'$;
    DEFINE WHITESPACE $(' '|'	'|'')$;
    DEFINE LINEBREAKS $('
'|''|'
')$;
  }
  TO longest
  TOKENSTYLES {
    "NAME" COLOR #000000;
    "SL_COMMENT", "ML_COMMENT" COLOR #00bb00;
    "STRING_LITERAL" COLOR #2A00FF;
  }
  RULES {
```
### Appendix B. SumUp Case Study

**Listing B.2:** Syntax specification of SumUp sheet component.

```java
package org.langems.sumup.module.sheet;

abstract petrinet sheet {

    types <../model-gen/sheet.mdl.ecore>;

    types <platform:/plugin/org.eclipse.emf.ecore/model/Ecore.ecore>;

    { // sheet
        place sheet : Sheet
        place statement_execute_MUTEX : EInt
        place statement_in_execute : Statement
        place statement_execute : Statement

        sheet - sheet -> prepareStatement
        statement_execute_MUTEX - mutex -> prepareStatement
        prepareStatement - sheet() -> sheet
        prepareStatement - statement() -> statement_in_execute

        transition prepareStatement
        if (sheet.getStatements().size().greaterThan(mutex))
        do {
            statement = sheet.getStatements().get(mutex);
        }

        statement_execute - evaluated -> evaluateStatement
        sheet - s -> evaluateStatement
        evaluateStatement - mutex() -> statement_execute_MUTEX
        evaluateStatement - s() -> sheet

        transition evaluateStatement
        if (s.getStatements().indexOf(evaluated).greaterEqual(0))
        do {
            mutex = s.getStatements().indexOf(evaluated).add(1);
        }
    }
}
```

**Listing B.3:** CPN-based semantics specification of SumUp sheet component.

```java
package org.langems.sumup.module.sheet;

abstract petrinet sheet {

    types <../model-gen/sheet.mdl.ecore>;

    types <../model-gen/sheet.mdl.ecore.genmodel>;

    types <platform:/plugin/org.eclipse.emf.ecore/model/Ecore.ecore>;

    types <platform:/plugin/org.eclipse.emf.ecore/model/Ecore.genmodel>;

    { // sheet
        place sheet : Sheet
        place statement_execute_MUTEX : EInt
        place statement_in_execute : Statement
        place statement_execute : Statement

        sheet - sheet -> prepareStatement
        statement_execute_MUTEX - mutex -> prepareStatement
        prepareStatement - sheet() -> sheet
        prepareStatement - statement() -> statement_in_execute

        transition prepareStatement
        if (sheet.getStatements().size().greaterThan(mutex))
        do {
            statement = sheet.getStatements().get(mutex);
        }

        statement_execute - evaluated -> evaluateStatement
        sheet - s -> evaluateStatement
        evaluateStatement - mutex() -> statement_execute_MUTEX
        evaluateStatement - s() -> sheet

        transition evaluateStatement
        if (s.getStatements().indexOf(evaluated).greaterEqual(0))
        do {
            mutex = s.getStatements().indexOf(evaluated).add(1);
        }
    }
}
```

### B.2. math Component

```java
component math {

    exported natural MathExpressionContainer {
        attribute EDouble result;
        containment reference MathExpression exp (1..1);
    }

    abstract natural MathExpression {
```
attribute EDouble result;

abstract natural BinaryExpression extends MathExpression {
    containment reference MathExpression left (1..1);
    containment reference MathExpression right (1..1);
}

natural Additive extends BinaryExpression {
    attribute AdditiveOperator operator (1..1);
}

natural Multiplicative extends BinaryExpression {
    attribute MultiplicativeOperator operator (1..1);
}

natural MathPrimitiveContainer extends MathExpression {
    containment reference MathPrimitive primitive (1..1);
}

role MathPrimitive {
    attribute EDouble value;
}

natural NumberLiteral extends MathExpression {
    attribute AdditiveOperator sign (0..1);
    attribute EDouble value (1..1);
}

natural Nested extends MathExpression {
    containment reference MathExpression expression (1..1);
}

serializable enum AdditiveOperator {
    1 : ADD = "+";
    2 : MINUS = "-";
}

serializable enum MultiplicativeOperator {
    1 : TIMES = "*";
    2 : DIV = "/";
}

Listing B.4: Role-based specification of SumUp math component in LCSL.
Appendix B. SumUp Case Study

Listing B.5: Syntax specification of SumUp math component.

