Object-Oriented Development for Reconfigurable Architectures

Von der Fakultät für Mathematik und Informatik
der Technischen Universität Bergakademie Freiberg
genehmigte

DISSERTATION
zur Erlangung des akademischen Grades
Doktor Ingenieur
Dr.-Ing.,
vorgelegt

von Dipl.-Inf. (FH) Dominik Fröhlich

geboren am 19. Februar 1974

Gutachter: Prof. Dr.-Ing. habil. Bernd Steinbach (Freiberg)
Prof. Dr.-Ing. Thomas Beierlein (Mittweida)
PD Dr.-Ing. habil. Michael Ryba (Osnabrück)

To my parents.
Reconfigurable hardware architectures have been available now for several years. Yet the application development for such architectures is still a challenging and error-prone task, since the methods, languages, and tools being used for development are inappropriate to handle the complexity of the problem. This hampers the widespread utilization, despite of the numerous advantages offered by this type of architecture in terms of computational power, flexibility, and cost.

This thesis introduces a novel approach that tackles the complexity challenge by raising the level of abstraction to system-level and increasing the degree of automation. The approach is centered around the paradigms of object-orientation, platforms, and modeling. An application and all platforms being used for its design, implementation, and deployment are modeled with objects using UML and an action language. The application model is then transformed into an implementation, whereby the transformation is steered by the platform models.

In this thesis solutions for the relevant problems behind this approach are discussed. It is shown how UML can be used for complete and precise modeling of applications and platforms. Application development is done at the system-level using a set of well-defined, orthogonal platform models. Thereby the core features of object-orientation - data abstraction, encapsulation, inheritance, and polymorphism - are fully supported.

Novel algorithms are presented, that allow for an automatic mapping of such application models to the target architecture. Thereby the problems of platform mapping, estimation of implementation characteristics, and synthesis of UML models are discussed. The thesis explores the utilization of platform models for generation of highly optimized implementations in an automatic yet adaptable way. The approach is evaluated by a number of relevant applications.

The execution of the generated implementations is supported by a run-time service. This service manages the hardware configurations and objects comprising the application. Moreover, it serves as broker for hardware objects. The efficient management of configurations and objects at run-time is discussed and optimized life cycles for these entities are proposed. Mechanisms are presented that make the approach portable among different physical hardware architectures.

Further, this thesis presents UML profiles and example platforms that support system-level design. These extensions are embodied in a novel type of model compiler. The compiler is accompanied by an implementation of the run-time service. Both have been used to evaluate and improve the presented concepts and algorithms.
ACKNOWLEDGEMENTS

This work would have never been started or even finished without the support of many people. I am particularly grateful to my advisors Prof. Steinbach and Prof. Beierlein for the long hours they spent with me in discussing the tangible pieces of this work. Especially thankful I am to Prof. Beierlein and my colleague Thomas Oehme for providing me the time and encouragement to finish this thesis.

I am greatly indebted to my parents Regina and Elmar for their seamlessly never-ending emotional and financial support, and patience while I was doing my academic endeavours. Without them this work would have never been possible.

The development of only the most important parts of the MOCCA project took many man-years of effort. Clearly, this can not be done by a single person in such a restricted time frame. Therefore, I am grateful to Isabel Drost, Alexander Faber, Andre Gauter, Thomas Mathiebe, Henning Riedel, and Vitali Tomm for the discussion and development of parts of MOCCA. Peter Grünberg helped me preparing most of the experimental results. Particular thanks go to Frank Anke for his work in improving the quality of the compiler and performing the experiments on model-driven graphical user interface generation.

Many thanks to Jana, Anja, Susi, Tina, Erik, Lars, and many more who offered my their friendship, and provided me an emotional refuge. Particularly I’d like to thank Mandy. We met in a time of my life when I was almost giving up finishing this thesis. It was her emotional support and kind nature that made me smile again and continue to the end.
CONTENTS

List of Figures .................................................. vii
List of Tables ................................................... xi
List of Algorithms .............................................. xv
List of Listings .................................................. xvii
List of Acronyms ............................................... xix
List of Symbols ............................................... xxiii

1. Introduction .................................................. 1
   1.1 Motivation ................................................. 1
   1.2 Related Work .............................................. 3
   1.3 Contributions and Restrictions ........................... 4
   1.4 Overview .................................................. 5

2. Theoretical and Technological Foundations .................. 7
   2.1 Reconfigurable Computing ................................. 7
      2.1.1 Design Space of Reconfigurable Computing ............ 7
      2.1.2 System-Level Design .................................. 9
   2.2 Hardware Design with High-Level Languages ............. 14
      2.2.1 High-Level Languages for Hardware Design .......... 14
      2.2.2 Design Space of VHDL ................................. 15
      2.2.3 Hardware Design Flow ................................ 16
   2.3 The Unified Modeling Language ........................... 18
      2.3.1 Design Space of UML ................................ 18
      2.3.2 System Design with UML .............................. 22

   3.1 System-Level Design with UML .......................... 25
      3.1.1 UML as System-Level Design Language .............. 25
      3.1.2 MOCCA Action Language .............................. 26
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>Model-Driven Development Methodology</td>
<td>31</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Co-Design, Platform-based Design and Model-Driven Architecture</td>
<td>31</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Model-Driven, Platform-Based System-Level Design</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>Platforms and Models</td>
<td>36</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Use-Case Model</td>
<td>36</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Design Platform Model</td>
<td>36</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Design Model</td>
<td>41</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Implementation Platform Model</td>
<td>42</td>
</tr>
<tr>
<td>3.3.5</td>
<td>Implementation Model</td>
<td>49</td>
</tr>
<tr>
<td>3.3.6</td>
<td>Deployment Platform Model</td>
<td>52</td>
</tr>
<tr>
<td>3.3.7</td>
<td>Deployment Model</td>
<td>53</td>
</tr>
<tr>
<td>4.</td>
<td>Platform Mapping</td>
<td>55</td>
</tr>
<tr>
<td>4.1</td>
<td>Platform Mapping for Object-Oriented Specifications</td>
<td>55</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Definition of the Mapping Problem</td>
<td>55</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Challenges</td>
<td>55</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Structure of the Design Space</td>
<td>56</td>
</tr>
<tr>
<td>4.2</td>
<td>Target Platform Architecture</td>
<td>60</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Architectural Illusions</td>
<td>60</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Implementation Options</td>
<td>61</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Architectural Constraints</td>
<td>62</td>
</tr>
<tr>
<td>4.3</td>
<td>Platform Mapping Algorithms</td>
<td>63</td>
</tr>
<tr>
<td>4.3.1</td>
<td>A Platform-Based Distributed Mapping Approach</td>
<td>63</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Mapping Control</td>
<td>65</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Breeding of Mappings</td>
<td>66</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Computation of Candidate Mappings</td>
<td>73</td>
</tr>
<tr>
<td>4.3.5</td>
<td>Mapping Evaluation</td>
<td>76</td>
</tr>
<tr>
<td>4.4</td>
<td>Estimation of Model Characteristics</td>
<td>77</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Estimation of Execution Characteristics</td>
<td>77</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Estimation of Implementation Characteristics</td>
<td>78</td>
</tr>
<tr>
<td>5.</td>
<td>Synthesis</td>
<td>83</td>
</tr>
<tr>
<td>5.1</td>
<td>Synthesis for Object-Oriented Specifications</td>
<td>83</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Definition of the Synthesis Problem</td>
<td>83</td>
</tr>
<tr>
<td>5.1.2</td>
<td>UML-to-Implementation Mappings</td>
<td>83</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Synthesis Flow</td>
<td>84</td>
</tr>
<tr>
<td>5.2</td>
<td>Hardware/Software Interface</td>
<td>84</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Hardware Object and Component Life Cycle</td>
<td>84</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Logical Hardware Object Interface</td>
<td>85</td>
</tr>
</tbody>
</table>
Appendix

A. MOCCA Modeling Framework .......................... 135
   A.1 MOCCA Action Language .......................... 135
      A.1.1 Restrictions and Extensions to Java ........... 135
      A.1.2 Mapping of MAL to UML Actions and Activities . 136
   A.2 Core Data Types and Operations .................. 141
      A.2.1 Core Data Types .............................. 141
      A.2.2 Core Operations ............................... 143
   A.3 MOCCA Profile Definitions ......................... 149
      A.3.1 Overview ...................................... 149
      A.3.2 Related Profiles ............................... 150
      A.3.3 Notation ...................................... 152
   A.4 Constraint and Tag Value Definition Profile ......... 152
      A.4.1 Syntactic Meta-Language ....................... 152
      A.4.2 Syntax Definitions ............................. 153
   A.5 Design Profiles .................................... 158
      A.5.1 Design-Platform Profile ......................... 158
      A.5.2 Design-Model Profile ........................... 162
      A.5.3 Estimation Profile .............................. 163
   A.6 Target-Platform Profiles .......................... 166
      A.6.1 Implementation-Platform Profile ................. 166
      A.6.2 C/C++ Platform Profile ......................... 175
      A.6.3 VHDL Platform Profile .......................... 178
      A.6.4 Deployment-Platform Profile ..................... 182

B. Platform Models .................................... 187
   B.1 Design Platform .................................... 187
      B.1.1 Design Platform Types .......................... 187
      B.1.2 Design Platform Types Constraints ............... 187
   B.2 C/C++ Implementation-Platform ..................... 189
      B.2.1 Packages ...................................... 189
      B.2.2 Implementation Types ............................ 190
      B.2.3 Type Mappings .................................. 194
      B.2.4 Model Compiler Components ...................... 194
      B.2.5 UML-to-C++ Mapping ............................ 195
   B.3 VHDL Implementation-Platform ...................... 195
      B.3.1 Packages ...................................... 196
      B.3.2 Implementation Types ............................ 198
B.3.3 Implementation Components .......................................................... 202
B.3.4 Type Mappings ............................................................................. 207
B.3.5 Model Compiler Components .......................................................... 207
B.3.6 UML-to-VHDL Mapping ................................................................. 212
B.4 Deployment Platform Model ............................................................... 213

C. Model Transformations ................................................................. 215
C.1 Primitive Transformations ................................................................. 215
C.2 Technology Independent Transformations .......................................... 216
C.3 Technology Dependent Transformations ............................................ 217

D. Experimental Results ................................................................. 219
D.1 Run-Time Reconfiguration Characteristics ........................................ 219
D.2 Boolean Neural Network ................................................................. 220
   D.2.1 Description of BNN Tests ............................................................ 220
   D.2.2 Hardware Implementation of the BNNs ....................................... 225
   D.2.3 Software Implementation of the BNNs ....................................... 249
D.3 Online Compression of Audio Streams ............................................ 252
   D.3.1 Description of the Audio Server .................................................. 252
   D.3.2 Implementation of the Audio Server ......................................... 256

Bibliography ....................................................................................... 257
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Makimoto’s Wave and Hardware Design</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>General System-Level Design Flow</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>SA Design Space Exploration</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Example: VHDL design of a 2-bit D-latch</td>
<td>15</td>
</tr>
<tr>
<td>2.4</td>
<td>Example: Mapping of if and case statements to hardware</td>
<td>16</td>
</tr>
<tr>
<td>2.5</td>
<td>General Hardware Design Flow</td>
<td>17</td>
</tr>
<tr>
<td>2.6</td>
<td>UML Class Diagram Example</td>
<td>19</td>
</tr>
<tr>
<td>2.7</td>
<td>UML Behavior Modeling</td>
<td>20</td>
</tr>
<tr>
<td>2.8</td>
<td>UML Behavioral Diagrams Example</td>
<td>22</td>
</tr>
<tr>
<td>2.9</td>
<td>UML Sequence Diagram Example</td>
<td>22</td>
</tr>
<tr>
<td>3.1</td>
<td>Mapping of MAL Operators</td>
<td>29</td>
</tr>
<tr>
<td>3.2</td>
<td>Mapping of Loop in Example 3.1 to Actions and Activities</td>
<td>30</td>
</tr>
<tr>
<td>3.3</td>
<td>Object Diagram Compaction Rules</td>
<td>30</td>
</tr>
<tr>
<td>3.4</td>
<td>Compacted Mapping of Loop in Example 3.1 to Actions and Activities</td>
<td>31</td>
</tr>
<tr>
<td>3.5</td>
<td>Y-Chart Design</td>
<td>32</td>
</tr>
<tr>
<td>3.6</td>
<td>Model-Driven Architecture based on Platforms</td>
<td>33</td>
</tr>
<tr>
<td>3.7</td>
<td>Model-Driven Architecture and Platform-Based Design</td>
<td>34</td>
</tr>
<tr>
<td>3.8</td>
<td>Model-Driven Architecture Methodology Overview</td>
<td>35</td>
</tr>
<tr>
<td>3.9</td>
<td>Platform-Independent Model Meta-Model</td>
<td>37</td>
</tr>
<tr>
<td>3.10</td>
<td>Design Platform Model Example</td>
<td>38</td>
</tr>
<tr>
<td>3.11</td>
<td>Mapping of Actions in Listing 3.2 to Core Operations</td>
<td>39</td>
</tr>
<tr>
<td>3.12</td>
<td>Implementation Models Meta-Model</td>
<td>42</td>
</tr>
<tr>
<td>3.13</td>
<td>GRM Core Resource Meta-Model</td>
<td>43</td>
</tr>
<tr>
<td>3.14</td>
<td>Resources and Resource Service Meta-Model</td>
<td>44</td>
</tr>
<tr>
<td>3.15</td>
<td>Mappings of Implementation Types Meta-Model</td>
<td>44</td>
</tr>
<tr>
<td>3.16</td>
<td>Realization Graph Example</td>
<td>45</td>
</tr>
<tr>
<td>3.17</td>
<td>Implementation Components Meta-Model</td>
<td>45</td>
</tr>
<tr>
<td>3.18</td>
<td>Implementation Component Example</td>
<td>46</td>
</tr>
<tr>
<td>3.19</td>
<td>Implementation Platform Model Example: FIFO Component and Realizations</td>
<td>47</td>
</tr>
<tr>
<td>3.20</td>
<td>MOCCA Model Compiler Components Meta-Model</td>
<td>48</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.21</td>
<td>Implementation Platform Model Example: Compiler Components</td>
<td>48</td>
</tr>
<tr>
<td>3.22</td>
<td>VHDL Type Mapping Example</td>
<td>50</td>
</tr>
<tr>
<td>3.23</td>
<td>VHDL Component Resource Mapping Example</td>
<td>51</td>
</tr>
<tr>
<td>3.24</td>
<td>MAL Statement Mapping Example</td>
<td>52</td>
</tr>
<tr>
<td>3.25</td>
<td>Deployment Models Meta-Model</td>
<td>53</td>
</tr>
<tr>
<td>3.26</td>
<td>Relationship of Node to Implementation Platform Model Meta-Model</td>
<td>53</td>
</tr>
<tr>
<td>4.1</td>
<td>Hierarchical Mapping Neighborhoods and DSE</td>
<td>58</td>
</tr>
<tr>
<td>4.2</td>
<td>Data-flow Graph and Scheduling with multi-cycle Operation Example</td>
<td>60</td>
</tr>
<tr>
<td>4.3</td>
<td>Object Implementation Options</td>
<td>62</td>
</tr>
<tr>
<td>4.4</td>
<td>Platform Mapping Algorithm Design</td>
<td>64</td>
</tr>
<tr>
<td>4.5</td>
<td>Propagation of Mapping Activity in Classifier Hierarchies</td>
<td>71</td>
</tr>
<tr>
<td>4.6</td>
<td>Re-Mapping of Classifiers</td>
<td>71</td>
</tr>
<tr>
<td>4.7</td>
<td>Profiling Methodology</td>
<td>78</td>
</tr>
<tr>
<td>5.1</td>
<td>Synthesis Flow</td>
<td>84</td>
</tr>
<tr>
<td>5.2</td>
<td>Hardware Object and Component Life Cycles</td>
<td>85</td>
</tr>
<tr>
<td>5.3</td>
<td>Remote Object Example</td>
<td>87</td>
</tr>
<tr>
<td>5.4</td>
<td>Hardware Design Hierarchy</td>
<td>89</td>
</tr>
<tr>
<td>5.5</td>
<td>Hardware Design Example</td>
<td>89</td>
</tr>
<tr>
<td>5.6</td>
<td>Implementation Component Instantiation Example</td>
<td>91</td>
</tr>
<tr>
<td>5.7</td>
<td>FSMD Communication Interface and Register File</td>
<td>92</td>
</tr>
<tr>
<td>5.8</td>
<td>Timing of MOB Write Transfer and Read Transfer</td>
<td>92</td>
</tr>
<tr>
<td>5.9</td>
<td>FSMD-based Implementation Dynamic Message Dispatch</td>
<td>94</td>
</tr>
<tr>
<td>5.10</td>
<td>Address Mapping Example</td>
<td>96</td>
</tr>
<tr>
<td>5.11</td>
<td>Principal Behavior Execution</td>
<td>98</td>
</tr>
<tr>
<td>5.12</td>
<td>Behavior Implementation using a FSMD</td>
<td>99</td>
</tr>
<tr>
<td>5.13</td>
<td>FSM for Actions</td>
<td>100</td>
</tr>
<tr>
<td>5.14</td>
<td>FSM for ConditionalNode (if)</td>
<td>100</td>
</tr>
<tr>
<td>5.15</td>
<td>FSM for ConditionalNode (switch)</td>
<td>101</td>
</tr>
<tr>
<td>5.16</td>
<td>FSM for LoopNode</td>
<td>101</td>
</tr>
<tr>
<td>5.17</td>
<td>FSM for Design Example in Listing 5.2</td>
<td>102</td>
</tr>
<tr>
<td>5.18</td>
<td>FSMD for Design Example 5.7</td>
<td>103</td>
</tr>
<tr>
<td>5.19</td>
<td>Data-Path for Design Example 5.9</td>
<td>104</td>
</tr>
<tr>
<td>5.20</td>
<td>Three-State Conversion</td>
<td>105</td>
</tr>
<tr>
<td>6.1</td>
<td>Hardware Abstraction Layer for Object-Oriented Systems</td>
<td>107</td>
</tr>
<tr>
<td>6.2</td>
<td>RTR-Manager Initialization</td>
<td>108</td>
</tr>
<tr>
<td>6.3</td>
<td>Hardware Object Utilization</td>
<td>109</td>
</tr>
</tbody>
</table>
6.4 Hardware Object Communication ...................................................... 111
7.1 MOCCA Development Environment .................................................. 113
7.2 MOCCA Compilation Flow ............................................................... 115
7.3 MOCCA Compiler Architecture ......................................................... 116
7.4 Design and Structure of BNN Example ............................................... 118
7.5 Latencies of the BNN FPGA Implementations ....................................... 119
7.6 FPGA Area Characteristics of Component of BNNs (L9) .......................... 120
7.7 FPGA Area Characteristics of calculate(...) of BNNs (L9) ...................... 120
7.8 FPGA Area Characteristics of calculate(...) of BNN8 ............................ 121
7.9 Average MOCCA Compilation Times for FPGA Implementation of BNNs .... 121
7.10 Latencies of the BNN Software Implementations (L9) ......................... 122
7.11 Execution Latencies of calculate(...) (L9) ........................................ 122
7.12 Average MOCCA Compilation Times for Software Implementation of BNNs . 123
7.13 Functionality of the Audio Server and Audio Clients ............................ 124
7.14 Design Model of the Audio Server .................................................... 125
7.15 Latencies of the Audio Server ........................................................ 126
A.1 Sequencing between Statements ....................................................... 136
A.2 Mapping of try-catch-Statement ...................................................... 137
A.3 Mapping of if-Statement ................................................................. 137
A.4 Mapping of switch-Statement ........................................................... 138
A.5 Mapping of for-Statement ............................................................... 139
A.6 Mapping of while-Statement ............................................................ 139
A.7 Mapping of do-Statement ............................................................... 140
A.8 MOCCA Profiles .............................................................................. 150
A.9 Reference to Extended Meta-Model Element Notation ......................... 152
A.10 Design Platform Profile ................................................................. 158
A.11 Design Model Profile ..................................................................... 162
A.12 Estimation Profile .......................................................................... 163
A.13 Implementation Platform Profile: Implementation Components ............ 166
A.14 Implementation Platform Profile: Features and Parameters ................. 166
A.15 Implementation Platform Profile: Model Compiler Components ............ 167
A.16 Implementation Platform Profile: Miscellaneous Stereotypes ............... 167
A.17 C/C++ Implementation Platform Profile ............................................ 175
A.18 VHDL Implementation Platform Profile: Implementation Types ............ 179
A.19 VHDL Implementation Platform Profile: Miscellaneous Constructs .......... 179
A.20 Deployment Platform Profile .......................................................... 182
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1</td>
<td>Base Types</td>
<td>187</td>
</tr>
<tr>
<td>B.2</td>
<td>Number Types</td>
<td>189</td>
</tr>
<tr>
<td>B.3</td>
<td>System Types</td>
<td>189</td>
</tr>
<tr>
<td>B.4</td>
<td>Packages</td>
<td>190</td>
</tr>
<tr>
<td>B.5</td>
<td>Package Dependencies</td>
<td>191</td>
</tr>
<tr>
<td>B.6</td>
<td>Primitive Data Types</td>
<td>191</td>
</tr>
<tr>
<td>B.7</td>
<td>Complex Data Types</td>
<td>192</td>
</tr>
<tr>
<td>B.8</td>
<td>Miscellaneous Data Types</td>
<td>192</td>
</tr>
<tr>
<td>B.9</td>
<td>Base Primitive Type Mappings</td>
<td>194</td>
</tr>
<tr>
<td>B.10</td>
<td>Primitive Type Mappings</td>
<td>195</td>
</tr>
<tr>
<td>B.11</td>
<td>Complex Type Mappings</td>
<td>195</td>
</tr>
<tr>
<td>B.12</td>
<td>MOCCA Compiler Components</td>
<td>196</td>
</tr>
<tr>
<td>B.13</td>
<td>Packages</td>
<td>196</td>
</tr>
<tr>
<td>B.14</td>
<td>Package Dependencies</td>
<td>198</td>
</tr>
<tr>
<td>B.15</td>
<td>Primitive Data Types</td>
<td>199</td>
</tr>
<tr>
<td>B.16</td>
<td>Clocking and Communication Implementation Types</td>
<td>199</td>
</tr>
<tr>
<td>B.17</td>
<td>Register Implementation Types</td>
<td>200</td>
</tr>
<tr>
<td>B.18</td>
<td>Memory Blocks Implementation Types</td>
<td>201</td>
</tr>
<tr>
<td>B.19</td>
<td>Clocking and Reset Interfaces</td>
<td>203</td>
</tr>
<tr>
<td>B.20</td>
<td>Communication Interface</td>
<td>203</td>
</tr>
<tr>
<td>B.21</td>
<td>Register Interfaces</td>
<td>203</td>
</tr>
<tr>
<td>B.22</td>
<td>Memory Block Interfaces</td>
<td>206</td>
</tr>
<tr>
<td>B.23</td>
<td>Clocking and Communication Implementation Components</td>
<td>207</td>
</tr>
<tr>
<td>B.24</td>
<td>Register Storage Components</td>
<td>208</td>
</tr>
<tr>
<td>B.25</td>
<td>Memory Block Components</td>
<td>209</td>
</tr>
<tr>
<td>B.26</td>
<td>Clocking of Implementation Components</td>
<td>209</td>
</tr>
<tr>
<td>B.27</td>
<td>Data Exchange between Implementation Components</td>
<td>210</td>
</tr>
<tr>
<td>B.28</td>
<td>Primitive Type Mappings</td>
<td>210</td>
</tr>
<tr>
<td>B.29</td>
<td>std_logic Type Mappings</td>
<td>210</td>
</tr>
<tr>
<td>B.30</td>
<td>MOCCA Compiler Components</td>
<td>211</td>
</tr>
<tr>
<td>B.31</td>
<td>Deployment Platform Overview</td>
<td>213</td>
</tr>
<tr>
<td>D.1</td>
<td>Design Model of the Audio Server</td>
<td>255</td>
</tr>
</tbody>
</table>
LIST OF TABLES

3.1 Model Elements Relevant to Resource Mapping .................................. 50
4.1 Granularity of Model Elements ..................................................... 56
4.2 QoS-Estimation of Structural Elements of FSMD-Implementations ............ 79
4.3 QoS-Estimation of Behavioral Elements of FSMD-Implementations ............ 80
5.1 Synchronization Sets .................................................................. 104
A.1 MOCCA Base Types .................................................................... 142
A.2 MOCCA Integral Types .................................................................. 142
A.3 MOCCA Floating Point Types ..................................................... 142
A.4 MOCCA Character Types ............................................................ 143
A.5 MOCCA Auxiliary Types ............................................................. 143
A.6 Core Operations of Base Types .................................................... 144
A.7 Core Operations of Boolean Types ................................................ 144
A.8 Core Operations of Integral Types ................................................ 145
A.9 Core Operations of Floating Point Types ........................................ 146
A.10 Core Operations of Time Types .................................................. 147
A.11 Core Operations of Character Types ............................................ 147
A.12 Core Operations of Auxiliary Types ............................................ 148
B.1 Design Platform Integral Types Constraints .................................... 188
B.2 Design Platform Floating Point Types Constraints ............................... 188
B.3 Design Platform Time Type Constraints ......................................... 188
B.4 Design Platform Character Type Constraints .................................... 188
B.5 IHwObject Interface Description ................................................... 192
B.6 UML-to-C++ Mappings .............................................................. 197
B.12 Memory Block Interface Description ............................................ 201
B.7 Clocking Interfaces Description .................................................... 203
B.8 Reset Interfaces Description ....................................................... 204
B.9 Communication Interface Description .......................................... 204
B.10 Data Register Interfaces Description .......................................... 205
B.11 Control Register Interfaces Description ....................................... 206
B.13 UML-to-VHDL Mappings ...................................................... 212
B.14 Deployment Platform Nodes Constraints ................................. 214
C.1 MOCCA Primitive Transformations ......................................... 215
C.2 MOCCA Technology Independent Optimizations .......................... 216
C.3 MOCCA Technology Independent Optimizations .......................... 217
D.1 FPGA Reconfiguration Latency on the PC-based Platform ............... 219
D.2 Creation and Destruction of Hardware Objects on the PC-based Platform 219
D.3 Remote Communication Overhead on the PC-based Platform ............ 220
D.4 FPGA Communication Latencies of the BNNs (L9) ......................... 220
D.5 FPGA Execution Latencies of the BNNs (L9) ............................... 226
D.6 FPGA Execution Latencies of Bnn::calculate(...) (L9) .................... 227
D.7 FPGA Implementation Characteristics Component BNN0 .................. 228
D.8 FPGA Implementation Characteristics Class Bnn BNN0 .................... 228
D.9 FPGA Implementation Characteristics Bnn::calculate(...) BNN0 .......... 228
D.10 FPGA Implementation Characteristics Component BNN1 .................. 229
D.11 FPGA Implementation Characteristics Class Bnn BNN1 .................... 229
D.12 FPGA Implementation Characteristics Bnn::calculate(...) BNN1 .......... 229
D.13 FPGA Implementation Characteristics Component BNN2 .................. 230
D.14 FPGA Implementation Characteristics Class Bnn BNN2 .................... 230
D.15 FPGA Implementation Characteristics Bnn::calculate(...) BNN2 .......... 230
D.16 FPGA Implementation Characteristics Component BNN3 .................. 231
D.17 FPGA Implementation Characteristics Class Bnn BNN3 .................... 231
D.18 FPGA Implementation Characteristics Bnn::calculate(...) BNN3 .......... 231
D.19 FPGA Implementation Characteristics Component BNN4 .................. 232
D.20 FPGA Implementation Characteristics Class Bnn BNN4 .................... 232
D.21 FPGA Implementation Characteristics Bnn::calculate(...) BNN4 .......... 232
D.22 FPGA Implementation Characteristics Component BNN5 .................. 233
D.23 FPGA Implementation Characteristics Class Bnn BNN5 .................... 233
D.24 FPGA Implementation Characteristics Bnn::calculate(...) BNN5 .......... 233
D.25 FPGA Implementation Characteristics Component BNN6 .................. 234
D.26 FPGA Implementation Characteristics Class Bnn BNN6 .................... 234
D.27 FPGA Implementation Characteristics Bnn::calculate(...) BNN6 .......... 234
D.28 FPGA Implementation Characteristics Component BNN7 .................. 235
D.29 FPGA Implementation Characteristics Class Bnn BNN7 .................... 235
D.30 FPGA Implementation Characteristics Bnn::calculate(...) BNN7 .......... 235
D.31 FPGA Implementation Characteristics Component BNN8 .................. 236
D.32 FPGA Implementation Characteristics Class Bnn BNN8 .................... 236
D.33 FPGA Implementation Characteristics \texttt{Bnn::calculate(...)} BNN8 236
D.34 FPGA Implementation Characteristics Component BNN9 237
D.35 FPGA Implementation Characteristics Class \texttt{Bnn} BNN9 237
D.36 FPGA Implementation Characteristics \texttt{Bnn::calculate(...)} BNN9 237
D.37 FPGA Implementation Characteristics Component BNN10 238
D.38 FPGA Implementation Characteristics Class \texttt{Bnn} BNN10 238
D.39 FPGA Implementation Characteristics \texttt{Bnn::calculate(...)} BNN10 238
D.40 FPGA Implementation Characteristics Component BNN11 239
D.41 FPGA Implementation Characteristics Class \texttt{Bnn} BNN11 239
D.42 FPGA Implementation Characteristics \texttt{Bnn::calculate(...)} BNN11 239
D.43 FPGA Implementation Characteristics Component BNN12 240
D.44 FPGA Implementation Characteristics Class \texttt{Bnn} BNN12 240
D.45 FPGA Implementation Characteristics \texttt{Bnn::calculate(...)} BNN12 240
D.46 FPGA Implementation Characteristics Component BNN13 241
D.47 FPGA Implementation Characteristics Class \texttt{Bnn} BNN13 241
D.48 FPGA Implementation Characteristics \texttt{Bnn::calculate(...)} BNN13 241
D.49 FPGA Implementation Characteristics Component BNN14 242
D.50 FPGA Implementation Characteristics Class \texttt{Bnn} BNN14 242
D.51 FPGA Implementation Characteristics \texttt{Bnn::calculate(...)} BNN14 242
D.52 FPGA Implementation Area Estimation Component BNN0 243
D.53 FPGA Implementation Area Estimation Component BNN1 243
D.54 FPGA Implementation Area Estimation Component BNN2 243
D.55 FPGA Implementation Area Estimation Component BNN3 244
D.56 FPGA Implementation Area Estimation Component BNN4 244
D.57 FPGA Implementation Area Estimation Component BNN5 244
D.58 FPGA Implementation Area Estimation Component BNN6 245
D.59 FPGA Implementation Area Estimation Component BNN7 245
D.60 FPGA Implementation Area Estimation Component BNN8 245
D.61 FPGA Implementation Area Estimation Component BNN9 246
D.62 FPGA Implementation Area Estimation Component BNN10 246
D.63 FPGA Implementation Area Estimation Component BNN11 246
D.64 FPGA Implementation Area Estimation Component BNN12 247
D.65 FPGA Implementation Area Estimation Component BNN13 247
D.66 FPGA Implementation Area Estimation Component BNN14 247
D.67 Average Compilation Times of the FPGA Implementation of BNN Designs 248
D.68 Software Communication Latencies of the BNNs (L9) 249
D.69 Software Execution Latencies of the BNNs (L9) 250
D.70 Average Compilation Times of the Software Implementation of BNN Designs 251
D.71 Communication Timing of the Audio Server (L9) ........................ 256
D.72 Execution Timing of the Audio Server (L9) ................................. 256
D.73 FPGA Implementation Characteristics of the Audio Server Component (L9) ........ 256
D.74 FPGA Implementation Area Estimation of Audio Server Component (L9) ........... 256
D.75 Compilation Times of the FPGA Implementation of the Audio Server (L9) ............ 256
## LIST OF ALGORITHMS

<table>
<thead>
<tr>
<th>Number</th>
<th>Algorithm Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Find the most specific operation - <code>findMSO(classifier, name, ArgList)</code></td>
<td>40</td>
</tr>
<tr>
<td>4.1</td>
<td>SA controller - <code>control(d_{n+1}^{0}, temp^{0}, temp_{min})</code></td>
<td>65</td>
</tr>
<tr>
<td>4.2</td>
<td>Breeder initialization - <code>initialize(d_{n+1}^{0})</code></td>
<td>69</td>
</tr>
<tr>
<td>4.3</td>
<td>Set the activity of mappings - <code>setActivity(m_{e,di,activity})</code></td>
<td>70</td>
</tr>
<tr>
<td>4.4</td>
<td>Breeding of mappings - <code>breed(d_{n+1}^{step-1})</code></td>
<td>70</td>
</tr>
<tr>
<td>4.5</td>
<td>Re-mapping of an element - <code>remap(m_{e,di,goal,lthres,rthres})</code></td>
<td>70</td>
</tr>
<tr>
<td>4.6</td>
<td>Select a mapping - <code>selectMapping(M_{i,goal})</code></td>
<td>72</td>
</tr>
<tr>
<td>4.7</td>
<td>Compute candidate mappings - <code>computeCandidateMappings(e, recurse)</code></td>
<td>74</td>
</tr>
<tr>
<td>4.8</td>
<td>Find implementation type - <code>findImplementationType(pf, cs, rs, if)</code></td>
<td>75</td>
</tr>
<tr>
<td>4.9</td>
<td>Merge-function for resource service instance sharing - <code>ϕ_{share}(label, QV)</code></td>
<td>81</td>
</tr>
<tr>
<td>5.1</td>
<td>Address Mapping - <code>computeAddressMap(C, DL)</code></td>
<td>97</td>
</tr>
<tr>
<td>6.1</td>
<td>Hardware Object Creation - <code>createObject(cl)</code></td>
<td>110</td>
</tr>
<tr>
<td>6.2</td>
<td>Hardware Object Destruction - <code>destroyObject(o)</code></td>
<td>110</td>
</tr>
</tbody>
</table>
### LIST OF LISTINGS

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>MAL Additional Operators and Statements Example</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>MAL Statement Example</td>
<td>39</td>
</tr>
<tr>
<td>3.3</td>
<td>MOCCA Optimization-Constraint Specification Example</td>
<td>41</td>
</tr>
<tr>
<td>3.4</td>
<td>MOCCA Execution-Constraint Specification Example</td>
<td>42</td>
</tr>
<tr>
<td>3.5</td>
<td>MOCCA QoS-Constraint Specification Example</td>
<td>48</td>
</tr>
<tr>
<td>3.6</td>
<td>MOCCA Synthesis-Constraint Specification Example</td>
<td>49</td>
</tr>
<tr>
<td>3.7</td>
<td>MAL Statement Mapping Example</td>
<td>51</td>
</tr>
<tr>
<td>5.1</td>
<td>Instantiation, Communication, and Destruction of <code>RemoteClass</code> Example</td>
<td>87</td>
</tr>
<tr>
<td>5.2</td>
<td>FSM Design Example</td>
<td>102</td>
</tr>
<tr>
<td>5.3</td>
<td>Operation Sharing Design Example</td>
<td>103</td>
</tr>
<tr>
<td>5.4</td>
<td>Hardware Object Model Example</td>
<td>106</td>
</tr>
<tr>
<td>D.1</td>
<td>Design of <code>calculate(...)</code> of BNN0</td>
<td>220</td>
</tr>
<tr>
<td>D.2</td>
<td>Design of <code>init_x(...)</code> of BNN0</td>
<td>220</td>
</tr>
<tr>
<td>D.3</td>
<td>Design of <code>get_y(...)</code> of BNN0</td>
<td>220</td>
</tr>
<tr>
<td>D.4</td>
<td>Design of <code>init_x(...)</code> of BNN1</td>
<td>221</td>
</tr>
<tr>
<td>D.5</td>
<td>Design of <code>get_y(...)</code> of BNN1</td>
<td>221</td>
</tr>
<tr>
<td>D.6</td>
<td>Design of <code>get_y(...)</code> of BNN3</td>
<td>221</td>
</tr>
<tr>
<td>D.7</td>
<td>Design of <code>calculate(...)</code> of BNN4</td>
<td>222</td>
</tr>
<tr>
<td>D.8</td>
<td>Design of <code>calculate(...)</code> of BNN5</td>
<td>222</td>
</tr>
<tr>
<td>D.9</td>
<td>Design of <code>calculate(...)</code> of BNN7</td>
<td>222</td>
</tr>
<tr>
<td>D.10</td>
<td>Design of <code>calculate(...)</code> of BNN8</td>
<td>223</td>
</tr>
<tr>
<td>D.11</td>
<td>Design of <code>calculate(...)</code> of BNN13</td>
<td>223</td>
</tr>
<tr>
<td>D.12</td>
<td>Design of <code>calculate(...)</code> of BNN14</td>
<td>224</td>
</tr>
<tr>
<td>D.13</td>
<td>Behavior of <code>encode()</code></td>
<td>252</td>
</tr>
<tr>
<td>D.14</td>
<td>Core Loop of <code>main(...)</code></td>
<td>254</td>
</tr>
</tbody>
</table>
# LIST OF ACRONYMS

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALAP</td>
<td>as-late-as-possible</td>
</tr>
<tr>
<td>ALSU</td>
<td>arithmetic-logic-shift unit</td>
</tr>
<tr>
<td>ALU</td>
<td>arithmetic-logic unit</td>
</tr>
<tr>
<td>ASAP</td>
<td>as-soon-as-possible</td>
</tr>
<tr>
<td>ASIC</td>
<td>application specific integrated circuit</td>
</tr>
<tr>
<td>ATP</td>
<td>area-time-power</td>
</tr>
<tr>
<td>BNN</td>
<td>Boolean neural network</td>
</tr>
<tr>
<td>BN</td>
<td>Boolean neuron</td>
</tr>
<tr>
<td>CLB</td>
<td>configurable logic block</td>
</tr>
<tr>
<td>CPU</td>
<td>central processing unit</td>
</tr>
<tr>
<td>DAG</td>
<td>directed acyclic graph</td>
</tr>
<tr>
<td>DCM</td>
<td>digital clock manager</td>
</tr>
<tr>
<td>DFG</td>
<td>data-flow graph</td>
</tr>
<tr>
<td>DMA</td>
<td>direct memory access</td>
</tr>
<tr>
<td>DPM</td>
<td>design-platform model</td>
</tr>
<tr>
<td>DSE</td>
<td>design space exploration</td>
</tr>
<tr>
<td>FIFO</td>
<td>first in first out</td>
</tr>
<tr>
<td>FLAC</td>
<td>free lossless audio codec</td>
</tr>
<tr>
<td>FPGA</td>
<td>field-programmable gate array</td>
</tr>
<tr>
<td>FSM</td>
<td>finite state machine</td>
</tr>
<tr>
<td>FSMD</td>
<td>finite state machine with data-path</td>
</tr>
<tr>
<td>GNU</td>
<td>GNU is Not Unix</td>
</tr>
<tr>
<td>GRM</td>
<td>general resource modeling framework</td>
</tr>
<tr>
<td>GUI</td>
<td>graphical user interface</td>
</tr>
<tr>
<td>HAL</td>
<td>hardware abstraction layer</td>
</tr>
<tr>
<td>HDL</td>
<td>hardware description language</td>
</tr>
<tr>
<td>HLS</td>
<td>high-level synthesis</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>ILP</td>
<td>integer linear programming</td>
</tr>
<tr>
<td>I/O</td>
<td>input/output</td>
</tr>
<tr>
<td>IP</td>
<td>intellectual property</td>
</tr>
<tr>
<td>ISA</td>
<td>instruction set architecture</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>LAN</td>
<td>local area network</td>
</tr>
<tr>
<td>LP</td>
<td>linear programming</td>
</tr>
<tr>
<td>LUT</td>
<td>lookup table</td>
</tr>
<tr>
<td>MAL</td>
<td>Model Compiler for reConfigurable Architectures (MOCCA) Action Language</td>
</tr>
<tr>
<td>MDA</td>
<td>model-driven architecture</td>
</tr>
<tr>
<td>MOB</td>
<td>MOCCA Object-Bus</td>
</tr>
<tr>
<td>MOCCA</td>
<td>Model Compiler for reConfigurable Architectures</td>
</tr>
<tr>
<td>uP</td>
<td>microprocessor</td>
</tr>
<tr>
<td>NP</td>
<td>non-polynomial</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>OMG</td>
<td>Object-Management Group</td>
</tr>
<tr>
<td>OO</td>
<td>object-orientation</td>
</tr>
<tr>
<td>OSLF</td>
<td>operating system abstraction layer framework</td>
</tr>
<tr>
<td>PC</td>
<td>program counter</td>
</tr>
<tr>
<td>PCB</td>
<td>printed circuit board</td>
</tr>
<tr>
<td>PBD</td>
<td>platform-based design</td>
</tr>
<tr>
<td>PE</td>
<td>processing element</td>
</tr>
<tr>
<td>PCI</td>
<td>peripheral component interconnect</td>
</tr>
<tr>
<td>PIM</td>
<td>platform-independent model</td>
</tr>
<tr>
<td>PSI</td>
<td>platform-specific implementation</td>
</tr>
<tr>
<td>PSM</td>
<td>platform-specific model</td>
</tr>
<tr>
<td>QoS</td>
<td>quality-of-service</td>
</tr>
<tr>
<td>RAM</td>
<td>random-access memory</td>
</tr>
<tr>
<td>RCS</td>
<td>resource-constrained scheduling</td>
</tr>
<tr>
<td>RF</td>
<td>reconfigurable fabric</td>
</tr>
<tr>
<td>RFU</td>
<td>reconfigurable function unit</td>
</tr>
<tr>
<td>RISC</td>
<td>reduced instruction set computer</td>
</tr>
<tr>
<td>RTL</td>
<td>register-transfer level</td>
</tr>
<tr>
<td>RTR</td>
<td>run-time reconfiguration</td>
</tr>
<tr>
<td>Acronym</td>
<td>Definition</td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td>SA</td>
<td>simulated annealing</td>
</tr>
<tr>
<td>SNOW</td>
<td>services for nomadic workers</td>
</tr>
<tr>
<td>SoC</td>
<td>system-on-chip</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>transmission control protocol/internet protocol</td>
</tr>
<tr>
<td>TCS</td>
<td>time-constrained scheduling</td>
</tr>
<tr>
<td>TRCS</td>
<td>time-resource constrained scheduling</td>
</tr>
<tr>
<td>TPM</td>
<td>target-platform model</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>VLIW</td>
<td>very long instruction word</td>
</tr>
<tr>
<td>VMT</td>
<td>virtual method table</td>
</tr>
<tr>
<td>VHDL</td>
<td>very high speed integrated circuit (VHSIC) hardware description language (HDL)</td>
</tr>
<tr>
<td>VHSIC</td>
<td>very high speed integrated circuit</td>
</tr>
<tr>
<td>XMI</td>
<td>Extensible Markup Language (XML) Metadata Interchange</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>XST</td>
<td>Xilinx Synthesis Tools</td>
</tr>
<tr>
<td>ZBT</td>
<td>zero bus turnaround time</td>
</tr>
</tbody>
</table>
LIST OF SYMBOLS

Vectors
v vector of elements \( v_0 \ldots v_n \)

Sets
\( \mathbb{N}^* \) set of positive integers \( \mathbb{N}^* = \{1, 2, \ldots\} \)
\( \mathbb{R} \) set of real numbers

Set Operators
\( \{a\} \) set of elements
\( \{a\mid P\} \) set of elements with property \( P \)
\( |A| \) cardinality of set \( A \), the number of elements of \( A \)
\( A \subset B, (A \subseteq B) \) \( A \) is a real subset of \( B \), (or equal to \( B \))
\( A \mapsto B \) mapping from set \( A \) to set \( B \)
\( A \times B \) Cartesian product of sets \( A \) and \( B \)
\( P(X) = \{ A \mid A \subseteq X \} \) set \( A \) defines a partial order of its elements
\( \forall \) universal quantifier
\( \exists^1, (\exists) \) exactly one (no) element with a property exists [1]
\( \exists^{\leq n}, (\exists^{\geq n}) \) at most (least) \( n \) elements with a property exists [1]

Intervals
\( [a, b] \) interval including all values between \( a \) and \( b \), including \( a \) and \( b \).
\( (a, b] \) interval including all values between \( a \) and \( b \), including \( b \) but not including \( a \).
\( [a, b) \) interval including all values between \( a \) and \( b \), including \( a \) but not including \( b \).
\( (a, b) \) interval including all values between \( a \) and \( b \), not including \( a \) or \( b \).

Boolean Operators
\( \pi \) not \( a \)
\( a \lor b \) a or \( b \)
\( a \land b \) a and \( b \)
\( a \rightarrow b \) a implies \( b \)

Relational Operators
\( a = b \) a equals \( b \)
\( a \neq b \) a is not equal to \( b \)
\( a \preceq b \) a is a predecessor of \( b \) or equal to \( b \) with respect to some order relationship
\( a \succeq b \) a is a successor of \( b \) or equal to \( b \) with respect to some order relationship

Integer Operators
\( \lceil r \rceil \) smallest integer \( i \) with \( i > r \)
\( \sum \) sum of integer values
**Design Space Specific Sets**

- $F_s$: functionality of system $s$
- $P_s$: properties of system $s$
- $Q_s$: quality characteristics of system $s$
- $C_s$: constraints of system $s$
- $DS_n$: design space at abstraction level $n$
- $R_n$: set of resource services at abstraction level $n$
- $O_{r,n}$: set of implementation options implementing $r \in R_n$
- $Q_n$: set of quality characteristics at abstraction level $n$

**Design Space Specific Functions**

- $qos : DS_n \mapsto Q_n$: quality of service of design space $DS_n$ with respect to the quality characteristics $Q_n$
- $cost : DS_n \times Q_s \mapsto \mathbb{R}$: cost of design space $DS_n$ with respect to the quality characteristics $Q_s$

**Model Specific Sets**

- $E$: set of model elements
- $EI$: set of model element instances
- $CL$: set of classifiers
- $CO$: set of components
- $DL$: set of deployment locations
- $M$: set of mappings
- $M_e$: set of mappings of a model element $e$
- $M_{e,dl}$: set of mappings of a model element $e$ on a deployment location $dl$
- $A$: set of allocated resource services
- $AI$: set of allocated resource service instances

**Model Specific Functions**

- $distance : CL \times CL \mapsto \mathbb{N}$: distance of two classifiers in the inheritance hierarchy
- $dchildren : E \mapsto P(E)$: set of deployment children of a model element
- $dparents : E \mapsto P(E)$: set of deployment parents of a model element
- $dchildren : EI \mapsto P(EI)$: set of deployment children of a model element instance
- $dparent : EI \mapsto EI$: deployment parent of a model element instance
- $binding : E \mapsto P(A)$: binding of a model element to a set of resource services
- $binding : EI \mapsto P(AI)$: binding of a model element instance to a set of resource service instances
- $schedule : EI \mapsto P(\mathbb{N}^*)$: schedule of a model element instance
- $mparents : M \mapsto P(M)$: set of parent mappings of a mapping
- $generalizations : CL \mapsto P(CL)$: set of generalizations of a classifier
- $specializations : CL \mapsto P(CL)$: set of specializations of a classifier
- $isActive : M \mapsto \{true, false\}$: evaluates $true$ if a mapping is active, otherwise the function evaluates $false$
- $utilization : E \mapsto \mathbb{R}$: utilization of a model element

**Names of Functional Units**

- ADD: functional unit performs an addition
- SUB: functional unit performs a subtraction
- MUL: functional unit performs a multiplication
- LT: functional unit performs a less than test
- EQ: functional unit performs an equality test
- GT: functional unit performs a greater than test
- NEG: functional unit performs a bitwise negation
- AND: functional unit performs a bitwise AND
- OR: functional unit performs a bitwise OR
Other Symbols

\(\langle x_1, \ldots, x_n \rangle\) \hspace{1cm} n-tuple of elements
\(\emptyset\) \hspace{1cm} empty variable
\(\sigma\) \hspace{1cm} standard deviation
\(\emptyset\) \hspace{1cm} empty set
\(\leftarrow\) \hspace{1cm} assignment in algorithms
\(:=\) \hspace{1cm} assignment in constraint specifications
\(\rightarrow\) \hspace{1cm} reference to an element in the thesis
1. INTRODUCTION

1.1 Motivation

In principle there are two possibilities to implement algorithms on computers: software and hardware. A software solution is a sequence of instructions, i.e. a program, that define the operations being performed by a processor. The processor continuously fetches the instructions from an attached memory and performs their operation on a general-purpose processing unit. The computed task may be changed by altering the instructions in the memory. The operations performed by the instructions are bound to the processor hardware on the basis of execution cycles. Thus, software implementations are commonly very flexible. However, software solutions are inefficient, if the processor instruction set architecture (ISA) poorly matches the requirements of the executed algorithm. The timely successive execution of instructions makes software solutions relatively slow.

Hardware implementations execute algorithms spatially. The implementations are tailored to the specific algorithm and keep the implementation overhead, that is spent to provide more general solutions, to a minimum. As a result, hardware implementations of algorithms are commonly very fast and efficient. Traditionally, the operation performed by custom hardware designs is manifested in the layers of integrated circuits and, in multi-chip designs, the interconnects between the individual circuits. As a consequence, the operation is bound to the hardware during manufacturing; thereafter, the circuits and their interconnections can not be altered. In addition, such solutions expose high development and implementation cost, which often do not amortize with production quantities in common applications.

In contrast, reconfigurable hardware is customized after the fabrication of the circuit. The performance and efficiency of custom hardware are maintained at a higher grade of flexibility. The application of a reusable fabric reduces the implementation cost in comparison to solutions with mask-programmed application specific integrated circuits (ASICs) or standard logic circuits. Hence, reconfigurable logic adds a new degree of freedom to the overall architectural space of computer systems, which enables the construction of novel computer architectures.

In 1960 Gerald Estrin proposed in his paper "Organization of Computer Systems - The fixed plus variable structure Computer" a novel kind of computer architecture [2]. This computer architecture couples a standard microprocessor with an array of reconfigurable logic resources. Thus, this kind of computer architecture is called reconfigurable computer architecture. Although this concept has been alive for many years, only the advent of field-programmable gate arrays (FPGAs) at the end of the previous decade enabled its adoption to general purpose computing. FPGAs build the hardware basis for the implementation of reconfigurable logic. Such architectures facilitate implementations that match the requirements of the application and use an optimal mix of resources. Criteria for optimality are performance, implementation effort, power dissipation, and many others. The identification of an optimal mapping of applications to the underlying computer hardware is still a major problem, not only in reconfigurable computing.

The existing architectures vary in the actual computer architecture as well as the methods, languages, and tools being employed for application development. While for coarse-grained and data path-coupled architectures hardware and software have evolved in conjunction, this is not the case for fine-grained, network-coupled architectures. These architectures are frequently utilized in system-on-chip design, embedded systems, and general-purpose computing, in applications such as cryptography, robotics, media-processing, and scientific computing. A wide range of hardware add-ons for standard computers are commercially available at affordable cost.

The usage of fine-grained, network-coupled architectures still requires a fair amount of manual hardware design, which naturally increases development time and cost, and limits the solution quality. The currently
starting customization process in this field goes toward heterogeneous architectures, comprising combinations of FPGAs, ASICs, and microprocessors. In future, novel architectures will continue to evolve as the need for customization is maintained by novel application areas and market demand for differentiation and value-addition. Customization and thereby fragmentation is enabled by respective design and manufacturing methods and tools. The reverse trend of standardization is mainly pushed by innovations in architecture and software, and the need to reduce development cost and time. This observation has been captured in the so called Makimoto wave for the semiconductor industry (→ Fig. 1.1) [3], but basically we can observe similar alternating cycles of customization and standardization in most areas of computer systems development.

The added heterogeneity and the complexity challenge demand novel development approaches. According to Fig. 1.1, the next step will push current behavioral-level design to the system-level. System-level design must be able to handle growing application complexity and exploit the capabilities of the hardware. Moreover, any meaningful methodology must provide a high degree of portability in order to handle the fragmentation caused by the forthcoming wave of customization. The raised level of abstraction must be accomplished by novel languages and methodologies which will require novel algorithms for design space exploration and synthesis. Currently, there is no complete system-level design flow for reconfigurable computer architectures available. Owing to the early state of respective hardware/software co-design there is also a lack of mature tool chains that enable the automated transition from system-level models to implementation. The progress in hardware technology can not be exploited sufficiently by the existing design approaches.

The complexity challenge is permanent in other areas of computer systems development as well. In software development this challenge is commonly tackled by taking of object-oriented approaches. Object-orientation supports the handling of complexity because it encourages data abstraction, design decomposition, and reuse. Inheritance, inclusion polymorphism, and encapsulation are the supporting design mechanisms. The object-based model of computation supports the problem-driven decomposition. Although there is agreement that this paradigm can be applied successfully to system-level design as well, object-orientation has not yet found its way into this domain.