```java
package org.langems.sumup.module.math;

abstract petrinet math

types <../model-gen/math.mdl.ecore>
<../model-gen/math.mdl.ecore.genmodel>;

types <platform:/plugin/org.eclipse.emf.ecore/model/Ecore.ecore>
<platform:/plugin/org.eclipse.emf.ecore/model/Ecore.genmodel> ;
{

// math

place mathExpressionContainer : MathExpressionContainer
place mathExpressionContainer_waitingForExpression : MathExpressionContainer
place mathExpressionContainer_evaluate : MathExpressionContainer

mathExpressionContainer - mec -> initExpression
initExpression - e() -> expressions
initExpression - mec() -> mathExpressionContainer_waitingForExpression

transition initExpression
do {
  e = mec.getExp();
}

expressions_evaluate - e -> evalMec
mathExpressionContainer_waitingForExpression - mec -> evalMec
evalMec - mec(result := result) -> mathExpressionContainer_evaluate

transition evalMec
if {
  mec.getExp().equals(e)
}
do {
  result = e.getResult();
}

place expressions : MathExpression
place expressions_in_evaluate : MathExpression
place expressions_evaluate : MathExpression
place mathPrimitive_in_value : MathPrimitive
place mathPrimitive_value : MathPrimitive
place mathPrimitiveContainer_waiting : MathPrimitiveContainer
```


```java
Additive ::= left operator[ADD : "+", MINUS : "-" ] right;

@Operator(type="binary_left_associative", weight="2", superclass="MathExpression")
Multiplicative ::= left operator[TIMES : "*", DIV : "/" ] right;

@Operator(type="primitive", weight="3", superclass="MathExpression")
NumberLiteral ::= sign[MINUS : "-", ADD : ""]? value[FLOAT];

@Operator(type="primitive", weight="3", superclass="MathExpression")
Nested ::= "(" expression ");"

MathPrimitiveContainer ::= primitive;
```
B.2. math Component

expressions - e -> prepare_left_and_right
prepare_left_and_right - left() -> expressions
prepare_left_and_right - right() -> expressions
prepare_left_and_right - e() -> expressions_in_evaluate

transition prepare_left_and_right
if (@BinaryExpression.isInstance(e))
do {
    left = ((BinaryExpression) e).getLeft();
    right = ((BinaryExpression) e).getRight();
}
expressions - e -> prepare_non_binary
prepare_non_binary - e() -> expressions_in_evaluate

transition prepare_non_binary
if (@BinaryExpression.isInstance(e).not() && @Nested.isInstance(e).not())
do {
}
expressions - e -> prepare_nested
prepare_nested - e() -> expressions_in_evaluate
prepare_nested - inner() -> expressions

transition prepare_nested
if ( @Nested.isInstance(e) )
do {
    inner = ((Nested) e).getExpression();
}
expressions_in_evaluate - e -> add
expressions_evaluate - a -> add
expressions_evaluate - b -> add
add - e(result := result) -> expressions_evaluate

transition add
if ( @Additive.isInstance(e) && ((BinaryExpression) e).getLeft().equals(a) && ((BinaryExpression) e).getRight().equals(b) && ((Additive) e).getOperator().toString().equals("+"))
do {
    result = a.getResult().add(b.getResult());
}
expressions_in_evaluate - e -> sub
expressions_evaluate - a -> sub
expressions_evaluate - b -> sub
sub - e(result := result) -> expressions_evaluate

transition sub
if ( @Additive.isInstance(e) && ((BinaryExpression) e).getLeft().equals(a) && ((BinaryExpression) e).getRight().equals(b) && ((Additive) e).getOperator().toString().equals("-"))
}
Appendix B. SumUp Case Study

```java
appendix
108  do {
109      result = a.getResult().sub(b.getResult());
110  }
111
expressions_in_evaluate - e -> mult
expressions_evaluate - a -> mult
expressions_evaluate - b -> mult
mult - e(result := result) -> expressions_evaluate

117  transition mult
118  if (OMultiplicative.isInstance(e)
119      && ((BinaryExpression) e).getLeft().equals(a)
120      && ((BinaryExpression) e).getRight().equals(b)
121      && ((OMultiplicative) e).getOperator().toString().equals("*")
122      )
123  do {
124      result = a.getResult().mult(b.getResult());
125  }
126
expressions_in_evaluate - e -> div
expressions_evaluate - a -> div
expressions_evaluate - b -> div
div - e(result := result) -> expressions_evaluate

132  transition div
133  if (OMultiplicative.isInstance(e)
134      && ((BinaryExpression) e).getLeft().equals(a)
135      && ((BinaryExpression) e).getRight().equals(b)
136      && ((OMultiplicative) e).getOperator().toString().equals("/")
137      )
138  do {
139      result = a.getResult().div(b.getResult());
140  }
141
expressions_in_evaluate - e -> literal
literal - e(result := result) -> expressions_evaluate

146  transition literal
147  if (e instanceof NumberLiteral)
148      && ((NumberLiteral) e).getSign().toString().equals("+")
149  )
150  do {
151      result = ((NumberLiteral) e).getValue();
152  }
153
expressions_in_evaluate - e -> literalMinus
literalMinus - e(result := result) -> expressions_evaluate

157  transition literalMinus
158  if (e instanceof NumberLiteral)
159      && ((NumberLiteral) e).getSign().toString().equals("-")
160  )
161  do {
162      result = ((NumberLiteral) e).getValue().mult(- 1.0);
163  }
164
expressions_in_evaluate - e -> nested
expressions_evaluate - inner -> nested
nested - e(result := result) -> expressions_evaluate

208
```
B.3. conditional Component

Listing B.6: CPN-based semantics specification of SumUp math component.

```java
transition nested
if (
  @Nested.isInstance(e)
  && ((Nested) e).getExpression().equals(inner)
) do {
  result = inner.getResult();
}

expression_in_evaluate - e -> prepare_mathPrimitive
prepare_mathPrimitive - m() -> mathPrimitive_in_value
prepare_mathPrimitive - mc() -> mathPrimitiveContainer_waiting

transition prepare_mathPrimitive
if (  
  @MathPrimitiveContainer.isInstance(e) 
) do { 
  mc = (MathPrimitiveContainer) e;
  m = mc.getPrimitive();
}

mathPrimitiveContainer_waiting - mc -> eval_container
mathPrimitive_value - m -> eval_container
eval_container - mc(result := result) -> expressions_evaluate

transition eval_container
if (  
  mc.getPrimitive().equals(m) 
) do { 
  result = m.getValue();
}
}
```

B.3. conditional Component

```java
component conditional {
  exported natural ConditionalExpression {
    containment reference BooleanExpression condition (1..1);
    containment reference Consequence consequence (1..1);
  }
  role Consequence {
    roleOperation void evaluate();
  }
  role Argument {
    attribute EDouble numericalValue (1..1);
  }
  abstract natural BooleanExpression {
    attribute EBoolean result (1..1);
  }
}
```
Appendix B. SumUp Case Study

natural Comparison extends BooleanExpression {
    attribute CompareOperator operator (1..1);
    containment reference Argument argumentA (1..1);
    containment reference Argument argumentB (1..1);
}

natural BooleanLiteral extends BooleanExpression, Argument {
    attribute EBoolean value (1..1);
}

enum CompareOperator {
    1 : GREATER = ">";
    2 : GREATER_EQUAL = ">=";
    3 : LESS = "<";
    4 : LESS_EQUAL = "<=";
    5 : EQUAL = "=";
}

Listing B.7: Role-based specification of SumUp conditional component in LCSL.

ABSTRACT SYNTAXDEF sumup
FOR <http://de.tudresden/conditional>
<../model-gen/conditional.mdl.ecore.genmodel>
OPTIONS {
}
RULES {
    ConditionalExpression ::= "when" condition ":" consequence;
    Comparison ::= argumentA (operator[GREATER : ">", GREATER_EQUAL : ">=",
    LESS : "<", LESS_EQUAL : "<=", EQUAL : "="] argumentB);
    BooleanLiteral ::= value["true":"false"];
}

Listing B.8: Syntax specification of SumUp conditional component.

package org.langems.sumup.module.conditional;
abstract petrinet conditional

types <../model-gen/conditional.mdl.ecore>
<../model-gen/conditional.mdl.ecore.genmodel>;

types <platform:/plugin/org.eclipse.emf.ecore/model/Ecore.ecore>
<platform:/plugin/org.eclipse.emf.ecore/model/Ecore.genmodel> ;
{
    // conditional

    place conditional : ConditionalExpression
    place conditional_waiting : ConditionalExpression
    place conditional_evaluate : ConditionalExpression
    place condition_in_evaluate : BooleanExpression
    place consequence_in_evaluate : Consequence
    place condition_waiting : Comparison
    place condition_evaluate : BooleanExpression

210
B.3. conditional Component