The goal of this thesis is the systematic investigation and development of an object-oriented system-level development approach for fine-grained, network-coupled, reconfigurable architectures (→ Section 2.1.1). This approach is required to fulfill at least the following objectives:

**Efficiency** - The approach must exploit the features of the hardware platform for the acceleration of executed applications. In particular, it must exploit the special features of the reconfigurable hardware, such as run-time reconfigurability and spatial execution. Real-time constraints are not in the focus of discussion, because these systems commonly require specialized methodologies.

**Object-Orientation** - The success of object-orientation (OO) in the field of complex software development suggests to use this paradigm also for the development of hardware/software systems. Object-orientation shall not be a mere specification paradigm as in present approaches, but the first-class paradigm of the entire development process, including specification, design space exploration, and synthesis. The structure and functionality of object-orientation map well to network-coupled multiprocessor systems which can exploited to improve efficiency.

**Automation** - To avoid the costly and timely ineffective amount of manual hardware/software design the approach must allow for an automated translation of object-oriented system-level designs to directly
executable implementations, comprising of hardware and software modules. Automation supports the correctness of implementations ("correctness-by-construction"). Fast automation encourages the exploration of different design alternatives and therefore helps to improve overall quality.

**Portability** - As a direct consequence of the fragmentation in hardware architectures the approach must not be designed toward one particular architecture or design environment. Also, designs should not be enforced to be specified towards particular platforms, because target platforms continue to change significantly faster than application designs. This requirement generally interferes with efficiency.

**Reuse** - The degree of reuse is a key to the commercial success of design efforts. The development approach must support the definition, provision, and integration of reusable components. This requires ability to define and document architectural components and interfaces, as well as integrated validation and verification support.

Of course, these requirements are quite general and impose grave challenges, whereas the most important and pressing problems will be discussed in this thesis. Although some related work has already been done in this field, this thesis presents the first systematic approach that tackles these requirements in conjunction in the context of reconfigurable computing. The outcome of this thesis is a development approach and a set of novel algorithms and tools that might not only effect reconfigurable computing, but also object-oriented software development and design automation in general.

The set of applications which was initially targeted by this thesis are application-specific accelerators for logic optimization problems, neural networks, and multimedia applications. These applications expose mixed control- and dataflow and can be modeled advantageously using the OO-paradigm.

### 1.2 Related Work

The overall development approach, as presented in chapter 3, is unique in co-design in that it consequently builds upon the principles of model-driven architecture (MDA) and platform-based design (PBD) [4, 5]. Currently there is no complete, consistent UML-based system-level design flow, and tool chains for reconfigurable computer architectures available [6]. Recently a number of research projects that go into this direction have been started in the system-on-chip domain. Due to the popularity of SystemC in this field the majority of these approaches is based on this language [7].

In [8, 9] Schattkowsky et al. present an approach for the object-oriented design of hardware and software. Designs are specified using UML classes, interfaces, operations, state machines, and activities. From such Unified Modeling Language (UML) design models software or hardware implementations can be generated. The software generation is considered state-of-art. Hardware implementations targeting reconfigurable fabrics are described using Handel-C [10]. The support of inclusion polymorphism is restricted in that polymorphic operations are not allowed to access the same attributes. Object lifetimes and size are determined at compile time using an approach similar to SystemC [7] and Forge [11]. Dynamically created and destroyed hardware objects are not supported. The support of design space exploration (DSE), different target platforms, and co-synthesis are not discussed. Handel-C code is generated which is then synthesized into register-transfer level (RTL) descriptions for FPGAs by a commercial tool.

Basu et al. use UML as design entry point of their co-design flow [12]. Designs are specified using classes, state machines, activities, and interactions. The model of computation is discrete events. From such specifications SystemC HW/SW implementations are generated automatically. Inclusion polymorphism is not discussed. Objects are instantiatied statically at compile time. Manual design space exploration is provided by the environment on the level of SystemC implementations. The modeling of target platforms is restricted to deployment models. The environment is capable of generating synthesizable RTL for SystemC modules and code for NEC V850 and ARM946 microprocessors.

The work presented by Nguyen et al. [13, 14] is a representative of a variety of approaches that closely model particular implementations. In the presented approach transaction-level modeling and SystemC implementations are defined using UML classes and state machines. Neither inclusion polymorphism nor dynamic object instantiation are supported. The authors stress the importance of platform models at UML-level in
order to support DSE early in the design flow. The synthesis of the SystemC descriptions into hardware
description has been open work at the time of their proposal.

Schulz-Key et al. model the SystemC implementation of hardware/software systems with UML and auto-
matically generate the respective C++ and SystemC implementations from such models [15]. The system-
level design space exploration is performed manually and expressed through the inheritance hierarchy in
the model [16]. The synthesis of complete RTL designs is delegated to back-end tools. This results again in
a significant loss of control over the generated results.

The presented approaches have in common that they have been carried out in parallel to the work described
in this thesis. Also they may be embedded into the presented work. They currently lack the full support of
the key characteristics of object-oriented specifications. However, these works, among many others, offer
notable extensions to the approach presented in this thesis. Particularly, the support of UML state machines
and the respective synthesis algorithms are notable, because they are a natural means for behavior definition
in application areas like system-on-chip, embedded systems, and communication systems.

1.3 Contributions and Restrictions

As discussed in the previous sections, the subject of this thesis is the exploration of a system-level approach
for the object-oriented design and implementation of applications for run-time reconfigurable computer
architectures. The focus is on the modeling of applications and platforms, and the mapping of application
designs to the target architecture. The major contributions of this thesis to this field of research are:

- This thesis presents the first complete and consistent approach to the object-oriented design, mapping,
  and synthesis to the system-level design of reconfigurable architectures using the Unified Mod-
eling Language. In particular, the applicability and implementation of the main features of object-
orientation - inheritance, polymorphism, data abstraction, and encapsulation - is explored (→ Chapter
3-6).

- This thesis provides a systematic discussion of the major design approaches from the software- and
  the embedded systems domain, namely, model-driven architecture, platform-based design, and hard-
ware/software co-design, and unifies them into a common approach (→ Section 3.2).

- This thesis proposes a novel approach to the modeling of object-oriented applications and the devel-
  opment environment. The modeling approach is platform-centric, whereas the platform concept is
  consistently used in all phases of development. The applicability of UML as system-level modeling
  language is investigated (→ Section 3.3).

- This thesis presents novel algorithms for the mapping, estimation, synthesis, and run-time manage-
  ment of applications of run-time reconfigurable architectures from object-oriented UML design mod-
  els. A scalable and flexible architecture for the implementation of objects using logic resources is
  proposed (→ Chapter 4-6). The algorithms and architectures comprise the core of the first model
  compiler for such architectures, called MOCCA (→ Section 7.1).

The subject of this thesis is quite general, so naturally only a very restricted sub-set of relevant issues can
be explored with limited resources in the context of a rapidly advancing technological environment. Even
within the chosen sub-set the space of possible research directions and solutions is often too vast to be
investigated in its entirety. Consequently, different approaches are possible, than the one presented in this
thesis. The most promising directions and their relationship to this work for future research will be pointed
out as part of the conclusions. The most notable and influential restrictions to the subject explored in this
thesis are:

- Reconfigurable architectures are becoming important to embedded systems and real-time systems.
  Although reconfigurable architectures have several advantages over traditional microprocessor-based
  architectures also in this domain, the effect of implementations of object-oriented features and run-
time reconfiguration on real-time behavior is still to be investigated. Since such analysis is outside
  the scope of this thesis real-time systems will not be considered.
1.4 Overview

- Another important issue, which is partly related to real-time systems and embedded systems development, are analysis models. In the recent past various approaches to the analysis of such systems have been proposed. In the context of UML these approaches are backed by specialized profiles for system analysis (→ Section A.3.2). In the context of this thesis, investigation starts at system design; analysis models are not explored.

- UML offers a rich set of capabilities for modeling behavioral and structural aspects of systems. In this thesis the activity sub-set of UML is used for behavior modeling. Arguably, activities are the most fundamental means of behavioral modeling, since they are partly reused in the other modeling facilities as well. Other approaches have been discussed in the related work (→ Section 1.2 and 2.3).

1.4 Overview

Apart from the appendices, the chapters of this thesis are organized top-down in the order of the typical development flow. Since each chapter builds on the issues and notations discussed in the previous chapters, the chapters can and should be read in order. The list of symbols supports the skimming and skipping of the text.

Chapter 2 - Theoretical and Technological Foundations discusses the fundamentals and technological context of the presented work. A brief introduction on the issues of reconfigurable computing, hardware design, and UML is given. Moreover, the important related work is reviewed in detail. The sections in this chapter may be read in arbitrary order.

Chapter 3 - Model-Driven Architecture for Reconfigurable Computer Architectures overviews important requirements for system-level design languages. Then an unified development approach is proposed, which is based on the paradigms of model-based and platform-based development, and co-design. An incarnation of this approach for run-time reconfigurable architectures is presented in the remainder of the chapter.

Chapter 4 - Platform Mapping presents a novel approach to the multi-granular mapping of object-oriented design models to the resources of a target architecture. This includes a thorough discussion of the design space, the mapping algorithm, and algorithms for the estimation of implementation characteristics of such models.

Chapter 5 - Synthesis proposes a life cycle for hardware objects and configurations which specifically addresses performance issues imposed by run-time reconfiguration. Then a system architecture for the implementation of object-oriented design models using microprocessors and reconfigurable logic resources is proposed.

Chapter 6 - Run-Time Reconfiguration discusses a principal hardware execution environment for run-time reconfigurable applications. The challenges of dynamic object creation and destruction of hardware objects and the communication between software and hardware are discussed.

Chapter 7 - Experimental Results first presents the MOCCA development environment, which is the implementation of the concepts and algorithms discussed in this thesis. The proposed development approach is evaluated by several experiments.

Chapter 8 - Conclusions concludes this thesis and discusses important future directions of research in this field.

Appendix A - MOCCA Modeling Framework presents the MOCCA action language and the profiles that have been developed specifically for this thesis. The relationship to profiles that have been proposed in this domain is discussed.

Appendix B - Platform Models overviews the platforms and their models that are used throughout this thesis. This appendix builds logically on Appendix A.
Appendix C - Model Transformations provides an overview of the model transformations that are provided by the MOCCA compiler. This discussion provides additional information to Chapter 4.

Appendix D - Experimental Results presents detailed information on the experiments described in Chapter 7 as well as the respective data.
2. THEORETICAL AND TECHNOLOGICAL FOUNDATIONS

2.1 Reconfigurable Computing

2.1.1 Design Space of Reconfigurable Computing

In reconfigurable computing computer architectures and software tools are used that allow for the adaption of the structure and behavior of the processing elements to the requirements of the application. Reconfigurable fabrics (RF) are arrays of concurrently executing reconfigurable function units (RFUs) which are used to perform computations. In general, reconfigurable computer systems couple reconfigurable hardware with a microprocessor (μP). Thereby the temporal von-Neumann execution scheme, which is used in standard computing, is complemented with spatial execution. The operation to be executed by the RFUs is bound to the physical hardware after the manufacturing of the device. This process is called configuration.

Late operation binding supports the implementation of specialized algorithms. Instead of implementing general algorithms, that may solve all instantiations of a class of problems, specialized solutions can be realized at the time the problem instance is sufficiently known. Configuration can be performed recurrently and dynamically. Recurrent operation binding, called reconfiguration, temporally adapts the physical hardware resources to changes in the computational requirements. Dynamic reconfiguration, which is also called run-time reconfiguration, exploits the locality of execution of common algorithms by reconfiguring the hardware during the execution of an algorithm. Partial reconfiguration allows to configure subsets of the RFUs of a reconfigurable fabric while the other RFUs of the same fabric are still in operation.

Reconfigurable computer architectures are mostly used to accelerate performance and I/O-critical operations. An application is mapped to the architecture such that the reconfigurable hardware executes the performance and/or I/O-critical parts of the application, if this is feasible and efficient. The processor executes the overall control flow and uncritical parts of the application at a medium performance. The performance advantage is delivered by the spatial execution scheme of the reconfigurable fabric. However, the configuration and the delegation of computations to a separate hardware may also imply timely overhead. Thus, in order to get a performance gain when executing an operation in a reconfigurable fabric (RF), in general, the central unequation of reconfigurable computing must hold:

\[
\text{t}_{\text{exec,op}_i}^{\text{Proc}_m} \geq \frac{t_{\text{RF}_n}^{\text{conf}}}{n} + t_{\text{comm,op}_i}^{\text{Proc}_m-RF_n} + t_{\text{exec,op}_i}^{\text{RF}_n}
\]

\( n \) – Number of consecutive executions of operation \( \text{op}_i \) on \( RF_n \)

\( m \) – Index of processor

The sum of the execution time \( t_{\text{exec,op}_i}^{\text{RF}_n} \) of an operation \( \text{op}_i \) on reconfigurable fabric \( RF_n \), the communication overhead \( t_{\text{comm,op}_i}^{\text{Proc}_m-RF_n} \) and the proportional reconfiguration time \( t_{\text{exec,op}_i}^{\text{RF}_n} \) should be less than the execution time \( t_{\text{exec,op}_i}^{\text{Proc}_m} \) on processor \( \text{Proc}_m \).

In the past, a variety of reconfigurable computer architectures have been published. The presented systems aim at different application domains, which resulted in architectures with different characteristics. These architectures can be differentiated by their granularity, interconnection network, reconfiguration support, and the coupling between the processor and the reconfigurable fabric.
Granularity of the RFUs

The granularity of reconfigurable fabric is the size of the smallest reconfigurable function unit of the hardware, which is determined by its functionality.

In fine-grained architectures the RFU supports operations on individual bit operands. These architectures suit operations on non-standard word-sized operands and bit-level operations. Complex operations are constructed from a sea of fine-grained RFUs. This offers large optimization potential, but also implies a relatively high area and interconnection overhead in terms of allocated routing resources and routing delay. The fine granularity imposes a large number of configuration bits, which consequently increases the size of configuration contexts and the configuration time. Thus, fine-grained architectures are most suitable to applications working at relatively low reconfiguration rates. Fine-grained RFUs are mostly used in FPGAs, e.g. [17–21].

The RFU of medium- and coarse grained architectures (e.g. [22–39]) supports standard operations on word-sized operands. The operations are optimized for performance, implementation area, and power dissipation. In comparison to fine-grained architectures, the reconfigurable hardware is build from orders of magnitude less RFUs. This implies smaller structures and fewer configuration points. Thus, coarse grained architectures are most suitable to applications working at high reconfiguration rates. The optimization toward standard data-paths makes medium and coarse grained architectures less efficient for operations on operands whose size does not correspond to their native word-size, because specialized operations must be constructed from multiple RFUs.

Interconnection Network between RFUs

The interconnection network defines the applicable communication patterns and the communication bandwidth of the reconfigurable fabric.

The granularity and the interconnection network are interdependent. Architectures with higher granularity utilize less RFUs than lower granular architectures. Due to the higher granularity, wide interconnects with multiple bits are used. This features data parallelism and thereby fast communication. Consequently, less routing resources are required, which implies less implementation area and fewer configuration bits.

In coarse-grained architectures each RFU executes relatively large chunks of the algorithm, which reduces the overall communication effort. This enables communication structures, like time-multiplexed buses, that connect large sets of RFUs. Wider communication paths often make relatively slow interconnection networks acceptable. Fine- and medium-granular architectures construct complex functions with multiple RFUs, which necessitates a small interconnection delay. Programmable interconnects imply larger interconnection delays, implementation area, and power dissipation, which again impose additional mapping and synthesis problems, as pipelining and re-timing [40].

The existing reconfigurable architectures use crossbars, linear arrays, meshes, and combinations of them, to interconnect RFUs. Crossbar architectures (e.g. [22, 23, 36, 37]) feature RFUs with arbitrary or nearly-arbitrary communication patterns. Due to practical implementation limits full crossbars are only applicable as main interconnection structure in coarse-grained architectures with a few (10..50) RFUs. If the communication patterns of the application are statically known to be less arbitrary specialized architectures can be used. Pipelined architectures are targeted toward straight data-flow computations. The RFUs are organized as single or multiple linear arrays, whereas each RFU can communicate with its next neighbor (e.g. [30, 36, 37]). Meshes are the most commonly used interconnection structure in reconfigurable computing, e.g. [25–27, 29, 31–35, 38, 39]. They support a high degree of parallelism and efficient implementations. Thus, they provide a compromise between closely connected, dynamic networks and pipelines. The RFUs are organized such that each RFU is connected with its direct neighbors. Some implementations connect the RFUs at the borders of the mesh with the RFUs on opposite border to build more sophisticated structures.

Reconfiguration of the Reconfigurable Fabric

The ability to adapt the hardware after its fabrication is the distinguishing property of reconfigurable computer architectures. Since acceleration is the most frequent aim of reconfigurable computing, significant
reconfiguration efforts may hamper this goal. The reconfiguration effort is determined by the characteristics of the reconfigurable hardware and the application. Due to their large number of configuration points, reconfiguration is critical particularly in fine-grained architectures.\footnote{Depending on the number RFUs and the configuration interface, the reconfiguration of modern FPGAs can require several milliseconds; the size of the configuration contexts can be a few MByte.}

In the last years, a number of optimizations reducing the negative effects of reconfiguration have been published. Most optimizations aim at decreasing the reconfiguration time. Examples include configuration prefetching \cite{41}, configuration compression \cite{42–45}, partial reconfiguration \cite{20, 21, 29, 31, 32, 36, 37, 39}, configuration relocation and defragmentation \cite{46, 47}, configuration caching \cite{45}, and multi-context devices \cite{48}.

Another approach is to reduce the overall number of reconfigurations. Thereby it is sought to increase the use of the currently active configuration contexts by increasing the number of consecutive operation executions (parameter $n$ in equation 2.1). The basic principles of these optimizations are to exploit the regularity inherent to the problem and to transform the problem to expose a greater regularity \cite{40}. This problem can be addressed statically and dynamically, during mapping, synthesis, and scheduling.

Virtual hardware machines overcome the restricted portability of reconfigurable applications, which is due to the binary incompatibility of the reconfiguration contexts, by defining an abstract hardware layer. This layer is targeted by mapping and synthesis. Online optimization, technology mapping, place and route, configuration context generation, and deployment are performed to adapt the virtual contexts to the specific reconfigurable fabric \cite{49, 50}.

### Coupling of Reconfigurable Fabric and Processor

The coupling of a reconfigurable architecture defines the functional interface between the processor and the reconfigurable fabric. Thereby it determines the operations that can be implemented in the reconfigurable fabric and the according access mechanism, which impacts the employable communication and synchronization mechanisms.

We differentiate between data-path-coupled and network-coupled architectures. In data-path-coupled architectures a reconfigurable fabric is integrated into the data-path of the processor \cite{51–61}. Application specific instruction processors extend the native instruction set of a processor core with custom, application specific instructions \cite{62}. The RF is used to realize operations for custom instructions that are executed in the fetch-execute-store cycle of the processor. The number and type of custom instructions is, however, limited by the ISA.

In network-coupled architectures the RFs are connected to the processor with an interconnection network \cite{34, 39, 63–66}. The coupling between the processor is determined by the architecture of the computer system. The network may be the processor bus, direct connections, local network switches, or even internetworks. The reconfigurable hardware operations are accessed with explicit transfers over the network, rather than the implicit instruction triggered activation. A shared or distributed memory or a combination of both may be used \cite{67}.

### 2.1.2 System-Level Design

Since the offspring of reconfigurable computing in the early 1990s, the development of applications for reconfigurable architectures has been a challenging and complex task. The goal of system-level design for reconfigurable computing is the translation of a target architecture independent requirements specification into a design that is executable on the target architecture. The design must implement the system functionality and satisfy the required properties and constraints. The principle design flow of system-level design is illustrated in Fig. 2.1. The development process is separated into three phases: specification, DSE, and refinement \cite{68, 69}. During development these phases are commonly performed repeatedly at decreasing levels of abstraction.
2. Theoretical and Technological Foundations

Verification/Validation

Lower Level Design Flows

Exploration

Refinement

Functional Specification

Non-Functional Specification

Specification

Exploration

Refinement

Specification

Fig. 2.1: General System-Level Design Flow

**Specification**

During specification, we capture a model of the design of a system. The input may be a set of requirements or a design generated by a higher level design flow, both of which specify a model of the system under development. Output of the specification phase is a design of the system with respect to a particular design space. Formally, a system specification \( s \) can be defined as the following quadruple [70]:

**DEFINITION 2.1:** A system specification is defined as:

\[
    s = (F_s, P_s, Q_s, C_s)
\]

\( F_s \) is a specification of the functionality of the system. \( P_s \) is a set of properties that must be satisfied by the specification, e.g. safety, robustness and, liveness. \( Q_s \) is a set of quality characteristics that describe the quality of the specification, e.g. performance, area, and response-time to external stimuli. \( C_s \) is a set of constraints, that bind the quality characteristics to certain values, e.g. minimum performance, maximum area and maximum response-time. Designs that do not implement the system function or do not satisfy the properties and constraints are considered invalid. The adherence to these characteristics must be ensured for all designs into which the system specification is translated. Adherence is given either implicitly in the specification, or must be explicitly checked by validation, verification, and simulation.

Each design is defined with respect to a particular design space.

**DEFINITION 2.2:** A design space offers instances of a particular set of resource services at some level of abstraction \( n \):

\[
    R_n = \{r_i\} - \text{set of resource services}
\]

\[
    O_{r,n} = \{o_j\} - \text{set of implementation options implementing } r \in R_n
\]

\[
    O_n = \{o_j\} = \bigcup_{r \in R_n} O_{r,n} - \text{set of implementation options}
\]

\( n \) – level of abstraction

Each implementation option possesses a set of properties \( P_j \), quality characteristics \( Q_j \), and constraints \( C_j \) on these characteristics:

\[
    o_j = (P_j, Q_j, C_j) - \text{implementation option}
\]

An instance of a resource service is then defined as:

\[
    ri = \{id, r_i, o_j\} - \text{resource service instance}
\]

whereas \( id \) represents an unique identifier of the instance. Then a design space is a set of resource service instances:

\[
    DS_n = P(\{ri\}) - \text{set of subsets of resource services instances}
\]
Resource services can be as primitive as transistors, wires, and logic operations, but may be also complex services for processing, storage, I/O and communication, or even as abstract as classes of a UML design model. Each service may be realized by a set of resources, like processors, storages, and communication adapters. Thereby each resource may realize several services. For each resource service a multitude of implementation options exist. For instance, a resource service "adder" may be realized serially, with ripple-carry, or carry-look ahead. An implementation may realize several resource services, e.g. an arithmetic-logic unit (ALU) implements various primitive operations. There may be multiple instances of the same resource service and implementation option.

**DEFINITION 2.3:** A design is an element of the design space and defined as an allocation of a set of resource service instances:

$$d_n \in DS_n$$

The design realizes the functionality of system s. Then the quality of designs is evaluated by its quality-of-service (QoS) which is a function of the set of quality characteristics $Q_s$ being assessed:

$$qos: DS_n \mapsto Q_n^{Q_s}$$

$Q_n$ – set of quality characteristics at abstraction level n

$Q_s$ – quality characteristics defined for system s

Consequently, a design realizes the functionality of the system s with a sufficient set of resource service instances that are offered by the design space. The design quality is determined by the characteristics of the specific selection of resource service implementations. A design space represents a, possibly abstract, computer architecture, which may comprise physical hardware or abstract machines (e.g. execution environments and tools).

**DEFINITION 2.4:** The architecture defines an interface that describes the functionality of an implementation, while being independent of the actual implementation [71, 72].

An architecture provides interfaces that define coherent sets of functionality and non-functional information on the characteristics of its services, implementations, and usage patterns. It should hide implementation details of its lower level refinements from the systems that depend on it. The implementation of the architecture is called micro-architecture.

**DEFINITION 2.5:** The micro-architecture defines the realization of functionality as a composition of modules and components, along with their associated software [71, 72].

For example, the architecture of a microprocessor is defined by its ISA, specifying the instruction set and registers, without defining the actual realization of these components. The micro-architecture of this ISA defines a specific processors data-path, control-unit, and other components, that realize the ISA. The distinction between architecture and micro-architecture depends on the considered level of abstraction. What is considered as architecture at some level of abstraction may be considered to be a micro-architecture at a higher level.

In general, the generated design is captured in a language (or a set of languages) and with respect to a model of computation (or a set of computation models) [70, 73–75]. Present approaches to system specification vary in the employed models of computation, the languages, the degree of implementation detail included in the design, and the degree of automation.

There is a significant difference between fine-grained and coarse grained architectures regarding the specification. The design flows for coarse-grained architectures mostly use programming languages (e.g. assembler [24, 31, 32, 39], C [26, 27, 30, 34–37, 76–82] and complementary data-flow languages [22, 79, 81]). This is because these systems have been outset as hardware/software systems with the goal to accelerate algorithms described in software.

In contrast, fine-grained architectures were initially developed as replacement of traditional ASICs. The technology specific parts of existing ASIC tool-chains have been adapted respectively. Even today the
integrated hardware/software development for these architectures is still rare [83–88]. The majority of applications for fine-grained reconfigurable architectures is defined using mixed-language approaches. A software programming language is employed for software design and the hardware is designed using an HDL (e.g. VHDL [89], Verilog [90]).

Design Space Exploration

During the DSE, we seek a transformation $f$ that transforms a design $d_n$ of system $s$, to design space $DS_{n+1}$, such that $f$ implements the functionality $F_s$, satisfies the properties $P_s$, optimizes the quality characteristics $Q_s$ and fulfills the constraints $C_s$. To evaluate the quality of the target design we can define a cost function $cost$, whereas smaller outcomes of $cost$ mean higher quality while higher values reflect worse quality.

**Definition 2.6:** A cost-function is defined as:

$$cost : DS_{n+1} \times Q_s \mapsto \mathbb{R}$$

$$f : DS_n \mapsto DS_{n+1}$$

$$d_{n+1} = f(d_n) \land cost(d_{n+1}, Q_s) \rightarrow \min$$

$$d_n \in DS_n, \quad d_{n+1} \in DS_{n+1}$$

The design space is searched for refinements of the resource service implementations, which are used in $d_n$, into resource services and respective implementations contained in $DS_{n+1}$. Such a transformation generally decreases the level of abstraction, which may necessitate the decomposition of the $o_{l,n} \in O_n$ into several functionally equivalent instances $o_{l,n+1} \in O_{n+1}$. Decompositions can be combined with compositions and further transformations to enable synthesis or to optimize the target design. On system-level, the design is partitioned among the resources of the target architecture, such that the functionality is realized, the non-functional requirements are satisfied, and the quality is optimized. To accomplish this goal, we compute a number of partitions, ascertain their characteristics with respect to a set of metrics, and evaluate the quality with the cost function. The partition with the best quality (minimum cost) is chosen for further refinement.

According to the description of the partitioned entities functional and structural partitioning are distinguished [91]. At system-level, functional partitioning is common today, because multiple design alternatives can be easily explored. The design is partitioned spatially and/or temporally amongst the hardware resources. Spatial partitioning divides the design among different resources. Temporal partitioning computes configuration contexts (programs) for a single run-time reconfigurable (programmable) resource that execute in mutual exclusion on the resource. Hardware/software partitioning is the spatial bi-partitioning between a custom circuit (e.g. ASIC, FPGA) and a processor. Partitioning approaches are further classified by their granularity, the metrics and cost function, the algorithm, and the degree of automation.

**Granularity.** The granularity of the partitioned entities may range from Boolean functions [92], single operations [93], operation networks (e.g. data-flow graphs, control/data-flow graphs) [39, 56, 76, 78, 79, 81, 82, 94–109], functions and methods [87, 110, 111], variables and channels [68, 69, 112], objects [16, 113, 114], processes [112, 115] to entire subsystems. The finer the granularity, the better the control, accuracy, and optimization potential, but the more time is required to compute a partition. To circumvent this problem Henkel and Ernst proposed to use dynamically determined granularities at the level of operator networks [101].

**Metrics and Cost-Function.** DSE generally relies on metrics on the implementation characteristics of the generated solutions. The cost function weighs the assessed metrics with respect to the optimization goal. Thereby a reliable, robust, and fast estimation of system properties $P_s$ and quality characteristics $Q_s$ is necessary in order to avoid design iterations. The area-time-power (ATP)-characteristics are frequently used quality characteristics, whereas in the context of reconfigurable computing area and latency estimates are of most interest. Such characteristics are either measured or estimated.

There exists a number of related work in this area in both the hardware and the software domain. Bilavarn et al. [116] proposed a bottom approach to the estimation of ATP-characteristics for data-paths that are
implemented on FPGAs. Area estimates are based on functional-units and registers. Although they support resource sharing, multiplexers are not taken into account. An estimation approach that is incorporated into partitioning is proposed by Henkel and Ernst [117]. They allow for the estimation of the datapath and controllers of finite state machines. Other approaches support only the estimation of data-flow graphs (DFGs) [118–121]. Two recent proposals use empirically derived formulae that are parameterized by simple metrics of the DFG, the actual platform mapping is not considered however [120, 121]. Others use combinations of integer linear programming (ILP) and graph matching [118].

In the software domain, the estimation of worst case execution time is of most interest, particularly for real-time systems. Early approaches focused on the estimation of execution probabilities and frequencies. Recent research shifted towards the static analysis of execution time in presence of advanced uP-features, including pipelining, super-scalarity (multiple instruction issue), caching, and dynamic branch prediction [122–130]. These features aggravate static timing analysis since they cause the instruction latency depending on the current state of the machine. Moreover, they are not independent from each other what makes their effect hard to analyze. Current approaches model the micro-architecture of real or virtual target machines. Typically, ILP-based path analysis techniques are combined with instruction set simulation. The effects are studied mostly in isolation, so there is no current solution that supports even the most common features.

**Partitioning Algorithms.** Partitioning approaches fall into deterministic approaches and heuristics. Deterministic algorithms compute identical solutions for identical inputs with identical time and resource requirements. A number of deterministic approaches has been proposed for automated solution to partitioning problems. Mathematical approaches, like linear programming (LP), model the partitioning problem as equation system whose solution is the solution to the partitioning problem [99,100,103,106,131]. Others use dynamic programming to construct accurate solutions [96, 98].

However, modeling partitioning problems for LP and dynamic programming is a quite complex task and the tractable problem size is restricted. Thus, the majority of partitioning algorithms computes approximative solutions with simple algorithms. Important algorithms are based on clustering [39, 95,112,132] and group migration, using variants of tabu search, such as Kernighan/Lin variable depth search [109,113,133,134], scheduling [97,108,115,135,136], binary constraint search [137], and combinations of several approaches [114].

Heuristics do not guarantee the computation of a solution with time and resource requirements, but for the most cases they allow for the quick computation of high quality solutions2 [134]. Because they are very robust and cheap to use, heuristics are currently the preferred way to solve partitioning problems. *Simulated annealing (SA)* is a memory-less, randomized local search, which models the optimization problem as process for obtaining low-energy states of a solid [134]. Starting with an initial solution, iteratively new, randomly selected solutions are searched in the neighborhood of the current solution. A solution is accepted as current solution if it costs less than the current solution, or, in order to overcome local extremities, it is randomly accepted with steadily decreasing probability even if the cost is worse than the current cost [81, 94, 101, 102, 105, 111]. At most one solution is active at any point in time. If this is a disadvantage, multi-start SA can be used. The local search is illustrated in Fig. 2.2. *Genetic algorithms* solve optimization problems by simulating the process of evolution of populations of individuals [134,138]. Starting with an initial population, new populations are computed by iteratively combining (crossover), changing (mutation), and selecting individuals. The individual with the minimum cost

---

2 Notice that the term "heuristic" is used wrongly and inconsistently in the literature to refer to partitioning problems that do not guarantee to compute the optimum solution. We use the definition given in [134] instead: "[A heuristic] provides randomized algorithms for which one is not able to guarantee the efficiency and the quality of the computed feasible solutions, [...]."
(best fitness) is chosen for refinement. This heuristic is good at coming out of local minima in the design space [104, 105, 107, 139, 140]. Another biologically inspired method is ant colony optimization [141, 142]. This algorithm searches the design space by imitating the cooperative path finding strategy of ant populations. Promising directions of the design space (mapped to ant tracks between towns) are examined more intensively than unfruitful parts.

**Automation.** Partitioning does not have to be performed automatically. In fact, the most current approaches to system-level design utilize manual partitioning, because the research in this field is still in its infancy. Manual partitioning, however, may be complemented by automatically generated suggestions, generated by design space exploration systems (e.g. [28, 143–145]).

**Specification Refinement**

During specification refinement, the mapping, that was computed during DSE, is actually manifested. The design that was determined in the previous phase is generated into a respective description. The computation components and communication structures of the higher level description are implemented with respective elements of the lower level architecture. The generated design is commonly described in a different language and model of computation, both of which are defined by the target architecture.

According to the partitioning software and hardware modules are generated for functionality being mapped to processors and the reconfigurable hardware resources respectively. Additionally, interfaces for the communication between hardware and software must be generated, if the target architecture does not define standard interfaces. The generated modules and the non-functional requirements specification are input to specialized design flows which further translate the design into lower level behavioral or structural representations.

### 2.2 Hardware Design with High-Level Languages

#### 2.2.1 High-Level Languages for Hardware Design

To overcome the deficiencies of the schematic-based design capture at gate-level, in the early 1980s several efforts toward HDLs have been started, which improve the lack of documentation and the poor integration of verification. The languages were required to be technology-independent and to have a well-defined, simulation-oriented semantics, and a high descriptiveness. The two most important outcomes of these efforts are Verilog [90, 146] and VHSIC HDL (VHDL) [89, 147]. The behavior and structure of the design is captured textually at RTL.

In the late 1980s, efforts were started to raise the level abstraction to behavioral-level. Some approaches use a HDL for design capture [148], other efforts utilize software programming languages, such as variants of C [10, 78–80, 94, 149–160], C++ [7, 15, 86, 161–165], Java [11, 166, 167], and specialized languages such as SAFL [110], ‘e’ [168] and Matlab [169, 170]. These approaches implement behavioral/architectural synthesis using various timing and concurrency models [171]. Timing and concurrency must be specified either explicitly or they are imposed by tool-specific rules. The explicit specification is more common, because the resulting design style is similar to traditional hardware design. The designer is given the full control over the generated implementation. Rule-based approaches argue that on the behavioral-level full specification of timing and concurrency is often not necessary and even not desirable, because this complicates the design and reduces the optimization potential. On the other hand, the specification of a particular timing can become a tedious task, because the behavior must be encoded with respect to the specific rule set.

In this thesis, VHDL is used as language to describe hardware designs at register-transfer level. This section gives a brief introduction on the most important concepts of VHDL and the basic hardware design flow. More thorough introductory material may be found elsewhere (e.g. [147, 172]).
2.2. Design Space of VHDL

VHDL supports the specification and design of digital and analog hardware circuits. For the specification the language integrates concepts to describe the algorithms and timing of the system at behavioral-level. The purpose of this specification is documentation and early simulation in the design flow. Behavioral specifications are not directly synthesizable, they must be translated into equivalent RTL designs. This translation is often done manually. Automated translation is performed by behavioral synthesis tools. Synthesis tools refine RTL designs into structural descriptions of the hardware (e.g., netlists, mask data, configuration contexts).

VHDL designs at register-transfer level represent the overall architecture of the system. Depending on the amount of implementation options that have been fixed in the design, the potential of the DSE performed by the synthesis tools is limited. Consequently, the technology independence of RTL designs is also limited. Thus, in order to get a certain degree of technology independence and to provide some optimization potential to synthesis, register-transfer level designs often use both, structural and behavioral style.

A VHDL design comprises a hierarchy of instances of hardware building blocks, called components. There is one top-level instance that contains all other instances, whereas each instance is contained in at most one other instance. VHDL features the structural and behavioral description of designs. Structural descriptions comprise circuits (components) that are connected by signals. Structural nesting of components supports complex designs. Behavior is defined at register-transfer level. RTL designs comprise combinational and sequential logic, whereas the combinational logic is located between memory elements (registers), which may be defined explicitly or implicitly. The supported model of computation is discrete events. If other models are required, they must be constructed explicitly from this basic model!

EXAMPLE 2.1: Fig. 2.3 illustrates these concepts using a 2-bit D-latch. The latch instantiates two 1-bit latches and connects their ports by appropriate signals. The definition of a 1-bit latch is shown on the left hand side. The sequential behavior of the latch is located in a process-statement. The value of the input signal is assigned (<=) to signal DFF synchronous to the rising edge of a clock-signal (CLK). Since not all possible evaluation paths in the process contain an assignment of this signal, an implicit register is inferred for DFF. The current value of DFF is steadily written to the output of the latch.

Fig. 2.3: Example: VHDL design of a 2-bit D-latch

The semantics of the most important VHDL constructs used in the example are summarized as follows:

signal - Signals are the fundamental carrier of information and correspond to wires. Hence, they are used for the static connection of component instances. Each signal has a type. VHDL employs a strong typing scheme and an extensible type hierarchy that comprises scalar and composite data types. The set of synthesizable types depend on the tool chain. For portability reasons designs should restrict to std_logic (single, three-stateable bit) and std_logic_vector (vector of std_logic). These types define extensible sets of operators, e.g., for assignment, arithmetic, logic, and relational operations.
entity - Entities define the named interface of hardware modules by means of port signals. For each port signal the type and its direction type must be defined. In addition to port signals, an entity declaration may contain generic signals, which can be used for static design parameterization. The interface may be implemented by one or more architectures.

architecture - Architectures define the implementation of an entity interface as behavior, data flow or structural description, or mixtures of them. Architectures contain statements that describe combinational and sequential logic. The statements contained in the top-level of an architecture body, that is outside of a process, are evaluated concurrently and permanently. All behavior and structure of a design is described in architectures.

cOMPONENT - Components are structural elements that, like entities, describe the external interface of a hardware building block in the architecture that instantiates the block. An instance of any entity with this interface may be used as instance of this component. Which entity and architecture are actually used is determined by implicit or explicit configurations. Example 2.1 uses implicit configurations for the instances dff0 and dff1 of component d_latch in architecture rtl of entity d_latch2. Component instantiations are static. The port signals of the instance are connected to the respective signals of the instantiating architecture via explicit mappings (port map).

process - Process statements are used in architectures mostly to describe sequential behavior. The statements in a process body are evaluated sequentially, often synchronously to a dedicated clock signal. Processes may contain special sequential statements to describe conditional and repeated signal evaluation and assignments (e.g. if, case, loop).

Example 2.2: Fig. 2.4 illustrates the mapping of the VHDL if and case statements to hardware. It is important to notice that although the definitions are familiar from programming languages, the semantics of these constructs is different from software! Functionally equivalent statements may be mapped to different hardware. The nested if statements are implemented as priority encoder, while for the case statement balanced logic with shorter delay and less resources is synthesized. Due to the static nature of hardware, the flexibility of the language constructs is restricted. For instance, for synthesizable loops the bounds and increment must be fix and statically known, because loops are completely unrolled at synthesis time. The same considerations apply for arrays, for which all accesses must be statically known.

The language supports the handling of complex designs and reuse. Behavioral hierarchies are supported by means of subprograms and functions. Their principle semantics is similar to common programming languages. In contrast to software implementation, in the hardware the body of each subprogram or function is always inlined, because VHDLs computation model has no means of explicit control transfer. The organization of design units at the source level is supported by packages.

2.2.3 Hardware Design Flow

The goal of the hardware design flow is to translate a hardware design and the non-functional requirements of a system into an equivalent hardware architecture. As shown in Fig. 2.5, the hardware design
flow comprises the same fundamental phases as the system-level design flow, for which the same principal considerations apply (→ Section 2.1.2).

**Fig. 2.5: General Hardware Design Flow**

**Specification**

During specification, we capture a design of the hardware and the non-functional requirements. As on system-level, the design entry can be homogeneous or heterogeneous with respect to the employed languages and computation models. The language for design entry may be a dedicated HDL, a software language, or a graphical formalism (e.g. finite state machine (FSM)-graphs, schematics). Combinations of different description types are common. Accordingly, the level of abstraction and the computation model vary between the approaches. Non-functional requirements comprise design constraints (e.g. latency, area, power), exploration and refinement constraints (e.g. FSM-encoding, implementation, and optimization options for arithmetic/logic functions), and interface constraints (e.g. pin-out, I/O-timing, drivers).

**Design Space Exploration**

During DSE, we map the design to the target hardware architecture. Base of the DSE is a netlist. The nodes of the netlist are operations and the edges represent the respective data flows. The netlist is optimized, e.g. by using arithmetic/logic simplification and decomposition, dead expression elimination, tree-height reduction, and common sub-expression elimination transformations.

The operators in the netlist often do not match the operators provided by the target architecture. Technology mapping converts the original netlist into a functional equivalent netlist, called *circuit netlist*, that uses only operators of the target architecture. There are different approaches to technology mapping according to the operators in the netlist and the target technology. For netlists whose operator granularity resembles the granularity of the target technology (→ Section 2.1) there are three main approaches to technology mapping. *Rule-based techniques* replace local subnets in the original list by functionally equivalent netlists according to a set of replacement rules. The replaced netlists are identified by pattern matching [173–175]. *Library-based techniques* decompose the original netlist into a netlist consisting of some standard gate (e.g. m-input NAND/NOR). Then an optimum covering of the decomposed netlist by logic functions of the target technology, which are stored in a library, is searched [175–177]. In contrast, *cell-based techniques*, used for ASIC development, employ sets of parameterized standard cells rather than logic functions [178]. Netlists with complex operators are mapped to fine-grained architectures by mapping each operator, or groups of operators, to pre-mapped modules [179, 180].

Placement is the optimization problem of assigning the nodes of the circuit netlist to physical processing units (e.g. RFUs or standard cells), whereas all edges in the netlist must be routable and no two nodes are allowed to be placed onto the same processing unit. The placement goal is captured by a cost function,
2. Theoretical and Technological Foundations

that directs the algorithm toward some set of optimization goals (e.g. minimum delay, minimum area, wire length, temperature). Due to the similarities between the general partitioning problem and the placement problem, current approaches use similar algorithms as in system-level partitioning [181] (→ Section 2.1.2).

Routing is the optimization problem of assigning physical wires to each edge of a placed circuit netlist, whereas no two edges are allowed to be mapped to the same wire. Routing is constrained by the placement and the routing resources of the target architecture. This is specifically a problem for architectures with restricted routing resources, as reconfigurable architectures. Routers commonly use a cost function to evaluate a routing with respect to metrics, like routing area, routability, and timing. An overview of routing approaches to reconfigurable computing can be found in [28].

Specification Refinement

During specification refinement, we generate a structural representation of the placed and routed circuit netlist. This representation may be a configuration context\(^3\) for a reconfigurable architecture or mask data for the chip- and printed circuit board (PCB)-manufacturing. The format of the configuration data is highly dependent on the hardware architecture. The configuration information commonly comprises the raw configuration bits, checksums, and address information (for partial configurations). For information protection the configuration data may be encrypted.

2.3 The Unified Modeling Language

2.3.1 Design Space of UML

The Unified Modeling Language is a non-proprietary modeling language for the visual specification, design, and documentation of the artifacts of systems [6, 182–186]. UML is independent of specific domains, architectures, and methods. This language was introduced in 1997 in response to the demand for uniform and consistent notations for object-oriented software systems and processes. Despite of the large area of applications, the core language requires only moderate learning effort, which is because of the fine-grained hierarchical organization of the language, supporting various degrees of compliance, and the massive reuse of design principles [182, 185]. The model exchange between different organizations and tools is supported through XML Metadata Interchange (XMI) [183, 186].

The execution model of UML is founded on the object-based model of computation [75]:

**Definition 2.7:** An object encapsulates state and behavior and can send and receive messages. A computation is performed by objects that exchange structured messages.

In addition, UML features OO by means of its specification mechanisms and the supported development principles\(^4\) [75]. With these mechanisms the modeling of systems and their relationship to the environment is supported.

The language supports the specification of structures and behaviors on different levels of abstraction, and the relationships between the model elements. Thus, the successive refinement of design elements is supported. Additionally, UML enables the modeling of general activities, which enables the modeling of processes, e.g. for system development and business operations. The stakeholders, their relationship, activities as well as the participating documents can be captured. UML does not define specific processes or methodologies however.

To adapt the language to particular domains, architectures, or methods, the language constructs can be specialized. Specialization is done either implicitly, by means of the employed tools, stakeholder agreement et cetera, or explicitly, by means of extension mechanisms. Each extension is exclusively additive, that

---

\(^3\) Because configuration contexts are frequently loaded sequentially into the device they are called bitstreams.

\(^4\) In literature there is ongoing discussion on the features of object-oriented languages, which is mainly driven by marketing interest (e.g. [187]). We agree with the discussion in [75], which considers a language object-oriented if it provides first class mechanisms for encapsulation and inheritance.
is, it may add constraints to language constructs, but it must not take existing constraints away. Sets of coherent extensions are organized in profiles. Profiles extend the UML meta-model by constructs specific to an application domain and thereby define domain-specific languages. For instance the "UML Profile for Schedulability, Performance, and Time" [188] extends the UML by constructs for modeling resources and resource services which are commonly found in embedded computer systems. Other examples include "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms" [189] and "Systems Modeling Language (SysML)" [190] (→ Section 2.3.2).

**Modeling Structure**

The fundamental constructs for modeling the structure of UML designs are classes, interfaces, packages, and relationships. Classes describe coherent sets of features and constraints. As such, classes serve as templates for the creation of class instances, called objects. An attribute is a typed, structural feature, which is associated with a class and is a part of the state of the respective objects. Operations and receptions are behavioral features that specify the access protocol and type of associated object behaviors. Constraints impose additional conditions on objects and features, such as pre- and post-conditions, the set of raised exceptions, and eventual side effects. As a result, classes implicitly specify object interfaces, which can be made explicit through the model element *interface*. Classes and interfaces can be organized in inheritance and implementation hierarchies that are modeled by means of generalization and implementation relationships, which are constrained according to the basic principles of OO. Classes can have association relationships, which define message exchanges that can occur between instances. Dependency relationships make additional semantic dependencies between model elements explicit, such as usages, type mappings, access permissions, and traces of refinements through different models. To facilitate the management, exchange, and understandability of UML models, model elements are organized in packages and models.

**Example 2.3:** Fig. 2.6 shows a UML class diagram comprising two classes, **Client** and **Server**. Clients request computations from the server using the compute(...) operation. This is done by sending messages over the compute association between the classes. Both classes contain constructor and destructor operations, which are marked with stereotypes *create* and *destroy* respectively.

![UML Class Diagram Example](image)

While the aforementioned constructs are used in models at different levels of abstraction, additional elements for structural modeling specific to design, implementation, and deployment are provided, namely components, artifacts, nodes, and communication-paths. These constructs are basically specialized classifiers and relationships. Components are replaceable parts of a system and encapsulate coherent sets of state and behavior. They may have associations and dependencies. Components are realized by classifiers and can own sets of model elements. They offer a partitioning mechanism that is, in contrast to packages, related to system realization rather than to model management. The content of a component is accessible through its interfaces which are defined by ports and implemented by the realizing classifiers.

Components are abstract entities; they are manifested by artifacts, such as files, libraries, and processes. Artifacts may be deployed on the nodes of a computer architecture. Nodes are specialized classes that define the computational resources of a hardware architecture. Networks of nodes can be constructed by means of communication-paths. In general, a node comprises a processing element, memory and I/O facilities. The architecture and services of nodes and communication paths is not modeled in detail. Examples of nodes include workstations, servers, and embedded devices; the respective communication paths may represent buses, direct connections and wireless links.
Modeling Function and Behavior

Behavioral modeling with UML is based on the object-oriented model of computation [75]. The objects may be instances of classes, components, nodes, and auxiliary classifiers such as actors and data types. Messages can be synchronous and asynchronous operation calls and signals. The sending and reception evoke discrete events in the sender and receiver respectively, which can be used to trigger some associated behavior. Behavior specifications owned by classifiers, define the conditions and activities for the state transitions of the respective objects. Behavior specifications owned by operations, represent the implementation of the particular service provided by the operation. Receptions specify the capability of objects to react to particular signal types.

The protocol and type to access object behaviors are specified by operations and receptions. As can be seen in Fig. 2.7(a), the actual behavior can be modeled in detail using various language constructs, such as actions, activities, state machines, and interactions. These constructs allow for the complete, precise and therefore executable behavior specifications. They may also be used for rather imprecise specifications, as they are often required during requirements capture and system analysis. Different methods for behavior specification are frequently used together in the same UML model, whatever is most convenient and appropriate to a particular behavior. These constructs are not mutual exclusive. That is, different types of behavior specification can be embedded into each other. For example, in state machines state behavior is defined using any of the specification methods. The same applies for the other constructs.

Actions and Activities. In UML data transformations can be defined using actions or the Object Constraint Language (OCL) [184]. OCL specifications are declarative. In contrast to actions, they do not support the definition of flow control or side effects to the model. OCL is very efficient for defining relations over data, and is frequently used in database and web applications, and as verification language. UML actions foster an imperative specification style. The UML predefines a large number of actions, such as actions for the creation and destruction of objects, communication, and for reading and writing structural features. The OpaqueAction enables the definition of actions with particular semantics using a user-defined language.
2.3. The Unified Modeling Language

Fig. 2.7(b) exemplifies this for an add action. The instances of the action can be represented in object notation or action notation, whereas both notations are equivalent.

The coordination and communication of actions is accomplished by activities. Activities define the execution context of sets of activity nodes (ActivityNode) and activity edges (ActivityEdge). The execution model within an activity can be object flow oriented, control flow oriented, or both. Both models are based on token exchange between activity nodes. Actions and objects are special activity nodes. Object flows and control flows are special activity edges. In this thesis opaque actions are used to define specific transformations on objects. The used action names can be found in Section A.2.2 in the appendix.

UML defines a graphical notation for the specification of basic actions and activity nodes. For some activity nodes and edges no graphical notation is defined. The UML encourages the definition of action languages or other notations for these language constructs. Action languages define a textual syntax for a subset of UML actions. They are designed to suit a particular application domain, and commonly fix some of the degree of freedom provided by the general execution model. In the past, several action languages have been proposed, such as ASL, AL, OAL, and SMALL [191–194]. These languages are designed specifically for software-only applications in business, telecommunication, and network systems.

Opaque Behaviors. Opaque behaviors provide a method of behavior specification that can be used if none of the other constructs is appropriate. As Fig. 2.7(a) shows, opaque behaviors can be used to extend the UML by application-specific languages, such as action languages and programming languages. Different languages can be used to define the same behavior. The UML specification provides a means to express common language constructs using activity nodes and edges. This enables the entire model semantics being defined in UML.

State Machines. State machines are often used to specify global object behavior. UML state machines closely resemble Harel state charts [195] in that states can be hierarchically nested and multiple states may be active simultaneously. The state transitions are triggered by events, which again are evoked directly or indirectly by actions. Each state can execute activities at its activation, finalization, and while it is active. This enables the modeling of behavioral hierarchies.

**EXAMPLE 2.4:** In Fig. 2.8 two examples of state machine and activity diagram specifications in UML are illustrated. The state machine diagram defines the behavior of instances of class Client. Transitions fire when specific conditions are fulfilled. Such conditions are noted next to the transition in square brackets. Dedicated sub-machine states are used to represent parts of a state machine that are specified in a separate state machine. In activity diagrams transitions between activity nodes represent control flow. Objects flowing into and out of activity nodes are decoupled through pins, which are denoted by small rectangles. The shown activity diagram describes the compute(...) operation of the class Server.

Interactions. Interactions focus on the modeling of scenarios of the message exchange between classifiers. Timing diagrams are specialized interactions that model the relationship between system state and time.

**EXAMPLE 2.5:** Fig. 2.9 presents a sequence diagram that describes the interaction between the classes of the running example. After Client created an instance of Server it asynchronously calls the compute(...) operation with some arguments x, y, and z. The server returns the results later on to the client. Finally, the client destroys its server instance.

Auxiliary Behavioral Constructs. There is a number of diagrams which complement the aforementioned constructs. They allow to focus on specific properties of system behavior. Use case diagrams specify the function a system provides to its environment and the respective message exchange. Information flows represent the exchange of information between system building blocks at a high level of abstraction.
2. Theoretical and Technological Foundations

![State Diagram Example](image1)

![Activity Diagram Example](image2)

Fig. 2.8: UML Behavioral Diagrams Example

![Sequence Diagram Example](image3)

Fig. 2.9: UML Sequence Diagram Example

2.3.2 System Design with UML

The UML is the lingua franca for the object-oriented modeling of software systems today. Support for UML-based software development is state-of-the-art and has been implemented in various approaches and tools, such as xUML [196], xtUML [194], Rhapsody [197], Rational ROSE [198], Artisan Real-time Studio [199], and many others. To enable the detailed definition of behavior, the first two approaches use a dedicated action language, while the other approaches allow for the integration of software programming language code into the models.