```java
place consequence_evaluate : Consequence
place argument_in_numericalValue : Argument
place argument_numericalValue : Argument

conditional - condi -> prepareCondition
prepareCondition - guard() -> condition_in_evaluate
prepareCondition - condi() -> conditional_waiting
transition prepareCondition
do {
    guard = condi.getCondition();
}

condition_in_evaluate - condi -> evalBoolLiteral
evalBoolLiteral - condi(result := result) -> condition_evaluate

transition evalBoolLiteral
if (BooleanLiteral.isInstance(condi))
do {
    result = ((BooleanLiteral) condi).isValue();
}

argument_in_numericalValue - e -> evalBoolLiteralArg
evalBoolLiteralArg - e(numericalValue := result) -> argument_numericalValue

transition evalBoolLiteralArg
if (BooleanLiteral.isInstance(e))
do {
    result = ((BooleanLiteral) e).isValue().hashValue().doubleValue();
}

condition_in_evaluate - condi -> prepareBoolExpression
prepareBoolExpression - comparison() -> condition_waiting
prepareBoolExpression - a1() -> argument_in_numericalValue
prepareBoolExpression - a2() -> argument_in_numericalValue

transition prepareBoolExpression
if (Comparison.isInstance(condi))
do {
    a1 = ((Comparison) condi).getArgumentA();
    a2 = ((Comparison) condi).getArgumentB();
    comparison = (Comparison) condi;
}

condition_waiting - comparison -> evalEquals
argument_numericalValue - a1 -> evalEquals
argument_numericalValue - a2 -> evalEquals
evalEquals - comparison(result := result) -> condition_evaluate

transition evalEquals
if (comparison.getArgumentA().equals(a1) &&
    comparison.getArgumentB().equals(a2) &&
    comparison.getOperator().toString().equals("="))
do {
    result = a1.getNumericalValue().equals(a2.getNumericalValue());
}

condition_waiting - comparison -> evalG
argument_numericalValue - a1 -> evalG
argument_numerical_value - a2 -> evalG
evalG - comparison(result := result) -> condition_evaluate

transition evalG
```
if (comparison.getArgumentA().equals(a1) &&
    comparison.getArgumentB().equals(a2) &&
    comparison.getOperator().toString().equals(">")
) do {
    result = a1.getNumericalValue().greaterThan(a2.getNumericalValue());
}

transition evalGE
if (comparison.getArgumentA().equals(a1) &&
    comparison.getArgumentB().equals(a2) &&
    comparison.getOperator().toString().equals(">=")
) do {
    result = a1.getNumericalValue().greaterEqual(a2.getNumericalValue());
}

transition evalLE
if (comparison.getArgumentA().equals(a1) &&
    comparison.getArgumentB().equals(a2) &&
    comparison.getOperator().toString().equals("<=")
) do {
    result = a1.getNumericalValue().lessEqual(a2.getNumericalValue());
}

transition evalL
if (comparison.getArgumentA().equals(a1) &&
    comparison.getArgumentB().equals(a2) &&
    comparison.getOperator().toString().equals("<")
) do {
    result = a1.getNumericalValue().lessThan(a2.getNumericalValue());
}

conditional_waiting - condi -> evaluateConsequence
condition_evaluate - guard -> evaluateConsequence
evaluateConsequence - consequence() -> consequence_in_evaluate
evaluateConsequence - condi() -> conditional_waiting

transition evaluateConsequence
if (condi.getCondition().equals(guard)
    && guard.isResult() ) do {
    consequence = condi.getConsequence();
}

conditional_waiting - condi -> evaluateConditional
consequence_evaluate - consequence -> evaluateConditional
evaluateConditional - condi() -> conditional_evaluate
Listing B.9: CPN-based semantics specification of SumUp conditional component.

B.4. units Component

Listing B.10: Role-based specification of SumUp units component in LCSL.

ABSTRACT SYNTAX DEF unit FOR <http://de.tudresden/units> <../model-gen/units.mdl.ecore.genmodel> OPTIONS {

TOKENS {

DEFINE NAME $(a'..z' | 'A'..'Z')+(a'..'z'| 'A'..'Z' | '0'..'9')+(::'
(a'..'z' | 'A' .. 'Z')+(a'..'z'| 'A'..'Z' | '0'..'9')+ )$;$

DEFINE FLOAT $((('1'..'9') ('0'..'9')+ | '0') ('.' ('0'..'9')+ )+)?$;

RULES {

Unit ::= name[NAME];

UnitDeclaration ::= "unit" unit;

}
Appendix B. SumUp Case Study

Listing B.11: Syntax specification of SumUp units component.