Various approaches for the UML-based hardware implementation have been presented. The mapping of relevant parts of UML models to hardware descriptions is done either manually or (semi-) automatically. Manual mapping requires the user to directly model the hardware implementation in some high-level language (→ Section 2.2.1). For this, the UML meta-model is extended by constructs that map one-to-one to respective language constructs. From such user models the respective hardware descriptions can be generated in a straightforward manner. Relevant examples include solutions based on SystemC [15, 200–205], VHDL [204], and SpecC [206].

Automated mapping approaches use tool-specific transformation rule sets to convert UML models to hardware descriptions. Common target languages are SystemC [13,207], Handel-C [8,9], and VHDL [208,209]. The current research in this field concentrates merely on the synthesis of state machines, and sequence diagrams. The synthesis of activities, actions, complete objects, and groups of objects is not well-researched however. Also component-based hardware development approaches are rarely discussed in literature.

The UML was originally developed as modeling language for software systems, the development of systems
comprising hardware and software components was not supported. As a response to the ongoing demand for a modeling language for such systems the systems engineering support of the latest version UML 2.0 was strengthened by various constructs, such as timing diagrams and information flows. Still UML is no complete system-level modeling language. Instead, in order to not confuse the core language, system engineering constructs are added by means of the recently proposed profile "Systems Modeling Language (SysML) Specification" [190]. Notable extensions of this profile are assemblies. These provide a mechanism for functional modeling. Parametrics are a means of integrating engineering analysis, e.g. of timing and reliability characteristics, into assembly models. Moreover, this profile extends the semantics of UML activities and strengthens requirements specification. Other research on the development of profiles for the analysis of embedded real-time systems can be found here [188, 189, 210, 211]. These profiles have been evolving in the embedded systems domain and focus on design analysis and architecture modeling. However, currently there is no profile that is sufficient for synthesis-oriented system development. A brief overview of the relevant profiles is given in Section A.3.2.

Currently there is no complete, consistent UML-based system-level design flow, and tool-chains for reconfigurable computer architectures. Recently, a number of research projects that go into this direction have been started in the system-on-chip domain. Due to the popularity of C/C++-based approaches in this field, the majority of solutions is based on variants of these languages. A particular implementation of the system is modeled using UML, whereas a range of abstractions from untimed functional modeling to RTL modeling is used. Automated system-level DSE is only rarely used. The generation of the C/C++ code from the model is mostly straightforward. The detailed synthesis is delegated to back-end tools which translate this code into RTL HDL descriptions, or directly into hardware implementations. Examples of such approaches include [12, 13, 15, 212, 213].
3. MODEL-DRIVEN ARCHITECTURE FOR RECONFIGURABLE COMPUTER ARCHITECTURES

3.1 System-Level Design with UML

3.1.1 UML as System-Level Design Language

In the previous chapter a number of system-level design approaches for run-time reconfigurable computer architectures, and, more general, for embedded computer systems has been examined. Despite of some effort in the direction of UML-based system-level design there is no consistent design flow, neither for RTR-architectures nor for embedded systems. In this chapter an approach is proposed, which has been developed for the former class of architectures, and that may be applied also in other domains. At first, the suitability of UML and required extensions for system-level design are examined. Then a respective development methodology is presented.

System-level design languages should support at least the following properties: expressiveness, modularity, synthesizability, and verification/validation support. As has been discussed in the previous chapter current approaches to system-level design utilize a wide range of computation models and languages with very different characteristics. Owing to their convenience and familiarity, developers commonly use software programming languages or hardware description languages. These languages have the advantage of being directly executable or simulatable. As a result, they support early verification, and validation, to some degree. Nevertheless, these languages lack system-level support, because

- the different aspects of a system, e.g. requirements, design, implementation, and deployment, are not sufficiently separated. For instance, the system function is captured in one of many possible implementations. There is no direct way to separate the design from particular implementations.
- important system aspects can not be represented adequately. For instance, often the specification of non-functional requirements, design constraints, and deployment information is captured in code, comments, and/or compiler directives. Concurrency and time cannot be expressed directly using native language constructs. Instead, accordingly extended derivatives, such as HardwareC, Handel C, or add-on libraries, must be used.
- the specification is inherently platform- and technology dependent. Examples are dependencies on specific hardware, operating systems, programming language semantics, physical data organization, and prerequisite libraries. Thus, the system analysis, estimation, verification, and construction is aggravated.

Consequently, these languages can only be an intermediate solution to system-level design. UML-based system-level design does not replace the existing approaches however. Instead, it builds atop of them and adds full system-level modeling, verification/validation, design space exploration, and synthesis capabilities.

Expressiveness and Modularity

It is a common observation in system design that the abstractions that are usable with a language should closely match the application requirements. If not, the result will be rather stilted designs. UML is mainly targeted toward object-oriented and component-based designs. The language provides constructs to model
function, structure, and behavior of such designs on different levels of abstraction. The expressiveness and modularization support for such systems is very high. This has been shown in the previous chapter and is confirmed by a recent study on languages and computation models for system-level design [214]. According to this study the object model is very suitable for the specification of data- and control-oriented applications. Remarkable are the support for concurrency specification, behavioral hierarchies, and integration of other computation models. The weaknesses in inherent verification support, which have been criticized by the study, have been relieved by UML.

Synthesizability

The synthesis support of UML depends on the abstractions employed by the designs. Synthesizing software implementations from object-oriented UML models is state-of-the-art today. Mostly high-level software programming languages are employed as target languages. Software synthesis is particularly simple if there are direct language mappings. This is the case for the common object-oriented languages such as Java and C++. Other languages become usable when unsupported features are emulated using native language constructs. For instance, in C polymorphic operations can be implemented by emulating C++-virtual method tables (VMTs), using structures and function pointers. Alternatively, critical features can be neglected or prohibited in the UML model.

Similar considerations apply for hardware synthesis. In approaches that directly model a particular hardware implementation synthesizability support is immediately given. The same applies for approaches that neglect the problematic object-oriented features and use behavioral-level languages. The respective language mappings are very similar to software synthesis but commonly require some tool-specific coding style. If the detailed synthesis of RTL hardware descriptions is delegated to back-end behavioral synthesis tools these approaches lack control and estimatability of the final implementation.

The full synthesis of object-oriented specifications into hardware and software, the synthesis of communication interfaces, and component-based synthesis are not well addressed by the current research however.

Verification and Validation

Formal verification is supported by the strict definition of the UML semantics. The high level of abstraction and hierarchical models support a wide range of verification methods and make them more tractable. The reason for this is the reduced number of elements to be considered on each hierarchical level. Constraints and assertions can be included directly into the models using a suitable constraint language. UML uses the OCL for this purpose [184]. For instance, the OCL is a very efficient means of defining preconditions and post-conditions of operations.

UML enables the presentation of different vertical and horizontal abstractions in the same model. The relations between the elements in different models can be captured explicitly or they are implicit through the model structure. For instance, the realization of use-cases is traceable through interactions and the class hierarchy down to the action level. Traceability is also given for structural features such as classes and components and their refinements. Such redundancy can be exploited to perform validations on the feasibility of models.

The most common approach to validate UML models is model execution. Generally, a model is considered correct when it fulfills the requirements. A number of test-cases is formulated for the requirements that, if model execution is applied, must be correct in order for the model to be right. Owing to the high level of abstraction model execution is fast even for complex models. Advanced modeling environments allow for integrated model execution. Other approaches synthesize the model into some software implementation and execute the software. Lower level refinements can use the whole range of available techniques for analysis, test, and debugging. However, these approaches rely on the correctness of the utilized interpretation system.

3.1.2 MOCCA Action Language

Actions are the fundamental unit of behavior. The specification defines a set of actions and their semantics but it does not define a syntax. Rather than having an predefined and probably over-designed language that
tries to fit all applications equally well users are free to define an action language that is most suitable to their particular application domain. Users select the actions and activities that are suitable to their domain and do not have to implement the full specification. The UML action semantics are a premise for building executable and interchangeable UML-models. An action language should allow for full model access, abstract from physical data organization, and not overemphasize sequential execution [192]. Additionally, action languages for system-level design must support the requirements defined in the previous section. Therefore, the language should facilitate analysis and optimization of control and data flow, and estimation of non-functional system characteristics.

Existing action languages, like ASL, AL, and SMALL [191, 192, 194], have been designed for software applications in business, telecommunication, and network systems with mixed control/dataflow. They have a high level of abstraction in that they do not make any assumptions about data organization. Although this provides some potential for performance optimization and broadens the available range of implementation opportunities it complicates estimation. Moreover, the support of arithmetic and logic operations provided by these languages is insufficient for the targeted application domains. Thus, the MOCCA Action Language (MAL) was designed.

This language is compliant to the UML action semantics. In comparison to the named action languages it has a medium level of abstraction because it requires the developer to make data organization and data access explicit. However, this enables the employment of standard analysis, optimization, and estimation techniques. Nevertheless, the development of higher level action languages for system-level design is considered a major research issue for the future.

MAL allows for the specification of sequential logic, arithmetical/logical/relational operations, instance manipulation, and class access. To make it easy to use and to reduce the learning effort, the syntax and semantics of the language orients toward the statement and expressions of the Java programming language [215]. The language does not overlap with concepts that already exist in UML. The action language supports only the constructs for control- and data flow specification. All constructs related to design modularization, such as packages, classes and interfaces, have been removed, because they are redundant to UML. The extensions and restrictions are summarized in Appendix A.1.1.

In this thesis MAL is used to define the behavior of operations. Each operation is associated with an opaque behavior whose body attribute is defined using this action language (→ Fig. 2.7(a) on page 20). As has been discussed in Section 2.3.1, this approach can be combined with other behavior specifications. Since actions are the fundamental functional units of UML heterogeneous specification models can benefit from the presented work as well.

Statements and Expressions adopted from Java

The MAL syntax and semantics of statements and expressions is similar to Java. That is, the common statements for blocks, conditional execution, loops and exceptions are directly applicable. The same applies for the expressions, with two exceptions. In MAL the conditional operator (?:) is not available, because this operator is a common source of errors and restricts understandability. It can always be emulated by using the if-statement. Moreover, MAL does not adopt Java’s synchronized-statement, because it is redundant to UML’s guarded operations.

Additional Operators and Statements

In MAL there is no inherent distinction between primitive and complex classifiers. Designers can define arbitrary operators on all classifiers. These operators are accessible in MAL using the same syntax as for method invocations. In addition, MAL merely defines the syntax of the built-in operators, but does not restrict the types on which operators can be applied. Designers can freely redefine the built-in operators to be used with any type. This allows to extend the available set of operators without having to change the action language.

In addition to the Java language specification [215], MAL defines a number of operators with a predefined syntax and semantics. Some of them, e.g. for the access of associations and model navigation, are experimental or not fully supported by the current version of the MOCCA compiler. The definition of these
operators can be found on the MOCCA co-development platform [216]. They will not be used in the course of this thesis. The additional operators being used in this thesis are:

countof – The countof-operator determines the current length of an array. The operand is an array reference. It returns the current length of the first dimension of an array.

**Syntax:**

\[ \text{CountofExpression} = 'countof' \text{ArrayReference} \]

**Priority:** same as operator `new`

**Objective:** Java provides similar functionality by appropriate methods or attributes. Because this is not a generic and portable mechanism in MAL a dedicated operator is used.

destroy – The destroy-operator is used to delete instances and arrays. The operand is a reference to the instance or array. The operator has no return value. The operator works nested; if an array is destroyed all referenced elements are destroyed.

**Syntax:**

\[ \text{DestroyExpression} = 'destroy' (\text{InstanceReference} | \text{ArrayReference}) \]

**Objective:** UML models cannot generally rely on the existence of garbage collection. Unused instances and arrays must be explicitly freed.

async – The async-operator marks an operation call to be asynchronous. The operand defines the receiver and the operation to be called as well as a number of optional arguments. The operator has no return value.

**Syntax:**

\[ \text{AsyncExpression} = 'async' \text{MethodInvocation} \]

**Objective:** The purpose of this statement is to overcome the restriction of the Java execution model to synchronous method invocations in order to be able to better exploit multi-processor hardware.

**Example 3.1:** Fig. 3.1 illustrates the additional operators. The example creates instance `someInst` of class `SomeClass` and an integer array `someArray`. In a loop all elements of the array are read and the number of zero bits of each value is computed. The data type `int` is specified to have a native operation `countZero()`. The number of zero bits is passed asynchronously to `someInst`. At the end `someInst` and `someArray` are destroyed.

**Listing 3.1:** MAL Additional Operators and Statements Example

```java
1 SomeClass someInst = new SomeClass();
2 int[] someArray = new int[100];
3 // fill values into array
4 for(int i=0; i<countof someArray; i++) {
5     async someInst.process(someArray[i].countZero());
6 }
7 destroy someArray;
8 destroy someInst;
```

1 Syntax definitions are given in a standardized version of extended Bachus-Naur form [217] (→ Section A.4.1).
MAL is an action language that provides a syntax for the most of the UML actions. The mapping between this language and UML is straightforward. This mapping enables reasoning about model behavior entirely in terms of UML. The action language defines just the surface representation of actions. Example 3.1 already gave an example. The employed mapping rules are presented in Appendix A.1.

The control-flow statements of MAL are mapped to UML activity nodes, such as LoopNode, ConditionalNode, and Clause, for which the UML does not define a notation. The reason for this is that UML expects these elements being presented by an action language. They are not used in activity diagrams since the semantic of these elements does not fully align with the token based execution model. Object diagrams are the only means of presenting these elements graphically in a UML compliant manner. Whenever these elements are presented graphically an object diagram is used. Otherwise activity diagrams are used to present activities.

**Example 3.2:** As a first example Fig. 3.1 illustrates the mapping of four different MAL operators to UML actions. Since for the CreateObjectAction and DestroyObjectAction no notation exists an object diagram is used for representation. The mapping of the unary operators is straightforward. To preserve the semantics of non-commutative binary operators, such as "%" (mod) in the right part of the figure, two designated input pins named left and right are defined. The left (right) operand is always mapped to the input pin named left (right). If commutativity matters the names are shown in the diagrams, otherwise they are frequently omitted in order to avoid cluttering the diagrams.

**Fig. 3.1:** Mapping of MAL Operators

**Example 3.3:** Fig. 3.4 illustrates the mapping of the for-loop in Listing 3.1 to UML actions and activities. The loop management is mapped to a LoopNode. The operators (++, <, countof, {}) are mapped to instances of OpaqueAction.

Clearly, this representation hampers readability and understandability. The major problem of the full notation is the large number of pin and flow objects. To improve readability, a compacted notation is proposed, which is less intrusive as the one presented in [218]. Input pins and output pins are shown as small circles, which are filled white or black respectively. To distinguish control-flow edges from data-flow edges, control-flow edges are stereotyped ControlFlow. If ambiguities can occur, data-flow edges are stereotyped ObjectFlow. These rules are illustrated in Fig.3.3. Accordingly, Fig. 3.4 shows the object diagram of Fig. 3.2 in compacted notation.

This representation can fully reflect the control- and data flow MAL specifications. The format combines features of abstract syntax-trees, control/data-flow graphs and hierarchical task graphs. It is intended to provide a common foundation for detailed behavior specification using UML. It may also serve as language independent exchange format for such specifications. Because no existing action language and tool supports the action semantics completely, the exchangeability of such specifications is likely to be restricted to particular domains and tool chains.
Fig. 3.2: Mapping of Loop in Example 3.1 to Actions and Activities

Fig. 3.3: Object Diagram Compaction Rules
3.2 Model-Driven Development Methodology

3.2.1 Co-Design, Platform-based Design and Model-Driven Architecture

**Hardware/Software Co-Design**

Hardware/software co-design is the traditional paradigm to the tool-based design of hardware/software systems. This discipline evolved in response to the increasing complexity of application development for powerful microprocessor systems in the mid-1980s. Co-design transforms system specifications into implementations that realize the specified function and meet all constraints. Whilst early approaches to co-design put emphasis on the problem of coordinating hardware and software development, later approaches moved toward higher degrees of integration and automation. The goal is to perform partitioning and refinement as late as possible in the design flow. Thereby the risk of design iterations, owing to unsuitable partitions missing the system constraints, is reduced.

Traditional co-design creates an individual architecture/micro-architecture for each problem. As a result, an optimum implementation is developed individually for each problem. This is beneficial for high volume productions with negligible development cost. Problem-specific solutions pose severe challenges on estimation, verification, and synthesis however. Owing to the reduced reusability, portability, and flexibility, the non-recurring engineering cost and manufacturing cost increase. Low-volume productions and systems utilizing fixed architectures for different problems are not supported well by traditional co-design. Reconfigurable architectures introduced a higher degree of system programmability. Although this helps in relieving these problems the majority of challenges remains.

**Platform-Based Design**

To overcome the problems inherent to traditional co-design, in the recent past a novel paradigm for system design evolved. The core of this paradigm are reusable architecture platforms, which can be used to implement a broad range of systems [72, 219–221].

**Definition 3.1:** A Platform is an architecture comprising fixed sets of components that define coherent sets of functionality and non-functional characteristics of its services. The components may be variable
to provide some degree of flexibility for lower level refinements and hide their micro-architecture from the systems that depend on the provided services.

**Definition 3.2:** A Platform Instance is a platform that fixes all variability of its components to specific settings.

Platforms are a means of standardization and facilitate design reuse, portability, and flexibility. Pre-characterization of architectural components with their implementation characteristics supports high-level estimation and helps to avoid design iterations. Formal definitions of platforms and applications support early verification and synthesis. Architecture platforms may be abstract, such as software platforms, operating systems and libraries, or physical, such as specific combinations of processing elements, storage, and peripherals. The core methodology of PBD is the Y-chart approach to system design [5], which is illustrated in Fig. 3.5. To maximize the implementation opportunities, the system function and the architecture platform are isolated. The implementation of the function with the architecture platform is established in a dedicated mapping step. Mapping is done by relating system behavior and structure to appropriate architectural elements. Transformations are required to explore design alternatives and to optimize results. The best design, with respect to some cost function, is chosen for synthesis. The synthesized result may be analyzed and refined by subsequent flows. During the traversal of the design flows the level of abstraction of both, the function specification and the architectures, is steadily lowered, and an increasing number of implementation parameters is fixed. This approach is an instance of the principal system-level design flow (→ Section 2.1.2).

The core of PBD is the orthogonalization of system function and implementation architecture, DSE, synthesis, and analysis. PBD is basically a specialization of hardware/software co-design that specifically addresses the challenges of cost-effective development in the today’s economical environments. A tremendous increase in design productivity and quality of hardware and hardware/software systems is necessary in the future. The key stones of productivity improvement are reuse and higher abstraction levels. PBD puts strong emphasis on component-based design since components are an efficient means of reuse. Pre-verified and pre-tested components ease the system verification and test and help improving their tractability. Raising the level of abstraction from levels that are too close to particular implementations, in order to reveal the full range of implementation opportunities for a function, is emphasized. In contrast, current system-level design lacks the low abstraction at register-transfer level. Higher levels of abstraction will require vast progress in methodologies, DSE, synthesis, and automation. Early modeling is another important issue in system-level design. Current RTL design approaches use modeling very late in the design flow, e.g. for creating behavioral models of hardware parts. In the future, modeling must move into the center of the design flow for modeling system function [222, 223].

**Model-Driven Architecture**

At the same time as platform-based design evolved in the embedded systems domain in the software domain the paradigm of MDA has been established [4]. MDA is a development paradigm that features the model-based development of software systems. The principal goal and approach of MDA is similar to PBD. As embedded systems, the software development of telecommunication systems, web-services and data base management systems faces highly fragmented technological environments. Moreover, MDA also aims at providing design reuse, portability, and flexibility through the strict separation of application logic from the platforms utilized for its implementation and execution. Although there is major research and development effort in this direction up to now there is no concise definition of MDA. Instead, there is just some agreement on the processed models and steps.
The MDA approach builds upon a set of UML profiles (→ Section 2.3.1), called core models. Core models define a particular type of platform that is independent from specific platform instances. A platform is indirectly linked into the application by using the elements of the respective profile. The link is substantiated into the final implementation by applying a series of transformations. The most important transformation types are model-to-model and model-to-implementation mappings. Applications and core-models are captured using UML. The application models access services provided by the utilized platform in terms of the appropriate core models. UML models extend the semantics of UML by domain specific semantics. As a result, the application models are platform-independent models (PIMs) with respect to a specific core model instance. PIMs are successively transformed into platform-specific models (PSMs). PSMs are UML models that link the application into particular platforms. Finally, PSMs are transformed into platform-specific implementations (PSIs). Owing to the focus on software and the quite direct link between design and implementation, MDA has no notion of DSE and does not define metrics and estimates for the selection of platform instances.

The core of MDA is the orthogonalization of system function and implementation architecture, the principal models and relationships, and the common model representation using UML. The level of abstraction is raised, to enable the development of applications that are widely independent of a particular technological basis. UML models are put into the center of development. Their content and semantics is not defined by the MDA specification. MDA promotes modeling and object/component-based design using UML.

3.2.2 Model-Driven, Platform-Based System-Level Design

Incorporating Platform-Based Design and Model-Driven Architecture

Both, platform-based design and model-driven architecture, share a common set of goals and principal approaches though they have evolved in different domains. In some sense both approaches are complementary to each other, which suggests to incorporate them into a common framework. As illustrated by Fig. 3.6, PBD can be implemented as specialization of MDA. The system design and architecture platforms can be captured using UML models. Initially, the requirements are translated into a PIM, which is based on a design platform. The PIM is then refined into a PSM and a PSI with respect to a target platform. Each of these artifacts selects a particular point in the platform architecture space. Mapping is the fundamental transformation applied to the application-specific models, whereas additional transformations are conceivable. The grayed refinement pyramid illustrates how different sets of requirements translate to different designs and implementations.

Fig. 3.7 formalizes this relationship as meta-model for design methodologies. The advantages of incorporating PBD and MDA are appealing. Modeling technology is put into the center of development. UMLs hierarchical modeling capability supports design productivity and quality. At the same time system verification and test become more tractable in the reality of complex designs.

Standardized representations are key to higher levels of abstraction. In comparison to register-transfer level, the current behavioral-level abstraction offers large gains in design productivity and quality. To cope with the complexity of future systems, raising the specification to the system-level is a premise. Ideally, systems can be modeled using a single specification and standardized representation. Model compilers automatically transform the specification into final implementations, whereas the implementation is parameterized by respective platform models.
To enable such environments the development of appropriate model-to-model mappings, model-to-implementation mappings, and respective tool support is required. The gap between the models and back-end design flows is bridged using language mappings. For instance, hardware designs may be generated from a UML model via UML-to-VHDL or UML-to-SystemC mappings. UML-to-C++ mappings can be used for software respectively. Transformations between different models of computation can also be achieved with appropriate language mappings.

In the longer term this approach may lead to the unification of hardware and software development. For this a convergence in the applied methods and tools is a prerequisite. This will not implicate, however, the replacement of existing languages, models of computation, and tool infrastructure. They will be used as entry into specialized design flows in the back-end of UML-based system-level design. By means of the standardized representation different tools, e.g. for modeling, verification, and synthesis, can be coupled more tightly in order to further improve productivity and quality.

The presented relationship between PBD and MDA is not generally applicable however. MDA obliges on using UML and object/component-based design and particular model types to some extent. Not all specializations of platform-based design may benefit from such constraints today. As recent publications indicate, system-on-chip as the major driver of system-level design can benefit from using UML and a model-based approaches, e.g. [12,13,196,205,212,213]. Arguably, as mature methodologies, core-models, and mappings become available for specialized domains (e.g. mixed analog/digital, mechanical components) the number of application domains will gradually increase. How this is accomplished for object-oriented applications of run-time reconfigurable architectures is discussed in the remainder of this thesis.

**General Development Activities and Artifacts**

Fig. 3.8 illustrates the model-driven methodology for object-oriented applications of reconfigurable architectures in detail. This methodology is a specialization of the proposed incorporation of PBD and MDA in that it specifically defines the development activities and artifacts. Other approaches may specialize this framework differently according to their specific requirements. In this section focus is on the discussion of the development activities. In order to give the reader a better understanding the content of the employed artifacts is briefly outlined. Section 3.3 provides a thorough discussion of the artifacts.

The development of applications is based on platforms, whereas different platforms are used for design, implementation, and deployment. Thereby a strict separation of development concerns is accomplished. Moreover, this approach eases validation, portability, adaptability, and reuse. Platforms represent sets of assumptions, which are the foundation of any development effort. Present design efforts mostly capture platforms implicitly in language reference manuals, libraries, and tools. Implicit platforms limit the adaption to changing requirements and hamper the automated interpretation. In contrast, the presented approach
makes platforms explicit by using platform models, whereas each platform is specified by a dedicated platform model. Platform models abstract from the details of the platform described but carry enough information to avoid iterations in the design flow. They are a direct consequence of the incorporation of MDA and PBD in Fig. 3.7.

**Specification.** Purpose of the specification phase is the creation of a design that realizes the system function. Input to this phase is the requirements specification and a design-platform model (DPM). The requirements specification is a formal or informal statement on the functional and non-functional requirements of the system. It is first translated into an use-case model. For each use-case a number of test-cases is defined as basis of subsequent functional tests. As the specification proceeds the use-case model is elaborated in detail and a design model is constructed. The design model represents an executable and implementation independent realization of the specified use-cases. Elaboration and construction are finished when the design model successfully executes all test-cases. Each design model is based on a design platform. A design platform is the fundamental set of types and constraints being used for system design. During elaboration further models, such as analysis models and test models, may be created. Such models aim at improving human understanding and management. Their application and content is highly dependent on the application domain, project, and organization. These models do not directly effect the implementation though they may help in creating the design model. Consequently, the presented approach does not constraint the specification in such detail.

**Platform Mapping.** Given a platform-independent model of the system implementation proceeds by transforming this model into a platform-specific model. The PSM is an UML model that defines a mapping of the design to the target platform. Consequently, this model fixes all free implementation parameters of the relevant elements of the design. It is created from the design model either manually or (semi-) automatically. The PSM defines an implementation model and a deployment model for the design model. These models define a realization of the design model using the services provided by the target platform. The target platform reflects the basic assumptions and restrictions when implementing and deploying systems. This platform is captured in a target-platform model (TPM). The transformation of the PIM into the PSM may be direct or it may require the creation of intermediate models. The approach being taken mainly depends on the employed transformations. The core of this activity is design space exploration including partitioning and system-level estimation. An approach that specifically addresses object-oriented specifications is presented in Chapter 4.

**Synthesis.** Given a platform specific model of a system, implementation proceeds by implementing it into a platform-dependent application. Central activity is the synthesis of the hardware and software modules and communication interfaces. The synthesis may be performed manually or (semi-) automatically. Synthesis is based on the PSM and the TPM. Components being deployed on nodes representing microprocessors are
implemented as software modules. Hardware modules are synthesized for components being deployed on nodes representing reconfigurable fabrics. The implementation of hardware- and software-modules can be described using different languages. Each particular commitment to a set of languages must be reflected in the target-platform model. Whilst the synthesis of software from object-oriented models is state-of-the-art today, the synthesis of respective hardware and automated synthesis of communication interfaces has not been addressed yet sufficiently. In Chapter 5 respective approaches are presented.

**Verification and Validation.** In the presented approach quality checks can be performed on all models in the refinement hierarchy. In contrast to traditional simulation, the automated synthesis capability enables the direct execution of the applications. Functional simulation can be performed by synthesizing a software implementation whose adherence to the defined test-cases is tested. Subsequently system function can be gradually migrated and tested in hardware.

**Tool Support and Automation**

The approach is backed by a novel type of model compiler called MOCCA. Given complete and precise models of the application design and the employed platforms this compiler can automatically perform validation, platform mapping, optimization, and synthesis. The synthesized implementation is executable and exploits the capabilities of the hardware architecture. The algorithms presented in this thesis have been implemented in MOCCA. The current version of the compiler accepts its input in XMI format. Designers can use any UML modeling tool that supports this format and MOCCA’s UML subset for design entry. The compiler provides an extension interface to enable the adaption to tools that do not support this format. MOCCA is presented in Chapter 7.1.

### 3.3 Platforms and Models

#### 3.3.1 Use-Case Model

**Definition 3.3:** An *use-case model* specifies the principal system function and the interaction of the system with one or more actors, being located in the system environment, to access this function.

Use-case models are the main tool of developers for capturing the system requirements. Analysts utilize such models to get a better understanding of the system. These models are the basis for the construction and test of design models and their implementations. Use-case models represent the system function at the highest possible level of abstraction. As such, they do not commit to any particular design or implementation. They demarcate the scope of the system and its environment. The environment comprises the principal actors who are going to work with the system [6, 194, 224–226].

#### 3.3.2 Design Platform Model

**Definition 3.4:** A *design platform* is an architecture platform that comprises a set of basic data types, their relationships, and constraints, that build the foundation for the construction of designs.

**Definition 3.5:** A *design platform model* is an UML model that represents a design platform.

DPMs are the basis for constructing design models for applications in a particular domain. A design-platform model must be semantically autonomous of particular designs and implementations and should not define or use such designs. Moreover, this model must not be used as a replacement for domain-specific models. The relationship of design models and design-platform models is formalized in Fig. 3.9.
3.3. Platforms and Models

Fig. 3.9: Platform-Independent Model Meta-Model (continues Fig. 3.7)

Each design-platform model represents a set of design data types.

**Definition 3.6:** A design data type is a data type that is used to construct design models.

Design data types fall into core data types and auxiliary data types. The core data types are specific to the particular domain and provide the basis for the definition of other data types. Examples of core data types are integer-, boolean-, or real-types. The definition of the core data types is mandatory, because the types defined by the UML specification are defined too loosely in order to be useful in real-world designs. Auxiliary data types are not as fundamental as core data types but they are used by virtually all applications within the domain. Examples include data types for input/output, access to actors/sensors, and system control. For each type the relationship to other types in terms of generalizations and dependencies the supported operations, and constraints are defined.

In contrast to the present approaches, DPMs make all classifiers and their services explicit. This has the advantage that relationships and constraints can be adapted to the domain to some extent. Moreover, all types may be extended by arbitrary operations. Consequently, the designer is not restricted to a fixed set of predefined operations. Instead, she can extend all types as it is useful for design understandability and construction. This approach emphasizes human-friendliness rather than machine-friendliness. An example has already been given in Example 3.1 on page 28, using the `int`-type that has been extended by the `countZero()`-operation.

**Example 3.4:** Fig. 3.10 shows a small part of a DPM. The example illustrates some types which may be used in design models. For the `boolean`-type the set of available operations is illustrated. The operations represent the core operations of this data type (→ Example 3.5). Constraints are exemplified using the `int`-type, for which the domain and its distance to other types is shown. The types are organized in an inheritance hierarchy. Multiple inheritance is supported for interfaces, which enables the simulation of multiple inheritance of classes. The illustrated DPM is captured using the “Design-Platform Profile”. This profile is described in Appendix A.5.1. The design platform model used for the examples in this thesis is described in Section B.1.

**Language- and Compiler-Specific Core Data Types and Core Operations**

Although the DPM is independent of a particular action language and model compiler, both of them may require the design-platform model to define a specific set of core data types. Model compilers require particular core data types to accomplish optimization, platform mapping, and synthesis. Such classifiers are part of the definition of all present action languages.
For each core data type its domain and the set of operations must be defined. In principle, arbitrary operations can be defined for a type. However, there is a minimum set of operations that the type is expected to offer. This set of operations is called the core operations of the data type. In contrast to all other operations, the core operations are associated with a predefined semantic. This semantic is exploited in order to perform model transformations such as arithmetic optimizations. If a data type does not define the expected core operations this can render a model invalid. The reason for this is that model compilers test if for each feasible action being used in a model an appropriate operation exists in the data type of the object on which the action is invoked. If not so, the model is considered invalid. Moreover, the missing of core operations can impede the applicability of transformations that are defined with the missing operation.

**Example 3.5:** The design-platform model in Fig. 3.10 shows the core operations for the boolean-type. This set includes logical operations (cond_and, cond_or, xor, not), tests of (un-) equality (eq, neq), a cast to an instance of data type bit (cast), and the assignment of another instance of type boolean to the current instance (asgn). MOCCA’s set of core data types and core operations is described in Appendix A.2.1. Notice, that cond_and and cond_or are the operation names for the conditional AND and OR, while and and or denote the operation names for the bitwise AND and OR respectively.

**Mapping Actions to Core Operations**

By Definition 2.7, the only inherent capability of each object is to send and receive messages. The set of messages an object can respond to is defined exclusively by the classifier specification of the object. As discussed in Section 3.1.2, action language specifications are mapped to UML actions. Straightforwardly, in a true object-oriented approach, only such actions can be executed on an object for which a respective service exists. On the other hand, UML defines actions separately from objects rather than as object services; objects are flowing into and out of actions. Hence, actions must be mapped to operations.

**Example 3.6:** Listing 3.2 and Fig. 3.11 illustrate this concept. The listing shows statements that declare two boolean variables a and b. Variable c represents an array whose elements are also of boolean-type. The figure shows the representation of the statement in line 4 using actions, and the mapping of the actions to respective operations in the data types. The read access to the array ([ ]) maps to a get action (→ Tab. A.6), that reads the value that is stored at index i in array c.
Listing 3.2: MAL Statement Example

```java
boolean a, b; int i;
boolean c = new boolean[100];
...
a = c[ i ] && b;
```

![Mapping of Actions in Listing 3.2 to Core Operations](image)

This problem is solved in two steps:

1. unambiguously associate the action with an unique object,
2. unambiguously determine the most specific operation in the classifier of the associated object.

**Associating the action with an object.** Actions are associated with an object that is carried by an incoming or outgoing object flow. Notice, that apart from InvocationActions, in the presented approach, actions have at most two incoming object flows and exactly one outgoing object flow. So, actions are associated according to the following rules, which apply in the order of definition:

1. each action that has no incoming object flow or that writes to an instance of ObjectNode is associated with the object that is represented by this object node.
2. each action that has a single incoming object flow is associated with the object that is carried by this object flow.
3. each action that has a two incoming object flows is associated with the object that is carried by the incoming object flow named left (Example 3.2 on page 29).

These rules resemble the associativity of the MAL operators. All actions for object creation, destruction, writing, test, and computation are mapped to respective core operations. Reading actions on a scalar variable are directly associated with the output pin that carries the variable. To read elements of ordered non-scalars, i.e. lists, the variable carried by the output pin depends on a selection operation which is controlled by an index variable. The same considerations apply for writing actions on elements of non-scalar variables. To make this selection explicit, all non-scalar types must provide a get-operation to read elements, and a put-operation to write elements. Invocation and reply actions are not required to be mapped since they realize the object inherent capability of message exchange (Definition 2.7).

**Searching the most specific operation.** The operation that realizes a particular action is selected by the declaring classifier of the object, the operation signature, and the list of arguments. The mapping of action names to operation names is defined in Tables A.6-A.12. The argument list comprises all object input pins in the order of their definition, except the input pin carrying the object with which the action is associated, if applicable. The most specific operation is searched in the classifier of this object. Thereby inheritance and polymorphism can be applied. The selection algorithm for most specific operation is presented in Algorithm 3.1. The algorithm uses similar rules for operation resolution as Java [215].
Algorithm 3.1: Find the most specific operation - \texttt{findMSO(classifier, name, ArgList)}

\textbf{Input}: Classifier for which to search the operation: \texttt{classifier}. Name of the searched operation: \texttt{name}. Argument list: \texttt{ArgList}.

\textbf{Output}: Operation \texttt{msop} defined by \texttt{classifier} and most specifically matching \texttt{name} and \texttt{ArgList}.

\texttt{msop} \leftarrow \emptyset, \texttt{distance} \leftarrow \infty;

\texttt{while classifier} \neq \emptyset \land (\texttt{msop} = \emptyset \lor \texttt{distance} > 0) \texttt{do}

\texttt{OpSet} \leftarrow \text{Operations defined by classifier, named name and having |ArgList| parameters;}

\texttt{foreach op} \in \texttt{OpSet} \texttt{do}

\texttt{ParamList} \leftarrow \text{List of parameters of op;}

\texttt{ldistance} \leftarrow 0;

\texttt{for} \texttt{i} = 1 \texttt{to} |\texttt{ArgList}| \texttt{do}

\texttt{carg} \leftarrow \text{Classifier of argument at position i in ArgList;}

\texttt{cparam} \leftarrow \text{Classifier of parameter at position i in ParamList;}

\texttt{ldistance} \leftarrow \texttt{ldistance} + \texttt{distance}(\texttt{carg}, \texttt{cparam});

\texttt{if} \texttt{ldistance} = \texttt{distance} \texttt{then}

\texttt{error because op ambiguates other operation}

\texttt{else if} \texttt{ldistance} < \texttt{distance} \texttt{then}

\texttt{msop} \leftarrow \texttt{op}, \texttt{distance} \leftarrow \texttt{ldistance;}

\texttt{classifier} \leftarrow \text{next generalizing classifier;}

\texttt{return \texttt{msop;}}

At the heart of the algorithm is the computation of the combined distance of the argument classifiers to the formal parameter classifiers. The distance is a relation on classifiers. It represents the minimum number of generalization relationships between two classifiers, whereas only classifiers that are in sub-type relationship have a finite distance. The distance between two classifiers \texttt{c}_0 \text{ and } \texttt{c}_1 \text{ is:}

\[
\text{Distance} : C \times C \rightarrow \mathbb{N}
\]

\[
\text{Distance}(\texttt{c}_0, \texttt{c}_1) = \begin{cases} 
0 & \text{if } \texttt{c}_0 = \texttt{c}_1, \\
\min(\text{length of generalization path from } \texttt{c}_0 \text{ to } \texttt{c}_1) & \text{if } \texttt{c}_0 \text{ specializes } \texttt{c}_1 \\
-\infty & \text{else}
\end{cases}
\]

The computation of the most specific operation considers the type hierarchy. For the list of argument classifiers the operation with the best corresponding list of parameter classifiers is searched, whereas the correspondence of each argument is searched in the generalization hierarchy of the argument classifier. This strategy can cause a loss of information however, if the generalizations cannot fully represent the domains of their specializations, as it is the case for hierarchies of primitive data types (→ Fig. 3.10). Moreover, in some cases type reclassification might be desired. To avoid loss of information, sometimes one would like to reclassify objects in a way that is not reflected in the classifier hierarchy. For instance, in the example hierarchy the best approximation of the \texttt{boolean}-type is \texttt{bit} rather than \texttt{object}, although both data types are not related to each other in terms of generalization\textsuperscript{2}.

To handle such situations, designers can model the distance between classifiers using the Distance-Vector constraint. A distance vector is specified for individual classifiers and defines the distance of the classifier to a list of other classifiers\textsuperscript{3}. If a distance vector is specified for a classifier it overrides the default distance computation. Distance vectors can be used to link classifiers in ways that are not reflected in the classifier hierarchy. These vectors are not required to be exhaustive. The distance to a type that is not listed is computed as the sum of distances between individual classifiers, whereas the default computation is used whenever no distance vector is given. The syntax of distance vectors and all other specifications in this thesis that employ constraints and tag values is defined in Section A.4.

\textit{Example 3.7: } Fig. 3.10 exemplifies the use of distance vectors with the \texttt{int}-type. The vector states that

\textsuperscript{2} A reversed hierarchy would be semantically wrong, because specializations would be restrictive in the supported messages. This would violate the open-closed principle and Liskov Substitution Principle [227,228].

\textsuperscript{3} Notice, that this definition differs from the distance vectors used in compilers to capture loop dependencies [229].
the distance of int to itself is zero. Using the standard distance the distance of int to float and double would be infinite. In contrast, the given vector assigns finite distance values.

3.3.3 Design Model

**Definition 3.7:** A design model defines an executable and implementation-independent realization of the use-cases of a system.

The relationship of design models and use-case models is formalized in Fig. 3.9. The design model defines the structure and behavior of the system. System structure is defined using packages, classes and interfaces and their various relationships. Behavior is defined using operations and structured activities, whereas detailed behavior is defined with actions at the intermediate level with regard to the UML specification. The MOCCA Action Language is used as action language. State machines are not supported by MOCCA although it can be extended accordingly. For the specification of concurrent behavior active classes are supported. Synchronization of concurrent control flow is accomplished using guarded operations. The model of computation is objects communicating through structured messages.

In order to facilitate design reuse and portability, each design model should be partitioned into multiple, domain-specific sub-models. Domain-specific models are a tool for the orthogonalization of a system into the subject matters it involves, e.g. networking, user-interface, and data-processing. The design model for each domain should be functionally cohesive and semantically autonomous of other domains [194].

Design models must be closed under the set of classifiers. That is, all classifiers that are used in the design must be defined either in the design model or in the DPM. Consequently, the domain of classifiers depends only on the information provided in these models. This property is used by model compilers in order to perform design validation and optimization. The validity of a design model is determined entirely by the design platform model and the constraints imposed by the UML specification. To use a specific design platform, a design model must import the respective design-platform model. Though, according to MDA, the design model is platform-independent, this independence refers only to the target platform. Each design model depends on a design platform however.

**Auxiliary Design and Optimization Extensions**

Design model elements can carry additional information that is used to support optimization and implementation of the design:

*Optimization-Constraints/Properties* - Optimization-constraints and properties give designers a fine-grained control of the optimizations performed by model compilers. In contrast to traditional approaches, that allowed designers to enable/disable optimizations only globally, in the presented approach such constraints can be defined for each relevant element in the design model.

**Example 3.8:** Listing 3.3 shows examples for the specification of optimization-constraints as implemented in MOCCA. Optimizations can be defined per project, or they may be associated individually to the regarded model elements. The scope of optimization-constraints is the model element and all model element below with respect to the containment hierarchy. For instance, if the constraints in the example are assigned to an operation then they apply for the operation and all elements below, if they are not overridden. If they are assigned to a class the scope is the class and all features of the class.

**Listing 3.3:** MOCCA Optimization-Constraint Specification Example

```plaintext
mocca.optimization.local_loop_unrolling:=yes
mocca.optimization.local_loop_unrolling_level:=10
mocca.optimization.arithmetic_transformations:=yes
```

4 For traditional reasons, the author continues to use the term “platform-independent” and is aware that this specifically refers to the target platform.
**Execution-Constraints/Properties** - Execution-constraints and properties are used to define execution characteristics of model elements. These characteristics only depend on the design and its input data (→ Section 4.4.1). They are frequently used to control the implementation. Like optimization-constraints, execution-constraints can be defined for each relevant model element in the design model.

**EXAMPLE 3.9:** Listing 3.4 illustrates the specification of important execution-constraints. The execution probability and frequency constraint is associated with behavioral model elements, such as activities, activity groups, and actions. The last two constraints define the maximum number of elements of an array and the maximum number of concurrent instances of a classifier.

**Listing 3.4: MOCCA Execution-Constraint Specification Example**

1. EstimatedFrequency := 250 // associated with behavioral elements
2. EstimatedProbability := 0.025 // ditto
3. EstimatedMaxArrayElements := 1152 // associated with arrays
4. EstimatedMaxConcInstances := 32 // associated with classifiers

### 3.3.4 Implementation Platform Model

**DEFINITION 3.8:** An implementation platform is an architecture platform that comprises a set of types, their relationships, constraints, and mappings, that build the foundation for the implementation of designs.

**DEFINITION 3.9:** An implementation platform model is an UML model that represents an implementation platform.

Implementation-platform models are the foundation for the construction of implementation models for design models. The model is, however, semantically autonomous of particular design- and implementation models and must not define implementations of particular designs. An implementation-platform model defines the realization of a design-platform model. As a result, each implementation-platform depends on the application domain to some extent. Fig. 3.12 defines the models and relationships formally.

![Implementation Models Meta-Model](image)

Fig. 3.12: Implementation Models Meta-Model (continues Fig. 3.9)

Examples of implementation-platforms are software programming language environments, such as C/C++ and Java. These environments define sets of types and components that can be used for implementation.
Implementation-platform models define this information, that is relevant for platform mapping and synthesis, formally and explicitly. To reflect heterogeneous development environments and refinement hierarchies, implementation-platform models can be nested and stacked. For instance, some behavioral synthesis environment may build atop of a VHDL environment for subsequent RTL-synthesis which itself is built atop of a gate-level synthesis platform (→ Section 2.2.1). The resulting platform stack can be modeled using a separate model for each platform and detailed associations between their types and components. The implementation platform models for C/C++ and VHDL, which are used in this thesis, are presented in Section B.2 and B.3.

**Resource Model**

Implementation-platform models enable the access to the resources and resource services offered by computer architecture. This resource-centered view, which is implicit to the implementation-platform model, is called resource model. Fig. 3.13 illustrates the general resource modeling framework (GRM) which is the foundation for resources modeling of the presented work [188]. The GRM evolved in the real-time community as a means of UML-based quantitative analysis and realization specification.

Resource services are the building blocks of architecture implementations. A resource is a model element that provides dedicated services to clients. Resources services and resources are characterized by their QoS. Resources and resource services are classifiers and must be instantiated in order to be useful. The GRM definition of resource services differs from Definition 2.2 in that each service is associated with a single implementation option only. Although the GRM has been developed mainly for the analysis of real-time domain models, its key concepts for modeling resources, resource services, and their QoS integrate smoothly into model-driven, platform-based design for reconfigurable architectures. Intentionally, the GRM specifies abstract concepts rather than defining particular extensions to UML. The model defines what information needs to be defined by domain specific models, instead of prescribing concrete notations, because different domains and tools will require different integrations of the provided concepts. Fig. 3.14 defines the integration of the GRM core model into the presented approach to model-driven, platform-based design.

Resources and resources services are the common concept of implementation and deployment. Thereby the resource model defines the link between these development phases and their underlying models. The functionality offered by a resource service is accessed via the feature interface of implementation types. The particular access mechanism depends on the implementation platform. Implementation components proxy the resource services of their realized implementation types. Designers of target platforms perceive the resource model through the QoS-constraints.
Implementation Types

**Definition 3.10:** An implementation type is an abstract or concrete type that provides resource services which are used to realize design types or other implementation types.

Examples of implementation types are primitive data types, such as C/C++ integer and VHDL std_logic, and complex types, such as types that represent sensors/actors or I/O facilities. Implementation types are either predefined in the implementation-platform model, or created during platform mapping in the implementation model. These types can be organized in type hierarchies, whereas the hierarchy can differ from the design type hierarchy. As formalized in Fig. 3.15, design types and implementation types are mapped to implementation types via realization relationships.

For each supplier type there may be multiple mappings to implementation types. A supplier type may participate in at most one realization dependency regarding a particular implementation platform. This leads to the notions of realization paths and realization graphs.

**Definition 3.11:** A realization path \( R_S = (CL, E) \) of a supplier type \( c_S \in CL \), with \( CL \) being the set of classifiers, is the directed sequence of mappings that define the realization of the type on a specific implementation platform. The \( c_i \in CL \) represent the types in the path. The edges \( e_i = (c_i, c_j) \in E \) represent the realization relationships, whereas \( c_i \) is the client type and \( c_j \) is the supplier type. The type \( c_S \) does not realize any other type with respect to path \( R_S \): \( \nexists \ e \in E : e = (c_S, c_i) \). A sub-path of a realization path is a realization path. There is a dedicated classifier \( c_p \) that is not realized by any other classifier in the path: \( \nexists \ e \in E : e = (c_i, c_p) \). The path is continuous: \( \forall c_i \in CL, c_i \neq c_p, c_i \neq c_j : \exists^1 e \in E : e = (c_j, c_i) \). The services of the supplier type must be sufficient to realize the services of the client type.

In presence of implementation-platform hierarchies, a realization path may span several implementation-platform models. Because multiple implementation-platforms can be used concurrently, each type may have multiple realization paths.
**Definition 3.12:** A realization graph \( R = (CL, E) \) combines the realization paths of one or more types \( c_i \in CL \). The realization graph is a directed acyclic graph (DAG).

The realization graphs of all types defined by platforms (design/implementation) must be statically known. Automated platform mapping is not applicable on these types, because they are considered primitive.

**Example 3.10:** Fig. 3.16 shows a realization graph that integrates the realization paths of the design types `boolean` and `byte`. These types are realized by implementation types of three platforms, whereby the VHDL XST platform is embedded in the VHDL MOCCA platform.

![Realization Graph Example](image)

If no explicit realization path exists for some type, model compilers try to infer an implementation path from existing types. This process is called type mapping and is discussed later in this section. For each implementation type QoS-constraints can be defined. Synthesis constraints define the link into lower level refinements.

**Implementation Components**

Implementation-platform models may specify implementation components. These components can represent software or hardware building blocks that can be integrated in implementation models. The advantages of component-based design have already been discussed earlier in this chapter. Implementation components are not required to be based on the object-based model and UML. Fig. 3.17 shows the meta-model of implementation components. In the presented approach, components are specified using the "Implementation-Platform Profile" (→ Appendix A.6.1).

![Implementation Components Meta-Model](image)

Depending on the implementation platform, different component types can be distinguished. For example, in hardware implementation platforms components for memories, communication, processing, and auxiliary components, e.g. clock- and reset-generators, can be found. Each component has an interface which is implemented by the classifiers that realize the component. For each component multiple realizations with different quality characteristics, interfaces, or dissimilar physical resources, can exist.
Standardized Component Interfaces. The components and their realizing types have predefined semantics, which is reflected in UML-models using stereotypes and interfaces. To enable the utilization of a component only its interfaces must be defined. In order to make components applicable to automated instantiation, it is necessary to standardize the component interfaces. Such interfaces must standardize the features that must be visible to the users of a component. For each feature the component designer must define at least its data type and the protocol to access the feature. Additionally, the interface may be characterized by constraints in order to enable users to estimate the cost of the resulting implementation. The predefined component interfaces of MOCCA are described in Sections B.3.3 in the appendix.

Example 3.11: Fig. 3.18 shows an example for the specification of implementation components. Different specialized implementation components are connected using provided and used interfaces. For instance, the reconfigurable fabric (RF) provides the interface SetClock. This interface is used by the clocking component SystemClock to clock the logic that is executed by the fabric. Similarly, the BlockRAM component provides two interfaces MemBlockLocalAccess and MemBlockExternalAccess (→ Section B.3.2) that enable the access to the storage resources from the user logic and via the communication component.

Modeling Hardware Component Interfaces. While the access mechanism to software components is mostly straightforward, it is not for hardware component interfaces. The basic idea of such a mechanism is to decompose the port signals of components according to the represented logical operation. This operation maps to an UML operation, while the port signals map to the respective parameters. The operation defines the protocol to access the represented behavior of the component. Moreover, the interface definition is structurally separated from the respective implementation. This enables the modeling of different implementations of the same interface. Each implementation must realize all external interfaces of the component. In common cases the port signal set is not decomposable into completely distinct subsets because logical operations may share physical port signals. This is handled during synthesis by mapping shared ports to the same name (→ Section 5.4.2). The name mapping is controlled by mapping-constraints, which will be discussed shortly.

Example 3.12: Fig. 3.19 illustrates these concepts using a FIFO component being part of a VHDL implementation platform. The component represents a queue-storage that provides an interface FIFOAccess. The component is realized by an interface that is implemented by a class. There may be multiple implementations, which may differ, for instance, in the capacity of the queue or the utilized physical resources. The logical operations are modeled using UML operations and parameters. During platform mapping the most appropriate implementation can be chosen for synthesis.

Implementation Environment

The modeled implementation components serve as proxies for actual implementations. They do not have to fully define the functionality of the represented building block. The physical files containing the implementations, e.g. VHDL-files, C++-files, libraries, et cetera, are modeled using artifacts. Again, the modeled
3.3. Platforms and Models

Figure 3.19: Implementation Platform Model Example: FIFO Component and Realizations

artifacts serve as proxies of the corresponding physical artifacts. In the implementation platform model these artifacts are related to components using manifestation relationships. Relationships to additionally required resources are modeled using respective usage-dependencies to their proxies. This way the coupling of all implementation components and types, and the implementation environment is captured.

Model Compiler Components

UML extensions can be interpreted by users and model compilers. In order to avoid design iterations, implementation models and implementation platform models must reflect the characteristics of the compiled/synthesized hardware/software artifacts as close as possible. Thus, it is important to give the model compiler control over the implementation process. Owing to the huge variety of implementation platforms, a model compiler should be able to adapt to the set of platforms being used. To make this adaption convenient and straightforward, the respective model compiler components are part of the implementation-platform model. As a result, the implementation platform, the employed profile, and the function for its automated interpretation are encapsulated. The core of the model compiler provides the central data-structures and algorithms, all specializations are delegated to implementation platform specific compiler components.

Example 3.13: Fig. 3.20 shows MOCCAs meta-model for the model of compiler components. MOCCA distinguishes four principal types of components: NodeGenerator-components implement the generator back-end of the implementation platform. The generator is responsible for translating UML model elements into descriptions that can be interpreted by lower level design flows. NodeMapper-components implement the platform-specific part of the platform mapping algorithm. The mapper is responsible for mapping relevant UML model elements to resources represented by the platform. NodeEstimator-components implement the platform-specific estimation algorithm used throughout platform mapping. The estimators responsibility is the estimation of QoS-characteristics of the mappings computed by the mapper. NodeInterpreter-components implement the link to lower level design flows. This component forwards the output of the NodeGenerator-component to back-end tools and triggers their execution.

Example 3.14: The approach is exemplified for the MOCCA compiler in Fig. 3.21. In this part of the model, the MOCCA components used for estimation, mapping, generation, and back-end tools are specified. The components are used by MOCCA to adapt to the implementation platform. During the compilation, these components are dynamically linked into the compiler. Users may implement new compiler components on their own or specialize existing components to adapt the compiler to their particular requirements.
As implementation components, the model compiler components must implement standardized interfaces. Standardized interfaces enable the dynamic linking of compiler components into the model compiler.

**Auxiliary Mapping and Synthesis Extensions**

To support platform mapping and synthesis, all types, operations, and components of an implementation platform may carry additional information:

**Mapping-Constraints/Properties** - Mapping-constraints and properties control and parameterize the platform mapping. They are used to parameterize allocation, binding, scheduling, and estimation. QoS-constraints are mapping constraints, that define the QoS-values of resource services.

**Example 3.15:** Listing 3.5 shows examples for the specification of QoS-constraints as implemented in MOCCA. As the example shows, the constraints can be scalar constants but may also be specified as probability density functions.

**Listing 3.5: MOCCA QoS-Constraint Specification Example**

```
ImplementationMaxInstances := (3, 'Qty')
ImplementationAddressSpace := (8, 'Byte')
ImplementationLatency := ((normal 5.0.12), 'Cycle')
ImplementationArea := ((histogram 0.0.2,10.0.3,20,0.5,30), 'Gate')
```
Synthesis-Constraints/Properties - Synthesis-constraints and properties are used to control synthesis. Examples include the parameterization of language mappings, e.g. for type names, syntactic rules, the mapping of parameter names to signal names, and the endianess.

EXAMPLE 3.16: Listing 3.6 illustrates the specification of such values, as implemented in MOCCA. The ImplementationName constraint specifies the name of a model element that is used for its implementation. The ImplementationLanguagePattern constraint defines invocations mappings for operations.

Listing 3.6: MOCCA Synthesis-Constraint Specification Example

```plaintext
ImplementationName := 'CLOCK'
ImplementationLanguagePattern := '($this\mathbin{\mathcal{\sim}}$other)'
```

The particular type and format of the mapping and synthesis extensions depends on the specific implementation platform. The examples 3.15 and 3.16 illustrated the format used by the current implementation of MOCCA.

3.3.5 Implementation Model

DEFINITION 3.13: An implementation model defines an implementation-specific realization of a design model.

The implementation model is a sub-model of the platform-specific model of a system. The model defines the implementation of a design model without considering the system deployment. The implementation model defines the mapping of model elements to resource services. The creation of an implementation model generally involves four tasks: type mapping, resource mapping, component mapping, and artifact mapping. These tasks are integral to platform mapping, which is discussed in the next chapter.

Type Mapping

Type mapping defines the realization paths of all design types. Design types are mapped to implementation types, whereas each implementation type implements the same functionality as the represented design type. That is, the types in the design platform model/design model and their counterparts in the implementation platform model/implementation model must fulfill the same contract. The contract of a design type must be realized by the contract of each individual implementation type to which the design type is mapped. Type mapping is the foundation of resource mapping, since all functionality is ultimately accessed through and captured by types.

The type mapping of design platform types is given by their predefined realization paths. For user-defined design types the realization paths are computed throughout platform mapping. All features of user-defined types are eventually constructed from design platform types. Thus, type mapping is performed on user-defined types recursively. This process regards all typed elements (TypedElement), i.e. structural features (attributes), parameters, and variables. The creation of the implementation type is performed bottom-up from the design platform types. If for some design platform type no realization path exists for some implementation platform model transformations become necessary.

EXAMPLE 3.17: Recursive type mapping is demonstrated in Fig. 3.22. The user-defined type UserType is mapped to a respective implementation type with the same name. The realization paths of all types of all typed elements defined by UserType are either predefined by the implementation platform or they are inferred automatically.

In contrast to resource mapping, type mapping does not allocate any resources or resource services to model elements. Type mapping uniquely relates types to each other and thereby ensures the uniformity and consistency of implementations on each particular implementation platform. Further, type mapping is responsible for the unique identification of resource services.

\footnote{In an OO approach the contract of a data type is defined as the set of services the type provides to the environment [75].}
Resource Mapping

Resource mapping binds model elements to appropriate resource services. Resource services are offered by implementation types and their operations, and implementation components. Only those model elements have to be considered that actually define state and/or behavior of the system. The following classes of model elements can be distinguished, according to their relevance to resource mapping. Tab. 3.1 associates the supported model elements according to these groups. The elements in square brackets are currently not supported by MOCCA.

**Mandatory** - The elements of this group are required to be mapped to resource services. These design model elements define the state and behavior of objects in detail.

**Platform** - The elements of this group might be mapped to resource services, depending on the particular implementation platform.

**Not Relevant** - The elements of this group are not relevant to implementation. These elements may be detailed by lower level refinements, which are associated recursively with one of the relevance groups.

<table>
<thead>
<tr>
<th>Relevance</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mandatory</td>
<td>Component, Class, Attribute, Operation, Parameter, Activity, ActivityGroup, ActivityNode, ActivityEdge, Variable, Action, [StateMachine, Association-Class, Enumeration]</td>
</tr>
<tr>
<td>Platform</td>
<td>Interface, PrimitiveType, DataType, [Trigger, Artifact]</td>
</tr>
<tr>
<td>Not Relevant</td>
<td>Others</td>
</tr>
</tbody>
</table>

Notice, that the relevance refers to model elements types. It does not imply, however, that each implementation platform must be capable of mapping each particular model element instance. Whether a model element instance can be mapped, depends exclusively on the chosen implementation platform. In case no direct mapping is available on a particular platform, model transformations may be necessary. The presented approach to model implementation platforms allows for a straightforward encapsulation of transformations by means of dedicated model compiler components. The particular approach being taken to map implementation models to resource services is the responsibility of the implementation platform itself. The platform mapping approach, which is presented in Chapter 4, merely defines the platform-independent parts of DSE and the respectively utilized data structures.

**Example 3.18:** The concept of resource mapping is illustrated in Fig. 3.23 for a component being mapped to hardware. The component is implemented using a VHDL implementation platform. Various predefined...
implementation types for communication, clocking and reset generation, and storage are allocated to the component. The operation \( \text{eq} \) is allocated to realize an address decoder. Obviously, the C++ mapping of this component would be fairly different, i.e. it would not allocate any local resource services at all.

![VHDL Component Resource Mapping Example](image)

**Fig. 3.23: VHDL Component Resource Mapping Example**

### Resource Mapping of Actions and Variables

There is a fundamental approach for the mapping of actions and variables that can be used over a wide range of platforms. Earlier in this section it was shown, how actions are mapped to operations of design types. This mapping is exploited to map actions to operations that are provided by implementation types. For design types, mappings to implementation types are defined by means of the realization path of the design type (→ Definition 3.11). Given the type mapping of a design type, the identification of the mapping of the design type features to the respective implementation type features is straightforward:

1. map the design type that defines the operation to the implementation type (denoted \( ct \)),
2. map all parameter types of the operation, including the return parameter, to their implementation types (denoted \( \langle cp_0, \ldots, cp_n \rangle \)), and
3. find the most specific operation in type \( ct \) with the mapped argument list, i.e. invoke algorithm \( \text{findM SO}(ct, \text{name}, \langle cp_0, \ldots, cp_n \rangle) \) (→ Algorithm 3.1).

For simplicity, the names of the operation in the design type and the implementation type are equal. Since the search of the operation respects the type hierarchy, the hierarchy of design types and implementation types can differ.

**Example 3.19:** Fig. 3.24 illustrates the mapping of the MAL statement in Listing 3.7. The design type \( \text{short} \) is mapped to an implementation type \( \text{std_logic_vector<16>} \), which is offered by a VHDL implementation platform. The sub action is mapped to an appropriate resource service offered by the sub operation of this type. This operation is realized by the functional unit \( \text{SUB} \). The instances of \( \text{std_logic_vector<16>} \), i.e. \( \text{diff1}, \text{csample}, \text{psample} \) are realized using flip-flops.

**Listing 3.7: MAL Statement Mapping Example**

```plaintext
short diff1, csample, psample;
...
diff1 = csample - psample;
```
Notably, the type mapping implicitly determines the realization of the variables. For instance, in software implementations instances of the implementation type will map to memory cells, while the operation is realized using processor instructions. In hardware implementations instances of this type will be realized using flip-flops. The operation is implemented by a functional-unit that may be realized on a RF using the available RFUs.

**Modeling the Deployment Hierarchy**

The mere mapping of the model elements to the employed resource services is not sufficient however. All resource services must be related to the elements of the target architecture which actually provides the resources at run-time. In order to establish this relationship the following additional properties must be defined by the implementation model:

- **Component Mapping** - The component mapping defines realization of components by implementation types. For each component a set of implementation types is identified that collaboratively realize the component. The same implementation type may participate in the realization of different components.

- **Artifact Mapping** - Each component is manifested by at least one artifact. On the model level artifacts serve as proxies of physical files, libraries, tables, etc. An artifact may manifest multiple components.

The actual relation of artifacts to the nodes of the target architecture is established in the deployment model using manifestation relationships, which is called **Node Binding**. Type mappings, component bindings, and resource allocations/bindings are captured using realization dependencies. The mappings define the partitioning of the design among the nodes of the target platform.

### 3.3.6 Deployment Platform Model

**Definition 3.14**: A deployment platform is an architecture platform comprising nodes, their relationships, and constraints, that build the foundation for the deployment of designs.

**Definition 3.15**: A deployment platform model is an UML model that represents an deployment platform.

The relationship of the deployment platform model to the implementation model and the platform-specific model is formalized in Fig. 3.25.

As a result of Definition 3.14 and 3.15, deployment-platform models define an abstract notion of hardware architectures by means of nodes and communication paths between them. The micro-architecture of neither
the nodes nor communication paths is defined in detail. According to the UML specification, nodes may comprise an abstract or concrete processing element (PE), dedicated memory, and peripherals. Communication paths represent associations between nodes. They can be used to represent all kind of networks. UMLs very general notion of nodes and communication paths is insufficient for the purpose of hardware/software co-design. Both elements offer resources and resource services which must be defined in detail. In the model these elements are characterized by their QoS. The set of according UML extensions is defined by the "Deployment-Platform Profile" \cite{section:6.4}. In Section B.4 the model of the deployment platform that is used in this thesis is given.

Each node that offers services being used by an implementation platform is related to this platform via a specialized realization relationship. The implementation platform provides abstractions of the resource services implemented by physical resources of the node. This notion is formalized in Fig. 3.26.

As the according platform models the implementation platform model and deployment platform model are complementary. Each implementation platform defines an abstract interface of a set of nodes of a computer system, while the deployment platform defines the actually existing nodes and their interconnection. The nodes may be real computers or abstract machines. Abstract machines may be execution environments or even refinements being interpreted by lower level design flows. Eventually they are build atop of some physical computer system. At the bottom level of this hierarchy all functionality is realized by hardware. This model resembles Gilois layered architectural model of computer systems \cite{230}.

\subsection{Deployment Model}

\textbf{Definition 3.16:} A deployment model defines the deployment of the components of an implementation model on the nodes of a deployment platform.
This model defines the relationship between the nodes of a deployment platform and the artifacts in the implementation model, which is called Node Binding\(^6\). Both models, the implementation model and the deployment model, complement each other. The deployment model and the implementation model build the PSM. The respective platform models comprise the target platform model.

\(^6\) The relationship of artifacts and nodes is defined in the "Deployments" section of the UML specification.
4. PLATFORM MAPPING

4.1 Platform Mapping for Object-Oriented Specifications

4.1.1 Definition of the Mapping Problem

The ultimate goal of system development is to create an implementation of the specification that executes on a given target platform. During this process, the designer performs a number of well-defined activities that will eventually map the functionality of the specification to the resources of the target. Platform mapping relates design model elements to the resources that will be used for their implementation. The central question of platform mapping is whether the target platform contains the appropriate resources, and resource service types, and if they can be combined such that the system functionality is implemented. In contrast to synthesis, the implementation is only simulated at an abstract level in order to compute more mappings in reasonable time.

Depending on the particular design representation, target architecture, and optimization goal different formulations of the mapping problem are possible. Definitions 2.1-2.6, 3.4-3.16 are specialized in the context of model-driven architecture and the mapping problem is defined as follows:

**Definition 4.1:** Given a design platform model $DS_n$, a design model $d_n \in DS_n$, a set of constraints $C_s$, specified by a target platform model $DS_{n+1}$, a set of properties $P_s$, a set of quality characteristics $Q_s$, and a cost-function $cost : DS_{n+1} \times Q_s \rightarrow \mathbb{R}$ the platform mapping problem is to find a transformation $\delta : DS_n \rightarrow DS_{n+1}$, whereas $d_{n+1} \in DS_{n+1}$ is a platform-specific model of $d_n$ that fulfills all properties $P_s$, satisfies all constraints $C_s$, and minimizes the cost-function cost.

This definition is very general indeed. The detailed definition of the target architecture, cost-function, properties, constraints, and quality characteristics depends on the particular application domain. For instance, while in safety-critical, resource constrained embedded systems the satisfaction of system safety and resource minimization might be important characteristics, in high-performance computing the minimization of the overall computation time will be important. Accordingly, different algorithms for design space exploration are necessary. In the remainder of this chapter, an according solution that is based on the simulated annealing heuristic is presented.

4.1.2 Challenges

In contrast to traditional approaches in system-level design, object-orientation and model-based development impose a number of challenging characteristics, namely inclusion polymorphism, dynamic object lifetimes and communication, dynamic object size, and different target platforms. These challenges are common to system design and cause severe implications on platform mapping, synthesis, and run-time support. Object-oriented system-level development should address these challenges in order to exploit the full potential of the overall approach.

**Inclusion Polymorphism** - Objects encapsulate state and behavior, both of which are defined by classes. Classes may be organized in inheritance hierarchies; a class (child) can specialize at most one immediate class (parent)\(^1\). Inheritance introduces the notion of inclusion polymorphism. Messages are dynamically dispatched to message handlers according to the dynamic type of the receiver object.

---

\(^1\) Multiple inheritance of classes is not considered in this thesis because it has minor practical importance and can be simulated by multiple inheritance and implementation of interfaces.
4. Platform Mapping

**Dynamic Object Lifetimes and Communication** - The dynamic creation and destruction of objects is a very powerful technique. Objects are created when they are actually required and disposed at the end of their lifetime. Objects may be created and destroyed by other objects on the local node or remote nodes. While for local objects the according mechanisms are commonly realized by lower level design flows and run-time support, transformations of the design model may be necessary for objects being deployed on remote nodes.

**Dynamic Object Size** - While dynamic object creation and destruction relates to *when* resources are bound to objects, the challenge of dynamic object sizes is *how many* resources are bound to them. An example are array objects whose number of elements is determined not before run-time. This will be a challenging feature not only to platform mapping if a node of the target platform does not directly support this dynamics. For example, the most execution environments based on microprocessors and memories support dynamic object sizes by means of their run-time environment. Similar support is not commonly provided by ASICs or RFs.

**Different Target Platforms** - Target platforms define the execution environment for design model implementations. They build the foundation for the generation of implementations for system designs. In practice, target platforms with very different functional and non-functional capabilities can be found. Target implementation languages, the available implementation types and components, the components of the model compiler, tools of lower level design flows, available run-time support, and the implementation options that are embodied into the model compiler components impact the final implementation.

### 4.1.3 Structure of the Design Space

The quality and computation time of DSE algorithms depend strongly on the chosen granularity. The mapped model elements (→ Tab. 3.1) have different granularity. Each mapped model element represents an implementation parameter that determines the mapping of the design model to the target platform, whereas the number of design model instances of an element type generally grows with finer granularity. Consequently, the number of model elements determines the control, accuracy, and quality of the generated mappings. On the other hand, the computation time for DSE also grows with the number of elements. Coarse granularity DSE is restricted in meeting the quality requirements, because less points of the design space can be explored. Fine-grained entities generally expose more optimization potential.

The identification of a good tradeoff between quality and computation time is an intrinsic problem of DSE approaches using a fixed granularity. Thus, Henkel and Ernst suggested to use elements with different granularity which is adapted dynamically to current requirements of the algorithm [101]. Up front hardware/software partitioning they compute partitioning objects, which are essentially groups of the operators of an operator network. Each partitioning object may consist of just one operator or multiple connected operators. A hardware/software partitioner, which is based on the SA heuristic, then randomly selects partitioning objects and moves them from hardware to software and vice versa. All partitioning objects have the same probability of being selected. Their work proved that it is advantageous to use multiple granularities.

The UML meta-model semantic provides an inherent notion of granularity by means of deployment hierarchies. Tab. 4.1 associates the model elements supported by the platform mapping approach to granularity levels.

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>G0 (coarse)</td>
<td>Component</td>
</tr>
<tr>
<td>G1</td>
<td>Classifier (Class, Interface, PrimitiveType, DataType)</td>
</tr>
<tr>
<td>G2</td>
<td>Attribute, Operation</td>
</tr>
<tr>
<td>G3</td>
<td>Activity, Parameter, Variable</td>
</tr>
<tr>
<td>G4 (fine)</td>
<td>ActivityGroup, ActivityEdge, ActivityNode, Action</td>
</tr>
</tbody>
</table>

The model elements at higher hierarchical levels have coarser granularity and may contain finer granular
4.1. Platform Mapping for Object-Oriented Specifications

elements via aggregation or composition relationships, but not in reverse. Consequently, the design space is structured by the UML semantics. Platform mapping must respect the semantics of the mapped elements and the containment hierarchy with regard to deployment, which is called deployment hierarchy. In this hierarchy, model elements are related to resources by means of realization- and manifestation relationships.

The function

\[ d\text{children} : E \mapsto P(E) \quad E = \{e_k\} - \text{set of mapped model elements} \]

returns the direct child elements of a model element in the deployment hierarchy, called deployment children. Primitive model elements, e.g. variables, actions, etc., are not composed from other model elements. Thus, they do not contain any (deployment) children. Model elements like components, artifacts, and classes are compositions of nested model elements so their set of deployment children is non-empty.

The function

\[ d\text{parents} : E \mapsto P(E) \quad d\text{parents}(e_k) = \{e_p | e_k \in \text{dchildren}(e_p)\} \]

returns the direct parent elements of a model element in the deployment hierarchy, which are called deployment parents. Notice, that the UML semantics allows a model element having multiple deployment parents. For instance, all components that realize the same class are deployment parents of this class.

Instances possess the same containment hierarchy as model elements. Thus, similarly to model elements, the function

\[ d\text{children} : EI \mapsto P(EI) \quad EI = \{e_k\} - \text{set of mapped model element instances} \]

returns the set of model element instances from which an instance is composed. Straightforwardly, the function

\[ d\text{parent} : EI \mapsto EI \quad d\text{parent}(e_{ik}) = \begin{cases} e_{ip} & \text{if } e_{ik} \in \text{dchildren}(e_{ip}) \\ \emptyset & \text{else} \end{cases} \]

returns the deployment parent element in this hierarchy. Obviously, on instance-level each instance can have at most one deployment parent.

**Allocation and Binding of Resource Services**

The mapping of a model element to the set of resource services and implementation options that are used to realize the element is called a binding. The binding corresponds to the design of the model element. If a model element has deployment children its design is defined by the designs of all child elements. A design allocates resource services of an appropriate number and type.

**Definition 4.2:** An allocation \( A \) is a set of resource services and implementation options pairs \( rs = \langle r_i, o_j \rangle, \) with \( r_i \in R_{n+1}, \) and \( o_j \in O_{r_i,n+1}. \)

**Definition 4.3:** The design \( d_{e_k,n+1} \) of a model element \( e_k \in E \) is a binding of resource services and implementation options that are sufficient to realize the instances of \( e_k. \)

\[ \text{mapping} : E \mapsto P(A) - \text{mapping of } e_k \in E \text{ to the locally bound resource services} \]

\[ d_{e_k,n+1} - \text{design of model element } e_k \text{ with respect to design space } DS_{n+1} \]

\[ d_{e_k,n+1} = \text{binding}(e_k) \cup \bigcup_{e_{ij} \in \text{dchildren}(e_k)} d_{e_{ij},n+1} \]

Optimality of an element with respect to its local design space does not imply global optimality. The impact of fine granular model elements on the global optimality decreases rapidly with system size. This suggests to use different granularities concurrently. Fig. 4.1(a) illustrates the size of the neighborhood as a function of the considered granularity. The finer the granularity the smaller the neighborhood of the current mapping. Large neighborhoods tend to contain more local extremities, which are a subset of the candidate mappings in the neighborhood, whereas smaller neighborhoods often better support the fast computation of new solutions. As Fig. 4.1(b) illustrates using simulated annealing, DSE algorithms continuously select candidate mappings from the neighborhood of the current mapping. Each selected mapping has a different neighborhood. The selection process iteratively approaches the final mapping.
Allocation, Binding, and Scheduling of Resource Service Instances

The design of each model element is defined by bindings of resource services while resource service instances are bound to model element instances at run-time. Consequently, the definition of the design of model elements must be reflected on the instance-level.

**DEFINITION 4.4:** An allocation $AI$ is a set of resource service instances and implementation options 3-tuples $ri = \langle id, r_i, o_j \rangle$, with an unique identifier $id$, $r_i \in R_{n+1}$, and $o_j \in O_{r_i, n+1}$.

**DEFINITION 4.5:** The design $d_{ei_k, n+1}$ of a model element instance $ei_k \in EI$ is a binding of resource service instances and implementation options that are sufficient to realize $ei_k$.

$EI = \{ei\} \rightarrow \text{set of mapped model element instances}$

$binding : EI \mapsto P(AI) \rightarrow \text{mapping of } ei_k \in EI \text{ to the locally bound resource service instances}$

$d_{ei_k, n+1} \in DS_{ei_k, n+1} \subseteq DS_{n+1} = P(AI)$

$d_{ei_k, n+1} = binding(ei_k) \cup \bigcup_{ei_j \in \text{children}(ei_k)} d_{ei_j, n+1}$

Scheduling determines for each model element instance the time steps $t_i, t_{i+1}, \ldots, t_{i+m} \in \mathbb{N}^*$, also called control steps, in which it is executed. Instances of model elements that execute in mutual exclusion may be bound to the same resource service instances. If a resource service instance is bound by different model element instances, a schedule must enforce the timely separation of their execution.

**DEFINITION 4.6:** A schedule is defined as:

$schedule : EI \mapsto P(\mathbb{N}^*)$

$length = \text{length of a schedule of a set of model element instances, with}$

$\forall ei_i \in EI_j \subseteq EI : \forall t_k \in \text{schedule}(ei_i) : length = \max(t_k)$

$width = \text{width of a schedule of a set of model element instances, with}$

$\forall t_k \in \mathbb{N}^* : width = \max|\{ei_i | ei_i \in EI \land \text{schedule}(ei_i) = t_k\}|$

Each resource service instance can execute at most one model element instance at any point in time. If any two model elements $ei_i$ and $ei_j$ can be executed concurrently the schedules must satisfy the following condition:

$\forall ei_i, ei_j \in EI, ei_i \neq ei_j : \text{schedule}(ei_i) \cap \text{schedule}(ei_j) = \emptyset$

The number of time steps to which an operation\footnote{The jargon of high-level synthesis (HLS) is used here in order to make the discussion transparent. In the context of UML replace operation by action.} is scheduled depends on the latency of the resource service instance to which the action is bound. Multi-cycle operations are scheduled to a continuous sequence of time steps.
Chained operations are operations with data-flow and/or control-flow dependencies between them being scheduled to the same time step. Depending on the model element and constraints, different formulations of the scheduling problem exist. In time-constrained scheduling (TCS), the condition $\text{length} < \text{t}_{\text{max}}$ must hold for an upper bound $\text{t}_{\text{max}}$. In resource-constrained scheduling (RCS) the number of instances of the resource services is restricted such that

$$\forall r_k \in R_{n+1} : \big|\{r_i | r_i = \langle id, r_k, o_m \rangle \in \bigcup_{e_i \in EI} \text{binding}(e_i)\}\big| \leq a_k$$

must hold for a maximum number of instances $a_k \in \mathbb{N}^*$ of each particular resource service $r_k$. In addition, resource constraints can be formulated indirectly by restricting the width of the schedule to an upper bound $s$:

$$\forall t_k \in \mathbb{N}^* : \big|\{e_i | e_i \in EI \land t_k \in \text{schedule}(e_i)\}\big| \leq s$$

Thereby $s$ represents the number of slots per time step, i.e. the maximum width of the schedule. This constraint restricts the resource services independent of individual resource service types. This is useful for technologies that allow the implementation and execution of different resource services using the same fabric. Examples are RFs, super-scalar uPs, and VLIW-uPs. In RFs the number of concurrent instances of a particular resource service is restricted while in uPs this constraint is determined by the number of instruction pipelines and arithmetic-logic-shift units (ALSUs).

TCS and RCS are NP-complete [231]. The combination of both problems, i.e. time-resource constrained scheduling (TRCS), can be solved by reformulating it either as TCS or RCS. Different approaches to scheduling exist for instruction scheduling and HLS. While the problem of instruction scheduling is to optimally utilize the specifics of a given architecture (e.g. pipeline, scalability, and VLIW) [229], in HLS an architecture is constructed that executes operations using an allocation of resource service instances that satisfies a specific set of constraints. Recently, there is increased effort to utilize software approaches in HLS [159].

Most scheduling approaches consider basic blocks, whereas a basic block is a (maximum) sequence of actions that is entered at the beginning and leaves at the end, without any possibility of halt or branch [232]. Basic blocks can be represented by DFGs, with $\text{DFG} = \langle V, E \rangle$, whose vertices $v_i \in V$ represent the operations to execute, and the edges $e_j \in E$ correspond to data-flow and control-flow between the operations. Thereby the DFG is directed and acyclic. Then the scheduling problem is to find an optimum topological order of the operations. Due to the NP-hardness of scheduling, commonly approximations\(^3\), branch-and-bound [233], or ILP-based approaches are used [234]. The most frequent approaches for basic block scheduling are:

**ASAP** - In as-soon-as-possible (ASAP) scheduling each operation is assigned to the first time step at which all its data-flow and control-flow dependencies are satisfied (→ Fig. 4.2(b)).

**ALAP** - In as-late-as-possible (ALAP) scheduling each operation is assigned to the latest possible time step before its output is needed (→ Fig. 4.2(c)).

**List Scheduling** - List scheduling is an approach which is frequently used for instruction scheduling and RCS [235]. A prioritized ready-list keeps all operations that are ready to schedule since their dependencies are satisfied. The operations with the highest priority are scheduled to the current time step until there are no more operations, or, in RCS, no more resource service instances. Instruction scheduling uses the execution time to prioritize operations. In HLS, the mobility\(^4\) ($t_{\text{alap}}(v_i) - t_{\text{asap}}(v_i) + 1$), the urgency ($t_{\text{alap}}(v_i) - t_{\text{ready}}(v_i) + 1$), and the number of successors of an operation, are frequent priority measures.

**Force-directed scheduling** - Force-directed scheduling is an extension to list scheduling, that solves TCS, and tries to satisfy resource constraints [236]. Each operation $v_i$ has an associated probability $p_{v_i,t_k} = \text{mobility}(v_i)^{-1}$ to be scheduled to a time step $t_k \in [t_{\text{asap}}(v_i), t_{\text{alap}}(v_i)]$, and $p_{v_i,t_k} = 0$ for $t_k \notin [t_{\text{asap}}(v_i), t_{\text{alap}}(v_i)]$. When an operation is scheduled to particular time step this effects the interval

---

\(^3\) In literature frequently the term heuristic is used, although actually approximation algorithms are meant.

\(^4\) For simplicity it is assumed that $v_i$ is no multi-cycle operation, otherwise the min/max time step of time intervals must be used in the equations.
to which its unscheduled predecessor and successors can be scheduled, which is modeled by a cost function. In each iteration the operation is scheduled to a time step that causes the least cost increment.

All of these approaches are used in HLS; instruction scheduling adopts list scheduling. The optimization potential of scheduling that operates on individual basic blocks can be rather restricted since these blocks are frequently short and contain only few parallel operations. Optimizations, such as loop unrolling, code motion, and speculative execution, extend the blocks and can reveal yet unused optimization potential. Cross-block scheduling algorithms, such as trace scheduling and percolation scheduling, take multiple basic blocks and their control-flow dependencies into account. Other approaches combine multiple basic blocks into an equivalent flat representation before applying traditional basic block scheduling [76, 237]. Recently, research in scheduling strategies has been started in the context of HLS. Thereby the focus is on increasing the degree of parallelism by integrating dynamic versions of well-known technology independent software compiler optimizations (→ Tab. C.2) into cross-block scheduling [159, 237]. Since only single data-paths are considered the scope of optimizations is local. System-level approaches, like the one presented in this thesis, create opportunities for global optimizations.

Notationally, UML activity diagrams are used to represent schedules. The association of a scheduled action to the time steps to which it has been assigned is denoted graphically by overlapping the respective time steps with the shape of the action. This notation is custom in the logic design and hardware synthesis. Notice, that conceptually an action takes zero time to execute, the latency is introduced by the functional unit that implements the action.

![Data-flow Graph and Scheduling](image)

Notationally, UML activity diagrams are used to represent schedules. The association of a scheduled action to the time steps to which it has been assigned is denoted graphically by overlapping the respective time steps with the shape of the action. This notation is custom in the logic design and hardware synthesis. Notice, that conceptually an action takes zero time to execute, the latency is introduced by the functional unit that implements the action.

4.2 Target Platform Architecture

4.2.1 Architectural Illusions

The target platform model provides an abstract view on the architecture of the computer system. To deal with the actual complexity and diversity of the possible physical realizations, the described abstract computer system must support a number of views, which are also called illusions [238]. The computer architecture is required to support the following illusions:

**Object Type Illusion** - Each node of the computer architecture is capable of processing the type instances defined by its associated implementation platform. The processed types can be either native or composed from native types.

**Simple Memory Illusion** - All objects being stored and executed by the computer system are stored once at a variable but unique address. Each object is accessible by read and write operations through an interface which is uniquely defined per object type.
Control Point Illusion - The computer architecture must define a single node from which the execution of the control flow starts. During execution concurrent control flows may exist. The node which evokes the initial control flow is called the master node.

Simple Communication Illusion - Objects communicate by exchanging structured messages. All message exchange is unicast. Broadcast messages must be handled at higher levels. A message is passed to the receiver through an unique object interface.

Operator Illusion - Each action executed by the computer architecture corresponds to an operator whose inputs, outputs, and effects are unambiguously specified by the target platform, and whose execution is atomic.

Existing physical computer architectures may or may not support these illusions directly. The support depends on the considered set of types, operations, control flow, and communication mechanisms. If other types, operations, or mechanisms are required, they can be realized logically atop of existing architectures. This logical layer may be implemented by design flows and tools, or by abstract execution environments [230, 238].

4.2.2 Implementation Options

Implementation options define how the described architectural illusions can be realized using a given implementation platform. Naturally, a huge number of options exist for the implementation of objects and their interconnect. Each commitment to a set of implementation options must enable the realization of the system functionality and should provide freedom to optimize the quality characteristics. In this thesis, the implementation of objects is based on the models of uP and finite state machine with data-path (FSMD).

FSMD - Implementations based on the FSMD-model [159, 175], support the implementation of mixed control- and dataflow behavior. A FSMD couples a FSM with a data-path. The FSM realizes the control-flow of the represented behavior. The FSM sequences through a number of states. In each state the data-path executes a number of computations. The result of some computations is used by the FSM to decide between alternative execution sequences. This model will be formalized in Section 5.4.4.

uP - The uP-based implementation is being used for those parts of the design which are uncritical to overall performance. A central processing unit (CPU) sequentially executes the behavior of objects. Sequential execution exposes relatively large execution latencies. The implementation effort of the CPU is amortized over the executed objects. Generally, a uP is a FSM whose function is the processing of instructions.

EXAMPLE 4.1: Fig. 4.3(a) shows an example of objects being executed by a single node of the target architecture. The uP-based implementation of objects is illustrated in Fig. 4.3(b). An universal automaton (CPU) sequentially executes instructions that define the behavior of objects. Object state is stored using a dedicated memory. One CPU executes the instructions of multiple objects using a program counter (PC). The uP defines the communication interface. The FSMD implementation (Fig. 4.3(c)) is defined by a behavior-specific FSM that controls the activation and switching of data-path components. Object state is stored using a register file, which is shared among different FSMDs, and local latches. An object comprises one FSMD per behavior. FSMD implementations are not shared among different objects. The list of symbols gives the name of the functional units used in this thesis and their associated semantics.

This combination of complementary models suits a wide range of requirements. The implementation of both models is not constrained. For instance, the uP perceived at system-level may be an abstract machine, such as a C/C++-compiler. Physical FSMD implementations may be covered by a VHDL machine. In order to avoid design iterations, details of lower level refinements are back-annotated in the implementation platform model by means of QoS-constraints.
4. Platform Mapping

Realization

uP-based implementations will typically use existing processors. FSMDs will commonly be implemented using network-coupled RFs at the full range of granularity. For the purpose of this thesis, a fine-grained, run-time reconfigurable RF is used, which is implemented physically by a FPGA. The clocking of the synthesized data-paths is single-edged and synchronous. The communication among the data-path units uses a multiplexer architecture.

The objects being executed by RFs must be accessible by the objects running on the master node (Simple Memory Illusion), whereas all objects may execute in parallel. The message exchange between objects executed by the same node is delegated to the node itself. For instance, message exchange on a uP may be realized using a stack and the control-flow instructions of the processor. The communication between objects running on different nodes is either supported by the node and its lower level refinements, or the communication must be delegated to a run-time environment. For example, if all nodes can become masters of a common bus, they might access their objects rather directly. If, in contrast, some nodes are coupled through some internetwork additional run-time support for routing, (un)marshalling, etc. will be necessary. To simplify implementation, a single data representation, i.e. endianness, data encoding, etcetera, is used. This must be accomplished by the resource services modeled in the implementation platform.

4.2.3 Architectural Constraints

The implementation of the architectural illusions is constrained. QoS-constraints are the major quantitative restriction (→ Section 3.3.4). These QoS-constraints are determined by a specific binding of resource service instances and the respective implementation options. A related problem occurs, if the QoS for the implementation of a design model is not statically known. For example, if the number and type of objects being created depends on input data. This kind of constraint can often be handled by resource extensions. If no direct extension exists, architectural changes or design iterations may become necessary. The employed implementation platforms do impose additional architectural constraints. In layered computer architectures often some lower level functionality can not be exploited on higher levels because it is not supported by the according implementation platforms. For example, the abstract machine model provided by the C/C++ language does not support direct bit manipulation operators although they might be available at the target processor. Implementation options typically impose additional constraints which directly relate to the challenges of object-orientation:

*Inclusion Polymorphism* - Inclusion polymorphism and the implementation of objects at all may become a problem for both uP-based and FSMD-based implementations. For instance, while the C++ programming language directly supports this type of polymorphism, in C it must be implemented explicitly.
4.3 Platform Mapping Algorithms

4.3.1 A Platform-Based Distributed Mapping Approach

The platform mapping problem is NP-hard. If at some granularity the number of elements is \( n \) and there are \( k \) implementation options for each element, the overall number of possible implementations is \( k^n \). If the platform mapping problem is restricted to a partitioning problem \( k \) corresponds to the number of nodes among which functionality is divided. This number increases dramatically if also resource mapping is performed. To make the problem tractable at all the design space is structured and restricted before DSE. For this, designers create mappings of subsets of the considered model elements manually. This approach is applicable to all model elements regardless of their specific level of granularity. The TPM specifies manually created mappings by means of realization paths for the DPM types (\( \rightarrow \) Section 3.3.4).

\[ ^5 \text{Proposals to overcome this restriction have been made on the MOCCA co-development platform [216].} \]
In upper hierarchical levels designers may want to define mappings as well. At these levels the partitioning of objects among the nodes of the target platform is of most interest. Due to the large amount of effected elements the partitioning at these levels has particular impact on the quality of the global solution. Partitions can be defined manually or automatically. The manual definition of resource mappings is possible in principle but it will fail for real world designs, due to the huge amount of model elements to be considered. Even if designers focus on those parts of the design that are most important the number of elements is still far beyond what is practicable. Partitions are defined transitively, using the appropriate UML relationships (type $\rightarrow$ component $\rightarrow$ artifact $\rightarrow$ node).

DSE algorithms complete partial PSMs by respective resource mappings. Design space restrictions may cause DSE missing the optimum solution or even worse to be not able to find a valid solution at all. Model transformations may be necessary to enable the mapping of design models. The advantage of transformations to design optimality has been demonstrated in the context of HLS, e.g. [76, 237, 239]. The approach presented in this thesis integrates transformations into platform mapping.

In the remainder of this chapter, an approach to the platform mapping problem of Definition 4.1 is presented. The proposed approach addresses the fundamental challenges of object-oriented specifications, is based on the platform concept, and imposes only minimum architectural constraints. Platform mapping is based on the principles of separation and delegation. All platform-independent portions of platform mapping algorithms are separated from the platform-specific parts. The platform-independent and the specific parts are delegated to distinguished components, which communicate through well-defined interfaces. Thereby the handling of different target platforms and the implementation of variable platform mapping algorithms is supported. As the research on design space exploration indicates, there is no single best approach. The suitability of a particular approach depends on the granularity, the number of the partitioned entities, the design representation, and the model of computation. Using the proposed approach, platform-independent parts are reusable by various target platforms. The platform-specifics are introduced by means of the implementation platform model. Fig. 4.4 clarifies the manifestation of these principles as overall design of platform mapping algorithms. This definition represent an extension of the advancing meta-model for model-driven architecture that has been presented in the course of this thesis.

---

**Fig. 4.4: Platform Mapping Algorithm Design (continues Fig. 3.20)**

The platform-independent portion of the algorithms is decomposed into three components:

**Controller** - The controller triggers the breeding and evaluation of mappings. This component steers the online selection of mappings and termination of the mapping process. A controller may use different breeders and evaluators throughout its execution, e.g. a breeder to compute some initial solution which is then optimized by another breeder. The evaluation of the computed mappings may change throughout platform mapping.

**Breeder** - The breeder coordinates the computation of platform-specific mappings for parts of the design model and combines them into a single mapping of the design model. All platform-specific mapping is delegated to respective mapper components. Breeders and mappers search feasible mappings in the overall design space. Input to the breeding algorithm is a TPM and a complete or partial
Thereby a design model is considered as a partial PSM, because it has no implementation or deployment parameters fixed. In case of successful execution the output of the breeder is a complete PSM.

Evaluator - The mappings, generated by breeders, may violate architectural constraints. Evaluators check if mappings satisfy all constraints while all platform-specific checks are delegated to the respective estimator component. These estimators are also responsible for the assessment of the metrics considered by the evaluated cost function. Moreover, evaluators compute the value of the cost function for each mapping.

Descriptions of the platform-specific components have already been given in Section 3.3.4. There will be different algorithms and implementations of each of these components. Notably, the definition in Fig. 4.4 shows the principal functionality that must be implemented. It does not prescribe a particular algorithm design however. For instance, a number of optimization algorithms, such as greedy search and dynamic programming, require a tight integration of the controller and breeder components.

### 4.3.2 Mapping Control

As has been shown in Section 2.1.2, a multitude of DSE algorithms has been developed. These algorithms can be transformed such that they fit into the presented algorithmic framework. The modeling of DSE problems for deterministic approaches is typically quite complex and the search of optimum solutions is often computationally intractable. Thus, virtually all practical approaches focus on heuristics. In this section a DSE algorithm is presented, which is based on simulated annealing [134].

**Controller Algorithm**

Algorithm 4.1 shows the SA-based controller algorithm being used in MOCCA. Input of the algorithm is an initial mapping, which comprises the starting point of optimization, a starting temperature, and the final temperature that must be reached to stop the search. The breeder selects candidate mappings from the neighborhood of the current mapping. With a small probability, or if the cost of the candidate mapping is less than the cost of the current mapping, valid candidate mappings become the current mapping.

**Algorithm 4.1:** SA controller - control 

**Input:** Initial valid design: $d_{n+1}^0 \in DS_{n+1}$. Initial temperature: $temp^0$. Minimum end temperature: $temp_{min}$.

**Output:** Mapping $d_{n+1}^1 \in DS_{n+1}$, a platform-specific model of $d_n$ to $DS_{n+1}$

1. Set $step \leftarrow 0$, $d_{n+1}^0 \leftarrow initialize(d_{n+1}^0)$;
2. If $d_{n+1}^0 = \emptyset$ then Mapping failed;
3. $cost_{n+1}^0 \leftarrow computeCost(d_{n+1}^0)$;
4. While $temp^{step} > temp_{min}$ do
   a. $step \leftarrow step + 1$, $temp^{step} \leftarrow nextTemperature(temp^0, step)$;
   b. $cand_{n+1}^{step} \leftarrow breed(d_{n+1}^{step-1})$;
   c. $cost_{n+1}^{step} \leftarrow computeCost(cand_{n+1}^{step})$;
   d. If $cost_{n+1}^{step} < cost_{n+1}^{step-1}$ then $d_{n+1}^{step} \leftarrow cand_{n+1}^{step}$ else $rand \leftarrow random()$ with $random : [0, 1]$;
   e. If $rand < e^{-\frac{cost_{n+1}^{step-1} - cost_{n+1}^{step}}{temp^{step-1}}}$ then $d_{n+1}^{step} \leftarrow cand_{n+1}^{step}$;
5. Return $d_{n+1}^{step}$.

**Initial Solution**

The initial solution may be a partial or complete PSM. This mapping is then completed and improved. If the output of the previous DSE step is a set of PSMs, such as in genetic algorithm (GA), an intermediate
selection step must be performed or multi-start SA is used. In the presented approach, designers may provide partial solutions to the system, which are then automatically refined by different adjustable DSE algorithms. This enables the integration of designer experience.

**Cooling Schedule**

Experience shows, that the cooling schedule and the random selection of solutions from the neighborhood of the current feasible solution are more critical to solution quality than the initial solution [134]. Equation 4.1 presents two very simple yet frequent static cooling schedules (CSs). A frequent choice of parameter \( n \) in cooling schedule 2 is two, higher (lower) values of \( n \) cause lower (higher) reduction rates. The cooling schedule 2 has less probability of accepting mappings with worse cost in the first iterations; this probability reduces slowly in later iterations.

\[
\text{nextTemperature}_{CSx} : \mathbb{R} \times \mathbb{N} \mapsto \mathbb{R}
\]

\[
\text{nextTemperature}_{CS1}(\text{temp}^0, \text{step}) = n^{\text{step}} \cdot \text{temp}^0 \quad \text{with typ.} \quad n \in [0.8, 1)
\]

\[
\text{nextTemperature}_{CS2}(\text{temp}^0, \text{step}) = \frac{\text{temp}^0}{\log_n(\text{step} + n)} \quad \text{with} \quad n > 1
\] (4.1)

Which cooling schedule is appropriate depends on the nature of design space and the local search function. If local extremities are likely even in later iterations, higher values of \( n \) perform better. For smoother design spaces smaller values of \( n \) should be chosen, because thereby the overall computation time is reduced. To determine the most appropriate cooling schedule theoretical or experimental analysis of the design space and the used search function is required.

4.3.3 Breeding of Mappings

**Representation of Mappings**

For each model element \( e \) a possibly empty set of mappings exists for each node, i.e. deployment location, \( dl \) with respect to a target design space \( DS_{n+1} \).

\[ e \in E - \text{mapped model element} \]

\[ DL = \{dl_i\} - \text{set of deployment locations of the target platform} \]

\[ dl \in DL - \text{deployment location of the mapped element} \]

\[ M_{e,dl} = \{m_{e,dl}\} - \text{set of mappings of element } e \text{ to node } dl \]

\[ M = \bigcup_{e \in E} \bigcup_{dl \in DL} M_{e,dl} - \text{set of all mappings} \]

A mapping \( m_{e,dl} \) is represented as 6-tuple:

\[ m_{e,dl} = (\text{subst}, \text{bind}, \text{sched}, \text{qos}, \text{smappings}, \text{act}) \]

\text{subst} \quad - \text{model element that substitutes } e \text{ in the implementation}

\text{bind} \quad = \{ri_i\} \subseteq \text{AI} \quad - \text{resource service instances bound by one model element instance}

\text{sched} \quad - \text{schedule for all instances of the model element/substitution}

\text{qos} \quad - \text{the QoS-characteristic of the mapping}

\text{smappings} \quad = \{m_{e_k,dl}\} \subseteq M_{e_k,dl} \quad - \text{set of mappings of all } e_k \in dchildren(e)

\text{act} \quad - \text{marks the mapping active/inactive } (\text{act} \in \{\text{true, false}\}, \text{default: false})

The \text{subst} field is used to integrate transformations into the mapping. If the \text{subst} field is not empty, the other fields are interpreted with respect to the substituting element. The substituting element and the substituted element must be functionally equivalent and preserve the model integrity. The binding (\text{bind}) and the eventual schedule (\text{schedule}) represent a local candidate design of the particular model element. Since
the TPM defines resource services rather than their instances, simulated instances are used to represent the binding of a model element instance. The reason to use instances instead of types is to reflect the actual resource service consumption of one model element instance directly. On the other hand, this approach requires all instances of a particular model element having the same implementation option. This is a drawback for targets with scarce resources. A QoS is associated with each mapping. The act field is used to activate a mapping. To enable the fast computation of new mappings it is advantageous to reflect the structure of the design space, which is mostly determined by the deployment hierarchy, in the hierarchy of mappings. The mapping hierarchy is organized through the smappings field. The field contains at least one mapping for each deployment child of the element:

\[ \forall e \in E : \forall e_i \in dchildren(e) : \exists \exists^1 m_{e_i,dl} : (\{m_{e_i,dl}\} \cap m_{e,dl} \cdot \text{smappings}) \neq \emptyset \]

All mappings define a set of DAGs. To simplify the algorithms, a single root mapping \( m_{mo} \) is introduced that serves as parent of all root mappings of the DAGs. The root mapping \( m_{mo} \) conceptually represents the mapping of the design model. This graph is called the mapping graph. In the following the important properties of mapping graphs are investigated.

The function \( m\text{parents} \) returns the set of parent mappings of a mapping in a mapping graph:

\[ m\text{parents} : M \mapsto P(M) \quad m\text{parents}(m_{e,dl}) = \{m_{e_i,dl} | m_{e,dl} \in m_{e_i,dl} \cdot \text{smappings} \} \]

An implementation platform may not be sufficient to realize a particular model element, e.g. because it does not offer sufficient resource services. If a model element is not implementable, its deployment parent is also not implementable on that implementation platform. Each model element must be implementable with at least one platform:

\[ \exists \exists m_{e,dl} \in M : e \in E \land dl \in DL \rightarrow e \text{ is not implementable on the target platform} \]

If a design contains at least one model element which is not implementable, this design is not implementable on the target platform. The implementability of a classifier additionally depends on the implementability of its generalizations. To capture inheritance relationships of classifiers two helper functions are defined:

\[ CL = \{cl_i\} \subseteq E - \text{set of classifiers} \]

\[ \text{generalizations} : CL \mapsto P(CL) \]

\[ \text{generalizations}(cl) = \{cl_i | cl_i \in CL \land cl_i \neq cl \land cl_i \text{ is a direct generalization of } cl \} \]

\[ \text{specializations} : CL \mapsto P(CL) \]

\[ \text{specializations}(cl) = \{cl_i | cl_i \in CL \land cl_i \neq cl \land cl_i \in \text{generalizations}(cl_i) \} \]

Then, in order to satisfy the object integrity constraint, a classifier \( cl \) is implementable on a node \( dl \) only if the following condition holds:

\[ M_{cl,dl} \neq \emptyset \land \forall cl_i \in \text{generalizations}(cl) : M_{cl_i,dl} \neq \emptyset \]

The activation of a mapping depends on the activation of its parent mappings. A mapping can be active only if itself is activated and at least one of its parent mappings is active:

\[ \text{isActive} : M \mapsto \{true, false\} \]

\[ \text{isActive}(m_{e,dl}) = \begin{cases} 
  true & \text{if } m_{e,dl} . \text{act} = true \land (m\text{parents}(m_{e,dl}) = \emptyset \lor \exists m_{e_p,dl} \in m\text{parents}(e) : \text{isActive}(m_{e_p,dl})) \lor m_{e,dl} = m_{mo} \\
  false & \text{else}
\end{cases} \]

In a mapping hierarchy with the root mapping \( m_{e,dl} \), at most one mapping can be active for a model element on a given node

\[ \forall m_{e,dl} \in M_{e,dl} : \forall e_i \in dchildren(e) : \exists \exists^1 m_{e_i,dl} \in m_{e,dl} \cdot \text{smappings} : \text{isActive}(m_{e_i,dl}) \]
while a mapping is only valid if
\[
\forall m_{e,dl} \in M : \forall e_i \in dchildren(e) : \exists^{1} m_{e_i,dl} \in m_{e,dl}.smappings : isActive(m_{e_i,dl})
\]
All active mappings in a mapping hierarchy comprise an activation graph, which, by definition, contains at most one active mapping for each element in a deployment hierarchy with respect to a particular node. This property is important because it enforces an unique implementation of each model element on a node.

**Breeding Algorithm**

The search of new mappings is the responsibility of the breeder component and must support the strategy employed by the controller. The DSE is modeled as constrained selection problem. Initially, for each mapped model element a set of candidate mappings is computed for each feasible node of the target platform. The mapping algorithm searches a selection of mappings that satisfies all constraints, i.e. both system constraints and architectural constraints, and minimizes the cost function.

**Initialization.** Algorithm 4.2 presents the initialization of the breeder component. The algorithm checks the implementability of the given design and computes an initial partition and candidate resource mappings for each model element. Components and classes are bound transitively to those deployment locations for which candidate resource mappings exist. In contrast to classes, the candidate mappings of components are computed without regarding their deployment children, which is controlled by the boolean parameter in the invocation. If the design does not define any component a default component is created and deployed on all nodes. Classes that are not realized by any component in the original design, are realized by all feasible components in the output. A mapping hierarchy is constructed whose roots are the candidate component mappings. To allow for a fast online estimation, all candidate mappings are pre-estimated. A consistent set of randomly activated mappings comprises the initial mapping of the design (Algorithm 4.3). The space and time complexity of the algorithm is \(O(|E||DL|)\), because at most one set of candidate mappings is computed for each model element on each node.

**Activation of Mappings.** Algorithm 4.3 presents a recursive algorithm for the activation of mappings in the mapping graph. Due to the definition of the function \(isActive\), the algorithm works lazily in that it performs (de-)activation mostly locally to a mapping with two exceptions. First, when a mapping is activated, then for each of its sub-mappings it is checked if there is an activated mapping. If not, a random mapping is activated. The second exception are classifiers. In order to enforce the object integrity constraint, the activation of classifier-mappings propagates upwards in the inheritance hierarchy, while deactivation propagates downwards.

**Example 4.2:** Fig. 4.5 exemplifies the propagation of mapping activity in classifier hierarchies. The upwards propagation of mapping activation is shown in Fig. 4.5(a). In response to activating mapping \(m_{C_2,dl}\), also the mappings \(m_{C_1,dl}\) and \(m_{C_0,dl}\) are activated. The deactivation of \(m_{C_0,dl}\), as shown in Fig. 4.5(b), triggers the deactivation of all mappings in this hierarchy on the particular deployment location \(dl\).

**Re-mapping of Model Elements.** Given the implementability of a design on a target platform, Algorithms 4.4 and 4.5 perform the local search. First, a mapping goal for the next search step is derived from the current mapping. The minimization of latency and area will be common goals. Then, in order to accomplish the goal, the current mapping is tried to re-map top-down. After each re-mapping step it is checked whether the mapping violates architectural constraints imposed by the chosen implementation options. If not so, all active mappings are applied to the current design and so a new design is computed. Otherwise, the resolution of the constraint violation is delegated to the respective node mapper component.

Re-mapping always starts at granularity G0 and passes on to G4, with some constant probability \(l_{thres}\), with typically \(l_{thres} \in [0.2, 0.6]\). Larger values of \(l_{thres}\) cause re-mapping to branch between relatively distant parts of the design space, but, in contrast to smaller values, the fine-adjustment capabilities of the
Algorithm 4.2: Breeder initialization - initialize($d_{n+1}^0$)

**Input**: Initial valid design: $d_{n+1}^0 \in D_{S_{n+1}}$.

**Output**: If the initialization is successful an initial yet complete PSM is returned, otherwise the algorithm returns $\emptyset$.