```java
package org.langems.sumup.module.units;

abstract petrinet units

types <../model-gen/units.mdl.ecore> ;
types <platform:/plugin/org.eclipse.emf.ecore/model/Ecore.ecore> ;
{
place elementWithUnit_in_unitsNumerator : ElementWithUnit
place elementWithUnit_in_unitsDenominator : ElementWithUnit
place elementWithUnit_unitsNumerator : ElementWithUnit
place elementWithUnit_unitsDenominator : ElementWithUnit
}

Listing B.12: CPN-based semantics specification of SumUp units component.

B.5. variables Component

Listing B.13: Role-based specification of SumUp variables component in LCSL.

```
```
Listing B.14: Syntax specification of SumUp variables component.
Appendix B. SumUp Case Study

Listing B.15: CPN-based semantics specification of SumUp variables component.

B.6. strings Component

Listing B.16: Role-based specification of SumUp strings component in LCSL.

Listing B.17: Syntax specification of SumUp strings component.
B.7. SumUp Composition Program

Listing B.18: CPN-based semantics specification of SumUp strings component.

```java
composer

import <platform:/resource/org.langems.sumup.module.sheet/model/sheet.mdl> as <sheet>
import <platform:/resource/org.langems.sumup.module.math/model/math.mdl> as <math>
import <platform:/resource/org.langems.sumup.module.variables/model/variables.mdl> as <variables>
import <platform:/resource/org.langems.sumup.module.conditional/model/conditional.mdl> as <conditional>
import <platform:/resource/org.langems.sumup.module.strings/model/strings.mdl> as <string>
import <platform:/resource/org.langems.sumup.module.units/model/units.mdl> as <units>

{ integrating: <sheet>

  // adds math
  <math> refines <sheet> {
    MathExpressionContainer plays Statement {
      execute() : in mathExpressionContainer out mathExpressionContainer_evaluate
    }
  }

  // adds variables
  <variables> refines <sheet> {
    VariableDeclaration plays Statement {
      execute() : in variableDeclaration out variableDeclaration_evaluate
    }
    Assignment plays Statement {
      execute() : in variableAssignment out variableAssignment_evaluate
    }
  }<math> refines <variables> {
    MathExpressionContainer plays AssignmentExpression {
      assignmentValue : get: in mathExpressionContainer out mathExpressionContainer_evaluate
      result *result = player.getResult();*
    }
  }
  <variables> refines <math> {
    VariableReference plays MathPrimitive {
      value : get: in variableReference_evaluate out variableReference_evaluate
      result *result = player.getValue();*
      net extension *place variableReference_evaluate : VariableReference*
    }
  }

  // adds conditional
  <conditional> refines <sheet> {
    ConditionalExpression plays Statement {
      execute() : in conditional out conditional_evaluate
    }
  }

217
```
Listing B.19: Composition program for complete SumUp language in LCL.
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>AG</td>
<td>Attribute Grammar</td>
<td>39</td>
</tr>
<tr>
<td>ADL</td>
<td>Architecture Description Language</td>
<td>92</td>
</tr>
<tr>
<td>AOP</td>
<td>Aspect-Oriented Programming</td>
<td>126</td>
</tr>
<tr>
<td>ASM</td>
<td>Abstract State Machine</td>
<td>40</td>
</tr>
<tr>
<td>BEDSL</td>
<td>Business Entities DSL</td>
<td>81</td>
</tr>
<tr>
<td>BPMN</td>
<td>Business Process Model and Notation</td>