**Data**: Computes candidate mappings for all mapped model elements on all feasible nodes of the target platform.

$CO \leftarrow$ set of mapped components of $d_{n+1}^0$;

if $CO = \emptyset$ then $CO \leftarrow \{ \text{new component} \}$;

foreach $co \in CO$ do

$DL \leftarrow$ set of deployment locations of $co$;

if $DL = \emptyset$ then $DL \leftarrow$ set of all nodes;

foreach $dl \in DL$ do

if $M_{co,dl} = \emptyset$ then $M_{co,dl} \leftarrow$ $dl$.NodeMapper.computeCandidateMappings($co$, false);

if $M_{co,dl} \neq \emptyset$ then

deploy $co$ on $dl$;

$m_{co,dl}.smappings \leftarrow m_{co,dl}.smappings \cup M_{co,dl}$;

$CL \leftarrow$ set of mapped classes of $d_{n+1}^0$;

foreach $cl \in CL$ do

$CO \leftarrow$ set of components realizing $cl$;

if $CO = \emptyset$ then $CO \leftarrow$ set of all components;

foreach $co \in CO$ do

$DL \leftarrow$ set of deployment locations of $co$;

foreach $dl \in DL$ do

if $M_{cl,dl} = \emptyset$ then $M_{cl,dl} \leftarrow$ $dl$.NodeMapper.computeCandidateMappings($cl$, true);

if $M_{cl,dl} \neq \emptyset$ then

foreach $m_{co,dl} \in M_{cl,dl}$ do

realize $cl$ by $co$;

$m_{co,dl}.smappings \leftarrow m_{co,dl}.smappings \cup M_{cl,dl}$;

if $\exists e \in E : e$ is not implementable then return $\emptyset$;

foreach $m_{e,dl} \in M$ do

$m_{e,dl}.qos \leftarrow$ $dl$.NodeEstimator.estimateQoS($m_{e,dl}$);

$CO \leftarrow$ set of mapped components of $d_{n+1}^0$;

foreach $co \in CO$ do

$DL \leftarrow$ set of deployment locations of $co$;

foreach $dl \in DL$ do

$m_{co,dl} \leftarrow$ random selection from $M_{co,dl}$;

setActivity($m_{co,dl}$, true);

$d_{n+1}^0 \leftarrow$ applyMappings($d_{n+1}^0$, $\bigcup_{e \in E} \bigcup_{dl \in DL} m_{e,dl}$ : isActive($m_{e,dl}$));

return $d_{n+1}^0$.
Algorithm 4.3: Set the activity of mappings - \texttt{setActivity}(m_{e,dl}, activity)

\begin{itemize}
\item \textbf{Input}: Mapping whose activity is to be set: \(m_{e,dl}\). Activity of the mapping: \(activity\).
\item \textbf{Data}: (De-)activates mappings in a mapping-DAG. For classifiers the satisfaction of the object integrity constraint is enforced.
\item \textbf{if} \(m_{e,dl}.act \neq activity\) \textbf{then}
\item \hspace{1em} \(m_{e,dl}.act \leftarrow activity\);
\item \textbf{if} \(activity = true\) \textbf{then}
\item \hspace{2em} \textbf{foreach} \(e_i \in \text{dechildren}(e)\) \textbf{do}
\item \hspace{3em} \textbf{if} \(\exists m_{e,dl} \in m_{e,dl}.smappings : \text{isActive}(m_{e,dl})\) \textbf{then}
\item \hspace{4em} \(m_{e,dl} \leftarrow \text{select random mapping from } m_{e,dl}.smappings\);
\item \hspace{4em} \texttt{setActivity}(m_{e,dl}, true);
\item \textbf{if} \(e\) is a classifier \textbf{then}
\item \hspace{2em} \textbf{if} \(activity = true\) \textbf{then} \(E_i \leftarrow \text{generalizations}(e)\) \textbf{else} \(E_i \leftarrow \text{specializations}(e)\);
\item \hspace{2em} \textbf{foreach} \(e_i \in E_i\) \textbf{do}
\item \hspace{3em} \(m_{e,dl} \leftarrow m_{e,dl} \in M_{e,dl} : \text{isActive}(m_{e,dl}) = true\);
\item \hspace{3em} \textbf{if} \(m_{e,dl} = \emptyset\) \textbf{then} \(m_{e,dl} \leftarrow \text{select random mapping from } M_{e,dl}\);
\item \hspace{3em} \texttt{setActivity}(m_{e,dl}, activity);
\end{itemize}

Algorithm 4.4: Breeding of mappings - \texttt{breed}(d^{step-1}_{n+1})

\begin{itemize}
\item \textbf{Input}: Current design: \(d^{step-1}_{n+1}\).
\item \textbf{Output}: Returns candidate design.
\item \(\texttt{qos} \leftarrow \text{estimateQoS}(d^{step-1}_{n+1}); \text{goal} \leftarrow \text{deriveGoal}(\text{qos}); \text{remap}(\text{mo}, \text{goal})\);
\item \textbf{if} \(\text{violatesConstraint}(\text{mo}) = false\) \textbf{then}
\item \hspace{1em} \(m_{e,dl} \leftarrow \text{get mapping that causes the constraint violation}\);
\item \hspace{1em} \text{dl.NodeMapper.resolveConstraintViolation}(m_{e,dl});
\item \textbf{return} \(\text{applyMappings}(d^{step-1}_{n+1}, \bigcup_{e \in E} \text{dl}, \text{DL} m_{e,dl} : \text{isActive}(m_{e,dl}))\);
\end{itemize}

Algorithm 4.5: Re-mapping of an element - \texttt{remap}(m_{e,dl}, goal, lthres, rthres)

\begin{itemize}
\item \textbf{Input}: Current mapping of an element \(e\) on node \(dl\): \(m_{e,dl}\). The optimization goal: \(goal\). Probability of re-mapping child elements \(lthres\). Re-activation probability \(rthres\).
\item \textbf{Output}: Returns true if re-mapping was successful, false is returned otherwise.
\item \textbf{if} \(m_{e,dl}.smappings = \emptyset\) \textbf{then return} false;
\item \hspace{1em} done \leftarrow false;
\item \textbf{while} done = false \textbf{do}
\item \hspace{2em} \(m_{e,dl} \leftarrow \text{selectMapping}(m_{e,dl}.smappings, goal)\);
\item \hspace{2em} \texttt{reactivate} \leftarrow false;
\item \hspace{2em} \textbf{if} isActive\((m_{e,dl})\) \land \(e_i\) is a classifier \textbf{then}
\item \hspace{3em} \texttt{reactivate} \leftarrow \exists m_{e,dl_j} \backslash \text{dl} \neq dl : \text{isActive}(m_{e,dl}) \vee \text{random}() < rthres;
\item \hspace{3em} \(m_{e,dl} \leftarrow \text{selectMapping}\{m_{e,dl} | m_{e,dl} \in m_{e,dl}.smappings\}, \text{goal}\);
\item \hspace{3em} \texttt{setActivity}(m_{e,dl}, false), done \leftarrow true;
\item \hspace{2em} \textbf{if} \texttt{reactivate} = true \textbf{then}
\item \hspace{3em} \(m_{e,dl} \leftarrow \text{select active mapping of } e_i\) from \(m_{e,dl}.smappings\);
\item \hspace{3em} \textbf{if} \(m_{e,dl} \neq \emptyset\) \textbf{then} \texttt{setActivity}(m_{e,dl}, false);
\item \hspace{3em} \texttt{setActivity}(m_{e,dl}, true), done \leftarrow false \text{\&} m_{e,dl} \neq m_{e,dl};
\item \hspace{3em} \textbf{if} done = false \text{\&} random() < \text{lthres} \textbf{then} done \leftarrow done \text{\&} \texttt{remap}(m_{e,dl}, goal);
\item \textbf{return} done;
\end{itemize}
4.3. Platform Mapping Algorithms

mapping quality are inferior. Smaller values require a larger number of steps to traverse reasonable parts of the design space. Future extensions of this approach should examine if a dynamically determined probability gains further improvements.

For each model element one mapping can be active at each node. This is an important extension to former approaches that allow for at most one active mapping of an element altogether. It is based on the observation that communication latency often dominates computation latency. If a model element performing some computation is accessed by various model elements that are not executed on the same node communication latency is inevitable. The execution of the critical element on all feasible nodes, whereby each accessor uses only the local implementation, reduces the overall latency at higher area cost.

The presented approach integrates the partitioning of objects. Partitioning is based on objects mainly for two reasons. First, objects are a fundamental source of concurrence. The object-based model of computation maps well to RTR-architectures. Second, objects encapsulate data and behavior to process the data. Objects migrate between nodes by means of their classifiers. If an active classifier mapping is encountered and chosen to re-map directly, the active mapping is deactivated and only re-activated if there is no other active mapping of the classifier, or with some constant probability \( r_{\text{thres}} \in [0.2, 0.7] \). Smaller values of \( r_{\text{thres}} \) steer mappings toward lower area consumption, while higher values tend to reduce latency. This approach integrates group migration into a resource-based mapping framework.

**Example 4.3:** The re-mapping of classifiers is demonstrated in Fig. 4.6 for the classifiers in the hierarchy shown in the previous example 4.2. Apart from classifier \( C_3 \), for all classifiers candidate mappings exist on the two deployment locations \( h_0 \) and \( h_1 \). The boxes represent the mappings, whereas for simplicity there is just one mapping on each location. In Fig. 4.6(b) the classifier \( C_2 \) is re-mapped from \( h_0 \) to \( h_1 \) since \( \text{random()} > r_{\text{thres}} \). This can be seen as moving \( C_2 \) from \( h_0 \) to \( h_1 \). If \( \text{random()} \geq r_{\text{thres}} \) the classifier is re-mapped to both locations, i.e. it is copied to \( h_1 \).

**Selection of Mappings.** Mappings can be chosen from the neighborhood of a mapping completely at random, cherishing hope that a chosen mapping helps improving the goal. On the other hand, a deterministic selection would undermine the foundation of SA. Algorithm 4.6 presents a Monte-Carlo selection approach that combines the advantages of both approaches. The mapping is selected from a randomly chosen subset.
of the given set of mappings \( M_k \), that satisfies the current goal best. The subset has cardinality \( \lceil c |M_k| \rceil \), whereas \( c \in [0.3, 0.7] \). Smaller values of \( c \) mean higher randomization and less computation time, while larger values increase the probability that the selected mapping is among the best \( c \cdot 100\% \) mappings regarding the goal.

<table>
<thead>
<tr>
<th>Algorithm 4.6: Select a mapping - <code>selectMapping(M, goal)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong> : Set of mappings: ( M_k \subseteq M ).</td>
</tr>
<tr>
<td><strong>Output</strong>: Returns a mapping ( m \in M_k ).</td>
</tr>
<tr>
<td><strong>Data</strong>: Constant value ( c ) that determines the number of selections: ( c )</td>
</tr>
<tr>
<td>( m \leftarrow \emptyset ), ( i \leftarrow 0 );</td>
</tr>
<tr>
<td>repeat ( m_i \leftarrow ) select random mapping from ( M_k );</td>
</tr>
<tr>
<td>( m \leftarrow ) m_{\text{eq}} ) if ( m_i \text{ satisfies goal better than } m ) or ( m = \emptyset ) then ( m \leftarrow m_i );</td>
</tr>
<tr>
<td>( i \leftarrow i + 1 );</td>
</tr>
<tr>
<td>until ( i &lt; \lceil c</td>
</tr>
<tr>
<td>return ( m );</td>
</tr>
</tbody>
</table>

To rank mappings a multitude of feasible approaches exist. If mappings of a single model element are considered, implementation characteristics, such as area, latency, or power dissipation, are used. However, if mappings of multiple model elements are considered concurrently and not related to the actual execution of the respective element, these characteristics can be quite meaningless. For instance, if an action will be executed seldomly a latency of several milliseconds can be acceptable, while the same latency may be intolerable if the action will be executed by an inner loop of a system. Thus, the effort is focused towards the highly utilized parts of a system, as designers commonly do in manual DSE-approaches as well.

**DEFINITION 4.7:** The metric utilization is defined as:

\[
\text{utilization} : E \rightarrow \mathbb{R} \\
\text{utilization}(e) = p_e \cdot f_e \\
p_e - \text{execution probability of element } e \\
f_e - \text{execution frequency of element } e
\]

Both values, \( p_e \) and \( f_e \), are implementation-independent characteristics of model elements. They are estimated before platform mapping (\( \rightarrow \) Section 4.4). Then, if the goal is the minimization of a value \( v_i \), of any two mappings \( m_{e_i,d_i} \), \( m_{e_k,d_k} \) mapping \( m_{e_i,d_i} \) satisfies the goal better than if

\[
\frac{m_{e_i,d_i}, \text{utilization}(e_i)}{m_{e_k,d_k}, \text{utilization}(e_k)} < \frac{m_{e_k,d_k}, \text{utilization}(e_k)}{m_{e_k,d_k}, \text{utilization}(e_k)} \quad \text{with} \quad v_i \in \{ \text{area, latency, power} \}.
\]

This condition defines a partial order \( m_{e_0,d_{i_0}} \preceq m_{e_1,d_{i_1}} \preceq \cdots \preceq m_{e_m,d_{i_m}} \). Notice that, in absence of an universal definition of area, this order is only applicable to areas that are comparable. If not, the order is undefined and the algorithm selects a random mapping.

**Resolution of Architectural Constraint Violations.** The breeded mappings might violate architectural constraints. Architecture constraint violations arise mainly due to resource over-consumption (QoS-constraints) or the breakage of constraints, which are imposed by the chosen implementation options. Mappings violating architectural constraints are invalid. The constraint violations must be resolved, whereas the following basic strategies are feasible:

- **Avoid** - Violation avoidance avoids to compute invalid mappings. This strategy is hard to realize due to the locality of re-mapping. The combination of valid platform-specific mappings does not necessarily result in valid model mappings.

- **Rollback** - Rollback resolution restarts mapping with the last valid mapping when the current mapping is evaluated to be invalid.
4.3. Platform Mapping Algorithms

**Neglect** - This head-in-the-sand strategy neglects constraint violations temporarily and keeps on re-mapping, cherishing hope that the violation will resolve in future mappings.

**New** - This strategy does not perform any conflict resolution. Instead, a new mapping is computed, which itself may violate architectural constraints.

All strategies have specific advantages and drawbacks in terms of space/time complexity, the ability to overcome design space discontinuities, and the algorithmic complexity of the resolution function. The presented approach uses a combination of conflict avoidance and the computation of new mappings. Platform-specific mappers try avoiding violations by computing mappings that are widely independent of other mappings. If a constraint violation occurs in spite of that, the conflict is resolved by performing a local or global re-mapping. On the platform-independent level constraints are enforced by the structure of the mapping graph and the employed algorithms.

4.3.4 Computation of Candidate Mappings

As has been previously shown, mapping graphs are used to represent and organize candidate mappings. All platform-specific details are delegated into specialized components for mapping and estimation. The computation of candidate mappings is highly platform-specific, as one would expect.

Allocation and Binding

It is an important goal of DSE, to allocate only the minimum amount of resource service instances that is necessary to execute a system. This is accomplished with optimized implementation options and sharing of resource service instances. In this context, the concept of sharing is inherently instance-based. Recall, however, that the PSM and the TPM are type-based. The binding in the mapping graph uses simulated resource service instances. Allocation and binding of resource services are done locally using a constructive, greedy approach. A simulated resource service instance is created, allocated, and bound whenever it is required to realize a model element instance. Sharing is factored in during estimation. The actual amount of required resource service instances is assessed and then reflected in the respective qos-field in the mapping graph.

Algorithm 4.7 presents a generic approach to the computation of candidate mappings. The algorithm performs allocation implicitly with binding and scheduling. It is specialized to the platform through the functions bind and schedule. For each model element it is checked if an model transformation is required, and if so, the transformation is applied to the element. In case a transformation is not applicable in the certain context or no appropriate transformation exists the element is not implementable with the platform. The context of the applicable transformations is local to the element. Transformations must not have side effects that interfere with mappings of other elements. Global transformations can be applied by choosing the appropriate level in the model hierarchy. Since a model element is only implementable if all its children are implementable, the algorithm fails if no mappings have been computed for some child. The binding of an element may be empty without affecting the implementability of the element. Examples are components and classes that do not bind any local resources, if mapped to software. In this case an empty mapping is created for the element. Binding and scheduling are closely intertwined. Although they can be performed together it is common to handle them separately. Some bindings, e.g. of multiplexers and temporary registers in data-paths, are performed after scheduling, which is done by the function bindPostSchedule.

Scheduling

Static schedules are computable for model element instances whose behavior and/or lifetime is exactly known at compile-time. This is the case for all model elements in the deployment hierarchy below and including operations. Scheduling is crucial to the QoS of a system, because it determines a tradeoff between area, power, and performance. The investigation of approaches to scheduling is far beyond the scope of this
Algorithm 4.7: Compute candidate mappings - computeCandidateMappings (e, recurse)

**Input**: Model element to map: $e \in E$. Recursion flag, set true if also mappings of deployment children are computed: $\text{recurse} \in \{\text{true, false}\}$.

**Output**: Returns candidate mappings for the element and the deployment location $M_{e,dl}$.

1. $\text{subst} \leftarrow e$;
2. if $\text{requiresTransformation} (\text{subst}) = \text{true}$ then
   1. $\text{subst} \leftarrow \text{applyTransformation} (\text{subst})$;
   2. if $\text{subst} = \emptyset$ then return $\emptyset$;
   3. $M_{e,dl} \leftarrow \text{bind} (\text{subst})$;
3. if $M_{e,dl} = \emptyset$ then $M_{e,dl} \leftarrow \{(e, \emptyset, \emptyset, \emptyset, \emptyset, \text{false})\}$;
4. foreach $m_{e,dl} \in M_{e,dl}$ do
   1. $m_{e,dl}.\text{subst} = \text{subst}$;
   2. if $\text{recurse} = \text{true}$ then foreach $e_c \in \text{dchildren} (e)$ do
      1. $M_{e_c,dl} \leftarrow \text{computeCandidateMappings} (e_c, \text{true})$;
   3. if $M_{e_c,dl} = \emptyset$ then return $\emptyset$;
   4. $m_{e_c,dl}.\text{smappings} \leftarrow m_{e,dl}.\text{smappings} \cup M_{e_c,dl}$;
   5. $m_{e,dl}.\text{sched} \leftarrow \text{schedule} (m_{e,dl})$;
   6. $\text{bindPostSchedule} (m_{e,dl})$;
5. return $M_{e,dl}$;

thesis. Scheduling is performed local to basic blocks using a basic scheduling strategy. The strategy can be selected and parameterized per behavior, using mapping constraints. Multi-cycle operations and operation chaining are supported.

In addition to static scheduling, model element instances are scheduled dynamically. This is the case for both, up-based implementations and FSMD-based implementations. Instructions are dynamically bound to a CPU per cycle. Software objects are bound to memory- and communication resources. The same applies for functionality that is executed using RFs. Here, the scheduling and dynamic binding usually happens at much coarser time frames, in the range of milliseconds to seconds, or just once when the execution is started. Dynamic scheduling is delegated to the execution environment (→ Chapter 6).

**Resource Service Selection**

For each model element a multitude of implementation options exist, whereas each implementation option is realized with a specific binding of resource services. These resource services are selected from the available options that have been modeled in the TPM.

**Type Mapping.** All model elements are realized with services offered by implementation types. Designs are composed entirely from design-platform types. The possible implementations of each design type are defined by its realization graph (→ Definition 3.12 on page 45). The implementation platform model defines the realization paths of design-platform types. For non-design platform types type mapping is performed implicitly during platform mapping. Candidate mappings are computed by Algorithm 4.7, if it is invoked with a type as input. The type mapping being used in the final implementation is selected from the candidate mappings.

**Implementation Component Selection.** The predefined type mappings are the foundation of resource service selection. Depending on the implementation option, more resource services may be required than those which are obvious from the design. For instance, in FSMD-based implementations classes must contain a register to store the object type, in order to realize the dynamic message dispatch. Implementation platform models specify proxies for such building blocks by means of implementation components. The predefined semantic of these elements is reflected in the model using stereotypes and interfaces, which are used to search a particular implementation. Algorithm 4.8 presents the search strategy. Since an implementation
component can be realized by multiple classifiers the algorithm chooses the realization with the least cost. As cost metric the implementation area or the area/latency-ratio is used. The function \textit{findMSO} is defined in Algorithm 3.1 on page 40.

\begin{algorithm}
\caption{Find implementation type - \textit{findImplementationType}(\textit{platform}, \textit{cstereos}, \textit{rstereos}, \textit{interface})}
\begin{algorithmic}
\REQUIRE Implementation platform: \textit{platform}, Implementation component stereotypes: \textit{cstereos}, Implementation type stereotypes: \textit{rstereos}, Implementation type interface: \textit{interface} = \{\textit{op}\_i\}, \textit{op} = \langle \text{name}, \text{ArgList} \rangle.
\ENSURE Returns the implementation type in implementation platform \textit{platform}, that realizes a component with the stereotypes \textit{cstereos}, has stereotypes \textit{rstereos}, implements \textit{interface}, and has least cost.
\STATE \textit{result} $\leftarrow \emptyset$;
\STATE \textit{CO} $\leftarrow \text{getComponents}(\textit{platform})$;
\FOR{\textit{co} $\in$ \textit{CO}}
\IF{\textit{co} has all stereotypes \textit{cstereos}}
\STATE \textit{CL} $\leftarrow \text{getRealizations}(\textit{co})$;
\FOR{\textit{cl} $\in$ \textit{CL}}
\IF{\textit{cl} has all stereotypes \textit{rstereos}}
\STATE \textit{success} $\leftarrow$ true;
\FOR{\textit{op} $\in$ \textit{interface}}
\STATE \textit{msop} $\leftarrow \text{findMSO}(\textit{cl}, \text{op}\.\text{name}, \text{op}\.\text{ArgList})$;
\ENDIF
\IF{\textit{msop} $= \emptyset$} \textit{success} $\leftarrow$ false;
\ENDIF
\ESAC
\IF{\textit{success} $= \text{true} \land \text{result} = \emptyset \lor \text{cost}(\textit{cl}) < \text{cost}(\text{result})$}
\STATE \textit{result} $\leftarrow \textit{cl}$;
\ENDIF
\ENDFOR
\ENDIF
\ENDFOR
\ENDFOR
\RETURN \textit{result};
\end{algorithmic}
\end{algorithm}

\textit{Behavior Selection}. In UML, actions are the fundamental units of behavior. As has been shown in Section 3.3.5, each action is mapped to an operation of an implementation type. The implementation type of the model elements on which an action operates is determined by the type mapping. The respective most specific operation is searched in this implementation type using Algorithm 3.1. The most specific operation is used to implement the action. Hence, the mapping of actions to operations is one-to-one. This constrains the possible mappings and thereby reduces the potential of implementation-dependent optimizations. Future extensions should overcome this restriction by adopting techniques being employed in technology mapping of hardware design flows and software code generation (\textit{\rightarrow} Section 2.2.3) [240]. Operations merely define the interface to the behavior that actually implements the action. An operation is implemented by at least one behavior. Each behavior corresponds to a point in the local design space of the operation. One of the available behaviors is selected and bound by the action, which corresponds to the module selection problem [241]. Due to the lack of mature UML design tools, that actually allow modeling multiple behaviors per operation, behavior selection has not been investigated in detail. One-to-one mappings of operations to behaviors are assumed.

\textit{Model Transformations}

Platform mapping is accompanied by behavior preserving model transformations. Transformations serve different purposes, they

- increase the explored part of the design space,
- optimize designs to get better implementations,
- adapt model elements to resource services, and
- apply mappings to the design.
To address this wide range of applications, a toolbox of transformations is defined. The transformations can be performed automatically or manually. Due to the common representation using UML, each automatic transformation can be applied manually as well. Thereby primitive transformations and optimizations are distinguished. Primitive transformations are mostly used when a mapping is applied to the model. Moreover, primitive transformations comprise the infrastructure for optimization transformations. For instance, the decomposition of a behavior into multiple behaviors, each of which is executed by an operation, requires a sequence of primitive transformations. Tab. C.1 in Appendix C.1 shows the primitive transformations implemented in MOCCA.

Optimizations are transformations that aim at improving the QoS of one or more QoS-values. Automatic transformations execute while candidate mappings are computed. To improve the QoS of the implementation, designers can perform a number of transformations manually. The most automatic transformations are parameterizable by means of optimization constraints. Optimizations are applied to the model until no more improvement is gained, or a maximum number of optimization passes is reached. Technology independent optimizations gain improvements regardless of the final implementation and are performed before the actual platform mapping step. Technology dependent optimizations are applied during platform mapping individually per implementation platform. See Tab. C.2 and C.3 for a description of the optimizations implemented in MOCCA.

4.3.5 Mapping Evaluation

Mapping evaluation is the process of quality assessment and testing if mappings satisfy the architectural constraints. Both tasks are based on the QoS of the active mappings, which is reflected in the mapping graph by the qos-field of each mapping. Given by the recursive definition of this graph, constraint evaluation and cost functions can be defined locally for each model element in principle. Although this would give designers a fine-grained control over the implementation, it required a fair amount of modeling. Also the compile-time is likely to increase significantly. Local constraints and cost functions are likely to be required in embedded systems and real-time systems. For the purpose of the applications envisaged by this thesis constraints are evaluated globally per deployment location. The cost function is computed for the entire model.

Constraint Evaluation

For each deployment location the TPM can define QoS-constraints. These constraints are compared with the QoS computed for the components deployed on this location. If some QoS-value exceeds its constraint, the mapping is invalid and a respective conflict resolution is started.

Cost Function

The value of the cost function is only defined if the mapping meets all mapping constraints on all deployment locations. In continuation of Definition 2.6 in the presented approach the cost function is defined as:

$$cost = \sum_{d \in DL} \sum_{v_i \in qos_{dl}} w_i \cdot v_i \cdot unit(v_i)^{-1}$$

$$w_i \in \mathbb{R} - \text{weights for the QoS-values } v_i$$

The ATP-characteristics are used as QoS-values. Other system properties may be integrated as well. The QoS-values are defined for each individual deployment location. This is important especially for area values, since there is no uniform definition of this property. The weights allow designers to prioritize individual QoS-values. The values are estimated by node specific estimator components of the model compiler. The estimates are based on the QoS-constraints defined in the TPM for the provided resources services.
4.4 Estimation of Model Characteristics

4.4.1 Estimation of Execution Characteristics

**Definition 4.8:** Execution characteristics are intrinsic properties of a design and its input data, that are invariant over all implementations of the design.

Execution characteristics are distinguished from implementation characteristics, such as the ATP-characteristics, because the latter are likely to change when the implementation changes. Clearly, there are many execution characteristics one can be interested in. For the purpose of this thesis, the following characteristics are considered:

- probability and frequency of execution of activities, activity groups, and actions,
- maximum number of concurrent instances of a model element, and
- maximum size of instances of a model element.

Different execution probabilities of behavioral elements are caused by the existence of conditionally executed behavior in a design model. Conditional behavior evokes from all activities that change control-flow, namely branches, loops, and exceptions. Varying execution frequencies are caused by the existence of loops and recursive messages in design models. The latter two characteristics directly refer to the challenges of object-orientation as discussed in Section 4.1.2, particularly to dynamic object lifetime and dynamic object size.

The probability and frequency are used to guide implementation effort, e.g. in terms of utilization (→ Definition 4.7). For these characteristics estimates are commonly acceptable as long as the estimation error is not too high. In contrast, the number of instances and object size directly control the binding of resource service instances, and synthesis. If any of these values is under-estimated run-time errors will occur. Over-estimation causes resource over-consumption and may render a design unimplementable. All characteristics depend on the input data.

For these reasons designers are allowed to define these characteristics using execution-constraints (→ Section 3.3.3). Thereby the definition of the number of concurrent instances and the instance size of a model element is mandatory in the sense that infinite values are assumed otherwise. Both values are considered during the QoS estimation. Hence, if a deployment location defines an according QoS constraint, no mapping can meet this constraint. Consequently, the design is not implementable on this target platform. In general, the existence of references makes lifetime analysis, and thereby the exact determination of the maximum number of concurrent instances of some model element, intractable [153, 229]. Designers must provide upper bounds of the number of instances.

**Estimation of Execution Probabilities and Frequencies**

The execution probability and frequency can be defined manually, but this obviously requires a large amount of modeling. Thus, these values can be estimated from the model automatically by means of dynamic profiling. An implementation of the design is executed several times using representative input data. For each considered behavior the number of executions is counted. After execution these counts are automatically back-annotated to the according model elements. For this, the execution constraints are used. From the execution counts the frequency is directly given. The probability is derived statistically. The accuracy of this approach depends on the chosen input data. The selection of representative input is not always straightforward and requires additional human effort and tool support. Since profiling typically delivers sufficient accuracy it is currently preferred for most design flows.

The development flow with profiling is illustrated in Fig. 4.7. If profiling is used MOCCA generates an instrumented implementation, which is for simplicity software. For each block a counter is inserted into the code. Also the generated program contains methods to output the counters to a XML-file at the end of the
execution. The profile is back-annotated to the according design model automatically and reflected using execution constraints.

In literature also static approaches to the automatic estimation, that are based on branch prediction and control-flow analysis, are discussed [242–247]. Due to its background in the software domain branch prediction defines characteristics on instructions of ISAs. Simple patterns and predefined probabilities, that are based on observations of a corpus of programs, are used to predict the probability that a conditional branch is taken. It was shown by Drost that the focus on ISAs renders these patterns not appropriate to the system-level [248]. Apart from profiling the automatic approaches to estimate the execution frequency of behavior are very few. The analysis of loops, recursion, and dynamic function calls, such as it is caused by inclusion polymorphism, is equivalent to the halting problem [124]. So commonly the existence of these features is prohibited. Only the iterations of loops with statically known boundaries are estimated correctly. For all other loops and recursions a typical number of iterations between 5 and 12 is assumed [229].

4.4.2 Estimation of Implementation Characteristics

**Definition 4.9:** Implementation characteristics are properties of a system that vary depending on its implementation. These properties are determined by the design and the target platform.

Implementation characteristics can refer directly to physical properties, such as ATP, and non-functional properties, like reliability and robustness. The estimation presented in this thesis is different from the present approaches in that it is performed after a mapping is computed (→ Section 2.1.2 on page 12). This has the important consequence that the scheduling and resource consumption are known to estimation. Also, the presented approach incorporates the estimation of multiplexers in FSMD-implementations, which is considered important, since these elements represent a significant amount of the overall area. Another important difference is that this approach is complete in that it supports the estimation of the entire system rather than just behaviors.

Due to the challenges of software estimation discussed in Section 2.1.2, the estimation of software execution characteristics is an open problem. This situation is unsatisfactory since platform mapping and particularly automatic partitioning depend on robust and reasonably accurate information on the software timing characteristics. Thus, in the following the estimation of software implementation characteristics and automatic partitioning are not considered any further. The presented approach merely discusses the estimation of FSMD implementations. Software estimation may be integrated by future extensions.

For unpartitioned elements the above described communication-oriented approach is used by default. That is, each unpartitioned element is deployed on all nodes for which implementations exist. This default partitioning can be overridden manually. Thereby system architects can integrate experience and detailed knowledge of the hardware platform. As will be shown in experimental results, relatively fast compilation times support the investigation of different partitions swiftly. Thereby the presented approach provides a powerful tool for the improvement of the overall design quality.

In reconfigurable architectures in particular the timing and area characteristics of those parts of the system are of particular interest which will be executed by the reconfigurable logic. Estimated latencies are
frequently used because their measurement is commonly a costly task that may require manual hardware
design effort. Area estimates are used by platform mapping in order to recognize resource constraint violations.
Estimates are computed by evaluating the QoS-characteristics of the mappings.

In the presented approach, the QoS-evaluation is performed bottom-up for mappings in the activation graph.
Estimation therefore reflects the deployment hierarchy of the model. To describe the estimation in a uniform
manner, merge-functions, which are denoted by \( \phi \), are used. Merge-functions combine the values of multi-
ple input QoS-characteristics. Since the values of a characteristic must be treated separately a projection is
defined that applies a merge-function to all QoS-values having some label \( i \)

\[
\phi : i \times \text{QoS}_0 \times \cdots \times \text{QoS}_n \mapsto v_i
\]

whereas \( v_i \) is a QoS-value. Primitive merge-functions are independent from particular model elements.
Common primitive \( \phi_{op} \), with \( op \) identifying the operator, are

\[
\phi_{\text{sum}}(i, \text{QV}) = \sum_{qos_j \in \text{QV}} qos_j \cdot v_i \\
\phi_{\text{avg}}(i, \text{QV}) = \frac{1}{|\text{QV}|} \sum_{qos_j \in \text{QV}} qos_j \cdot v_i
\]

with \( \text{QV} = \{ \text{qos}_j \} \) representing a set of QoS-characteristics. If some characteristic does not define a value
\( i \), the result of the function is not determined for that specific value, which is denoted by \( \emptyset \). In this thesis,
the discussion is restricted to maximum area and latency, whereby these values are abbreviated \( a_{max} \) and
\( t_{max} \), i.e. maximum area and worst case execution time, respectively. Other values can be integrated into
this framework in a straightforward manner.

Tab. 4.2-4.3 present the QoS-estimation for the various model element types. The presentation regards the
inheritance of model elements in the UML meta-model. To compute the QoS of a specific model element the
estimation function for the (parent) type with the least distance is used. Due to the large number of different
model elements to be considered, \( \phi_{\text{sum}} \) is used for all elements and values for which no other function is
explicitly defined. In general, the QoS of a model element depends on its locally bound resource service
instances. The QoS of a resource service instance is determined by the QoS-constraint of the respective
resource service \( r_i \in R_{n+1} \), which reflects the quality characteristics \( Q_j \) of the used implementation option
\( \alpha_j = \{ P_j, Q_j, C_j \} \in O_{n+1} \) (\( \rightarrow \) Definition 2.2 on page 10), that is: \( \text{qos}(r_i) = Q_j \). If a binding comprises
multiple resource service instances, the overall QoS is computed for FSMD implementations as

\[
\text{qos}\{\{ r_i \} \} = \langle \phi_{\text{sum}}(a_{max}, \{ \text{qos}(r_i) \}), \phi_{\text{max}}(t_{max}, \{ \text{qos}(r_i) \}) \rangle
\]

In case a schedule is defined for a model element, this schedule determines the estimated latency. If the
element is no leaf in the deployment hierarchy, its QoS is also determined by the QoS of the respective
deployment children.

Tab. 4.2: QoS-Estimation of Structural Elements of FSMD-Implementations

<table>
<thead>
<tr>
<th>( e \in \mathcal{E} ) is ...</th>
<th>QoS-Estimation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>( \text{qos}(e) = \langle \phi_{\text{sum}}(a_{max}, \text{QV}<em>j \cup m</em>{e,dt}.\text{bind}), \emptyset \rangle )</td>
</tr>
<tr>
<td></td>
<td>( QV_j = \bigcup_{e \in \text{classifiers}(e)} \bigcup_{f \in \text{features}(e)} \text{instances}(e, c) \cdot \text{qos}(f) )</td>
</tr>
<tr>
<td></td>
<td>( \text{classifiers}(e) = \text{dechildren}(e) - \text{classifiers realizing } e )</td>
</tr>
<tr>
<td></td>
<td>( \text{features}(c) = \text{dechildren}(c) - \text{local and inherited features of } c )</td>
</tr>
<tr>
<td></td>
<td>( \text{instances}(c, e) - \text{instances of } c \in e )</td>
</tr>
<tr>
<td>Classifier</td>
<td>( \text{qos}(e) = \langle \phi_{\text{sum}}(a_{max}, \text{QV}), \emptyset \rangle )</td>
</tr>
<tr>
<td></td>
<td>( QV = \text{qos}(m_{e,at}.\text{bind}) \cup \bigcup_{f \in \text{features}(e)} \text{qos}(f) )</td>
</tr>
<tr>
<td></td>
<td>( \text{features}(e) = \text{dechildren}(e) - \text{local and inherited features of } e )</td>
</tr>
<tr>
<td>Attribute, Variable, Parameter</td>
<td>( m_{e,dt}.\text{qos} = \langle \phi_{\text{sum}}(a_{max}, \text{qos}(m_{e,dt}.\text{bind})), \emptyset \rangle )</td>
</tr>
</tbody>
</table>
The area estimates of FSMD-implementations are computed by the merge function $\phi_{\text{share}}$. This function accounts for the potential sharing of resources service instances among model elements with mutual exclusive execution. In principle, FSMD-implementations can share functional-units, memories, and communi-

| Tab. 4.3: QoS-Estimation of Behavioral Elements of FSMD-Implementations |
|-----------------------------|-----------------------------|
| $e \in E$ is ...            | QoS-Estimation              |
| **Operation**               |                             |
| $qos(e) = \langle \phi_{\text{sum}}(a_{\text{max}}, QV), \phi_{\text{sum}}(l_{\text{max}}, QV) \rangle$ |                             |
| $QV = qos(m_{e, \text{bind}}) \cup qos(method(e)) \cup \bigcup_{p \in \text{parameters}(e)} qos(p)$ |                             |
| $\text{method}(e)$ - method of $e$, must be an activity |                             |
| $\text{parameters}(e)$ - parameters of $e$ |                             |
| $\text{method}(e) \cup \text{parameters}(e) = \text{dchildren}(e)$ |                             |
| **Activity**                |                             |
| $qos(e) = \langle \phi_{\text{share}}(a_{\text{max}}, QV), \phi_{\text{sum}}(l_{\text{max}}, QV) \rangle$ |                             |
| $QV = qos(m_{e, \text{bind}}) \cup \bigcup_{a \in \text{anodes}(e)} qos(a) \cup \bigcup_{v \in \text{variables}(e)} qos(v)$ |                             |
| $\text{anodes}(e)$ - activity nodes owned by $e$ |                             |
| $\text{variables}(e)$ - local variables of $e$ |                             |
| $\text{anodes}(e) \cup \text{variables}(e) = \text{dchildren}(e)$ |                             |
| $\phi_{\text{share}}$ - determines resource sharing ($\rightarrow$ Algorithm 4.9) |                             |
| **ActivityGroup**           |                             |
| (no basic block)            |                             |
| $qos(e) = \langle \phi_{\text{share}}(a_{\text{max}}, QV), \phi_{\text{sum}}(l_{\text{max}}, QV) \rangle$ |                             |
| $QV = qos(m_{e, \text{bind}}) \cup \bigcup_{a \in \text{anodes}(e)} qos(a) \cup \bigcup_{v \in \text{variables}(e)} qos(v)$ |                             |
| $\text{anodes}(e)$ - activity nodes contained in $e$ |                             |
| $\text{variables}(e)$ - local variables of $e$ |                             |
| $\text{anodes}(e) \cup \text{variables}(e) = \text{dchildren}(e)$ |                             |
| $\phi_{\text{share}}$ - determines resource sharing ($\rightarrow$ Algorithm 4.9) |                             |
| **ActivityGroup**           |                             |
| (basic block)               |                             |
| $qos(e) = \langle \phi_{\text{share}}(a_{\text{max}}, QV), \phi_{\text{sum}}(l_{\text{max}}, QV) \rangle$ |                             |
| $QV = qos(m_{e, \text{bind}}) \cup \bigcup_{a \in \text{anodes}(e)} qos(a)$ |                             |
| $\text{anodes}(e)$ - activity nodes contained in $e$ |                             |
| $\text{variables}(e)$ - local variables of $e$ |                             |
| $\text{anodes}(e) \cup \text{variables}(e) = \text{dchildren}(e)$ |                             |
| $\phi_{\text{share}}$ - determines resource sharing ($\rightarrow$ Algorithm 4.9) |                             |
| **ConditionalNode**         |                             |
| $qos(e) = \langle \phi_{\text{share}}(a_{\text{max}}, QV), \phi_{\text{sum}}(l_{\text{max}}, QV) \rangle$ |                             |
| $QV = qos(m_{e, \text{bind}}) \cup \bigcup_{c \in \text{clauses}(e)} qos(a)$ |                             |
| $\text{clauses}(e) = \text{dchildren}(e)$ - clauses of the conditional node |                             |
| $\phi_{\text{share}}$ - determines resource sharing ($\rightarrow$ Algorithm 4.9) |                             |
| **Clause**                  |                             |
| $qos(e) = \langle \phi_{\text{share}}(a_{\text{max}}, QV), \phi_{\text{sum}}(l_{\text{max}}, QV) \rangle$ |                             |
| $QV = qos(m_{e, \text{bind}}) \cup qos(\text{test}(e)) \cup qos(\text{body}(e))$ |                             |
| $\text{body}(e)$ - body of the clause |                             |
| $\text{test}(e)$ - test of the clause |                             |
| $\text{body}(e) \cup \text{test}(e) = \text{dchildren}(e)$ |                             |
| $\phi_{\text{share}}$ - determines resource sharing ($\rightarrow$ Algorithm 4.9) |                             |
| **LoopNode**                |                             |
| $qos(e) = \langle \phi_{\text{share}}(a_{\text{max}}, QV, QV_1 \cup QV_2), \phi_{\text{sum}}(l_{\text{max}}, QV_1 \cup QV_2, \text{iterations}_{\text{max}}) \rangle$ |                             |
| $QV_1 = qos(m_{e, \text{bind}}) \cup qos(\text{setupPart}(e))$ |                             |
| $QV_2 = qos(\text{test}(e)) \cup qos(\text{bodyPart}(e))$ |                             |
| $\text{setupPart}(e)$ - setup of the loop |                             |
| $\text{bodyPart}(e)$ - body of the loop |                             |
| $\text{test}(e)$ - test of the loop condition |                             |
| $\text{setupPart}(e) \cup \text{bodyPart}(e) \cup \text{test}(e) = \text{dchildren}(e)$ |                             |
| $\phi_{\text{share}}$ - determines resource sharing ($\rightarrow$ Algorithm 4.9) |                             |
| **ActivityNode, ActivityEdge, Action** |                             |
| $qos(e) = qos(m_{e, \text{bind}})$ |                             |
| **CallOperationAction, CreateObjectAction, DestroyObjectAction** | not supported ($\rightarrow$ Section 4.2.3) |

The area estimates of FSMD-implementations are computed by the merge function $\phi_{\text{share}}$. This function accounts for the potential sharing of resources service instances among model elements with mutual exclusive execution. In principle, FSMD-implementations can share functional-units, memories, and communi-
4.4. Estimation of Model Characteristics

Estimation of Model Characteristics. However, the sharing of communication services is no issue due to the multiplexer-based architecture. Algorithm 4.9 presents a greedy approach for the sharing of functional-units and registers in the data-path. For all resource services it is checked whether they can be shared among different uses and if the sharing actually decreases the required area. Thereby the amount of multiplexer logic being required to implement the sharing is compared to the size of the instance without sharing. To get accurate estimates of the area, the multiplexer logic is also considered when merging the QoS of the optimized mapping.

Algorithm 4.9: Merge-function for resource service instance sharing - \( \phi_{\text{share}} \) (label, QV)

<table>
<thead>
<tr>
<th>Input</th>
<th>Label of the QoS-value being merged: label.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Set of QoS-characteristics being merged: ( QV = {qos_i} ).</td>
</tr>
<tr>
<td>Output</td>
<td>Returns merged QoS-value label that regards the resource sharing.</td>
</tr>
<tr>
<td>Data</td>
<td>Set of shared resource service instances ( shared = {ru_i} ). The elements have the format ( ru_i = (ri, uses) ), whereas ( ri ) represents a resource service instance and ( uses ) counts the number of elements that use this instance.</td>
</tr>
</tbody>
</table>

\[
\text{shared} \leftarrow \emptyset; \\
\text{foreach } qos_i \in QV \text{ do} \\
\quad m_{e,dl} \leftarrow m_{e,dl} \in M | m_{e,dl}.qos = qos_i; \\
\quad \text{foreach } ri_j \in m_{e,dl}.bind \text{ do} \\
\quad \quad RU \leftarrow \{ru \in shared | ru.ri.r = ri_j.r \land ru.ri.o = ri_j.o\}; \\
\quad \quad \text{if } RU = \emptyset \text{ then } shared \leftarrow shared \cup \langle ri_j, 1 \rangle; \\
\quad \quad \text{else } ru_{opt} = \emptyset, amux_{opt} \leftarrow \infty, uqos \leftarrow dl.\text{NodeEstimator}.\text{estimateQoS}(ri_j); \\
\quad \quad \text{foreach } ru_k \in RU \text{ do} \\
\quad \quad \quad \text{if all uses of } ru_k \text{ execute in mutual exclusion with } e \text{ then} \\
\quad \quad \quad \quad mux \leftarrow dl.\text{NodeMapper}.\text{createMultiplexer}(ru_k.ri, ru_k.uses + 1); \\
\quad \quad \quad \quad mqos \leftarrow dl.\text{NodeEstimator}.\text{estimateQoS}(mux); \\
\quad \quad \quad \quad \text{if } mqos.\text{area} < amux_{opt} \text{ then } ru_{opt} \leftarrow ru_k, amux_{opt} \leftarrow mqos.\text{area}; \\
\quad \quad \quad \quad \text{else } ru_{opt} \neq \emptyset \lor amux_{opt} > uqos.\text{area} \text{ then } shared \leftarrow shared \cup \langle ri_j, 1 \rangle; \\
\quad \quad \quad \quad ru_k.\text{uses} \leftarrow ru_k.\text{uses} + 1; \\
\quad \quad \text{foreach } ru_k \in shared \text{ do} \\
\quad \quad \quad muxers \leftarrow muxers \cup dl.\text{NodeMapper}.\text{createMultiplexer}(ru_k.ri, ru_k.uses); \\
\quad \text{return } \phi_{\text{sum}}(\text{label}, qos(\{ri_j | ru \in shared \land ru.ri = ri_j\} \cup muxers)); \]

Considering sharing only locally in data-paths seems rather restrictive. It is thinkable to employ sharing also among data-paths or even instances. However, as discussed earlier in this section, on upper hierarchical levels it is hard, if not impossible, to determine if these model elements execute in mutual exclusion. However, due to the support of inclusion polymorphism, an indirect yet significant sharing among instances can be observed. All attributes and operations of an instance of a generalization are available in the respective instance of the specializations. The saving in resource service instances depends on the particular inheritance hierarchy.

The estimation model orients towards the deployment hierarchy of the design. Obviously, the quality of the estimation results depends on the quality of the QoS- and execution-constraints such as the number of iterations of loops and the number of instances of a classifier realizing a component instance. For the purpose of the previously defined estimates these constraints must define good, if not least, upper bounds. This is critical for both \( a_{\text{max}} \) and \( t_{\text{max}} \). The presented platform mapping approach will be evaluated in Chapter 7.
4. Platform Mapping
5. SYNTHESIS

5.1 Synthesis for Object-Oriented Specifications

5.1.1 Definition of the Synthesis Problem

**Definition 5.1**: Given a PSM $d_{n+1} \in DS_{n+1}$, a set of language-processor pairs $\{\langle L, P \rangle\}$, with target language $L_i$ and (abstract) machine $P_i$ that can interpret representations in $L_i$, the synthesis problem is to find a transformation $\delta : DS_{n+1} \mapsto P(L)$, that if interpreted by the respective target machine will evoke a behavior that is equivalent to $d_{n+1}$.

This definition is based on Gilois definition of computer architectures as stacked interpretation systems [230]. This view contrasts platform mapping which seeks a mapping of a design model to the resource services offered by the lower level machine. Also, while platform mapping perceives its building blocks through their QoS, synthesis focuses on the synthesis constraints which define the link into the target micro-architecture. Both views are connected through the resource model. For the implementation of a model element solely the resource service instances can be used that were bound during platform mapping.

5.1.2 UML-to-Implementation Mappings

Synthesis must generate implementations of all model elements that define the behavior of a system. The set of regarded model elements includes all elements that were previously mapped to resource services. In addition, model elements that structure the behavior must be implemented. As Table 3.1 shows these structural elements may be handled by resource mapping as well, which is specific to the particular platform. In general only considering this aspect of the implementation is not sufficient. Further synthesis must support:

* **IP-integration** - Implementations are constructed from building blocks (intellectual property (IP)) provided by the target platform. These blocks are commonly accessed and integrated through libraries, packages, databases, et cetera.

* **Artifact Organization** - The generated artifacts comprising the implementation must be organized such that they are accessible by the target machine. The according organization scheme may be relatively free or restricted by the implementation language.

* **Interpretation Control** - The target machine that interprets the generated artifacts is commonly configurable in order to provide control of its operation. For instance, if the lower level machine is a compiler or synthesis environment the tool-chain can be configured.

The support of interpretation control seems to be a matter of user convenience at a first glance, but it is often more than that. Important decisions made during platform mapping build on assumptions of the detailed working of the target machine. For example, the implementation of a particular behavior with software/hardware may only be justified if certain optimizations are enabled in the lower level design flow.

The principal rules and patterns that define UML to implementation-language mappings are captured by the node-specific generators. These generators must reflect fundamental assumptions of the respective mapper and estimator components. Their detailed configuration is done using synthesis constraints that have been defined in the implementation platform model. Each implementation may define multiple generators that serve the different aspects of the implementation. Interpreter components provide the link into lower level flows.
5.1.3 Synthesis Flow

The principal synthesis flow is illustrated in Fig. 5.1, which details the flow shown in Fig. 3.8. The final application consists of executables, hardware configuration contexts, and a model of the hardware objects.

![Synthesis Flow Diagram](image)

The hardware and software modules and the hardware object model are co-synthesized and translated into the final application artifacts using a distributed approach. For this the generator and interpreter components that have been specified in the TPM are used. Each generator/interpreter is responsible for synthesizing the modules of one specific node in the deployment platform. The co-synthesis accomplishes IP-integration and artifact organization. The interface between the modules is determined prior to the actual synthesis and manifested in the co-synthesized modules. The interpretation control is either managed by the interpreter component or by a further generator. For example in C++ environments, dedicated generators are used for the creation of Makefiles.

5.2 Hardware/Software Interface

5.2.1 Hardware Object and Component Life Cycle

The hardware/software interface of object-oriented implementations with reconfigurable hardware defines the life cycle and access mechanisms of objects and components realized in reconfigurable hardware. The hardware/software interface can be viewed from a logical and physical perspective. The logical hardware/software interface can be realized by different physical implementations. The particular implementation depends upon the target platform and the model compiler.

For efficiency reasons the life cycle of hardware objects is different from the life cycle of software objects. Even if the hardware is partially reconfigurable, which would allow for the implementation of each class in a partial configuration, this raises significant problems during synthesis, verification, tool support, and also implementation. One problem is that the configurations are not independent from each other, because the objects have to share the same physical communication interface. The class instantiation per reconfigurable device is by far too expensive in terms of required and wasted area and device reconfiguration time.

Because of these problems another approach is chosen. Instead of mapping each class to a configuration, multiple objects (hardware objects) are clustered in a configuration (hardware component). In order to avoid costly reconfigurations hardware objects are reused as much as possible. Because a true dynamic instantiation/destruction of objects is not efficiently possible in hardware, these objects are pre-instantiated at compilation time and synthesized into configuration contexts. The objects are allocated dynamically on demand. The RTR-Manager, which will be described in Chapter 6, serves as object broker. Additionally, in RTR-systems the objects and hardware configurations are dynamically bound to logic resources.
5.2. Hardware/Software Interface

The dynamic instantiation and binding is reflected in the life cycle of hardware objects and configuration context instances, which is illustrated in Fig. 5.2. Because of the tight relationship between the hardware objects and their configuration context instances, as the container of the hardware objects, both life cycles influence each other. Each object and its configuration context instance will go through three states, X_UNBOUND, X_BOUND and X_ALLOCATED (where X is either OBJ for hardware objects or CT for configuration contexts). As long as the context is not loaded into the reconfigurable hardware, the context instance and the contained objects are in state X_UNBOUND. When a context is loaded, i.e. it is instantiated, the state of the instance becomes CT_BOUND. The objects contained go to the state OBJ_BOUND. Objects allocated by the application change go from state OBJ_BOUND to OBJ_ALLOCATED. All objects returned by the application will set their state back to OBJ_BOUND. When the last object of a context instance is destroyed the context instance state is set back to CT_BOUND. Until the context instance is unloaded from the hardware, i.e. destroyed, the objects will still be available for allocation. The context instance is not allowed to be unloaded from the hardware as long as it is in the state CT_ALLOCATED. For devices that do not support dynamic configuration the context instance is always in state CT_ALLOCATED.

5.2.2 Logical Hardware Object Interface

The mechanisms for the access of objects are defined by the object interfaces. The interface of each object consists of a control interface, a data interface, and an exception interface.

Control Interface - The control interface enables the message exchange with the object. The execution of message handlers, i.e. behaviors, is triggered. The interface also indicates when the message processing is finished.

Data Interface - The data interface allows one to access the object state and to pass message data to/from objects. The object state is public to enable the migration of objects between different deployment locations.

Exception Interface - The exception interface reflects exceptional conditions that occur in the object. Depending on the exception handling of the object the information on the position and type of exceptions

![Fig. 5.2: Hardware Object and Component Life Cycles](image-url)
is represented.

The interface of each component representing a configuration context comprises all interfaces of the objects accessible through the component interface. An implementation of the hardware object and component interfaces is presented in Section 5.4.

5.3 Implementation of Software Modules

5.3.1 UML-to-C++ Mapping

For both hardware and software realizations, the question for the appropriate level of abstraction of the final implementation and a suitable language arises. In an object-oriented approach it seems quite natural to choose a language supporting the object paradigm for software implementation. This approach makes the implementation convenient and straightforward. The final compilation is delegated to 3rd-party compilers. However, this results in loosing a fair amount of control over the final implementation. In performance and resource critical applications, this uncertainty can cause iterations in the development flow. To avoid this problem model compilers for critical application domains may generate microprocessor specific assembly language implementations. For the purpose of this thesis C++ is used to implement software modules.

Owing to different implementation patterns and styles multiple implementations are possible for an implementation model. These differences are reflected in the QoS in the implementation platform model so that it does not affect the quality of the design space exploration results. The implementation patterns and rules are either manifested in the respective components of the model compiler or in code generation annotations in the UML meta model. The latter approach is taken by xtUML [194]. It has the advantage of being defined entirely with UML models and dedicated generation languages (archetypal language). However, it orients towards single language software implementations. Design space exploration, estimation, (automated) model transformations, and mixed language implementations are not directly supported. Thus in this thesis the former approach is taken. In future adoptions of approaches like xtUML to multi-language environments should be investigated.

General Mapping Rules

The principal mapping of UML design models to C++ and other software languages is state-of-the-art and has been implemented in a number of UML tools. The implementation of binary associations and the proper realization of visibility is a research topic [249,250]. Due to the medium level of abstraction in the presented approach associations must be transformed to properties and behavior before. Table B.6 in the appendix presents the mapping of the most important UML model elements to C++ constructs.

Mapping of Actions

A crucial point is the mapping of UML actions to their implementations. As has been shown in Section 3.3.5 actions are intermediated mapped to operations in the implementation model. The mapping to the implementation is accomplished with a synthesis constraint called ImplementationLanguagePattern (→ Example 3.16 on page 49). To define such patterns a very simple language is used. All names that must be replaced by the generator are preceded with a dollar sign ($). The string following the $ until the next white space must refer to a parameter of the operation or the current object ($this). Thereby $this is replaced with the name of the object on which the action is executed. References to parameters are replaced by the respective arguments of the action.

EXAMPLE 5.1: For example, assume that the pattern defined in Example 3.16 on page 49 is associated with the sub operation in Example 3.19. Although this example actually refers to a VHDL implementation the same rules apply. For C++ just replace std_logic_vector<16> by say short. Since the sub action is invoked on object csample, $this is replaced by this name. The pattern $other is replaced by psample respectively.
5.3.2 Communication with Hardware Objects

The classes of the implementation model being deployed on the local node are implemented directly in C++ using the presented mapping rules. For each remote object a local proxy is used. Proxies realize the communication between local and remote objects. One proxy-type is used for all proxy instances. The proxy-type is explicitly modeled in the implementation platform model as \texttt{IHwObject}-type, which provides operations for creating, destroying, and accessing remote objects using basic read/write operations.

**Example 5.2:** Fig. 5.3 illustrates this concept. On some node \( h_0 \) a set of objects of type \textit{LocalClass} is executed. These objects access remote objects (\textit{RemoteClass}) that are execute on node \( h_1 \) via local proxies. The RTR-Manager performs the creation, destruction, and configuration of the remote objects. This can include loading configuration contexts that contain remote objects of the requested type into \( h_1 \) in case this node is run-time reconfigurable. Listing 5.1 shows an example C++ code that is automatically synthesized for the creation, access, and destruction of an instance of \textit{RemoteClass}. The proxy is implemented using smart-pointers (\texttt{smartptr}), that count the number of references to the respective remote object, and automatically destroys the object if the reference count gets zero.

**Listing 5.1:** Instantiation, Communication, and Destruction of \textit{RemoteClass} Example

```cpp
short * elements = ... // get elements and size
/* write elements to remote object, the elements parameter starts */
obj->write<short>(12, elements, size);
obj->write<int>(2316, size); // write size to address 2316
obj->execute(4, 2); // execute ‘add’
int sum = obj->read<int>(8); // read result
RTRManager::getInstance()->destroyObject(obj); // destroy object
```

**Fig. 5.3:** Remote Object Example

5.4 Implementation of Hardware Modules

5.4.1 UML-to-VHDL Mapping

Goal of the UML-to-VHDL mapping is the translation of a given PSM into a functionally equivalent description of a circuit. Due to its popularity and standardization VHDL is used as description language. This thesis presents the first approach to the fully automated direct synthesis of UML-models to RTL designs. On system-level a naive approach is to use behavioral synthesis in order to generate RTL-designs. In such an approach the isolated data-path implementations, which are generated by behavioral synthesis tools, are...
5. Synthesis

integrated into a complete description of the final hardware. Early work in this thesis has shown that this is a quite tedious and error-prone task which is complex to automate due to the specifics and obstacles of each tool [251].

Moreover, owing to the tight timing and resource constraints imposed by the hardware, it is even more important to reflect the implementation model directly in the implementation. In principle the implementation can be delegated to behavioral synthesis tools. However, as for software implementations it is hardly possible to predict the synthesized results. Moreover, the programming language based approaches are restricted by the employed languages and the directly synthesizable language subsets of the targeted HDLs. Thus model compilers synthesize hardware modules directly from UML models at the RTL. In this thesis, the hardware is described with synchronous VHDL-RTL designs. In order to increase the portability, reusability, simulation performance, and optimization opportunities in lower level design flows a mixed structural and behavioral style is used. The final synthesis of the generated hardware descriptions is delegated to lower level design flows.

This approach to synthesis enables the exploitation of advances in lower level flows using the same principal solutions to platform mapping and synthesis. The effects of lower level optimizations are regarded using merge functions. The assumptions driving system-level decisions are forwarded to the lower level flows by means of respective configurations or constraints. VHDL constraints are commonly attached to the impacted design element. Such constraints control resource-sharing, FSM state encoding, three-state to logic conversion, or enable or disable specific optimizations. Constraints are tool-specific and are propagated either in the VHDL design or by means of configuration scripts of the tool-chain. MOCCA employs the latter approach by means of specialized generators. The former approach is possible by extending the current generators accordingly.

The principal features and assumptions of the employed implementation options and architectural constraints have been presented in Section 4.2. Table B.13 in the appendix summarizes the mapping of UML model elements to VHDL constructs. Due to the large semantic gap between both languages the mappings are detailed in the following sections.

5.4.2 Hardware Implementation of Components

Each artifact that is stereotyped as “Configuration” maps to a hardware configuration, such as contexts of RTR-FPGAs, ASIC mask-data, and netlists. For the purpose of this thesis a run-time reconfigurable target technology is assumed. Each artifact manifests a component. The instantiation of the component corresponds to the loading of the context into the physical device. The dynamic instantiation of components is used to mimic the run-time instantiation of hardware objects. For efficiency reasons a component instance defines the execution context of multiple objects.

Design Hierarchy

Hardware designs are constructed hierarchically. At the top-level the infrastructure for the objects is synthesized. The lower layers are constructed according the deployment hierarchy. Fig. 5.4 illustrates the design hierarchy of hardware component implementations. The component instance provides the communication interface, register file, address decoders, and auxiliary services, such as clock refresh and reset generation, to the contained objects. The implementation of the communication interface, register file, and dispatch is explained in the course of this section.

In principle the entire design can be implemented flat by using a single level. In general, it is not advisable to put the entire logic into one layer because this increases time required for synthesis, simulation, and verification. Moreover, design reuse and readability are aggravated. While arguably the latter points are less important in model-driven approaches the tool-time is significant since the design productivity and size are increased as higher-levels of abstractions are used. For the same reasons classes are implemented in isolation with only minimal regard to eventual super-classes.
Instance Synthesis

In each hardware component all executed hardware objects are pre-instantiated. For each instance all state and behavior that is required to implement its class and all of its super-classes is replicated. This is relatively inefficient in terms of resources but it offers the highest performance since there is no need to synchronize concurrent executions of the same behavior.

**Example 5.3:** Fig. 5.5 shows an example design comprising two classes \( C_0 \) and \( C_1 \). The operation \( op_0 \) is polymorphic. In the hardware component \( n \) instances of both classes are created and connected at the top-level in the register file and through the dispatch logic. The registers are located in the register file; other components are not shown. Also note that the figure shows a particular implementation of the logical object interface. This implementation will be described in the course of this chapter.
The implementation of resource services, such as clock generators, communication interface, and storage, is inherently technology dependent. To maximize adaptability and design reuse the implementations are constructed from implementation components that have been defined in the implementation platform model. As has been shown in Section 3.3.4, this requires the definition of standard component interfaces that can be interpreted automatically by the model compiler. This has previously been exemplified in Example 3.12 for a FIFO storage. In hardware implementations the interface is interpreted structurally as representing the signals of a circuit. The same approach is used for the synthesis of the classes that define hardware objects.

The implementation type that is used to implement the implementation component is bound to the model element during platform mapping. Synthesis then must create a VHDL entity for the respective UML proxy. Since physical port signals may be shared among logical signals in the interface definition the proper mapping to the shared signal must be ensured. An UML entity or architecture proxy definition is transformed into the corresponding VHDL construct by converting all attributes and operation parameters into signals. The signal type is the type of the respective UML element. The direction is `inout` for attributes. For signals created from parameters the direction corresponds to the parameter direction, whereas direction kind `return` is mapped to the VHDL direction kind `out`. By default a signal name is generated automatically. This name can be overridden using the synthesis constraint `ImplementationName`. If a signal with the current name already exists no new signal is created. The same mapping rules approach are used for types, such that the name of the entity and architecture can be set. This approach gives the user maximum control over the implementation.

Using the created entity synthesis creates a VHDL component, instantiates the VHDL architecture, and binds it to the VHDL component. This approach implies that the interface of the architecture equals that of the entity, such as is defined in the VHDL specification. Also elements that map to the same name must be functionally equivalent.

**Example 5.4:** Fig. 5.6 illustrates this concept with two implementation components\(^1\). One component is a proxy of a PCI-bridge, which is realized by the VHDL entity `PLX9080_PCI_BIU`, that implements the communication interface. The other component is a memory block (`bram16_32x16_xst`). For the lack of space, the architectures of both entities are not shown. The tables define the name mapping rules which are specified in the model as synthesis constraints of the respective parameters. Both components are interconnected using an instance of the MOCCA Object-Bus (MOB). A special address decoder logic is synthesized for the memory block. This block is mapped to the address range `AR` of the MOB. The bus is explained later in this section.

### Interconnection Synthesis

Interconnect synthesis handles the connection of circuit instances. Such instances are created from implementation model elements and implementation components. As illustrated in Fig. 5.4, a design comprises generic circuitry (shown in the grayed area), which is represented by implementation component instances, and application specific circuit instances. The semantics of the interface of application specific circuits is defined by the model elements from which they have been created. At the side of the implementation component the semantic is defined by means of the standardized interfaces. This property enables the semantically correct synthesis of interconnect.

Some implementation components must be connected to the pins of the physical device. This interface is called *device connect interface*. Examples of such components are the communication interface, external storage, and peripheral devices. The interface of the top-level module of the hardware component includes the device connect interface of all instantiated implementation components. The device connect interface is part of the UML proxy definition. To give the user maximum freedom in modeling all non-interpreted port signals are considered to be part of this interface\(^2\). For the device connect interface the same name mapping rules apply as for interpreted interfaces.

---

\(^1\) Notice that, in accordance with common convention, thick lines are used in the notation to represent signal groups, while thin lines represent individual signals. A thickly drawn schematic symbol represents multiple instances of the element that are connected appropriately.

\(^2\) To improve readability and reuse it is recommended to capture the device connect interface using one operation, e.g. `device_connect`. 

---
Communication Interface

The communication interface defines the external interface of each hardware component. It enables the message exchange with objects that are executed in the context of the instances of the component. Thereby the interface couples some external interface to a standardized local interface called MOCCA Object-Bus (MOB). The concept of IP integration by means of implementation components supports the adaption of the synthesis to different communication networks. For this, designers must implement a bridge of the external communication interface to MOB and provide a proxy in the TPM. The external interface is captured in the proxy as device connect. Consequently, this interface is unrestricted, so it can be any network, such as serial lines, field buses, and advanced uP buses.

The MOB is a single-master bus that is designed towards high speed data transfers. To enable the adaption to different timing constraints the MOB is asynchronous. As illustrated in Fig. 5.7, the MOB comprises separate sub-buses for addresses, data, and control. These buses are scalable in that the width of the address-, data-, and byte-enable signals is not standardized. Instead, the model compiler automatically adapts to the width that is modeled in the proxy interface. For this, the compiler needs to know the width and representation of the signal types, which is defined using synthesis constraints. The MOB uses the same clock as the object logic. The synchronization between the clock of the external network and the local clock must be accomplished in the bridge. Addresses are divided into word addresses and byte addresses. A word comprises $n$ bytes. The bytes within each word are addressed using byte enables. The data bus contains $n$ byte lanes and there is one byte enable signal for each byte lane. All signals are active high. The endianess is adjusted in the bridge. To this bus the dual-ported register file is attached whereby the registers are mapped to individual addresses or address ranges. The objects are connected to the local port of one or more registers.

To illustrate the usage of the bus Fig. 5.8 presents the timing of read and write transfers. The principal bus timing was designed after the Wishbone-bus [252]. In contrast to Wishbone, the separate data buses of the
5. Synthesis

Fig. 5.7: FSMD Communication Interface and Register File

The delays \( t_{wa} \) (write-acknowledge delay) and \( t_{ra} \) (read-acknowledge delay) depend on the performance of the addressed slave components. The delay \( t_{ae} \) (acknowledge-enable delay) is a property of the FSM design. Typical values of these delays are in the range of 1..4 clock cycles.

Fig. 5.8: Timing of MOB Write Transfer and Read Transfer

Register File

The purpose of the register file is to

- store the state of all objects,
- store input and output parameters of messages,
- define the control interface of objects, and
- decouple object logic from the communication interface.

A similar object logic can be used with different communication interfaces. Hence, the logic is adapted to the specific incarnation of the MOB. For instance, the registers are connected to the proper byte-lane of the MOB data sub-bus.
The register file is constructed from primitive storage components that have been modeled in the implementation platform. Such storage components are either individual registers, with a small data width, or dedicated memory blocks. All storage components must be dual-ported. The interface that is connected to the MOB is called the external interface. The local interface is used to connect to the objects. To avoid synchronization of different clock domains the register file uses the same clock as the object logic. Design modularity is supported by requiring all registers having three-state buffers on both ports. Common hardware design flows transform three-state buffers into equivalent logic after flattening the design hierarchy (→ Section 5.4.4).

Registers are used to store scalar data types and the control logic. Designers can model registers with different read/write modes on the local interface, in order to safe logic resources. Registers with wider data ports are constructed from primitive registers. This reduces the modeling effort since designers do not have to define registers for all possible types that are mapped to hardware. Array types are mapped to memory blocks. Designers can provide storage components of different width and depth. Also their physical implementation may differ. For instance, many FPGAs contain embedded RAM blocks of relatively small size. Larger memories are attached to the FPGA externally. As for registers, memory blocks of different width and depth are constructed automatically from simple memories.

This simple mapping between variables and storage elements - scalar variables to registers and array variables to memory blocks - is sufficient for the most situations. However, this can affect both the latency and area requirements of the generated circuit. For instance, memory blocks have one interface being connected to behaviors. Consequently, multiple accesses to the memory block must execute sequentially. Similarly the utilization of a large number of registers may affect the routing delay and the allocated chip area. It can be advantageous to allow for more implementation options. Future extensions should investigate the implementation of arrays using individual registers. This option increases the number of available interfaces and consequently allows for more parallel accesses. Also the effect of collapsing multiple variables into a single memory block should be studied.

Address Decoders

Address decoders enable registers according to the current address on the MOB. Logically these decoders are part of the register file. The MOB address comprises the address bus and the byte enable signals. For each primitive register a comparator is used for the address and the byte enable. For each address range occupied by memory blocks address range decoders are used. Due to the addressing scheme of the MOB a full address range decoder requires the decoding of word and byte addresses. If the available address space is large enough, memory blocks are aligned to word boundaries which eliminates the byte decoders.

Dispatch

Dynamic message dispatch represents the technological foundation for the support of inclusion polymorphism. Depending on the dynamic type of an object, different subsets of the overall feature set are active and accessible from the environment of an object. Specifically, dynamic message dispatch is performed only among the active operations of an object. Recall, that the notion of inheritance implies that there can be at most one active message handler, i.e. operation, for each message type! Accordingly, dynamic message dispatch is implemented by selecting the currently active operations of each object. A type field in the object interface is used to dispatch polymorphic messages to the appropriate operation. This is generally accomplished using multiplexer structures. All control- and output data is multiplexed.

The execution of a FSMD is triggered using a dedicated GO signal. This signal enables the transitions of the FSM to be taken. Similarly, a dedicated DONE signal is activated if the FSM is in a final state and thereby it signals the termination of behavior execution. Both signals are defined formally later in this section.

EXAMPLE 5.5: Fig. 5.9 exemplifies the implementation options for dynamic message dispatch on the FSMD-model. Depending on the system inputs, some object may be either of class C_0 or C_1. Due to inheritance the operation op_0 is polymorphic.

3 In uP-based implementations dynamic message dispatch is accomplished using VMTs and indirect function calls. Each class is associated with a VMT. Objects have an implicit pointer to the VMT of their dynamic type.
5. Synthesis

The isolated synthesis of user classes necessitates the handling of polymorphic behaviors at the top-level. Since no behavior implementations are shared among different objects this structure is replicated for each set of instances of polymorphic behaviors.

**Clock- and Reset Generators**

The generated hardware is driven by a fixed clock. This clock is generated either from a dedicated source, or the clock of the external communication interface. The purpose of the clock generator component is to refresh the clock, buffer the clock signal, and to perform a clock multiplication/division if required. For this, dedicated clock manager components are used, which are available in commercial FPGA devices. After the configuration of the hardware is finished the logic is reset. The propagation delay of the power on reset of common FPGAs can take longer than the user clock cycle. This may bring the logic into an invalid state. Thus an additional reset generator is integrated, which performs a startup reset of several clock cycles. Since the clock and reset generator components are specific to the vendor and even the particular device, respective implementation components are used. The model compiler performs the distribution of both signals to the logic.

5.4.3 Hardware Implementation of Objects

The implementation of hardware objects is realized cooperatively by several components in the design hierarchy, namely the register file, the message dispatch logic, and the components of the individual classifiers in the type hierarchy of the object. Data-paths that access common data are connected to the common registers in the register file using a local MOB.

**Physical Object Interface**

The physical object interface realizes the logical object interface, which has been described in Section 5.2.2. In the presented approach this interface is implemented using registers that are mapped into the address space of the MOB. Thereby registers with different functionality are used.

**Control Interface** - The control interface of each class is independent from the control interfaces of its specializations. Each class defines a control interface for all operations that are visible in its interface and that are not inherited or overridden by any operation that is defined by the class. Behavior execution is controlled using GO/DONE signals.

Objects that contain polymorphic behaviors use a type register, which stores a numeric identifier of the dynamic object type. The type register is an ordinary register as used in the data interface. A type register is assigned to root classes and shared among all specializations of this class. This is possible due to the object integrity constraint.

Fig. 5.9: FSMD-based Implementation Dynamic Message Dispatch
5.4. Implementation of Hardware Modules

Data/Exception Interface - The data interface is realized using registers and memory blocks as has been explained earlier in this section. The exception interfaces stores the position and type of exceptions that occurred in each behavior. A common exception interface of all behaviors of an object is not advisable since behaviors may execute concurrently. The automated implementation of this interface is currently not supported by MOCCA.

Polymorphic Address Mapping

The address mapping defines which MOB addresses decode each particular storage component in the register file. In principle each storage component can be mapped to any free address range. As has been discussed in the previous section, this is not recommended because the address mapping impacts the size of the address decoders. Moreover, the realization of the external communication interface is affected. Practical experience has shown, that it is advantageous to use the addresses of the external network locally if possible since this reduces design effort, occupied chip area, and timing problems. As a consequence, address mapping must satisfy data alignment constraints that are imposed by the external network. For instance, common uPs require primitive data elements to start at addresses that are divisible by the data size. Address mapping must support inclusion polymorphism, and is an important means of decoupling the hardware objects from the other parts of the system. It should be transparent to the users of a hardware object which particular object is accessed as long as it has the proper type. Additionally, in order to support dynamic object migration, objects must be relocatable in the address space. Therefore the address mapping of hardware objects must satisfy the following constraints:

1. all hardware objects of the same type must have the same relative address mapping.
2. all polymorphic behavior implementations must have the same relative address mapping.
3. data must be aligned according to the constraints of the external communication network.

The relative address of each mapped element must be the same in all element instances. Notice that same constraints apply for the most implementations of software objects. They are enforced by the compiler tool chain.

Data alignment constraints are modeled in the deployment platform individually for each node using the synthesis constraint AddressAlignment. If the value of the constraint is a positive integer then all data is aligned to an address that is divisible by this value. Designers may also define the value TypeInstance-Size, which constrains data to be aligned to addresses that are divisible by their size.

EXAMPLE 5.6: A class hierarchy and some possible address mappings for objects created from the classes in the hierarchy is shown in Fig. 5.10. In unconstrained address mapping data is packed without any address gaps (illustrated as grayed boxes) in between, while constrained address mapping can cause address gaps.

Algorithm 5.1 maps the features of a set of classes to relative addresses. The set includes all classes that are instantiated by hardware objects either directly or by means of class inheritance. The algorithm satisfies all data alignment constraints that are defined by the deployment locations of the classes. Initially all classes that realize hardware objects are sorted in topological order. The topological order is a partial order \( C \preceq \) that defines that a classifier \( c_i \in C \) is predecessor of classifier \( c_j \in C \), if \( c_i \) is a generalization of \( c_j \):

\[ \forall c_i, c_j \in C : c_i \in \text{generalizations}(c_j) \rightarrow c_i \preceq c_j \]

The order ensures that the address mapping of a classifier is computed not before the address mapping of its generalizations was computed. The function visibleOperations(\( c_i \)) returns all operations that are visible in the interface of \( c_i \), including the inherited operations (inheritedOperations(\( c_i \))) and the operations defined by \( c_i \) that override operations of the generalizations (overridesOperations(\( c_i \))). For each newly defined operation, the algorithms allocates a bit in a control register. Using this bit the activation control of the respective operation and its overriding operations is performed. Each operation that overrides another operation inherits the control bit and the relative address mapping of the overridden operation. The function
5. Synthesis

**Fig. 5.10: Address Mapping Example**

\[ \text{nextAlignedAddress}(\text{type}, \text{current}, DL) \] returns the next address that is greater than \( \text{current} \) and that satisfies all data alignment constraints imposed by the type \( \text{type} \) and the deployment locations.

Since the local relative address mapping is performed with respect to the globally imposed alignment constraints it is ensured that the interfaces of hardware objects are equal in the entire system regardless of the particular deployment location. As a consequence the address mapping can be sub-optimal for a specific node. The presented algorithm implements a greedy strategy in order to reduce compilation time.

A similar address mapping step must be performed when a hardware component is synthesized. In this step all hardware objects contained in the component are mapped to relative addresses with respect to the base address of the hardware component. Thereby the relative start address of each object starts at an address that is divisible by the bus width. Hardware components and objects are mapped to absolute addresses at runtime when the hardware component is instantiated. For this mechanism to work, the RF must be mapped into the address space of the master. This embodies the Simple Communication Illusion (→ Section 4.2.1).

### 5.4.4 Hardware Implementation of Behavior

Behaviors are implemented at the bottom-most level of the hardware design hierarchy. The implementations evoke the same functionality that is modeled in the design model. To enable efficient mappings to the hardware resource services, extensive transformations are applied to design model behaviors. Important enabling transformations are array access transformations, that transform array accesses into explicit MOB transfers, and inlining, that replaces operation calls by the invoked behavior (→ Tables C.2 and C.3).

Behaviors are implemented in hardware using the FSMD model, which will be detailed in the course of this section. This model is especially suitable for control oriented applications and fits the message based computing paradigm of the object based model of computation. As described in Section 4.2.3, by principle
Algorithm 5.1: Address Mapping - computeAddressMap ($C, DL$)

**Input**: Set of classes that realize hardware objects: $C$. Set of deployment locations of hardware objects: $DL$.

**Output**: Address mapping of attributes and parameters of all $c_i \in C$, that satisfies the constraints imposed by polymorphic behaviors, uniform object interfaces, and data alignment. The address mapping is a set $AM = \langle e, address \rangle$, that assigns each mapped element $e \in E$ a relative address $address \in \mathbb{N}$.

**Data**: The function $\text{sizeof}(e)$, with $e \in E$ returns the number of bytes occupied by the element $e$ in the address space. This information is modeled in the TPM using the synthesis constraint ImplementationAddressSpace ($\leftrightarrow$ Example 3.15 on page 48).

$T_C \leftarrow \text{sortTopologically}(C), AM \leftarrow \emptyset$

foreach $c_i \in T_C$ do
  if $\text{generalization}(c_i) = \emptyset$ then current $\leftarrow \text{sizeof}(\text{type - reg})$;
  else current $\leftarrow \text{lastOccupiedAddress}(\text{generalization}(c_i))$;
  $OP \leftarrow \text{visibleOperations}(c_i) \setminus \text{overriddenOperations}(c_i) \setminus \text{inheritedOperations}(c_i)$;
  controlregs $\leftarrow \lceil \frac{|OP|}{\text{sizeof}(\text{byte})} \rceil$;
  if controlregs $> 0$ then current $\leftarrow \text{nextAlignedAddress}(\text{byte}, \text{current}, DL) + \text{controlregs}$;
  foreach $a_j \in \text{attributes}(c_i)$ do
    current $\leftarrow \text{nextAlignedAddress}(\text{type}(a_j), \text{current}, DL)$, $AM \leftarrow AM \cup \langle a_j, \text{current} \rangle$;
  foreach $op_j \in \text{operations}(c_i)$ do
    $op_{\text{override}} \leftarrow \text{overriddenOperation}(op_j)$;
    foreach $p_k \in \text{parameters}(op_j)$ do
      if $op_{\text{override}} \neq \emptyset$ then
        $p_{\text{override}} \leftarrow \text{overriddenParameter}(op_{\text{override}}, p_k)$;
        $am \leftarrow \langle p_{\text{override}}, \text{address} \rangle \in AM$, $AM \leftarrow AM \cup \langle p_k, \text{am}.address \rangle$;
      else
        current $\leftarrow \text{nextAlignedAddress}(\text{type}(p_j), \text{current}, DL)$;
        $AM \leftarrow AM \cup \langle p_j, \text{current} \rangle$;
  endforeach;
endforeach;

this model has no notion of recurrence since only one FSM state can be active at any point in time. Consequently, there must be as many implementations of the behavior as there can be concurrent executions of it. This causes a severe synchronization and interfacing problem if concurrent executions of a behavior of the same object exist. Thus, behaviors are implemented once in a hardware object. Additionally, it is required that concurrent invocations are sequentialized in the design model. Notice, that the missing notion of recurrence precludes the implementation of recursive behaviors in hardware.

Principal Behavior Execution

Fig. 5.11 illustrates the principal execution of behaviors in hardware. The execution is started when $GO$ is active at the rising edge of the clock ($\text{CLK}$). During the initial state inputs are optionally buffered to local latches of the data-path. Before the activation all input parameters of the processed message must be written to the data interface and must remain stable for at least one clock cycle after the activation of $\text{GO}$. As long as the behavior is executed $\text{GO}$ must remain active.

The end of execution is signalled to the environment by activating the $\text{DONE}$ signal. After activation, this signal remains active as long as $\text{GO}$ is still active. This signal can be used to synchronize the loading of output data to the register file. Some output signals, such as MOB bus signals, must be driven continuously while the behavior is executing. To drive outputs only at particular synchronisation points, for each output signal a gate signal can be defined that enables the respective output driver.
Definition of the FSMD Model

Behaviors are implemented in hardware according to the FSMD model.

**Definition 5.2:** A FSMD is defined as [175]:

\[ \text{FSMD} = \langle S, I, O, V, s_0, v_0, F, f, h \rangle \]

- \( S = \{S_i\} \) - set of states of the FSM
- \( s_0 \in S \) - initial state of FSM
- \( F \subseteq S \) - set of final FSM states
- \( I = \{I_j\} = I_C \times I_D \) - set of inputs of FSM (\( I_C \)) and data-path (\( I_D \))
- \( O = \{O_k\} = O_C \times O_D \) - set of outputs of FSM (\( O_C \)) and data-path (\( O_D \))

A FSM is extended by a data-path, whose state is defined as:

\[ V = \{V_l\} \] - state of the data-path

\( v_0 \in V \) - initial state of data-path

Then the next state function of the FSMD is defined as:

\[ f = \langle f_C, f_D \rangle \] - next state function of FSMD

\[ f_C : S \times I_C \mapsto S \] - next state function of FSM

\[ f_D : S \times V \times I_D \mapsto V \] - next state function of data-path

The output function of the FSMD is defined as:

\[ h = \langle h_C, h_D \rangle \] - output function of FSMD

\[ h_C : S \times I_C \mapsto O_C \] - output function of FSM

\[ h_D : S \times V \times I_D \mapsto O_D \] - output function of data-path

For the purpose of this thesis FSMDs are represented graphically using UML. UML state machines, an example of which has been given in Section 2.3.1, are used as graphical notation of the FSM part. To make the presentation easy to grasp also for non-UML experts no further extended features of UML state machines are used. Data-paths are represented by scheduled basic blocks, whose notation has been introduced in Section 4.1.3.

Integration of FSMDs into the Execution Environment

Practical implementations of this model require additional logic for reset and integration into the execution environment. Fig. 5.12 presents a respective implementation template. A synchronous Moore FSM controls the data-path components. The FSM determines the execution of all action instances that are executed in the context of the behavior. Thereby it goes through a sequence of FSM states. The execution sequence is determined by the FSM design and the input set \( I_C \). The input signals are feedbacks from the data-path,
while the output signals $O_C(dp)$ steer data-path components, such as multiplexers and registers. The actions are implemented by the data-path.

The synchronization logic ($sync$) resets the FSM to its initial state, and synchronizes the data-path inputs and outputs with the environment ($O_C(sync)$). This logic is functionally part of the next-state logic and the output-logic of the FSM. The next-state logic and the output logic only realize the functionality of the application logic. In order to simplify the overall design it is implemented separately.

The $GO$ signal triggers the execution of the behavior. Formally this signal is part of the input set of the FSM:

$$GO \in I_C \text{ - initialization input}$$

$$f_C(s_j \in S, i_0, \cdots, GO = 1, \cdots, i_k \in I_C) = s_0$$

The FSM is active as long as $GO$ is activated. If this signal is deactivated the next state is permanently set to $s_0$. The $DONE$ signal is activated when the FSM is active and is in a final state. This signal is part of the FSM output set:

$$DONE \in O_C \text{ - termination output}$$

$$h_C(s_j \in F, i_k \in I_C) = o_0, \cdots, DONE = 1, \cdots, o_l$$

Output drivers decouple the behavior from the environment and ensure that output data is only driven by the behavior at specific synchronization points as long as the behavior is active. This enables that several behaviors that modify the same data can be connected to the same data sink, i.e. the register file, using a MOB. The external inputs of the data-path are connected to the data source directly and latched to the initial state of the data-path $v_0$ upon activation of $s_0$.

**FSM Synthesis**

The next-state function $f_C$ and the state assignment capture the FSM design. The state assignment associates all action instances with FSM states. The assignment is based on the schedules of the action instances that were computed throughout platform mapping. For each schedule a unique FSM state is assigned to each time step. *Global slicing* associates time steps of schedules that are executed in mutual exclusive control paths to the same FSM state. This strategy generally reduces the overall number of FSM states. Each state that is associated with different time steps requires the definition of sub-states that distinguish between the time steps. Consequently, additional state registers and multiplexer logic is needed. *Local slicing* does not share states among different time steps at the expense of more overall FSM states and higher chip area requirements [159]. Fig. 5.13 illustrates the FSM design for actions comprising a basic block using the ASAP schedule of the DFG in Fig. 4.2 on page 60.

Conditionally executed actions, represented by *ConditionalNode* and *LoopNode* activity nodes, require that the computation of the next state and the actions in the data-path are sequentialized. This is because the next state can not be determined safely before the end of the evaluation of the condition. The
events evaluation result is available when the state that executes the according action is deactivated. Thus, in order to avoid any invalid states, a dedicated state is assigned to instances of ConditionalNode and LoopNode, if their execution depends on data-path results. Fig. 5.14-5.16 present the FSM design for these UML activity nodes.

The gray shaded boxes relate the UML elements to the parts of the FSM that realize these elements. Following the principal structure of UML activities also FSMs are hierarchically nested, so the FSM realizing an activity is constructed from a number of sub-FSMs that represent the nested activity nodes. Each FSM that is synthesized for an activity has a unique initial state and a unique final state. Observe that each sub-FSM also has a unique initial state since all behavior is eventually realized by actions. Consequently, all transitions between FSM states are deterministic. By definition actions comprise basic blocks, which always have a unique entry point, i.e. the state to which their first time step is mapped. Consequently, the FSM also has a unique initial state. A transition to the final state is synthesized at the exit of each FSMs that is generated for an activity node that does not have a successor. Given by the MAL some activity nodes are optional, such as the setupPart of the LoopNode, and the false branch of the ConditionalNode. If these elements are not modeled they are also skipped in the synthesized FSM. The same applies for empty activity groups. A special case are ConditionalNodes whose clauses can be evaluated concurrently, i.e. the cases of a switch statement. The evaluation of all clauses is performed by comparator components, which are assigned to the same state. This state assignment requires all comparators to execute concurrently and finish their execution before the end of the clock cycle. Activity edges and instance of ReplyAction are mapped to transitions. The activity of the GO signal is an implicit firing condition of all transitions. If
5.4. Implementation of Hardware Modules

not defined otherwise, transitions are taken as long as the GO signal is active, otherwise the FSM is reset to the initial state.

The current state vector of the FSM is stored using registers. The FSM states are described in VHDL as enumerated type. The detailed encoding of the states is not fixed in the design since the appropriateness of an encoding depends on various properties, such as the number of states, the control flow complexity, and the target technology. For instance, Xilinx recommends one-hot encoding for large FPGA designs, since the register representing a particular state, the next state logic, and the output logic of the state can commonly be implemented in a CLB\(^4\) [253]. This encoding maps each state to an individual flip-flop whose output

---

\(^4\) A configurable logic block (CLB) represents the RFU in Xilinx Virtex-II FPGA devices, and comprises four identical slices with each slice including two 4-input lookup tables (LUTs), two flip-flops, two multiplexers, and carry logic. Additionally, each CLB provides two three-state buffers [21].
reflects the activity of the state. Using enumerated types the detailed encoding can be selected in lower level flows without having to change the design. This approach supports the adaption to different hardware synthesis tools and target technologies. Obviously the next-state logic and the output logic depend on the encoding of the state vector. If not stated otherwise one-hot encoding and local slicing are used for the examples in this thesis.

EXAMPLE 5.7: To illustrate the FSMD design, Listing 5.2 shows a very simple sequence of MAL statements which will be used as running example. Fig. 5.17(a) shows the schedules for the two basic blocks in the code using ASAP. Fig. 5.17(b) presents a FSM of the entire statement block in graphical notation and as VHDL description. The FSM is constructed using the mapping given in Fig. 5.14, whereas the false branch is not modeled. An additional final state has been added to the FSM. The outcome of the comparison is carried by the signal \( \text{cond} \).

Listing 5.2: FSM Design Example

\[
\begin{align*}
y &= a + b; \\
\text{if } (y > c) \\
y &= y + 3;
\end{align*}
\]

Data-Path Synthesis

The goal of data-path synthesis is to realize the actions and local variables using the resource services to which they have been mapped. In FSMD implementations additionally feedback signals must be synthesized that steer the next-state logic of the FSM. In the presented approach the synthesized data-paths have a multiplexer-based architecture. In contrast to bus-based architectures multiplexers require more routing resources but are faster and less complex to design.

The synthesized data-paths comprise functional units, registers, and multiplexers. In accordance with common design practice these elements are instantiated implicitly, in order to support a certain level of technology independence. The detailed module selection and bit-level optimizations are delegated to lower level design flows. Thereby also the current restriction of behavior selection, which has been described on page 75, is circumvented. The downside of implicit instantiation is the reduced accuracy of the predicted implementation characteristics. The explicit modeling and instantiation of data-path elements using UML implementation components is possible as well. However, due to the inherent technology dependence and the significant modeling effort this approach is not used for the data-paths in this thesis.
Local variables in the design are mapped to registers, which are inferred implicitly from VHDL signals (Example 2.1 on page 15). For each action data-path synthesis must define the functional unit that executes the action, the signal that carries the output of the functional unit, and the FSM state in which the execution is started. As in C++, actions are mapped to functional units using the synthesis constraint ImplementationLanguagePattern (Section 5.3.1). By default, functional units whose actions have been mapped to different time steps are decoupled by registers. Direct connections are synthesized between functional units whose actions have been scheduled to the same time step and that are data-dependent. In VHDL this is implemented with VHDL variables. If extended operation chaining is used (Tab. C.3), registers are only inferred to store data that is communicated between different basic blocks.

In multiplexer-based architectures all data-flow that depends on control-flow and resource sharing is realized with multiplexers. Thereby multiplexers are inferred for all signals that are driven by multiple sources. The selector signals of the multiplexers are generated by the FSM. Multiplexers realizing control-flow dependent data-flow are fixed in the synthesized VHDL description. Due to the implicit instantiation of the data-path elements the detailed resource sharing, and hence the insertion of additional multiplexers, is done by lower level synthesis. On system-level the effect of resource sharing is captured during estimation (Algorithm 4.9).

**Example 5.8:** Fig. 5.18 presents the synthesized data-path for Example 5.17 in VHDL and the RTL schematic for the main part of the FSMD. The FSM states are one-hot encoded. The respective next-state logic and the output logic is realized with combinational logic. The clock inputs of the registers are connected to a single clock, which is not shown for the purpose of presentation. The RESET signal is pulsed high for one clock cycle at the rising edge of GO. Notice, that y is only an output of the design, since it is never read before it is written.

```vhdl
process (GO, CLOCK) is
begin
if GO = '0' then
  y <= "000...000";
  cond <= '0';
else if rising_edge(CLOCK) then
  case CS is
    when S0 => y <= (a + b);
    when S2 =>
      cond <= conv_std_logic(y > q);
    when S4 =>
      y <= (y + "00...11");
    when others => null;
  end case;
end if;
end if;
end process;
```

(a) VHDL Data-Path

(b) FSMD Schematic

**Fig. 5.18:** FSMD for Design Example 5.7

**Example 5.9:** Listing 5.3 and Fig. 5.19 illustrate the data-path synthesis when multiple data-dependent operations, i.e. actions, are scheduled to the same time steps. The respective functional units are connected directly using VHDL variables. Chaining creates commonly opportunity for further optimizations. Depending on the target technology and the data type of the chained actions, lower level design flows may jointly optimize the functional units.

**Listing 5.3:** Operation Sharing Design Example

```plaintext
1 y1 = a & b;
2 y2 = y1 | (~c);
3 y3 = ~(y1 | (~c)) | c;
```
Output Driver Synthesis

Output drivers are inferred from the design for all signals representing elements of the data interface, which are written by the modified data-paths. These elements are all attributes that are modified by a behavior and the parameters with a direction kind other than \( \text{in} \). Notice that the latter also includes parameter sets that represent a MOB interface of the behavior.

The purpose of the output drivers is to decouple the behavior from the register file and to increase the driver strength of the affected signals. Since multiple behaviors can drive the same signal in the register file output drivers are implemented using three-state buffers. These buffers are only activated when the respective behavior is active. The set of FSM states in which a particular output is activated is called the set of synchronization states \( S_{\text{sync}} \subseteq S \) of this driver. The main cases of synchronization sets are shown in Table 5.1. If a particular set falls into more than one group the enable signal is the conjunction of the enable signals of the individual sets.

### Tab. 5.1: Synchronization Sets

<table>
<thead>
<tr>
<th>( S_{\text{sync}} )</th>
<th>Enable Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F )</td>
<td>DONE</td>
<td>All modified data is synchronized to the register file at the end of behavior execution. This is the default.</td>
</tr>
<tr>
<td>( S )</td>
<td>GO</td>
<td>The driver is activated continuously. This property can be used to interface with peripheral devices or other IP using the native protocol of these components.</td>
</tr>
<tr>
<td>( S \setminus F )</td>
<td>signal(s) controlled by the data-path</td>
<td>The enable signal is generated by the data-path. For example, for the MOB signals driven by the behavior the synchronization states are all states in which the ENABLE signal of the bus is active. The enable of the data bus drivers is additionally qualified with the RW signal.</td>
</tr>
</tbody>
</table>

A particular problem in this architecture is to avoid bus contention, i.e. different sources driving the same signals concurrently, since this can damage the device. For each bus signal it must be ensured that always at most one output driver driving this signal is active. Bus contention occurs when different behaviors concurrently modify the same data. This is a race condition and a failure in the design model! Designers must avoid such conditions by synchronizing relevant parts of the design model appropriately.

To avoid device damage even in the presence of race conditions the multiple drivers must be resolved by the hardware. This can be done by converting three-state buffers into equivalent logic. If the enable signals
are orthogonal, the conversion must ensure that the equivalent logic behaves as a multiplexer. Otherwise an implicit resolution of the colliding signals must be performed. There are many ways this can be done. Fig. 5.20 illustrates the most common solution, which basically is a wired-OR emulation, for a MOB data bus. Fig. 5.20(a) shows three behaviors that can drive this bus and Fig. 5.20(b) shows the equivalent logic\textsuperscript{5}. The logic eliminates undefined values of the driven signal. The logical validity of the value of the signal is indicated to the sink with the actual enable signal of the former three-state buffer. If some behaviors drive the bus concurrently the result depends on the values of individual signals comprising the bus and will therefore be logically invalid.

It is strongly recommended always to convert the three-state buffers to equivalent logic if the presence of race conditions cannot be surely precluded. This can be done automatically by common hardware synthesis tools. Clearly, depending on the system size, this conversion can require a large amount of logic resources. This conversion can be done only after the design hierarchy is flattened, since it requires that all three-state buffers driving one line being visible at the same level. An alternative solution would be to synthesize supplementary synchronization logic that prevents the simultaneous activation of the individual enable signals driving one signal.

\section{5.5 Hardware Object Model}

The hardware object model describes the hardware objects and their association to the available configuration contexts. This model defines the interface between the software and hardware parts of a system. This information is used by the RTR-Manager at run-time to dynamically create hardware objects and to load and configure configuration contexts to the appropriate devices. For each configuration context this model describes

- the device type for which the context is synthesized,
- the device configuration (max. clock cycle),
- the file that contains the configuration context, and
- the contained hardware objects.

The description of hardware objects comprises

- the classifier and super classifier, of which the object is an instance, and

\textsuperscript{5} Notice, that the equivalent logic always carries a value at its output, while the three-state buffer does not. The logic does not have the same physical properties as neither a three-state buffer nor a wired-OR.
• the mapping of the object features to addresses relative to the start of the object.

The hardware object model is synthesized by the model compiler whereas XML is used for representation.

**Example 5.10:** Listing 5.4 illustrates this model for Example 5.2 on page 87. The model contains a configuration context. The context is manifested by the bitstream file HardwareComponent.bit. This file can be instantiated on all nodes that have the specified speed grade, package type, and model identifier. This information is sufficient to ensure the applicability of the bitstream to the device. The required logic clock cycle is 100MHz. The configuration context contains an instance of RemoteClass. The example shows the description of the add-operation of this object.