<td>81</td>
</tr>
<tr>
<td>CDA</td>
<td>Closed Domain Assumption</td>
<td>76</td>
</tr>
<tr>
<td>CPN</td>
<td>Coloured Petri Net</td>
<td>10</td>
</tr>
<tr>
<td>CSS</td>
<td>Cascading Style Sheets</td>
<td>33</td>
</tr>
<tr>
<td>CWA</td>
<td>Closed World Assumption</td>
<td>23</td>
</tr>
<tr>
<td>DAML</td>
<td>DARPA Agent Markup Language</td>
<td>63</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td><strong>DBDSL</strong></td>
<td>DataBase DSL</td>
<td>81</td>
</tr>
<tr>
<td><strong>DSL</strong></td>
<td>Domain-Specific Language</td>
<td>1</td>
</tr>
<tr>
<td><strong>DL</strong></td>
<td>Description Logic</td>
<td>23</td>
</tr>
<tr>
<td><strong>DGO</strong></td>
<td>Delegating Compiler Object</td>
<td>30</td>
</tr>
<tr>
<td><strong>EBNF</strong></td>
<td>Extended Backus-Naur Form</td>
<td>29</td>
</tr>
<tr>
<td><strong>EMF</strong></td>
<td>Eclipse Modelling Framework</td>
<td>18</td>
</tr>
<tr>
<td><strong>EOL</strong></td>
<td>Epsilon Object Language</td>
<td>4</td>
</tr>
<tr>
<td><strong>ER</strong></td>
<td>Entity-Relationship</td>
<td>22</td>
</tr>
<tr>
<td><strong>GEF</strong></td>
<td>Graphical Editing Framework</td>
<td>35</td>
</tr>
<tr>
<td><strong>GEMS</strong></td>
<td>Generic Eclipse Modeling System</td>
<td>33</td>
</tr>
<tr>
<td><strong>GMF</strong></td>
<td>Graphical Modelling Framework</td>
<td>32</td>
</tr>
<tr>
<td><strong>GPL</strong></td>
<td>General Purpose Language</td>
<td>2</td>
</tr>
<tr>
<td><strong>GrUML</strong></td>
<td>Graph Unified Modelling Language</td>
<td>22</td>
</tr>
<tr>
<td><strong>GXL</strong></td>
<td>Graph eXchange Language</td>
<td>21</td>
</tr>
<tr>
<td><strong>GRS</strong></td>
<td>Graph Rewrite System</td>
<td>50</td>
</tr>
<tr>
<td><strong>FOSD</strong></td>
<td>Feature-Oriented Software Development</td>
<td>8</td>
</tr>
<tr>
<td><strong>FODA</strong></td>
<td>Feature-Oriented Domain Analysis</td>
<td>47</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
<td>20</td>
</tr>
<tr>
<td>KM3</td>
<td>KM3 Metamodelling Language</td>
<td>18</td>
</tr>
<tr>
<td>LFE</td>
<td>Language Family Engineering</td>
<td>2</td>
</tr>
<tr>
<td>LCSL</td>
<td>Language Component Specification Language</td>
<td>98</td>
</tr>
<tr>
<td>LCL</td>
<td>Language Composition Language</td>
<td>98</td>
</tr>
<tr>
<td>MEDSL</td>
<td>Managed Entities DSL</td>
<td>81</td>
</tr>
<tr>
<td>MDE</td>
<td>Model-Driven Engineering</td>
<td>1</td>
</tr>
<tr>
<td>MDD</td>
<td>Model-Driven Development</td>
<td>1</td>
</tr>
<tr>
<td>MDSD</td>
<td>Model-Driven Software Development</td>
<td>1</td>
</tr>
<tr>
<td>MOF</td>
<td>MetaObject Facility</td>
<td>4</td>
</tr>
<tr>
<td>EMOF</td>
<td>Essential MOF</td>
<td>4</td>
</tr>
<tr>
<td>CMOF</td>
<td>Complete MOF</td>
<td>4</td>
</tr>
<tr>
<td>MOST TOPF</td>
<td>MOST TOol Product Family</td>
<td>80</td>
</tr>
<tr>
<td>OIL</td>
<td>Ontology Inference Layer</td>
<td>63</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
<td>4</td>
</tr>
<tr>
<td>OWA</td>
<td>Open World Assumption</td>
<td>23</td>
</tr>
<tr>
<td>OWL</td>
<td>Web Ontology Language</td>
<td>4</td>
</tr>
</tbody>
</table>
**VSM** Variant-Specific Model .......................................................... 50

**XML** eXtensible Markup Language .................................................. 21


239


240