**Listing 5.4: Hardware Object Model Example**

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<bitstreamConfiguration>
  <bitstreamList>
    <bitstream FileName="/data/myapp/HardwareComponent.bit"/>
    <node SpeedGrade="5" ClockCycle="(10, 'ns')"
      PackageType="FF1152" ModelName="XC2V3000"/>
    <hwobjectList>
      <hwobject ClassifierName="Data.RemoteClass" ClassifierId="0"
        RelativeStartAddress="0">
        <interface>
          <operationList>
            <operation Name="add" RelativeStartAddress="8">
              <parameter Name="elements" RelativeStartAddress="12"/>
              <parameter Name="size" RelativeStartAddress="2316"/>
            </operation>
          </operationList>
        </interface>
      </hwobjectList>
    </bitstreamConfiguration>
```

...
6. RUN-TIME RECONFIGURATION

6.1 Hardware Abstraction Layer

6.1.1 Resource Management

The final implementation of design models comprises software executables, hardware configuration contexts, and the hardware object model (→ Fig. 5.1 on page 84), which together represent the application. This implementation does not include any further specifics of the particular computer architecture on which it is going to execute. A tight coupling of the application to the computer architecture would hamper the execution of the application on similar environments. From the application point of view it is only important to execute on a computer architecture that provides at least the QoS that is modeled in the target-platform model. It should be able to benefit from hardware extensions without having to reiterate through the entire design flow.

The decoupling of the application and computer architecture is the task of a hardware abstraction layer (HAL). HALs enable the isolation of application and computer architecture (e.g. [254–256]). They implement the management, administration, communication, and configuration of reconfigurable hardware, configuration contexts and the objects, being executed on the reconfigurable fabric. Fig. 6.1 illustrates this concept of a HAL for object-oriented systems.

![Fig. 6.1: Hardware Abstraction Layer for Object-Oriented Systems](image)

The HAL shown in the figure abstracts from the specifics of the physical hardware and distribution of hardware objects among the reconfigurable fabrics. Applications are not required to handle individual configuration contexts and hardware devices. The relationship between hardware objects of a specific application and configuration contexts is defined by the hardware object model.

The MOCCA environment includes a HAL, called RTR-Manager [257]. The RTR-Manager is targeted toward object-oriented applications of network-coupled reconfigurable architectures. It is executed on the target platform and acts as middleware between the application and the hardware objects. The application interface supports the creation, destruction, and access of hardware objects in a virtualized hardware environment. This environment may comprise of reconfigurable fabrics attached to the processor bus, fabrics connected with an internetwork, or even virtual fabrics implemented by hardware simulators. This HAL makes the location, concurrency, and scaling of hardware objects and the target architecture transparent [258].
Fig. 6.2 illustrates the interaction of the application and the RTR-Manager. When the RTR-Manager is started, it is initialized with the hardware object model. The manager enumerates and configures the available reconfigurable hardware. Enumeration searches the hardware that actually exists on the target. This allows to extend and use the target hardware after the implementation. Thereby the reconfigurable hardware can be used if it is compatible with the hardware that has been modeled in the TPM. Notice, that the deployment hierarchy allows each configuration context being bound to multiple deployment locations. This location must be compatible to the deployment location the context was synthesized for, since configuration contexts are specific to the particular device type. When enumeration and configuration is finished the application can start working with hardware objects.

6.1.2 Resource Broking

The HAL serves as broker for hardware objects and resources and implements the life cycle of hardware objects and components (→ Section 5.2.1 on page 84). It schedules and binds the objects by means of their configuration contexts to the hardware and serves them to the respective application. Since in principle multiple applications may try to execute on the same reconfigurable hardware their concurrent requests must be coordinated.

Example 6.1: Fig. 6.3 illustrates the conceptual interaction between an application, the RTR-Manager, the proxy, and the hardware object for Example 5.2 on page 87. The application requests the creation of a hardware object from the RTR-Manager. This service in response conceptually creates the hardware object and a respective proxy. The proxy is configured and served to the application. All communication between the application and the hardware object is handled via the proxy. At the end of the hardware object life time the object is destroyed.

The algorithms that are used for object creation and destruction, and the communication are discussed in the next sections.

6.2 Dynamic Object Lifetimes and Communication

6.2.1 Object Creation

The scheduling and binding of hardware objects and configuration contexts to RFs is done dynamically, whenever the application demands a hardware object. Algorithm 6.1 presents the algorithm for the creation
of hardware objects. Thereby, in accordance with the hardware object life cycle, the following definitions apply:

\[ O = \{ o_i \} \subseteq EI \] - the set of hardware objects, \( o_i = (state_{o,i}, type) \)

\( state_{o,i} \) - the current object state

\( state_{o,i} \in \{OBJ_UNBOUND, OBJ_BOUND, OBJ_ALLOCATED\} \)

\( type \) - the current dynamic object type, with \( type \in C \)

\( CT = \{ ct_i \} \) - the set of configuration contexts, \( ct_i = (dl, objects) \)

\( dl \) - the deployment location of the context, with \( dl \in DL \)

\( objects \) - the contained hardware objects, with \( objects \subseteq O \)

\( CTI = \{ cti_i \} \) - the set of current configuration context instances, \( cti_i = (state_{ci}, ct) \)

\( state_{ci} \) - the current context instance state

\( state_{ci} \in \{CT_UNBOUND, CT_BOUND, CT_ALLOCATED\} \)

\( ct \) - the context from which the instance was created, with \( ct \in CT \)

\( first : P(X) \mapsto X \) - returns the first element of set \( X \)

Whenever an application creates a hardware object of a specific type, the HAL searches an unallocated object in all configuration contexts being bound to hardware resources. If so, it sets the state of the object and the context to OBJ_ALLOCATED and CT_ALLOCATED respectively. Then the relative address of the hardware object with respect to the start of the configuration context is converted into an absolute address in the address space of the master. Finally, the dynamic type of the object is set in the control interface and a proxy is returned to the application. Otherwise, the HAL tries to bind an appropriate configuration context to hardware resources first. If no feasible hardware object exists, the algorithm returns nothing. This occurs when all appropriate hardware objects are allocated, or no feasible configuration context can be instantiated since all suitable deployment locations are allocated. Increased resource requirements must be addressed by either extending the hardware or by reiterating the design flow with appropriate settings for the respective mapping constraints.

The algorithm selects feasible configuration contexts, their instances and hardware objects using a first-fit strategy. This strategy minimizes the negative effect of dynamic object creation on the application perfor-
mance. Other strategies, based on metrics such load balancing, performance criticality, and priority, are specific to the particular application and computer architecture. Such strategies can be straightforwardly integrated into the RTR-Manager.

6.2.2 Object Destruction

Hardware objects are destroyed on demand. Algorithm 6.2 presents the algorithm for the destruction of hardware objects. The object state is set from OBJ_ALLOCATED to OBJ_BOUND. If the application later on creates a new object of the same type or one of its sub-types the same object can be allocated again. If this object was the last object of the configuration context instance in whose context it is executed, the state of the context instance is reset to CT_BOUND. This ensures that the according deployment location is available to execute different configuration contexts.

Algorithm 6.2: Hardware Object Destruction - destroyObject(o)

<table>
<thead>
<tr>
<th>Input</th>
<th>Hardware object: o ∈ EI.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Returns nothing.</td>
</tr>
<tr>
<td>Data</td>
<td>The set of configuration contexts: CT. The set of configuration context instances: CTI.</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
o & \leftarrow 0, O & \leftarrow \{o_i | o_i = \langle OBJ\_BOUND, C \rangle \}; \\
& \text{if } O \neq 0 \text{ then } o & \leftarrow \text{first}(O); \\
& \text{else} \\
& \quad \text{foreach } c \in \text{specializations}(C) \text{ do} \\
& \quad \quad \quad o & \leftarrow \text{createObject}(c); \\
& \quad \quad \text{if } o \neq 0 \text{ then return } o; \\
& & \text{if } FDL = \emptyset \text{ then} \\
& \quad \quad \quad FDL & \leftarrow DL \setminus \{dl_k | dl_k \in DL \land ct_i \in CT \land \langle dl_k, objects \rangle \land \langle ct_i_j, CTI \land \langle ct_i_j = \langle CT\_BOUND, ci \rangle \land ci \in CT \land \langle dl_k, objects \rangle \land \langle ct_i_j = \langle CT\_ALLOCATED, ci \rangle \}; \\
& & \text{if } CT_C = \emptyset \text{ then} \\
& \quad \quad \quad \text{ct}_i, & \leftarrow \text{first}(CT_C), ct_i & \leftarrow \text{ instantiate}(ct_i); \\
& \quad \quad \quad \text{CTI} & \leftarrow CTI \cup \langle \text{ct}_i \rangle, \text{configure}(\text{CTI}) \text{, setState}(\text{CTI} \_\text{ALLOCATED}); \\
& \quad \quad \quad \text{configure}(dl), \text{configure}(ct_i); \\
& \quad \quad \quad o & \leftarrow \text{createObject}(c); \\
& \quad \quad \text{if } o \neq 0 \text{ then } o\_\text{setState}(\text{OBJ\_ALLOCATED}); \\
& \quad \text{return } o; \\
\end{align*}
\]
6.2.3 Communication with Hardware Objects

The communication between application and hardware objects is especially important in network-coupled architectures with distributed memory. The interaction between the application and the hardware object follows the client-server model. Since hardware objects can be executed on remote deployment locations, the RTR-Manager may have to provide typical gateway services. If a RF is not mapped to the address space of the host processor, the HAL must transform all communication accordingly. This can involve data conversion, marshalling, and unmarshalling.

The employed communication protocols are encapsulated in the hardware object proxies. This makes the physical distribution and communication of hardware objects transparent to the application.

**Example 6.2:** Fig. 6.4 exemplifies the communication between the software and hardware objects for the running Example 5.2. The application accesses to the hardware object are translated by the proxy into primitive operations for read/write of data words, and the set/test of bits. These primitive operations could, for instance, be provided by the ISA of the uP that executes the software object. This requires the RF to be mapped into the uP’s native address space.

This approach creates opportunity for new usage models for RTR, such as distributed RTR systems and dedicated computation servers providing RF to clients. Also, it is thinkable to integrate hardware simulators or emulators into this framework in order to support system-level verification.
7. EXPERIMENTAL RESULTS

7.1 MOCCA-Development Environment

7.1.1 Overview

Before the presentation of selected experimental results in the second part of this chapter the MOCCA development environment and relevant aspects of the MOCCA compiler are presented. This environment has been developed at the Hochschule Mittweida in cooperation with the Technische Universität Freiberg. The goal of the MOCCA development environment is to develop algorithms and tools for the automated implementation of object-oriented models as combined hardware and software solutions. Fig. 7.1 gives an overview of this environment. The MOCCA compiler and the RTR-Manager, which have been developed in this thesis, build the core of the environment. Important principles and algorithms of the compiler and the RTR-Manager have been presented in the previous chapters.

UML models are edited using a commercial modeling tool [259]. The output of the modeling tool is a XMI representation of the models [260]. Models are fed into a model repository. Currently, the integration of MOCCA with other tools being frequently used in system-level development is under development.

MOCCA reads UML models from a repository. Its flexible architecture supports the utilization of different repositories, and thereby the integration into different environments. The project-specific parameterization of MOCCA is provided in its execution context or via a dedicated project file. Output of the compiler are hardware and software modules, which are exemplified in the figure with VHDL designs and C++ code. The hardware object model is represented using XML. Additionally, MOCCA can generate synthesis scripts and Makefiles that execute and configure lower level design flows. The compiler can automatically trigger the execution of 3rd party tools.

The used software and hardware tool chains transform the output of the model compiler into executables and hardware configuration contexts. These artifacts, in conjunction with the hardware object model, comprise the platform-dependent application. Lower level design flows integrate predefined components using their native library mechanisms. These components implement the implementation components that have been modeled in the TPM (→ Section 3.3.4 on page 42).
A specific component, which has been developed as part of MOCCA, but that is not discussed further in this thesis, is the operating system abstraction layer framework (OSLF). The OSLF is a generic implementation of common operating system services, such as a thread-mechanism and primitives for the communication and synchronization of concurrent control flows. This framework is used to implement active classes in UML design models in a way that is source-code portable among different operating systems.

7.1.2 Model Compiler for Reconfigurable Architectures

Features and Restrictions

MOCCA is an embodiment of the system-level design flow that has been presented in Section 3.2.2. Thereby all principles, algorithms, and transformations that have been described in this thesis are fully supported. The MOCCA compiler has the following important features:

- Comprehensive system-level design flow that integrates MDA, PBD, and co-design.
- Automated translation of UML design models to hardware/software implementations.
- Integration with different development environments through an extensible architecture:
  - Input: Integration of user-defined parser components that interface with dedicated repositories, support different dialects and representations of UML, and handle obstacles introduced by modeling tools.
  - Implementation: Integration of user-defined, implementation platform specific components for platform mapping, estimation, synthesis, and interfacing with lower level design flows (→ Section 3.3.4).
  - Platform Mapping: Integration of user-defined components for platform mapping, that provide application-specific implementations of the controller, breeder, and evaluator components (→ Section 4.3.1).
- Support of a large sub-set of the latest UML specification.
- Support for detailed behavior specification using the MOCCA Action Language.
- Validation of UML models according to the syntax and semantics imposed by UML and the MOCCA Action Language.
- Portability, since MOCCA is implemented in Java.

The implementation of MOCCA was intended to be proof-of-concept of the principles and algorithms presented in this thesis in a non-commercial, academic environment, with a very restricted time frame. Consequently, there are restrictions and unimplemented elements.

- Initially MOCCA was developed to support the UML 1.4/1.5 specification. In the recent years, the UML was restructured and extended significantly. This effort is embodied in the UML 2.0 specification. At the time of writing the latest UML specification is not fully adopted into the compiler. This mainly affects the specification of activities, actions, and state machines:
  - The support of the activities and actions that have been used in this thesis is complete. Due to the medium level of abstraction of MAL, actions for links between objects are not supported (→ Section 3.1.2).
  - Intermediate activities and behavior state machines are not supported. The hardware/software synthesis from UML state machines is discussed in related works [208, 261]. Intermediate activities are not supported since they are not accessible through the MAL.
- Implementation Restrictions:
Only behaviors that represent leaf nodes with respect to the call graph or that can be transformed into leaf-nodes by means of inlining, can be implemented in hardware (→ Section 4.2.3).

Exceptions triggered by hardware designs are not supported (→ Section 5.4.3). Notice, that this is no conceptual restriction.

The implementation of multi-dimensional arrays as hardware design is not supported. This can be handled during platform mapping by transforming multi-dimensional arrays to one-dimensional arrays.

Compiler Activities

Fig. 7.2 illustrates the most important activities that are performed in a typical MOCCA compilation flow and relates them to the chapters in which they have been discussed.

![MOCCA Compilation Flow Diagram]

The key principles underlying the grayed actions have been presented in this thesis. Other activities, such as model parsing and validation, have not been discussed since they are realized similar to what is known from compiler construction [232].

bootstrap – Bootstrapping integrates application-specific components, such as for parsing and platform mapping, into the model compiler.

parse – Parsing fetches UML models from repositories and translates them into the compilers model representation. During parsing the syntactical correctness of the model is checked.

validate – Validation checks the semantic correctness of the model representation according to the semantics of UML and MAL.

optimize – Optimization performs the technology-independent optimizations (→ Table C.2 on page 216). Technology-dependent optimizations are triggered during platform mapping by the respective platform-specific mappers.

map/synthesize – The process of platform mapping and synthesis (→ Chapter 4 and 5).

The following important objects/artifacts are used and produced during compilation:

Project File – Definition of the project-specific settings of the model compiler. The dynamically linked compiler components and the individual compilation activities, that are specific to the project but independent of the design, are parameterized.

Design Platform Model, Design Model, Target Platform Model – These models define the employed platforms and the user design (→ Section 3.3).
Model Representation - An in-memory representation of the UML models. The language orients towards the UML meta-model. Linked instances of the meta-model elements represent the model structure, which is complemented by a symbol table [229, 232].

Hardware Modules, Software Modules, Hardware Object Model - The manifestation of the implemented hardware/software solution. The employed language and organization depends on the particular implementation platform (→ Chapter 5). The hardware object model is described in XML.

Synthesis Scripts, Makefiles - These artifacts are used to configure and execute lower level design flows. The employed language and organization is specific to the particular flow.

Compiler Architecture and Implementation

As described earlier, the presented approach is backed by a flexible compiler architecture that enables the dynamic integration of user-specific components. The relevant parts of the architecture have been described in Sections 3.3.4 and 4.3.1. Fig. 7.3 gives an overview over the top-level of the compiler architecture. MOCCA features a traditional compiler design, comprising of a front-end, an engine, and a back-end. All components access the common model representation.

The translation of UML models to the compilers native model representation is the responsibility of the front-end. Two distinct components are used for parsing models and action language specifications. This separation allows to use the same model parser with different action languages. The current implementation uses MAL. The compiler engine performs the optimizations and analyses. It implements the separate optimize activity (→ Fig. 7.2) and provides optimization services to the compiler back-end. In the back-end the platform mapping and synthesis are performed. The ModelMapper and ModelGenerator components implement the common algorithms for platform mapping and synthesis. Moreover, both components manage the implementation platform specific components. The ModelMapper can be replaced by a user-defined component. The figure shows a particular configuration of MOCCA using a VHDL and a C++ implementation platform.

![Fig. 7.3: MOCCA Compiler Architecture](image)

Although the implementation of the model compiler required a major part of the overall project effort, it is not discussed in this thesis for two reasons. First, the implementation is not mandatory to the approach and does not improve the comprehensibility. Second, a meaningful discussion would exceed the scope and length restrictions of this thesis by far (at the time of writing the model compiler comprises approximately 145000 lines of Java code, organized in 59 packages, and 637 classes). Implementation-specific details of this tool and its usage can be found on the MOCCA co-development platform [216].
7.2 Boolean Neural Networks

7.2.1 Problem Description

To evaluate the presented approach to system-level design a data structure for the representation of Boolean functions, called Boolean neural network (BNN), is used. Common applications of BNNs are classification, data mining, and Boolean optimizations. BNNs are a special type of artificial neural network consisting of Boolean neurons (BNs), which, in contrast to conventional neural networks, operate exclusively on Boolean values. The structure of this type of network is similar to traditional artificial neural networks. The activation function of each BN, however, is a Boolean function \[ f_B \in \{0, 1\} \]. A Boolean neuron is defined as follows:

\[
\begin{align*}
  y &= f_B(x_B, w_B), y, f_B \in \{0, 1\} \quad \text{Output and activation function of the neuron} \\
  x_B &= \{x_1, x_2, ..., x_N\}, x_i \in \{0, 1\} \quad \text{Input vector of the neuron} \\
  w_B &= \{w_1, w_2, ..., w_N\}, w_i \in \{0, 1\} \quad \text{Weight vector of the neuron}
\end{align*}
\]

The type of BNN being considered in the thesis is a 3-layer feed forward neural network. The activation functions and neurons are determined during a learning process which is done prior to system design.

Traditionally, artificial neural networks are implemented using microprocessors. Because of the sequential execution paradigm of microprocessors and the common lack of native bit type support, such implementations are often sub-optimal in terms of both, execution time and allocated memory. To improve efficiency, in the recent past FPGA implementations of artificial neural networks have been proposed. Known implementations require between 22 and 784 CLBs per neuron [262–267]. This diversity is mostly due to the different feature sets in terms of supported activation functions (e.g. Boolean, integer, real domain), the implementation of these functions, and on-chip training support.

In order to get space efficient implementations of BNN using FPGAs Kohut and Steinbach proposed to adapt the structure of the network to the implementation technology. A BN can be mapped to a LUT one-to-one, if the number of inputs of the neuron is at most the number of inputs \( n \) of the LUT (typ. \( 3 \leq n \leq 5 \)). The interconnection network can then be mapped directly to respective routing resources. Such a network can be derived from an arbitrary BNN by a decomposition of the \( y_i \in y \) into simpler Boolean functions having at most \( n \) inputs [262]. Clearly, this approach allows for the optimal implementation of a respectively structured network. On-chip training is not directly supported but might be implemented using partial reconfiguration.

Applications of BNNs may be implemented as hardware/software systems employing a run-time reconfigurable architecture. The actual neural network is implemented with logic resources, while the software component reconfigures the FPGA with the BNN, communicates the input vectors to the BNN, and returns the respective output vectors. In the following, a very small BNN application is used in order to study MOCCA and the implications of distributed computation on the target platform that is being used for all examples in this thesis.

7.2.2 Experiments

**Example 7.1:** Fig. 7.4(a) shows the structure of the example BNN. Each neuron in the hidden layer has an activation function which is different from the activation functions of the other neurons in this layer. This function depends on a sub-set of the input variables, while the functions of the BN in the output layer only depend on the neurons of the hidden layer. The activation functions are given in Eq. 7.2-7.6. An UML design model of a system using the BNN is shown in Fig. 7.4(b). The system comprises two classes `Main` and `Bnn`. The trained network is defined in the `calculate(...)` operation of `Bnn`. For simplicity, instances of class `Main` create instances of `Bnn`, compute the input vectors, synchronously call `calculate(...)`, and read back the respective output vectors.
The BNN design in Fig. 7.4(b) serves as a template; depending on the particular design there may be additional attributes, parameters, and operations. To evaluate different modeling styles, platform mapping, and synthesis 15 different design models (BNN0 .. BNN14) of this BNN have been created. Each of the BNN designs is implemented at 10 levels of optimization (L0 .. L9). This approach is taken to evaluate the effectiveness and efficiency of the optimizations and the compilation flow. The designs and optimization levels are described in detail in Section D.2.1 on page 220. For each design model two deployment models have been created that fix the deployment of $Bnn$ to device h0 (uP) or h1 (FPGA).

All designs are implemented with MOCCA using a target platform comprising a C/C++ implementation platform, a VHDL implementation platform, and a standard PC-based deployment platform. The according platform models are described in Section B.2 on page 189, Section B.3 on page 195, and Section B.4 on page 213 respectively. The overhead that is introduced by the run-time reconfiguration, the remote communication, and the RTR-Manager is summarized in Section D.1.

The QoS-characteristics of the target platform were derived from the data sheets of the Pentium 4 microprocessor and the Virtex-II FPGA [21, 268]. Throughput and latency measures were carried out for operations that require additional software support, such as the creation/destruction of objects and arrays, data transfers to peripherals, and FPGA reconfiguration. The QoS-characteristics of types and operations implemented using the FPGA according test designs are synthesized and mapped to the device.

The experimental results for the FPGA implementation can be found in Section D.2.2. Tab. D.4 and D.5 list the measured latencies for communication and execution of the BNNs on the FPGA\(^1\). Tab. D.6 and Tab. D.52-D.66 contain the latency and area estimates of MOCCA. The implementation characteristics of all designs are given in Tab. D.7-D.8. The respective compilation times are presented in D.67. Software implementation results are given in Section D.2.3. Like for the FPGA implementation, Tab. D.68 and D.69 show the measured latencies. The average software compilation times for the investigated designs are given in Tab. D.70.

\(^1\) Zero values in these tables indicate that the respective operation is not performed by the design.

---

Fig. 7.4: Design and Structure of BNN Example

\[
\begin{align*}
k_1 &= \overline{x_0} \land \overline{x_2} \lor x_0 \land \overline{x_1} \land x_2 \lor x_0 \land x_1 \land \overline{x_2} \\
k_2 &= x_0 \land \overline{x_1} \land \overline{x_2} \\
k_3 &= \overline{x_0} \land \overline{x_1} \\
y_0 &= k_1 \\
y_1 &= k_1 \lor \overline{k_1} \\
y_2 &= k_2 \lor \overline{k_2} \\
y_3 &= \overline{k_2} \lor k_3 \\
y_4 &= k_2 \lor \overline{k_3} \\
y_5 &= k_1 \lor \overline{k_1} \\
y_6 &= \overline{k_1} \lor \overline{k_4} \\
y_7 &= k_1 \lor \overline{k_4} \\
y_8 &= \overline{k_1} \lor \overline{k_3} \lor k_4 \\
y_9 &= \overline{k_2} \lor \overline{k_3} \lor \overline{k_4}
\end{align*}
\]
7.2. Boolean Neural Networks

7.2.3 Evaluation

Hardware Implementation of the BNNs

In the first experiment all designs are manually partitioned to the reconfigurable logic $h_1$ while the class $\text{Main}$ is associated to the microprocessor $h_0$. The operating frequencies are 100 MHz and 2.4 GHz respectively. The PCI-bus that connects both processing elements works at 33 MHz.

Fig. 7.5 depicts the latencies of the BNN designs that have been compiled at the highest level of optimization (L9). The shown latencies are those perceptible by the software. The execution time

$$t_{\text{exec}} = t_{\text{exec,init}_x} + t_{\text{exec,calculate}} + t_{\text{exec,get}_y}$$

(7.7)

is the total of all execution times of all invoked operations, including the communication effort for triggering and testing the GO/DONE signals. The communication time

$$t_{\text{comm}} = t_{\text{write,}x_n} + t_{\text{read,}y}$$

(7.8)

denotes the timely effort to transfer the input and output vectors to and from the network. As Tab. D.6 on page 227 shows, the execution time $t_{\text{exec,calculate}}^h$ that is required by the reconfigurable logic to compute the network is far less than the execution time that is perceived by the software part, i.e., $t_{\text{exec,calculate}}^h \ll t_{\text{exec}}^h$. The total effort is dominated by the communication overhead. This is particularly critical for all designs that use sequences of single element transfers, which is because some effort is spent in the proxies for address computation. Since the base address of the accessed item in the respective hardware object is computed only once block transfers can amortize this overhead. Most of all, as can be seen in Tab. D.3 on page 220, the latency of the transfers on the PCI-bus is significant in comparison to the performance of the microprocessor.

The use of direct memory access (DMA) to accelerate block transfers is not feasible for this small amount of data. Designs BNN1 and BNN9 encode the input and output vector in the bits of an 32 Bit integer word.

While BNN9 performs the extraction and the packing of the vectors in the $\text{calculate}(\ldots)$ operation, BNN1 uses two additional operations $\text{init}_x(\ldots)$ and $\text{get}_y(\ldots)$. Since the invocation of these operations evokes additional communication overhead BNN1 is nearly two times slower than BNN9.

The actual hardware execution times of the $\text{calculate}(\ldots)$ operations are shown in Tab. D.6 on page 227. Thereby the minimum execution time is 50 ns. This is optimal for the given functionality and an 2-input technology library, as defined by the implementation platform model since the logic depth is seven and on the employed FPGA each LUT has 6.14 ns latency, that is \(7 \cdot 6.14\text{ns} = 50\text{ns}\). If a technology library with more inputs is used, such as provided by 4-input LUTs in FPGAs, the logic depth can be further reduced. Notice, that the timing is controlled by the FSM however, which is not effected by the technology mapping of the data path elements.

Fig. 7.6 illustrates the area characteristics of the component that is realized by the class $\text{Bnn}$ at the highest level of optimization. The meaning of the values is given in Eq. 7.9. Those designs that use embedded

---

2 To improve accuracy in comparison to conventional software timers the time stamp counter of the IA-32 microprocessor architecture was used for latency measurement [268]. This counter is incremented each cycle of the internal microprocessor clock.

3 $t_{\text{exec,calculate}}^h$ was measured directly on the reconfigurable logic using an integrated logic analyzer.
memory blocks for the implementation of arrays allocate the most chip area. Designs with less FSM states require the fewer resources for their implementation. In the considered designs the number of states can be used as complexity measure. The designs with more states contain loops or conditionals which make the next state logic more complex.

\[
\text{#FSM states} - \text{Sum of states of all FSMs in the design element}
\]

\[
\text{A} - \text{Synthesized area in gate equivalents}
\]

\[
\text{#FF} - \text{Sum of flip flops in the design element}
\]

\[
\text{#LUT} - \text{Sum of LUTs in the design element}
\]

At the highest level of optimization the most calculate(...) operations have equivalent implementations, which is reflected in the implementation characteristics (→ Fig. 7.7). This implementation is optimal, because only thirteen 4-input LUTs are allocated. One lookup table implements the next-state logic while the other LUTs implement the BNs. Notice, that no LUTs are required to implement \(y_0\) and \(y_9\) because they are just copies of the \(k\)-functions \(k_1\) and \(k_4\) respectively. Apart from BNN0 and BNN3, the allocated resources correlate with the resources at the component level. BNN0 and BNN3 are different since they perform the extraction and packing of the input and output vectors in separate operations.

A representative example of the degree of optimization that MOCCA is able to perform is illustrated in Fig. 7.8. The number of FSM states is reduced from 347 to 34, which translates to an area decrement of 77% (→ Tab. D.32 on page 236). Thereby loop invariant code motion causes the most significant drop in the complexity. All computations and array accesses that are computed within the loop are moved before the loop. This creates further optimization opportunity, e.g. for common sub-expression elimination, because the computations of the BNs can be optimized together. Moreover, this optimization simplifies the design such that each array element is accessed at most once. Tab. D.7-D.51 show that operation chaining and pruning frequently cause a significant increase of the maximum operation frequency of the design.

Aggressive optimizations must not require unacceptably long compilation times. The average compilation times of MOCCA are given in Fig. 7.9. Thereby the following semantics hold:

\[
t_{\text{opt}} - \text{Optimization time}
\]

\[
t_{\text{map}} - \text{Platform mapping time}
\]

\[
t_{\text{syn}} - \text{Synthesis time}
\]

\[
t_{\text{sum}} - \text{Total compilation time}
\]
7.2. Boolean Neural Networks

The compilation time depends on the complexity of the design of course. Experience with other designs shows, that typical compilation times are less than five minutes\(^4\). This encourages incremental and iterative system development.

Tab. D.6 and Tab. D.52-D.66 list the timing and area estimates respectively, that are computed by the algorithms presented in Section 4.4.2. For the timing estimates the value \(\hat{t}_{\text{exec, calculate}}\) represents an estimate of the execution time of the \(\text{calculate(...)}\) operation on node \(h1\). The average percentage estimation error \(t_{\text{err}}\) is approximately 8\% which is reasonably accurate. In most cases the execution time is estimated accurately even for behaviors with complex control flow. The outliers of the estimates regard designs which are dominated by loops. Notice, that in designs whose control flow depends on the input data timing estimates can be arbitrary worse than the presented values, because input data can not be factored in by principle. In this case execution profiles are recommended (\(\rightarrow\) Section 4.4 on page 77).

As can be seen in Tab. D.52-D.66 on pages 243-247, the FPGA area estimation performs slightly worse. The percentage error \(A_{\text{err}}\) of the estimator \(\hat{A}\) is in the range of 5\% to 117\%, whereas the mean error \(A_{\text{err}} \approx 21\%\). In literature, approaches can be found that perform better (typ. 2\%..20\%), however, they are either restricted to individual behaviors or they operate directly on the hardware description [116, 117, 121, 269]. The reasons for the estimation error can be found in the greedy resource allocation performed by MOCCAs VHDL back-end and principal limitations of system-level estimation. Most important is the fact that the optimizations being performed by lower level design flows are impossible to predict by principle [117].

Software Implementation of the BNNs

The BNN designs BNN0..BNN14 were implemented at the levels of optimization L0..L9 using a C/C++ software implementation platform. It is important to emphasize, that no changes were made to the UML design model! The timing of the synthesized implementations is presented in Fig. 7.10 (\(\rightarrow\) Tab. D.68 and D.69). As in the previous setup, the execution and communication latencies that are perceptible by the

\(^4\) The peak shown in the diagram is due to the scheduling of the exceptionally large number of actions in the \(\text{calculate(...)}\) operation.
object that uses an instance of \texttt{Bnn} are measured. Thus, $t_{\text{exec}}$ commonly includes some communication effort, e.g. for the transfer of parameters. Because just references or integer values are transferred this latency is neglectable. The latency $t_{\text{comm}}$ denotes all communication effort that is evoked by all variable assignments that have been modeled explicitly.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{fig7_10.png}
\caption{Latencies of the BNN Software Implementations (L9)}
\end{figure}

If all communication effort is respected, the slowest software execution time is approximately five times faster than the fastest FPGA implementation (BNN9). On the other hand, if the mere execution times of the microprocessor are compared to the reconfigurable logic, as depicted in Fig. 7.11, the proportions get more diverse. The significantly higher operating frequency and the employed number of transistors of the microprocessor do not translate into an equivalent speedup.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{fig7_11.png}
\caption{Execution Latencies of calculate(...) (L9)}
\end{figure}

Although the most implementations of the considered operation are equivalent, such as on the reconfigurable logic, there is a significant variance in the execution times. This is due to side effects introduced by caching, dynamic branch prediction, and concurrently executing programs.

Fig. 7.12 illustrates the average software compilation times of MOCCA. Software compilation is nearly six times faster than hardware synthesis (→ Tab. D.70).

As already discussed in Section 4.4.2, the estimation of software implementation characteristics is not considered in this thesis.

For the purpose of presentation the given example is small. Tests with significantly more complex logic designs and BNNs with up to 22 logic levels showed similar results however. Although this is rather counterintuitive at a first glance this behavior is a property of this particular computer system. It illustrates impressively that platform mapping is a system-level decision which must be supported by the employed approaches and tools. For this it is of high importance, that different implementations can be generated rapidly without having to change the system design. If different implementations still do not meet the non-functional requirements changes in the design or the platform become necessary.

The example shows the importance of communication to platform mapping. In order to increase system performance, objects must be partitioned among the processing elements such that the communication effort
7.3 Online Compression of Audio Streams

7.3.1 Problem Description

The topic of the second project is the multi-channel streaming of high-quality digital audio information [270]. Thereby the audio data (16..24 Bits/sample, 44..192 kHz sample rate, 10..24 channels, lossless encoding) is distributed from a dedicated audio server to multiple audio clients over existing LAN infrastructure using the TCP/IP protocol family. Auxiliary traffic is explicitly allowed. The envisaged application domains are professional audio installations of buildings, theatres, concerts, and universities. In contrast to traditional solutions, the analog distribution of the audio data and the cabling effort is avoided, which reduces cost and increases flexibility. To cut cost, the server is built from standard PC components and the clients are realized as resource restricted embedded system using an embedded microprocessor.

Audio Server and Clients Functionality

Fig. 7.13 shows the functionality performed by the participating components. The server reads the input from audio sources, like files or sound cards (INPUT). Each input channel is optionally compressed in order to save network bandwidth (ENCODE). On the compressed data additional measures for the handling of packet loss are applied (INTERLEAVING and ERROR HANDLING). Finally, the data is transmitted over the network (NET). The system control and the synchronization of the audio system is realized in the function blocks CTRL and SYNC respectively. The clients must reverse all error handling and compression performed by the server. The final output of the data is performed by the function block PLAY.

Linear Predictive Audio Compression and Decompression

One challenge of this setup is the online compression and decompression of multiple audio streams (ENCODE, DECODE). Common algorithms for this problem are AudioPaK, free lossless audio codec (FLAC),

---

5 The project was executed in cooperation with an industrial partner and was funded by the Federal Ministry of Education and Research, Germany (FKZ 0314731B).
These algorithms use linear predictive coding to model the digital waveform. Linear prediction exploits the property of audio data that consecutive samples are typically correlated. The predicted waveform \( \hat{x}(t) \) is modeled as a linear combination of \( p \) previous samples:

\[
\hat{x}(t) = \sum_{i=1}^{p} a_i x(t - i)
\]

\( \hat{x}(t) \) – Predicted waveform  
\( x(t) \) – Original waveform  
\( t \) – Number of sample  
\( p \) – Number of previous samples  
\( a_i \) – Prediction coefficients

Then, the transmitted waveform is the residual signal of the predicted waveform and the original waveform:

\[
e(t) = x(t) - \hat{x}(t)
\]

\( \hat{x}(t) \) – Predictor waveform  
\( x(t) \) – Original waveform  
\( e(t) \) – Transmitted waveform

High quality predictors decorrelate the waveform and thereby reduce its frequency spectrum. The problem is the optimal setting of the prediction coefficients \( a_i \), because this setting depends on the waveform itself and varies over the time. Thus, frequently a fixed set of coefficients, called predictor, is used to predict reasonable sequences of samples, called frames or blocks. Commonly, a restricted form of linear prediction is used that selects the predictor from a set of predefined polynomial predictors of order \( p \) [271], whereas the following predictors \( \hat{x}_0, \hat{x}_1, \hat{x}_2, \hat{x}_3 \) are used:

\[
\begin{align*}
\hat{x}_0(t) &= 0 \\
\hat{x}_2(t) &= 2x(t - 1) - x(t - 2) \\
\hat{x}_1(t) &= x(t - 1) \\
\hat{x}_3(t) &= 3x(t - 1) - 3x(t - 2) + x(t - 3)
\end{align*}
\]

During intra-channel decorrelation the predictor is chosen which minimizes the total of the absolute values of the residuals in the block. The residual signal is encoded by a variable number of bits per sample using Golomb codes [274]. Golomb codes are optimal for positive integer values with an exponentially decaying probability distribution.

While these algorithms provide relatively high compression ratios they are not very efficient in terms of computation. First, compression handles each sample twice (intra-channel decorrelation and encoding). Further, the Golomb encoding requires many operations on single bits whose execution is expensive on microprocessors. Due to the large number of concurrent channels this is critical for the audio server. The clients are affected as well, due to the performance restrictions of the employed hardware. Consequently, when fixed predictors are used with FLAC, the algorithm defines the additional predictor \( \hat{x}_4(t) = 4x(t - 1) - 6x(t - 2) + 4x(t - 3) - x(t - 4) \) [273].

The compression ratio of these algorithms is approximately 0.53–0.55 [273]. However, this value depends on the encoded material and the detailed settings of the algorithm.
both hardware architecture were augmented by reconfigurable logic\(^8\). For the design of the respective algorithms of this sub-part of the system the presented approach was used. For the purpose of this thesis the experimental results of the server are presented.

### 7.3.2 Experiments

A design model was developed for the compression in the server. The development of the other function blocks was considered but not done due to the early state of MOCCA at that time. AudioPaK and FLAC have been chosen for encoding, because these algorithms offer a good compression ratio.

**Example 7.2:** Fig. 7.14 outlines the design of the audio server. The encoding algorithm is performed by the operation `encode()`, which is implemented by the classes `FLACEncoder` and `AudioPaKEncoder`. To make the system extensible by other algorithms using linear predictive encoding/decoding or Golomb codes the respective functionality and interfaces are decomposed into two abstract classes `GolombEncoder` and `LPEncoder`. The class `Main` instantiates one or more encoder objects and invokes their encoding operation. The previous examples suggested that the workload put on the reconfigurable logic must outweigh the communication overhead. Further, potential concurrency should be exploited as often as possible. Thus, the logic performs the intra-channel decorrelation and encoding of entire audio frames asynchronously to the microprocessor. Details of the design can be found in Section D.3.1 on page 252.

As target platform for the server the same platform as in the previous example was used (→ Section B.2 on page 189, Section B.3 on page 195, and Section B.4 on page 213).

As the BNN example suggested, the design was implemented at the highest level of optimization. The partitioning of the functional components is implicit by design, because C/C++ has no meaningful and portable native support for bit-arrays\(^9\). The encoder was tested at its maximum operation frequency (45 MHz) using different frames.

---

\(^8\) The clients have been implemented using a hardware architecture which augmented a Leon-3 RISC processor by a Spartan-3 FPGA, both running at 33 MHz [20, 275, 276]. Similarly to the server, the decoding (DecoDE) was implemented in the FPGA while the microprocessor performed the other function blocks in Fig. 7.13(b).

\(^9\) Some C compilers support single bit operations and bit-arrays natively, however, the according code is not portable among different compilers.
7.3.3 Evaluation

Fig. 7.15 shows the timing of the server implementation (→ Tab. D.71, D.72). The latencies depend linearly on the number of samples. The meaning of the values is as follows:

\[
\begin{align*}
    t_{\text{read}} &= \text{Time for reading the encoded output.} \\
    t_{\text{exec}} &= \text{Encoding time} \\
    t_{\text{write}} &= \text{Time for writing the output to be encoded} \\
    t_{\text{comm}} &= t_{\text{read}} + t_{\text{write}} \\
    t_{\text{sum}} &= t_{\text{comm}} + t_{\text{exec}}
\end{align*}
\]

The overall time taken for the largest frame adds up to approximately 2.3 ms, which, in theory, allows the encoding of about 435 frames per second. Since a respective 96 kHz mono audio stream comprises nearly 84 frames per second about five streams can be encoded using the same object.

![Fig. 7.15: Latencies of the Audio Server](image)

Practically, there is overhead for the execution of the other functions performed by the server. However, the main processor of the system can work concurrently to the reconfigurable hardware. Further, if DMA is used, the negative side effect of the communication can be reduced to almost zero. The employed FPGA allows for the instantiation of up to nine encoder objects (→ Tab. D.73). Thus, either multiple encoders can be used in parallel or a smaller device can be selected in order to reduce cost. The FSMD of the encoder has 91 states. This implementation allocates about 50% of the resources of our previously presented designs [277]. This is caused by improvements in the algorithm design and the model compiler.

The percentage area estimation error of this design is -7%. However, this value indicates that the employed estimator can not be used for worst-case estimation. Since the run-time of the algorithm depends on the frame size, the according timing estimates are inappropriate for reasons explained earlier.

7.4 Modeling of Graphical User Interfaces

7.4.1 Problem Description

The flexible, platform-independent modeling of graphical user interfaces (GUIs) has been a permanent problem in system-level design. Apart from common applications of desktop computers an increasing number of embedded devices, such as organizers, cellular telephones, and vehicles, contain such interfaces. More often than not, user interfaces are constructed from specialized GUI libraries that define the basic appearance and behavior of the dialog elements. Unfortunately, these libraries are incompatible to each other, which makes porting of a modeled interface among different libraries a very costly and lengthy task.
7.4.2 Experiments

Thus, a sub-project of MOCCA investigated whether the platform-based, model-driven development approach that has been presented in this thesis, offers solutions to this problem. The respective approach and the experimental results are just summarized here, a detailed discussion can be found in [278]. The basic observation was that the different GUI libraries are basically specialized implementation platforms. The idea was then to abstract from particular libraries and create a design platform that provides fundamental elements for dialog modeling. Then each application and its GUI is defined in terms of this abstract design platform, rather than a specialized library. Implementations for a particular library are generated automatically by using the according implementation platform.

This ambitious approach was evaluated using two reasonably complex application designs - a dialog manager and an address book. Further, a dedicated design platform was modeled, containing primitive types, i.e. the core types shown in Section B.1, and abstract types for modeling dialog elements, such as labels, buttons, events, and containers. For the GUI libraries Java Swing, which is a native part of the Java class libraries, and services for nomadic workers (SNOW)-web implementation platform models were created [279]. A standard desktop PC was used as experimental platform.

7.4.3 Evaluation

The automatic transformations of the applications to the Swing Framework was quite unproblematic. For the SNOW platform the implementation was not successful however. The current direct mapping between platform models requires the participating models not only to have an equivalent functionality, but also to share a similar principal structure. SNOW-web employs a hybrid approach for the description of dialog elements using Java and XML, whereas XML is used to serialize dialog elements as strings. This is no conceptual problem of the presented approach however. It can be solved straightforwardly by using an appropriate implementation platform. First of all, the generation of XML is not supported by the current version of MOCCA. Second, the experiment also showed that MOCCA can be extended by the required compiler components for mapping and generation. It was suggested to create a dedicated back-end for SNOW, which is accompanied by a implementation-platform profile specialized to this platform. In general, similar solutions are feasible for all implementation-platforms that use a mix of languages or different dialects of the same language.

A second conclusion of this project regards modeling. In this experiment the user interface had to be modeled using UML and MAL. The instantiation, configuration, and associations of all used dialog elements had to be done explicitly. Graphical GUI development is most appropriate and state-of-the-art. Thus, MOCCA should be accompanied by a graphical interface modeling tool that generates the according design model. The explicitly defined UML models for platforms and designs provide the appropriate means to decouple the modeling tool from the compiler.
8. CONCLUSIONS

This thesis discussed the modeling, platform mapping, synthesis, and execution of object-oriented system-level models. The presented work creates novel, interesting methodological opportunities for the system-level development of hardware/software systems. In the following, the contributed work in these fields is concluded and directions for future research are discussed.

Modeling

During modeling, a system is decomposed into the application specific logic and the platforms being used for its design, implementation, and deployment. Both, the application and the platform, are defined using models. The application specific models define different perspectives on the functionality that solves a particular problem. Platforms provide the foundation to define such perspectives. UML and MAL are used for the complete and precise modeling of applications. Thereby the modeling of generic designs that are independent from particular implementation platforms is encouraged. This enables the straightforward generation of different implementations from the same design.

UML 2.0 and object-orientation proved to be appropriate means for system-level modeling of hardware/software systems. It has been shown that UML already contains all important concepts and constructs being demanded by a system-level modeling language. Necessary extensions can be defined using the native extension mechanisms. There is only few need, if any, to enrich the core language through meta-model extensions. In fact, this should be considered the last alternative since such extensions hamper portability, comprehensibility, and standardization.

In this thesis, the adaption of the object paradigm to system-level design is emphasized. This paradigm is suitable to a wide range of applications and helps managing design complexity and comprehensibility. On the other hand, message-based communication can occasionally be a performance and conceptual bottleneck. Future extensions may overcome this by providing additional communication paradigms such as streaming\(^1\).

Similar considerations apply for the definition of computations using the imperative paradigm. Future extensions should support the full set of UML activities, as well as the other behavior modeling facilities. Further, declarative modeling such as OCL inclusions should be investigated. Thereby the embedding of at least the standardized constructs for behavior modeling into each other should be supported. This would enable users to choose the behavior modeling facility that is most appropriate to the particular problem. Adding behavior modeling facilities should be accompanied by the development of respective support for platform mapping and synthesis [208, 261].

Platform Mapping

The novel type of system-level modeling necessitated the development of according approaches to implementation. The implementation process is decomposed into two separate activities called platform mapping and synthesis. Platform mapping is the transformation of an application-specific design model into functionally equivalent models for implementation and deployment. All models are defined in terms of the respective platforms. For this, well-defined transformations are applied to the design and each implemented model element is associated with the resource services being required for its implementation.

\(^1\) As of UML 2.0 activities allow for the modeling of data flow and buffers with FIFO semantics.
To account for the model hierarchy, a novel, multi-granular mapping algorithm and a data-structure called *mapping graph* have been presented. This approach treats model elements with different semantics uniformly and thereby simplifies its integration with different algorithms for design space exploration. Multiple platforms for implementation and deployment are supported by separating the platform-specific parts of the algorithm from the common parts. Thereby the design of mapping algorithms is simplified; respective algorithms known from literature can be integrated in a straightforward manner. The algorithm takes as input partial mappings or completely unmapped designs. The output is an implementation model and a deployment model.

The principal approach and the employed transformations proved successful to create high quality implementations of application designs. The respective design space is defined by the principal system architecture and the architecture defined by the platforms. Lower level design flows often further improve the implementations. This aggravates system-level estimation of implementation characteristics. Because these characteristics steer platform mapping and automated partitioning better approaches must be developed that consider the aforementioned observations.

The presented estimation approach for area and time of FPGA implementations are reasonably accurate for the system-level. For software implementations the computation of reliable performance estimates is of high importance, because this information is required for automatic partitioning! Estimation must regard effects introduced by the micro-architecture and the lower level designs flows. However, this is by principle impossible to model accurately. Successful directions may be found in the field of machine learning, which has recently been applied to logic estimation [269].

The experiments performed for the flexible and portable modeling of GUIs suggested to define further transformations that allow for more ambitious mappings between platforms [278]. The current approach employs one-to-one mappings, which are too limited for this purpose due of the large differences between the GUI libraries. Experience showed, however, that the existing compiler infrastructure and the underlying approach allow for the straightforward integration of such specializations.

There is still much unused optimization potential that should be used to improve design quality and extend the design space being explored. A wide range of known optimizations will likely improve implementation quality, such as Boolean simplifications and speculative versions of the optimizations currently being supported [159, 229]. Another important extension will be the opportunity to select among different behaviors of the same operation (→ Section 4.3.4).

**Synthesis**

The full synthesis of the core features of object-orientation - inheritance, polymorphism, data abstraction, and encapsulation - encourages their usage in system design. For software the implementation of these features has been available for many years. This thesis has demonstrated their effective implementation using logic resources. The presented work has shown how complete UML design models can be synthesized automatically into optimal hardware/software implementations. The presented work is the first solution to this problem.

The synthesis approach and the underlying architecture proved appropriate for the envisaged application domains. Other architectures are imaginable and should be investigated in the future. For instance, related research observed that in presence of appropriate library primitives control flow can be transformed into equivalent data flow [27, 33, 76, 237], which can reveal new optimization opportunity. Application-specific or even object-specific instruction set processors are another option, which may offer high performance at low area cost per instruction in combination with the possibility of recurrent execution.

**Run-Time Reconfiguration**

Applications of run-time reconfigurable architectures require run-time management of configurations and objects. An optimized life cycle for these entities, the grouping of multiple objects into the same configuration, optimized algorithms for dynamic selection of configurations and objects, and the support of
inheritance help in reducing the reconfiguration overhead. The utilization of a HAL and proxies makes implementations relatively independent of the physical hardware architecture, because architecture specifics can be encapsulated by the proxies.

**Run-Time Reconfigurable Architectures**

In the recent years, different run-time reconfigurable architectures have become available at affordable cost. These computer architectures can be an efficient means for the implementation of high-performance systems in various application domains. In conjunction with appropriate development approaches and tools, these architectures can also reduce the design complexity, because the most suitable design style can be chosen for the formulation of a problem into a design.

The selection or development of a hardware architecture that is most appropriate to a problem is a permanent challenge in system development. System architects must consider both, economical and technological implications of a particular architecture. This thesis focused on the application development for network-coupled architectures, because they are arguably the most common type of reconfigurable hardware architecture. In contrast to data-path coupled architectures, network-coupling introduces significant communication overhead, which can dominate the overall computation time.

As the experimental results have shown, this problem can be tackled in the system design. But also better hardware architectures and usage models should be discussed. In particular, for high-performance applications the reconfigurable fabric must be integrated tightly with the microprocessor. Thereby the amount and type of work that is performed by the RF and the required flexibility must determine the coupling. To accomplish different requirements, a hierarchy of reconfigurable fabrics, similar to the memory hierarchy of modern computer architectures, is imaginable. Few but fast reconfigurable resources with high communication bandwidth are integrated directly with the microprocessor. They are complemented by slower, but large and cheap remote resources. According to the computational requirements of the application the tasks being performed by the system may migrate dynamically in the hierarchy.

**Methodology**

The presented approach meets the overall goals that have been defined in Chapter 1. Object-oriented system-level development reduces design complexity, because it supports abstraction, decomposition, and reuse. The high degree of automation encourages the exploration of different designs at system-level. Relatively low compilation times enable the incremental and iterative development of systems or system components. This system-in-the-loop approach offers major advantages over traditional development flows. Still, the approach closely aligns with the common process models. Future work shall investigate the problem of defining and selecting platforms that provide the proper abstractions being useful for the envisaged domains, yet allowing for optimal and efficient implementations.

The opportunity to synthesize different yet efficient implementations from the same design model offers an efficient means of system prototyping. Application development and functional tests can start very early using a software implementation platform. When the hardware becomes available the functionality can be partitioned respectively. This allows for an early evaluation of the hardware and can be used to identify bottlenecks in the system design. Also, the porting of the application to another hardware platform is supported. Respective target platform models can commonly be derived from existing models, because often just the QoS-characteristics and the implementation of the defined proxies must be adapted.

The necessity of system-level development and the strength of the proposed system-level approach is substantiated by the presented experimental results. Although the particular BNN in Section 7.2 turned out to be implemented best using the microprocessor on the employed hardware platform, this information has become available swiftly. This is indeed a tremendous progress in comparison to previous approaches since it enables seeking for the optimal system architecture at affordable cost and time.

Due to the focus of this work to hardware/software codesign the level of abstraction of the majority of employed platforms and designs is relatively low. The GUI example illustrated that the presented concepts can be applied to other application domains and levels of abstraction as well. Thereby the design platform
8. Conclusions

offers a means of standardization. Customization is accomplished through target platforms. As discussed in the introduction, the necessity of standardization and the market demand for customization are frequent contradictory requirements in most areas of computer systems development. This thesis offers solutions and an implementation framework to this challenge and thereby broadens the significance and applicability of the presented work.
APPENDIX
A. MOCCA MODELING FRAMEWORK

A.1 MOCCA Action Language

A.1.1 Restrictions and Extensions to Java

The MAL syntax and semantics of statements and expressions is similar to Java. All statements for blocks, conditional execution, loops, and exceptions are directly applicable. The following Java language constructs are not available in MAL, since they are either considered problematic or redundant to UML:

- all constructs that define design modules: packages, classes, interfaces,
- all constructs that define features: fields, operations,
- conditional operator (?:), and
- synchronized-statement.

In addition to the Java language specification [215], MAL defines the following statements and operators:

**countof** - The countof-operator determines the current length of a given array. The operand is an array reference. It returns the current length of the first dimension of an array.

**Syntax:**

```
CountofExpression =
    'countof' ArrayReference
```

**Priority:** same as operator **new**

**Objective:** Java provides similar functionality by appropriate methods or attributes. Because this is not a generic and portable mechanism, in MAL a dedicated operator is used.

destroy - The destroy-operator is used to delete instances and arrays. The operand is a reference to the instance or array. The operator has no return value. The operator works nested; if an array is destroyed all referenced elements are destroyed.

**Syntax:**

```
DestroyExpression =
    'destroy' ( InstanceReference | ArrayReference )
```

**Objective:** UML models cannot generally rely on the existence of garbage collection. Unused instances and arrays must be explicitly freed in order to avoid memory bottlenecks.

async - The async-operator marks an operation call to be asynchronous. The operand defines the receiver and the operation to be called as well as a number of the message arguments. The operator has no return value.

**Syntax:**

```
AsyncExpression =
    'async' MethodInvocation
```

**Objective:** The purpose of this statement is to overcome the restriction of the Java execution model to synchronous method invocations, in order to be able to better exploit multi-processor hardware.
A.1.2 Mapping of MAL to UML Actions and Activities

The graphical representation of the mappings in this section uses the compacted notation that has been presented in Fig. 3.3 on page 30. For clarity, the examples are presented as object diagrams.

Sequencing between Statements

The sequencing of statements can be determined by data dependencies and/or control dependencies. Each data dependency maps to an instance of InputPin, ObjectFlow, and OutputPin. The data source is connected to the output pin. The data sink is connected to the input pin.

If sequencing is not defined otherwise, each control dependency maps to an instance of InputPin, ControlFlow, and OutputPin. The control predecessor (source) is connected to the input pin. The control successor (target) is connected to the output pin.

![Object Flow Sequencing](imageA.png)

(a) Object Flow Sequencing

![Control Flow Sequencing](imageB.png)

(b) Control Flow Sequencing

Fig. A.1: Sequencing between Statements

Statement Blocks

Each statement block that does not represent an exception handler maps to an instance of SequenceNode. The statements in the block are contained nodes (containedNode) of the block. The sequencing between is modeled by the order in which the statements are stored in the containedNode property. Additionally, sequencing can be made explicit using instances of ControlFlow.

Exception handlers, i.e. catch-statement blocks, map to an instance of ExceptionHandler. The body of the block maps to an instance of ExecutableNode. All statement blocks that are protected by the exception handler, i.e. try-blocks, map to an instance of ExecutableNode. They are connected to the handler property of the ExceptionHandler instance. The exception object that is caught by the handler is connected to the exceptionInput property. The most specific type of the exception object maps to the exceptionType property. By definition, the exception object can have only one type.

Conditional Statements

Conditional statements map to instances of ConditionalNode and instances of Clause. If multiple clauses exist, the clauses are evaluated in sequence. The sequence is defined using the predecessorClause and successorClause property of Clause. Implementations can evaluate multiple clauses concurrently, when the tests evaluate to a constant value. The isDeterminate property is always true.
A.1. MOCCA Action Language

_activation_language_ try
{
  _body_
}
activate_language_ catch( Exception _ex_ )
{
  _body_
}

**Fig. A.2:** Mapping of *try-catch*-Statement

**if-Statement.** When the conditional statement is an *if*-statement, there is at least one instance of **Clause** for the true branch. The condition of the *if*-statement is the test of the clause of the true branch. By definition, the condition has one output pin, that carries a true value whenever the condition evaluates true. The **decider** maps to the output pin of the condition. The statement that represents the true branch maps to the body of the clause. If there is no false branch, the *isAssured* property is set false. Otherwise, this property is set true.

If a false branch exists, it maps to an instance of **Clause**. This clause is a **successorClause** of the clause representing the true branch. The statement executed by the false branch maps to the body of this clause. The test maps to an instance of **ValueSpecificationAction** whose **ValueSpecification** evaluates always true. The **decider** is connected to the output pin of the action instance.

**switch-Statement.** When the conditional statement is a *switch*-statement, there is at least one instance of **Clause**. The expression of the switch maps to an activity node with one output pin which carries the value of the expression. If there is no default branch, the *isAssured* property is set false. Otherwise, this property is set true.

Each case maps to an instance of **Clause**. The clauses are sequenced according to the order of specification in the MAL statement. The test of each clause is represented by an instance of **OpaqueAction**. This action compares the equality of the expression of the *switch* and the (constant) value of the *case*. If the comparison evaluates true, it carries true at its output pin. In this case the body of the respective clause is executed. The next clause is evaluated otherwise. If the last clause in the sequence does not evaluate true no body is executed.
The body of each case maps to a dedicated activity node. Fall-through between cases is modeled using control flow sequencing between their bodies. If the body of a case is empty, the body of its clause maps to the next non-empty body in the evaluation sequence of clauses. If the body of the last case is empty, the body property of the respective clause is also empty.

![Diagram](image)

**Fig. A.4:** Mapping of switch-Statement

All break statements within a switch map to an instance of OutputPin, ControlFlow, and InputPin. The output pin is linked to the activity node representing the statement that is executed immediately before the break. If the respective case is empty, the input pin is connected to the decider of the respective clause. In case of fall-through the input pin is additionally connected to the deciders, or, if exists, the body activity, of all clauses that are control flow predecessors of the break. The output pin is connected to the control flow input pin of the successor of the switch statement.

**Loops**

Loop statements map to instances of LoopNode. In case the loop is a while loop or a for loop the isTestedFirst property is set true. In case the loop is a do-while loop this property is set false.

Instances of the break-statement within a loop map to an instance of OutputPin, ControlFlow, and InputPin. The output pin is the control flow output pin of the activity that immediately precedes the statement. The input pin is connected to the control flow output pin of the loop by means of a control flow.

If a continue-statement is executed within a for loop that has a non-empty update, the statement maps to an instance of OutputPin, ControlFlow, and InputPin. The output pin is the control flow output pin of the activity that immediately precedes the statement. The input pin is the control flow input of the activity node that represents the update.

**for-Statement.** If not empty, the setup of the loop is represented by a respective activity node. This activity node represents the setupPart of the LoopNode instance. The condition maps to the test of the loop. The test must have one output pin that carries true if the condition is true. The body of the loop maps to an respective activity node that is connected to the bodyPart property. The update of the loop maps to an activity that is executed after the activity representing the body. This is accomplished using control flow sequencing.
while-Statement. The mapping of this statement is similar to the for-statement. By definition, the setup is empty. In contrast to the for-statement, no explicit update exists.

do-Statement. The body of the loop maps to an activity node that is connected to the bodyPart property of the LoopNode instance. The setupPart is always empty. The condition maps to an activity node. This activity node has one output pin that carries true if the condition holds. This activity node is connected to the test property. The respective output pin links to the decider.

Actions

Operation Calls. Each operation call maps to an instance of CallOperationAction. The target property carries the object on which the operation is called. The operation property denotes the called operation. If this operation is polymorphic the operation that is actually called depends on the dynamic type of the target object (inclusion polymorphism). In adherence to UML, the inputs to the action carry the arguments of the operation call. If asynchronous, (async) the isSynchronous property is set false, otherwise it is set true. To accept the call in the receiver, each activity can be invoked by a call that executes an instance of AcceptCallAction.
**return-Statement.** Each return-statement maps to an instance of `ReplyAction`. The `returnInformation` property directly connects to the respective output of the instance of `AcceptCallAction` of the operation that executes the `ReplyAction`. If a value is returned to the caller, the `replyValue` carries the object representing this value. The `replyToCall` property carries the call event.

**throw-Statement.** The throw-statement maps to an instance of `RaiseExceptionAction`. The input pin that is associated with the `exception` property carries the exception object being thrown.

**Access to Attributes/Variables.** Read and write accesses to attributes map to instances of `ReadStructuralFeatureAction` and `AddStructuralFeatureAction` respectively. The `object` property denotes the object whose attribute that is accessed. The `structuralFeature` property maps to the accessed attribute. In case of write access, the `isReplaceAll` property is set true for scalar attribute types, otherwise it is set false.

Read and write accesses to other variables, i.e. parameters and local variables, map to an instance of `ReadVariableAction` and `AddVariableValueAction` respectively. The `variable` property denotes the variable that is accessed. In case of `ReadVariableAction` the result pin carries the value that was read. In case of `AddVariableValueAction` the value pin carries the written value. The `isReplaceAll` property is set true for scalar variable types, otherwise it is set false.

**new-Operator.** Each new-operator maps to an instance of `CreateObjectAction`. The result property carries the created object. The `classifier` property maps to the instantiated classifier. By definition, this classifier is constrained to be a class. The constructor invocation maps to an operation call.

**destroy-Operator.** Each destroy-operator maps to an instance of `DestroyObjectAction`. The target output pin carries the destroyed object. The properties `isDestroyLinks` and `isDestroyOwnedObjects` are set to false. The destructor invocation maps to an operation call.

**instanceof-Operator.** Each instanceof-operator maps to an instance of `ReadIsClassifiedObjectAction`. The action tests if an object is classified by a given type, or one of its super-types. Hence, the `isDirect` property is set to false. The `classifier` property refers to the classifier to test. The object input pin carries the tested object, and the result output pin carries the outcome of the test.
Equality-Operator. Each equality-operator, i.e. `==`, if invoked on objects of user-defined classifiers, maps to an instance of `TestIdentityAction`. The action tests if two objects are the same identity. The input pins first and second refer to the tested objects. The result output pin carries the outcome of the test.

Other Expressions. Each expression for which none of the previously defined mappings applies, is mapped to an instance of `OpaqueAction` or, in case the expression evaluates to a constant value, an instance of `ValueSpecificationAction` and `OpaqueExpression`.

In case of expressions that do not evaluate to a constant value, the body of the action instance is the name of the executed core operation. See Section A.2 for the list of core types and operations. The language is MAL. Each operand of an operator is mapped to an input pin. The result is mapped to an output pin. For binary operators there are two designed inputs pin called `left` and `right`. They are mapped to the left and right operand respectively.

Instances of `ValueSpecificationAction` are used to represent constant values. The value property connects to an instance of `OpaqueExpression`. The body of the expression defines the constant value. The language is MAL. The language is frequently not shown in diagrams if it is clear from the context.

The data flow between expressions maps to an instance of `OutputPin`, `ObjectFlow`, and `InputPin`. The output pin is connected to the activity node that produces the value. The input action is connected to all activity nodes that consume the value.

A.2 Core Data Types and Operations

A.2.1 Core Data Types

MOCCA defines a number of core data types. These data types are necessary for automatic reasoning about design properties, and to implement model transformations. For each type its domain and the applicable operations must be defined. In principle both properties can be freely modeled. As discussed in Section 3.3.2, it is sensible to associate core data types with a predefined semantics and to require them to provide a minimum set of interpreted operations called the set of core operations of the type. In this section the core data types of MOCCA are presented. Section A.2.2 presents the core operations of these types.

Each core data type, being defined by the MOCCA modeling framework, falls into one of the following groups:

- **Base Types**: `object`, `object[]`, `classifier`
- **Boolean Types**: `boolean`
- **Magnitude Types**
  - **Integral Types**: `bit`, `byte`, `short`, `int`, `long`
  - **Floating Point Types**: `float`, `double`
  - **Time Types**: `time`
- **Character Types**: `char`, `string`
- **Auxiliary Types**: `type`, `remote`, `process`, `guard`

Apart from the base types, there is no implicit type hierarchy. For all other types, the super-type can be determined by the user. If for some type no explicit super-type is defined and the type is not declared to be a root type, MOCCA automatically infers the super-type. If not specified otherwise, the super-type is `object`. 
### A. MOCCA Modeling Framework

#### Tab. A.1: MOCCA Base Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Generalization</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>-</td>
<td>The object-type is the super-type of all other types. The null literal can be assigned to all variables of this type.</td>
</tr>
<tr>
<td>object[]</td>
<td>object</td>
<td>The object[]-type is the super-type of all arrays.</td>
</tr>
<tr>
<td>classifier</td>
<td>object</td>
<td>The classifier-type is the super-type of all classes and interfaces.</td>
</tr>
</tbody>
</table>

#### Base Types

Base types build the infrastructure of the type hierarchy. They represent the super-types of all types, including user defined types. All base types are abstract.

#### Boolean Types

There is one predefined type in this group; the boolean-type. Instances of this type are used to represent logical values. Instances of this type can be assigned with the literals true or false.

#### Magnitude Types

Magnitude types represent contiguous ranges of integral quantities, real quantities, or time quantities. The domain of magnitude types is not fixed. The boundaries of the domain of each type must be modeled using the LowerBound and UpperBound constraints (→ Section A.5.1).

#### Integral Types

Instances of integral types represent the common integer quantities. A relatively rich number of integral types is used in order to allow model compilers to select the most appropriate type whenever a transformation creates variables that are not part of the user design.

#### Tab. A.2: MOCCA Integral Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Generalization</th>
<th>Recommended Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit</td>
<td>user determined</td>
<td>[0, 1]</td>
</tr>
<tr>
<td>byte</td>
<td>user determined</td>
<td>[-128, 127]</td>
</tr>
<tr>
<td>short</td>
<td>user determined</td>
<td>[-32768, 32767]</td>
</tr>
<tr>
<td>int</td>
<td>user determined</td>
<td>[-2147483648, 2147483647]</td>
</tr>
<tr>
<td>long</td>
<td>user determined</td>
<td>[-9223372036854775808, 9223372036854775807]</td>
</tr>
</tbody>
</table>

#### Floating Point Types

Instances of floating point types represent approximations of real value quantities. For the same reasons as for integral types, different floating point values are distinguished.

#### Tab. A.3: MOCCA Floating Point Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Generalization</th>
<th>Recommended Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>user determined</td>
<td>[-1.40129846432481707e-45, 3.4028234663852885981170418348452e+38]</td>
</tr>
<tr>
<td>double</td>
<td>user determined</td>
<td>[-4.94065645841246544e-324, 1.79769313486231570e+308]</td>
</tr>
</tbody>
</table>
A.2. Core Data Types and Operations

**Time Types.** Time types represent time quantities. In current systems time quantities are emulated using integral types. Since there is no common agreement on a time basis however, such specifications are not portable among different platforms. Dedicated time types can circumvent this problem.

There is one predefined type in this group - the \texttt{time}-type. The recommended domain of this type is \([-9223372036854775808\text{ps}, 9223372036854775807\text{ps}]\). The super-type of this type is determined by the user, and defaults to \texttt{object} if it is left undefined.

**Character Types**

Instances of character types represent a single character or sequences of characters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Generalization</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>user determined</td>
<td>A single character. Recommended domain: ([\text{u0000}, \text{u00ff}], \text{i.e. [0, 65535]})</td>
</tr>
<tr>
<td>string</td>
<td>user determined (default: \texttt{object[]})</td>
<td>A sequence of characters.</td>
</tr>
</tbody>
</table>

**Auxiliary Types**

Auxiliary types are used by the model compiler to work with types, and to implemented specific features of the presented approach, such as active classes, guards, and remote objects.

<table>
<thead>
<tr>
<th>Name</th>
<th>Generalization</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>user determined</td>
<td>This type represents the type concept. It can be used whenever the type concept as such, and not a particular type is meant. An example is the \texttt{typeof}-operator, that is used to check if an object has a given type. The argument of the respective core operation is 'type' to refer to types in general.</td>
</tr>
<tr>
<td>process</td>
<td>user determined (default: \texttt{classifier})</td>
<td>Instances of this type represent a own thread of control, i.e. process, task, or thread. They are used to implement active classes. This type is the super-type of all active classes.</td>
</tr>
<tr>
<td>guard</td>
<td>user determined (default: \texttt{classifier})</td>
<td>Instances of this type represent a synchronization primitive for critical regions. Guards have a mutex semantics. Only one thread of control, i.e. process, can be the current owner of the guard. The execution of all other threads that request ownership, is blocked until the current owner releases the guard.</td>
</tr>
</tbody>
</table>

A.2.2 Core Operations

For each core data type a set of core operations is defined. As for all user-defined types the defined operations determine the functionality that can be realized with this type. Since types can be organized in hierarchies not all possible core operations of a type must be defined locally to the type definition. Instead, for each type the core operations of its super-types are also accessible.
### Tab. A.6: Core Operations of Base Types

<table>
<thead>
<tr>
<th>MAL Operator</th>
<th>Action Name</th>
<th>Core Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>new</td>
<td>CreateObjectAction create(in type: type, in elements: int):object[] - create an array with 'elements' elements of type 'type'.</td>
</tr>
<tr>
<td>destroy</td>
<td>DestroyObjectAction destroy():void - destroy the current array.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>asgn</td>
<td>asgn(in val: object[]):void - assign an array object to the array object on which the operation is invoked.</td>
</tr>
<tr>
<td>countof</td>
<td>countof</td>
<td>countof():int - get the number of elements.</td>
</tr>
<tr>
<td></td>
<td>get</td>
<td>get(in index: int):object - get the element at index 'index'.</td>
</tr>
<tr>
<td></td>
<td>put</td>
<td>put(in index:int, in val: object):void - write an element to a specific index.</td>
</tr>
</tbody>
</table>

### Tab. A.7: Core Operations of Boolean Types

<table>
<thead>
<tr>
<th>MAL Operator</th>
<th>Action Name</th>
<th>Core Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>asgn</td>
<td>asgn(in val: boolean):void - assign an instance of boolean to the current boolean object.</td>
</tr>
<tr>
<td></td>
<td>eq</td>
<td>eq(in val: boolean):boolean - test if two booleans have the same value.</td>
</tr>
<tr>
<td></td>
<td>neq</td>
<td>neq(in val: boolean):boolean - test if two booleans do not have the same value.</td>
</tr>
<tr>
<td></td>
<td>not</td>
<td>not():boolean - returns the complementary value of the boolean object.</td>
</tr>
<tr>
<td></td>
<td>cond_and</td>
<td>cond_and(in val: boolean):boolean - returns the logical AND of two boolean values.</td>
</tr>
<tr>
<td></td>
<td>cond_or</td>
<td>cond_or(in val: boolean):boolean - returns the logical OR of two boolean values.</td>
</tr>
</tbody>
</table>
### Tab. A.8: Core Operations of Integral Types

<table>
<thead>
<tr>
<th>MAL Operator</th>
<th>Action Name</th>
<th>Core Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>xor</td>
<td>xor(in val: boolean):boolean - returns the logical XOR of two boolean values.</td>
</tr>
</tbody>
</table>

**type X, with X one of {bit, byte, short, int, long}**

<table>
<thead>
<tr>
<th>MAL Operator</th>
<th>Action Name</th>
<th>Core Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>asgn</td>
<td>asgn(in val: X):void - assign an instance of X to the current instance of X.</td>
</tr>
<tr>
<td>==</td>
<td>eq</td>
<td>eq(in val: X):boolean - test if two instances have the same value.</td>
</tr>
<tr>
<td>!=</td>
<td>neq</td>
<td>neq(in val: X):boolean - test if two instances do not have the same value.</td>
</tr>
<tr>
<td>&amp;=</td>
<td>and_asgn</td>
<td>and_asgn(in val: X):X - compute bitwise AND of two instances and assign result to the instance on which the operation is invoked.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>or_asgn</td>
</tr>
<tr>
<td>^</td>
<td>xor</td>
<td>xor(in val: X):X - compute bitwise XOR of two instances.</td>
</tr>
<tr>
<td>^=</td>
<td>xor_asgn</td>
<td>xor_asgn(in val: X):X - compute bitwise XOR of two instances and assign result to the instance on which the operation is invoked.</td>
</tr>
</tbody>
</table>

**Additional Core Operations for X one of {byte, short, int, long}**

<table>
<thead>
<tr>
<th>MAL Operator</th>
<th>Action Name</th>
<th>Core Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+=</td>
<td>add_asgn</td>
<td>add_asgn(in val: X):X - add two instances and assign result to the instance on which the operation is invoked.</td>
</tr>
<tr>
<td>-</td>
<td>sub</td>
<td>sub(in val: X):X - subtract two instances.</td>
</tr>
<tr>
<td>-=</td>
<td>sub_asgn</td>
<td>sub_asgn(in val: X):X - subtract two instances and assign result to the instance on which the operation is invoked.</td>
</tr>
<tr>
<td>*</td>
<td>mul</td>
<td>mul(in val: X):X - multiply two instances.</td>
</tr>
<tr>
<td>*=</td>
<td>mul_asgn</td>
<td>mul_asgn(in val: X):X - multiply two instances and assign result to the instance on which the operation is invoked.</td>
</tr>
<tr>
<td>/=</td>
<td>div_asgn</td>
<td>div_asgn(in val: X):X - divide two instances and assign result to the instance on which the operation is invoked.</td>
</tr>
<tr>
<td>%</td>
<td>mod</td>
<td>mod(in val: X):X - compute instance modulo 'val'.</td>
</tr>
<tr>
<td>%=</td>
<td>mod_asgn</td>
<td>mod_asgn(in val: X):X - compute instance modulo 'val' and assign result to the instance on which the operation is invoked.</td>
</tr>
<tr>
<td>»</td>
<td>shr</td>
<td>shr(in pos: int):X - shift instance 'pos' positions to the right.</td>
</tr>
<tr>
<td>»=</td>
<td>shr_asgn</td>
<td>shr_asgn(in pos: int):X - shift instance 'pos' positions to the right and assign result to the instance.</td>
</tr>
<tr>
<td>«</td>
<td>shl</td>
<td>shl(in pos: int):X - shift instance 'pos' positions to the left.</td>
</tr>
<tr>
<td>«=</td>
<td>shl_asgn</td>
<td>shl_asgn(in val: X):X - shift instance 'pos' positions to the left and assign result to the instance.</td>
</tr>
<tr>
<td>&gt;</td>
<td>gt</td>
<td>gt(in val: X):boolean - test if current instance is greater than 'val'.</td>
</tr>
</tbody>
</table>

*continued on next page*
### MAL Operator

<table>
<thead>
<tr>
<th>MAL Operator</th>
<th>Action Name</th>
<th>Core Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;=</td>
<td>gteq</td>
<td>gteq(\text{in val: } X):boolean - test if current instance is greater than or equal to 'val'.</td>
</tr>
<tr>
<td>&lt;</td>
<td>lt</td>
<td>lt(\text{in val: } X):boolean - test if current instance is less than 'val'.</td>
</tr>
<tr>
<td>&lt;=</td>
<td>lteq</td>
<td>lteq(\text{in val: } X):boolean - test if current instance is less than or equal to 'val'.</td>
</tr>
<tr>
<td>-x</td>
<td>uminus</td>
<td>uminus():X - negate instance arithmetically (0-x).</td>
</tr>
<tr>
<td>+x</td>
<td>uplus</td>
<td>uplus():X - compute absolute value of instance.</td>
</tr>
</tbody>
</table>

---

**Tab. A.9: Core Operations of Floating Point Types**

<table>
<thead>
<tr>
<th>MAL Operator</th>
<th>Action Name</th>
<th>Core Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>asgn</td>
<td>asgn(\text{in val: } X):void - assign an instance of X to the current instance of X.</td>
</tr>
<tr>
<td>==</td>
<td>eq</td>
<td>eq(\text{in val: } X):boolean - test if two instances have the same value.</td>
</tr>
<tr>
<td>!=</td>
<td>neq</td>
<td>neq(\text{in val: } X):boolean - test if two instances do not have the same value.</td>
</tr>
<tr>
<td>+</td>
<td>add</td>
<td>add(\text{in val: } X):X - add two instances.</td>
</tr>
<tr>
<td>+=</td>
<td>add_asgn</td>
<td>add_asgn(\text{in val: } X):X - add two instances and assign result to the instance on which the operation is invoked.</td>
</tr>
<tr>
<td>-</td>
<td>sub</td>
<td>sub(\text{in val: } X):X - subtract two instances.</td>
</tr>
<tr>
<td>-=</td>
<td>sub_asgn</td>
<td>sub_asgn(\text{in val: } X):X - subtract two instances and assign result to the instance on which the operation is invoked.</td>
</tr>
<tr>
<td>*</td>
<td>mul</td>
<td>mul(\text{in val: } X):X - multiply two instances.</td>
</tr>
<tr>
<td>*=</td>
<td>mul_asgn</td>
<td>mul_asgn(\text{in val: } X):X - multiply two instances and assign result to the instance on which the operation is invoked.</td>
</tr>
<tr>
<td>/</td>
<td>div</td>
<td>div(\text{in val: } X):X - divide two instances.</td>
</tr>
<tr>
<td>/=</td>
<td>div_asgn</td>
<td>div_asgn(\text{in val: } X):X - divide two instances and assign result to the instance on which the operation is invoked.</td>
</tr>
<tr>
<td>&gt;</td>
<td>gt</td>
<td>gt(\text{in val: } X):boolean - test if current instance is greater than 'val'.</td>
</tr>
<tr>
<td>&gt;=</td>
<td>gteq</td>
<td>gteq(\text{in val: } X):boolean - test if current instance is greater than or equal to 'val'.</td>
</tr>
<tr>
<td>&lt;</td>
<td>lt</td>
<td>lt(\text{in val: } X):boolean - test if current instance is less than 'val'.</td>
</tr>
<tr>
<td>&lt;=</td>
<td>lteq</td>
<td>lteq(\text{in val: } X):boolean - test if current instance is less than or equal to 'val'.</td>
</tr>
<tr>
<td>-x</td>
<td>uminus</td>
<td>uminus():X - negate instance arithmetically (0-x).</td>
</tr>
<tr>
<td>+x</td>
<td>uplus</td>
<td>uplus():X - compute absolute value of instance.</td>
</tr>
</tbody>
</table>
### Tab. A.10: Core Operations of Time Types

<table>
<thead>
<tr>
<th>MAL Operator</th>
<th>Action Name</th>
<th>Core Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>=</code></td>
<td>asgn</td>
<td><code>asgn(val: time):void</code> - assign an instance to the current instance.</td>
</tr>
<tr>
<td><code>==</code></td>
<td>eq</td>
<td><code>eq(val: time):boolean</code> - test if two instances have the same value.</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>neq</td>
<td><code>neq(val: time):boolean</code> - test if two instances do not have the same value.</td>
</tr>
<tr>
<td><code>+</code></td>
<td>add</td>
<td><code>add(val: time):char</code> - add two instances.</td>
</tr>
<tr>
<td><code>+=</code></td>
<td>add_asgn</td>
<td><code>add_asgn(val: time):char</code> - add two instances and assign result to the instance on which the operation is invoked.</td>
</tr>
<tr>
<td><code>-</code></td>
<td>sub</td>
<td><code>sub(val: time):time</code> - subtract two instances.</td>
</tr>
<tr>
<td><code>-=</code></td>
<td>sub_asgn</td>
<td><code>sub_asgn(val: time):time</code> - subtract two instances and assign result to the instance on which the operation is invoked.</td>
</tr>
<tr>
<td><code>*</code></td>
<td>mul</td>
<td><code>mul(val: long):time</code> - multiply time value by some integral value.</td>
</tr>
<tr>
<td><code>*=</code></td>
<td>mul_asgn</td>
<td><code>mul_asgn(val: long):time</code> - multiply time value by some integral value and assign result to the instance on which the operation is invoked.</td>
</tr>
<tr>
<td><code>/</code></td>
<td>div</td>
<td><code>div(val: long):time</code> - divide time value by some integral value.</td>
</tr>
<tr>
<td>(<code>/=</code></td>
<td>div_asgn</td>
<td><code>div_asgn(val: long):time</code> - divide time value by some integral value and assign result to the instance on which the operation is invoked.</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>gt</td>
<td><code>gt(val: time):boolean</code> - test if current instance is greater than 'val'.</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>gteq</td>
<td><code>gteq(val: time):boolean</code> - test if current instance is greater than or equal to 'val'.</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>lt</td>
<td><code>lt(val: time):boolean</code> - test if current instance is less than 'val'.</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>lteq</td>
<td><code>lteq(val: time):boolean</code> - test if current instance is less than or equal to 'val'.</td>
</tr>
</tbody>
</table>

### Tab. A.11: Core Operations of Character Types

<table>
<thead>
<tr>
<th>MAL Operator</th>
<th>Action Name</th>
<th>Core Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>=</code></td>
<td>asgn</td>
<td><code>asgn(val: char):void</code> - assign an instance to the current instance.</td>
</tr>
<tr>
<td><code>==</code></td>
<td>eq</td>
<td><code>eq(val: char):boolean</code> - test if two instances have the same value.</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>neq</td>
<td><code>neq(val: char):boolean</code> - test if two instances do not have the same value.</td>
</tr>
<tr>
<td><code>+</code></td>
<td>add</td>
<td><code>add(val: char):char</code> - add two instances.</td>
</tr>
<tr>
<td><code>+=</code></td>
<td>add_asgn</td>
<td><code>add_asgn(val: char):char</code> - add two instances and assign result to the instance on which the operation is invoked.</td>
</tr>
<tr>
<td><code>-</code></td>
<td>sub</td>
<td><code>sub(val: char):char</code> - subtract two instances.</td>
</tr>
<tr>
<td><code>-=</code></td>
<td>sub_asgn</td>
<td><code>sub_asgn(val: char):char</code> - subtract two instances and assign result to the instance on which the operation is invoked.</td>
</tr>
</tbody>
</table>

*continued on next page*
<table>
<thead>
<tr>
<th>MAL Operator</th>
<th>Action Name</th>
<th>Core Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>gt</td>
<td>gt(in val: char):boolean - test if current instance is greater than 'val'.</td>
</tr>
<tr>
<td>&gt;=</td>
<td>gteq</td>
<td>gteq(in val: char):boolean - test if current instance is greater than or equal to 'val'.</td>
</tr>
<tr>
<td>&lt;</td>
<td>lt</td>
<td>lt(in val: char):boolean - test if current instance is less than 'val'.</td>
</tr>
<tr>
<td>&lt;=</td>
<td>lteq</td>
<td>lteq(in val: char):boolean - test if current instance is less than or equal to 'val'.</td>
</tr>
</tbody>
</table>

**string**

<table>
<thead>
<tr>
<th>MAL Operator</th>
<th>Action Name</th>
<th>Core Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>asgn</td>
<td>asgn(in val: char):void - assign another string instance to the current instance.</td>
</tr>
<tr>
<td>==</td>
<td>eq</td>
<td>eq(in val: string):boolean - test if two strings are equal, i.e. if they have the same sequence of characters.</td>
</tr>
<tr>
<td>!=</td>
<td>neq</td>
<td>neq(in val: string):boolean - test if two strings are not equal, i.e. if their sequence of characters differs.</td>
</tr>
<tr>
<td>+</td>
<td>add</td>
<td>add(in val: string):string - concatenate 'val' to the string on which the operation is invoked.</td>
</tr>
<tr>
<td>+=</td>
<td>add_asgn</td>
<td>add_asgn(in val: string):string - concatenate 'val' to the string on which the operation is invoked and assign the result to this string.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MAL Operator</th>
<th>Action Name</th>
<th>Core Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>get</td>
<td>get(in index: int):char - get the character at index 'index'.</td>
</tr>
<tr>
<td>put</td>
<td>put</td>
<td>put(in index:int, in val: char):void - write a character to a specific index.</td>
</tr>
</tbody>
</table>

**Tab. A.12: Core Operations of Auxiliary Types**

<table>
<thead>
<tr>
<th>MAL Operator</th>
<th>Action Name</th>
<th>Core Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>new</td>
<td>CreateObjectAction</td>
<td>create():process - create an instance of process.</td>
</tr>
<tr>
<td>destroy</td>
<td>DestroyObjectAction</td>
<td>destroy():void - destroy the process instance.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mapped to CallOperationAction start():boolean - start the process. Returns true if start was successful.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mapped to CallOperationAction stop():boolean - stop the process. Returns true if stop was successful.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mapped to CallOperationAction suspend(in interval: time):boolean - suspend the process for a time interval. Returns true if suspend was successful.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mapped to CallOperationAction resume():boolean - resumes a suspended process. Returns true if resumption was successful.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mapped to CallOperationAction run():void - implements the main method of the process. This method is executed when the process is started.</td>
</tr>
</tbody>
</table>

*continued from previous page*
A.3. MOCCA Profile Definitions

A.3.1 Overview

The profiles defined in this thesis extend the UML 2.0 meta-model [6]. The presented profiles are relatively light-weight and specific to the approach being presented in the previous chapters. The focus of the profiles is system implementation\(^1\). For this, the profiles offer constructs for mapping and mapping evaluation of designs to implementations using software programming languages and hardware descriptions languages. For each major translation step a set of profiles is defined. The presented profiles have been design specifically for the presented approach and are by no means generic in that they are useful as such to other domains and approaches. The development of such profiles is possible of course, but it is clearly outside the scope of this thesis. Section A.3.2 overviews the related profiles and puts them into the contexts of this work.

The main elements defined by profiles are stereotypes. Stereotypes extend meta-model elements. Stereotypes themselves are further described by constraint and tag definitions. The profile definitions given in this thesis define the constraints and tags imposed by some stereotype next to the respective stereotype definition. For some UML meta-model elements additional constraints are defined. As for stereotypes, the constrained element type is referred as base type of the constraint. Constraints can be applicable to multiple base types. All profiles defined in this thesis use OCL to define additional constraints on profile elements.

Fig. A.8 overviews the profiles being part of the MOCCA modeling framework. All profiles are build upon the "UML Standard Profile" which is part of the UML specification. This relatively rich set of profiles supports the orthogonalization the different issues involved in system development.

The main profiles build upon the "Constraint and Tag Value Definition Profile". This profile defines the language for the definition of constraints and tagged values that are defined by the other profiles. As such it serves as infrastructure for the definition of the other profiles, except the "Design Model Profile" which does currently not define tags or constraints.

For implementations a core profile was defined that defines the common concepts of all implementation platform specific extensions. The extensions use or specialize the concepts defined in the "Implementation Platform Profile".

\(^1\) The author is aware that the profiles are likely to miss features that may desirable in particular application domains. As such the defined profiles present a proof-of-concept and may be the basis for the development of more general profiles.
A.3.2 Related Profiles

Currently, there is major research effort in the development of UML profiles for systems engineering and system-on-chip (SoC). These profiles have a different development background than those being defined in the following sections. With exception of the "UML Profile for System-on-Chip" these profiles are mainly oriented toward system analysis; system implementation is a minor concern.

**UML Profile for Schedulability, Performance, and Time Specification**

This profile targets the modeling and analysis of real-time systems and resource-constrained systems [188]. Its main contribution is the introduction of the GRM. The GRM serves as foundation for the resource modeling presented in this thesis (→ Section 3.3.4). The resource model allows for modeling logical models and engineering models of applications. Logical models relate resources to the clients that use the resources, whereas both of them may coexist. Engineering models relate each client to the resources being used to implement the client. In that sense, the implementation models and deployment models presented in this thesis are engineering models.

Due to its background in the real-time domain this profile offers extensive support for modeling time, schedulability, performance, and concurrency. Time modeling allows for the definition of time values, events that are related to time, and time mechanisms and services. The notion of time value specification in MOCCA is inspired by this profile. Schedulability modeling supports the analysis whether all jobs executed by a system can meet their deadlines. The sub-profile for performance modeling assist analyses of performance related properties such as throughput, workload, and utilization. Finally, the concurrency sub-profile enables the modeling of those parts of a system that may execute in parallel.

**UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms**

This profile provides extensive support for modeling the QoS of systems and components [189]. The profile reuses the concepts of QoS characteristics and QoS values that have been introduced in the "UML Profile for Schedulability, Performance, and Time Specification". These concepts are extended and generalized to enable a generic notion of QoS modeling.

Additionally, the concept of QoS constraints is introduced, which define semantic conditions or restrictions to individual QoS values. Also, the QoS being offered and required by the components of a system are
modeled. Another contribution of the profile is a catalog of QoS value types. Each value type is defined by a set of attributes being important during system analysis. For instance, the set of attributes defined for the QoS value type latency includes the definition of arrival patterns, min/max values, jitter, and others. The definition of these concepts is based on [188]. MOCCA aligns with this profile yet it does not fully support it. Future extensions, may fully integrate the profile into the presented approach. On the other hand, MOCCA extends the QoS catalog by area definitions.

**UML Profile for Modeling and Analysis of Real-Time Embedded Systems**

This profile, also called MARTE, has been developed to extend the "UML Profile for Schedulability, Performance, and Time Specification" in order to enable better modeling of real-time systems and to align it to the latest UML specification [210]. Major extensions shall include:

- improved modeling support for timing, performance, schedulability, and concurrency, including a catalog of related QoS values,
- support of different relevant models of computation,
- modeling of software deployment on platforms, and
- modeling resource allocations to applications (i.e. engineering models).

Clearly, this profile has similar goals as the profiles presented in this thesis. This profile tries to overcome some of the shortcomings of the former two profiles and make them more accessible to users. The profile is compliant to the "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms" and extends it by concepts for quantitative analysis. Since the work on this profile is currently confidential at the Object-Management Group (OMG) it is not clear, however, how well it fulfills the requirements. The proposition of the first draft version was scheduled to June 2006 but has not been released yet.

**Systems Modeling Language Specification**

The goal of the SysML profile is the provision of a language for system-engineering that is based on UML [190]. Applications are complex systems comprising software and hardware components. The profile defines the notion of assemblies which describe the composition of systems as combination of components and interconnects. Assemblies can be used to define the logical or physical view of human, hardware, and software systems. Parametrics enable the definition of parametric constraints in assembly diagrams.

The most important extension of SysML regards UML activities. In contrast to UML activities, control tokens are now typed and may be processed by actions. Further, they can be stored in object nodes and may be used to stop the execution of actions rather than just starting them. Additional support for modeling continuous behavior has been added. Probabilities can be assigned to parameters and activity edges, which fosters the various QoS analyses. Another important extension is the modeling of allocations. Allocations are defined on assemblies and ports to enable, among other applications, capturing and relating logical models and engineering models of an application and target architectures. In the presented approach this notion is expressed using realization dependencies between elements of the user design and the model of the target architecture. This approach only partly reflects the structure of the implementation. Assemblies should be integrated into the approach to make the structural aspects of implementations clear and explicit.

**UML Profile for System-on-Chip**

This profile has been proposed in response to the growing demand of using the UML in the SoC domain [211, 280]. The specification is currently under revision at the OMG. Notable extensions include the modeling of SoC as composition of modules and channels, the specification of module types, and modeling the information being exchanged using channels.
A. MOCCA Modeling Framework

The focus of the profile is hardware modeling using high-level building blocks and interconnects called *modules* and *channels*. Modules may be structured hierarchically. For each channel a logical and a physical interface may be defined. The logical interface definition is the specification of the employed protocol while the physical interface defines the exchanged information using data types.

This profile is very similar to the implementation platform model, particularly the VHDL implementation platform (→ Section B.3), that are presented in this thesis. However, in this thesis components and interfaces are used to model building blocks. There is no distinction between the logical interface and the physical interface, since it is the common notion of an interface to define both the exchanged data and the protocol anyway. When the profile is finally adopted by the OMG, it may be checked whether it is useful to adopt it to the current approach. The profile is not sufficient, however, to replace the current implementation profile.

A.3.3 Notation

The extended element type, which is frequently called the *base type* of the extension, is shown in square brackets in the rectangle representing the stereotype. This notation is commonly used as a shortcut to avoid cluttering diagrams by loads of extension associations and meta-model elements.

![Fig. A.9: Reference to Extended Meta-Model Element Notation](image)

The properties of stereotypes, called *tag definitions*, are shown using the same notation as for classifier properties. This is feasible and conforming to the UML specification since stereotypes are specialized classes. Constraint definitions are not shown diagrammatically. For the application of constraints to model elements the standard notation, using parentheses, is used.

A.4 Constraint and Tag Value Definition Profile

A.4.1 Syntactic Meta-Language

To specify tagged values and constraints in this thesis a language was defined. This language definition itself is a profile that is imported by all models that use the language. In principle, the language definition may be decomposed into a hierarchy of language definitions with a core language at the root and extensions that are specialized to the various profiles. However, for the purpose of this presentation and user convenience it is sufficient and more comprehensible to present all language constructs together. This approach also fosters a clean design of the syntax and supports reuse of syntactical constructs wherever possible. This way the overall language is easier to grasp by users.

The syntax orients toward the Java language grammar [215]. The specification of physical quantities was inspired by the "UML Profile for Schedulability, Performance, and Time Specification" [188]. The time quantity modeling capabilities of this profile have been restricted to the ones being relevant for this thesis. Some extensions for modeling area quantities and number of elements have been added.
A.4. Constraint and Tag Value Definition Profile

The syntax of the language is presented exclusively in Extended Bachus-Naur Form. There are loads of versions of this form. In this thesis the standardized version is used [217]. Notable differences of the standard to common notations, that are used in this thesis, are

- "=" is used as defining symbol (rather than ::=),
- "." is used to exclude the value of the right operand from the domain of the left operand, and
- meta-identifiers are not enclosed in angle brackets (<, >).

For further information the reader is referred to [217].

A.4.2 Syntax Definitions

Boolean Literals

BooleanLiteral =
    'true' | 'false'

String Literals

StringLiteral =
    { CharacterLiteral }

Integer Numbers

IntegerNumberLiteral =
    DecimalIntegerNumberLiteral | HexIntegerNumberLiteral | OctalIntegerNumberLiteral

DecimalIntegerNumberLiteral =
    DecimalIntegerNumeral [ IntegerTypeSuffix ]

HexIntegerNumberLiteral =
    HexIntegerNumeral [ IntegerTypeSuffix ]

OctalIntegerNumberLiteral =
    OctalIntegerNumeral [ IntegerTypeSuffix ]

IntegerTypeSuffix =
    'l' | 'L'

DecimalIntegerNumeral =
    '0' | NonZeroDigit [ Digits ]

Digits =
    Digit | Digits Digit

Digit =
    '0' | NonZeroDigit

NonZeroDigit =
    '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

HexIntegerNumeral =
    '0' 'x' HexDigits | '0' 'X' HexDigits
HexDigits =
    HexDigit [ HexDigits ]
HexDigit =
    '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | 'a' | 'b'
    | 'c' | 'd' | 'e' | 'f' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'
OctalIntegerNumeral =
    '0' OctalDigits
OctalDigits =
    OctalDigit [ OctalDigits ]
OctalDigit =
    '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7'

Real Numbers

RealNumberLiteral =
    Digits '.' [Digits] [ExponentPart] [FloatTypeSuffix] |
    '.' Digits [ExponentPart] [FloatTypeSuffix] |
    Digits ExponentPart [FloatTypeSuffix] |
    Digits [ExponentPart] FloatTypeSuffix
ExponentPart =
    ExponentIndicator SignedInteger
ExponentIndicator =
    'e' | 'E'
SignedInteger =
    [ '+' | '-' ] Digits
FloatTypeSuffix =
    'd' | 'D' | 'f' | 'F'

Time Specifications

TimeSpecification =
    ConstantTimeSpecification | PDFTimeSpecification
PDFTimeSpecification =
    '(' PDFSpecification ',' "" TimeUnit "" ')'
    (* Example: ((histogram 0,0.1,1,0.2,3,0.7,4),'ps') *)
ConstantTimeSpecification =
    DateSpecification |
    HourMinSecSpecification |
    ConstantMetricTimeSpecification
    (* Examples: 2006/02/19 , 12:02:19, (0.5,'ms') *)
ConstantMetricTimeSpecification =
    '(' RealNumberLiteral ',' "" TimeUnit "" ')'
MetricTimeSpecification =
A.4. Constraint and Tag Value Definition Profile

ConstantMetricTimeSpecification | PDFTimeSpecification

TimeUnit =
   'ps' | 'ns' | 'us' | 'ms' | 'sec' | 'min' | 'hr' | 'days' | 'wks' |
   'mos' | 'yrs' | 'Cycle'

DateSpecification =
   YearIntegerNumberLiteral '/' MonthIntegerNumberLiteral '/'
   DayOfMonthIntegerNumberLiteral

YearIntegerNumberLiteral =
   IntegerNumberLiteral

MonthIntegerNumberLiteral =
   IntegerNumberLiteral (* must be in [1, 12] *)

DayOfMonthIntegerNumberLiteral =
   IntegerNumberLiteral (* must be in [1,31] *)

HourMinSecSpecification =
   HourLiteral [ ':' MinuteLiteral [ ':' SecondLiteral [ ':'
   CentisLiteral ] ] ]

HourLiteral =
   IntegerNumberLiteral (* Must be in [0, 23]*)

MinuteLiteral =
   IntegerNumberLiteral (* Must be in [0, 59]*)

SecondLiteral =
   IntegerNumberLiteral (* Must be in [0, 59]*)

CentisLiteral =
   IntegerNumberLiteral (* Must be in [0, 99]*)

Area Specifications

AreaSpecification =
   ConstantAreaSpecification | PDFAreaSpecification
   (* Examples: (0.5,'KByte'), ((binomial 0.7), 'Gate') *)

PDFAreaSpecification =
   '(' PDFSpecification ', ' '"' AreaUnit '"' ')

ConstantAreaSpecification =
   '(' RealNumberLiteral ', ' '"' AreaUnit '"' ')

AreaUnit =
   BitAreaUnit | LogicAreaUnit

BitAreaUnit =
   'Bit' | 'Byte' | 'KByte' | 'MByte' | 'GByte'

LogicAreaUnit =
   'Gate'
**Quantity Specifications**

QuantitySpecification =
   ConstantQuantitySpecification | PDFQuantitySpecification
   (* Examples: (300, 'Qty'), ((binomial 0.7), 'Qty') *)

PDFQuantitySpecification =
   (' PDFSpecification ','"" QuantityUnit "" ')

ConstantQuantitySpecification =
   (' IntegerNumberLiteral ','"" QuantityUnit "" ')

QuantityUnit =
   'Qty'

**Address Related Specifications**

ConstantAddressSpecification =
   IntegerNumberLiteral

AddressAlignmentSpecification =
   "TypeInstanceSize" | ConstantAddressRangeSpecification

AddressRangeSpecification =
   ConstantAddressRangeSpecification | PDFAddressRangeSpecification

ConstantAddressRangeSpecification =
   (' ConstantAddressSpecification ','"" BitAreaUnit "" ')

PDFAddressRangeSpecification =
   (' PDFSpecification ','"" BitAreaUnit "" ')

**Probability Distribution Functions**

PDFSpecification =
   (' BernoulliPDFSpecification | BinominalPDFSpecification |
   ExponentialPDFSpecification | GammaPDFSpecification |
   HistogramPDFSpecification | NormalPDFSpecification |
   PoissonPDFSpecification | UniformPDFSpecification ')

BernoulliPDFSpecification =
   'bernoulli' RealNumberLiteral
   (* Models a bernoulli probability distribution. The parameter represents the distribution probability and must be in [0,1). *)

BinominalPDFSpecification =
   'binomial' RealNumberLiteral ',' IntegerNumberLiteral
   (* Models a binomial probability distribution \( f(x) = \binom{n}{x} p^x (1-p)^{n-x} \).
   The first parameter represents the distribution probability \( p \) and must be in [0,1]. The second parameter represents the number of trials \( n \) and must be greater than 0. *)

ExponentialPDFSpecification =
   'exponential' RealNumberLiteral
A.4. Constraint and Tag Value Definition Profile

GammaPDFSpecification =
'gamma' IntegerNumberLiteral

(* Models a standard gamma probability distribution function
\[ f(x) = \frac{x^{\gamma-1} e^{-x/\gamma}}{\Gamma(\gamma)} \] for \( x \geq 0 \) and \( \gamma > 0 \). The parameter represents the shape
parameter \( \gamma \) of the distribution. *)

HistogramPDFSpecification =
'histogram' \{ RealNumberLiteral ',,' RealNumberLiteral \} ' ,'
RealNumberLiteral

(* Models a histogram probability distribution function as ordered set
of pairs. Each equivalence class is modeled as pair, whereas the
first value represents the start of the class and the second value is
the probability of the class. The last value is the end of the last
class. The starts of the classes must be given in strongly
monotonous, ascending order. The rightmost value must be larger than
the start of the rightmost class. *)

NormalPDFSpecification =
'normal' RealNumberLiteral ',,' RealNumberLiteral

(* Models a Gauss probability distribution
\[ f(x) = \frac{e^{-(x-\mu)^2/2\sigma^2}}{\sigma \sqrt{2\pi}} \]. The first
parameter represents the mean \( \mu \) and the second parameter is the
standard deviation \( \sigma \) of the distribution. *)

PoissonPDFSpecification =
'poisson' RealNumberLiteral

(* Models a Poisson probability distribution
\[ f(x) = \frac{e^{-\lambda} \lambda^x}{x!} \]. The
parameter represents the mean \( \lambda \) of the distribution. *)

UniformPDFSpecification =
'uniform' RealNumberLiteral ',,' RealNumberLiteral

(* Models an uniform probability distribution
\[ f(x) = \begin{cases} \frac{1}{b-a} & a \leq x \leq b \\ 0 & \text{else} \end{cases} \].
The parameters represent the start \( a \) and end \( b \) of the interval. *)

Scheduling Specifications

SchedulingPolicySpecification =
"asap" | "alap" | "force" | "sequential"

Distance Vectors

DistanceVectorSpecification =
QualifiedTypeName '=' IntegerNumberLiteral [ ',,' DistanceVector ]

(* Example: long=0,int=1,float=2,MyModel.MyPackage.MyType=3 *)

QualifiedTypeName =
[ QualifiedPackageName '.' ] TypeName

QualifiedPackageName =
[ QualifiedPackageName '.' ] PackageName

TypeName =
A. MOCCA Modeling Framework

Auxiliary Specifications

ExecutionModeSpecification =
"always" | "concurrent"

A.5 Design Profiles

A.5.1 Design-Platform Profile

The MOCCA design platform modeling profile enables the definition of design platforms. The design platform model is the foundation for the creation of design models that can be used with the model compiler.

![Design Platform Profile Diagram]

Fig. A.10: Design Platform Profile

DesignPlatform

Type: Stereotype

Base Type: Model

Description: This stereotype defines models to represent design platform models. A design platform model defines sets of design types (stereotyped DesignType), and constraints, that build the foundation for the construction of design models.

Additional Constraints: none

Tag Definitions: none
DesignType

Type: Stereotype

Base Type: Classifier

Description: This stereotype defines classifiers to represent design types. Design types are the foundation for the construction of design models. All user-defined types are created from design types.

Additional Constraints: none

Tag Definitions:

- DistanceVector
  - Description: The default distance between classifiers being organized in inheritance hierarchies is the minimum length of the generalization path from a specialization type to its generalizations. If two types are not related to each other by means of inheritance the distance is infinite. The distance of a type to itself is zero (→ Definition 3.3.2). Distance vectors override the default distance!
  - Syntax: DistanceVector:=DistanceVectorSpecification
    EXAMPLE A.1: The distance vector specification DistanceVector:=long=0, int =1, float=2, MyModel.MyPackage.MyType=3 sets the distance of the constrained design type to type long to be zero, to type int to be one, to type float to be two, and to type MyType in package MyPackage in package MyModel to be three. The defined distances represent starting points for the default distance computation (→ Definition 3.3.2).

Object

Type: Stereotype

Base Type: Classifier

Description: This stereotype defines a classifier to represent an object type. In MOCCA, object is the root of all design types. All other types inherit from this classifier.

Additional Constraints: none

Tag Definitions: none

Classifier

Type: Stereotype

Base Type: Classifier

Description: This stereotype defines a classifier to represent the classifier that is the root of all classes and interfaces.

Additional Constraints: none

Tag Definitions: none
Array

Type: Stereotype
Base Type: Classifier

Description: This stereotype defines a classifier to represent an array type. In MOCCA, this type is the root of all array types.

Additional Constraints: none
Tag Definitions: none

Boolean

Type: Stereotype
Base Type: Classifier

Description: This stereotype defines a classifier to represent the Boolean type. Instances of this type represent Boolean quantities and can carry the values true or false.

Additional Constraints: none
Tag Definitions: none

Magnitude

Type: Stereotype
Base Type: Classifier

Description: This stereotype defines a classifier to represent a magnitude type. Instances of this type represent scalar magnitudes. There are no direct instances of this stereotype. The value range of the type must be contiguous, and is defined by LowerBound and UpperBound, to be the interval [LowerBound, UpperBound].

Additional Constraints:

- LowerBound
  - Description: Defines the lower bound (including lower bound) of the range of values.
  - Syntax: The syntax is defined by the sub-types of Magnitude.
- UpperBound
  - Description: Defines the upper bound (including upper bound) of the range of values.
  - Syntax: The syntax is defined by the sub-types of Magnitude.

Tag Definitions: none

Number

Type: Stereotype
Base Type: Classifier

Description: This stereotype defines a classifier to represent a magnitude type that represents a range of numbers. Instances of this type represent scalar numbers. There are no direct instances of this stereotype.

Additional Constraints: LowerBound, UpperBound
Tag Definitions: none
Real

Type: Stereotype

Base Type: Classifier

Description: This stereotype defines a classifier to represent a real number type.

Additional Constraints:
- LowerBound
  - Description: Defines the lower bound (including lower bound) of the range of values.
  - Syntax: LowerBound:=RealNumberLiteral (→ Section A.4.2)

Example A.2: The constraint LowerBound:=-1.40e-45 constrains the lower bound of the constrained type. Instances of the type are not allowed to carry values less than the lower bound.

- UpperBound
  - Description: Defines the upper bound (including upper bound) of the range of values.
  - Syntax: UpperBound:=RealNumberLiteral (→ Section A.4.2)

Example A.3: The constraint UpperBound:=3.40e+38 constrains the upper bound of the constrained type. Instances of the type are not allowed to carry values greater than the upper bound.

Tag Definitions: none

Integer

Type: Stereotype

Base Type: Classifier

Description: This stereotype defines a classifier to represent a integer number type.

Additional Constraints:
- LowerBound
  - Description: Defines the lower bound (including lower bound) of the range of values.
  - Syntax: LowerBound:=IntegerNumberLiteral (→ Section A.4.2)

- UpperBound
  - Description: Defines the upper bound (including upper bound) of the range of values.
  - Syntax: UpperBound:=IntegerNumberLiteral (→ Section A.4.2)

Tag Definitions: none

Time

Type: Stereotype

Base Type: Classifier

Description: This stereotype defines a classifier to represent a time quantity type.

Additional Constraints:
- LowerBound
  - Description: Defines the lower bound (including lower bound) of the range of values.
A. MOCCA Modeling Framework

- **Syntax**: LowerBound:=ConstantMetricTimeSpecification
  (→ Section A.4.2)
  
  **Example A.4**: The constraint `LowerBound:= (-1000, ‘ms’)` constrains the lower bound of the constrained type to be -1000 milliseconds.

- **UpperBound**
  - **Description**: Defines the upper bound (including upper bound) of the range of values.
  - **Syntax**: UpperBound:=ConstantMetricTimeSpecification
    (→ Section A.4.2)
  
  **Example A.5**: The constraint `UpperBound:= (1000, ‘ms’)` constrains the upper bound of the constrained type to be 1000 milliseconds.

**Tag Definitions**: none

A.5.2 Design-Model Profile

The MOCCA design modeling profile enables the definition of design models. The design model defines the user-specific applications. Design models are build on design platform models.

![Design Model Profile Diagram]

**Fig. A.11**: Design Model Profile

**main**

**Type**: Stereotype

**Base Type**: Operation

**Description**: This stereotype defines an operation to represent the starting point of the overall control-flow of a design model. Exactly one operation must be declared to be the main operation.

**Constraints**: `context main inv: self.allInstances()->size() = 1`

**Additional Constraints**: none

**Tag Definitions**: none

**reactive**

**Type**: Stereotype

**Base Type**: Class

**Description**: This stereotype defines a class to represent a class that is capable of receiving and processing signals.

**Additional Constraints**: none

**Tag Definitions**: none
exception

Type: Stereotype
Base Type: Class
Description: This stereotype defines a class to represent an exception.
Additional Constraints: none
Tag Definitions: none

A.5.3 Estimation Profile

This profile is a light-weight profile for annotation model elements with execution characteristics. Such characteristics can be derived from real or estimated execution profiles. The profile is commonly not accessed by designers. Instead, the model compilers uses it to associate relevant information to models. Analyzers may use the annotated information to assess system properties and to control the implementation of a system. The information is commonly associated to design models.

Extensions of this profile should implement a better orthogonalization of the modeled concepts. Further it may be aligned to the "UML Profile for QoS and Fault Tolerance" [189].

![Fig. A.12: Estimation Profile](image-url)

CharacterizedElement

Type: Stereotype
Base Types: Element
Description: This stereotype is used to extend model elements whose implementation or execution characteristics are measured or estimated. This stereotype is used during profiling to back-annotate the characteristics of the element to the model.
Additional Constraints: none
Tag Definitions: none
CharacterizedBehaviorElement

**Type:** Stereotype

**Base Types:** BehavioralFeature, Behavior, Action

**Description:** This stereotype is used to extend behavior related model elements whose execution characteristics are modeled.

**Additional Constraints:** none

**Tag Definitions:**

- **ExecutionFrequency**
  - **Description:** This property specifies the execution frequency of an element.
  - **Syntax:** ExecutionFrequency:=IntegerNumberLiteral (→ Section A.4.2)

- **ExecutionProbability**
  - **Description:** This property specifies the execution probability of an element.
  - **Syntax:** ExecutionProbability:=RealNumberLiteral

- **ExecutionLoopCount**
  - **Description:** This property specifies the loop count of an instance of LoopNode. The loop count is the total loop count. That is, if the annotated loop is invoked within an outer loop (possibly in a different behavior) then the loop count is the loop count of the outer loop multiplied by the unnested loop count of the annotated loop.
  - **Syntax:** ExecutionLoopCount:=IntegerNumberLiteral

- **ExecutionUnnestedLoopCount**
  - **Description:** This property specifies the unnested loop count of an instance of LoopNode.
  - **Syntax:** EstimatedUnnestedLoopCount:=IntegerNumberLiteral

- **EstimatedUtilization**
  - **Description:** This derived property specifies the execution utilization of an element.
  - **Syntax:** EstimatedUtilization:=RealNumberLiteral

- **EstimatedMaxConcFlow**
  - **Description:** This property specifies the number of concurrent control-flows that can concurrently execute the characterized element.
  - **Syntax:** EstimatedMaxConcFlow:=IntegerNumberLiteral

MeasuredBehaviorElement

**Type:** Stereotype

**Base Types:** BehavioralFeature, Behavior, Action

**Description:** This stereotype is used to extend model elements whose implementation or execution characteristics are measured. This stereotype is used during profiling to back-annotate the measured characteristics of the element to the model.

**Additional Constraints:** none

**Tag Definitions:** none
EstimatedBehaviorElement

Type: Stereotype

Base Types: BehavioralFeature, Behavior, Action

Description: This stereotype is used to extend model elements whose implementation or execution characteristics are estimated. This stereotype is used during profiling to back-annotate the estimated characteristics of the element to the model.

Additional Constraints: none

Tag Definitions: none

CharacterizedStructuralElement

Type: Stereotype

Base Types: StructuralFeature

Description: This stereotype is used to extend structural features whose execution characteristics are modeled.

Additional Constraints: none

Tag Definitions:

- EstimatedMaxConcInstances
  - Description: This property specifies the estimated number of concurrent instances of an execution utilization of an element. The generator is advised to instantiate the according element the given number of times
  - Syntax: EstimatedMaxConcInstances:=IntegerNumberLiteral

- EstimatedMaxArrayElements
  - Description: This property specifies the estimated maximum number of elements that instances of an array must store concurrently.
  - Syntax: EstimatedMaxArrayElements:=IntegerNumberLiteral

MeasuredStructuralElement

Type: Stereotype

Base Types: StructuralFeature

Description: This stereotype is used to extend structural features whose implementation or execution characteristics are measured. This stereotype is used during profiling to back-annotate the measured characteristics of the element to the model.

Additional Constraints: none

Tag Definitions: none
**EstimatedStructuralElement**

**Type:** Stereotype

**Base Types:** StructuralFeature

**Description:** This stereotype is used to extend structural features whose implementation or execution characteristics are estimated. This stereotype is used during profiling to back-annotate the estimated characteristics of the element to the model.

**Additional Constraints:** none

**Tag Definitions:** none

### A.6 Target-Platform Profiles

#### A.6.1 Implementation-Platform Profile

The MOCCA implementation platform modeling profile enables the definition of implementation platforms. The implementation platform model is the foundation for the creation of implementation models that can be used with the model compiler. In order to be useful, this profile must commonly be specialized by platform-specific profiles, e.g. for C/C++, Java, and VHDL. This profile is by far not generic and sufficient to support all requirements of implementation platforms for real-time systems, embedded systems, or even reconfigurable architectures in general. It serves, however, as a starting point for the specification of further more generic profiles.

![Fig. A.13: Implementation Platform Profile: Implementation Components](image)

![Fig. A.14: Implementation Platform Profile: Features and Parameters](image)
ImplementationPlatform

Type: Stereotype

Base Type: Model

Description: This stereotype declares a model to represent an implementation platform. All elements that comprise the represented platform must be packaged directly or nestedly by this model.

Additional Constraints: none

Tag Definitions: none

ImplementationType

Type: Stereotype

Base Type: Classifier

Description: This stereotype declares a classifier to represent an implementation type. Implementation types are used to realize design types. Each implementation type must fulfill the same contract as the design types it realizes.

Additional Constraints:

- ImplementationMaxInstances
  - Description: This constraint defines the maximum number of instances of the constrained element that can be used by implementations. The satisfaction of this QoS-constraint is enforced during platform mapping. If this constraint is not defined for some model element an infinite number of possible instances is assumed. Implementations may, however, be constrained otherwise, e.g. by the implementation area.
  - Syntax: ImplementationMaxInstances:=ConstantQuantitySpecification

Tag Definitions:
• DistanceVector (→ Section A.5.1 on page 158)

• ImplementationName
  – **Description:** This property defines the type name that is to be used in the implementation. The implementation name is used as name of the element in its implementation and, if applicable, all usages of the element. If this tagged value is not defined the native name of the element is used.
  – **Syntax:** ImplementationName:=StringLiteral
    Example A.6: The tagged value ImplementationName:=CLOCK being associated with some model element, such as a parameter or attribute, causes the according generator to use CLOCK in the implementation of the element.

• ImportName
  – **Description:** This property defines the name of the type that is to be used in import declarations. Frequently, implementation types are imported from packages, libraries, etc. For some implementation platforms and implementation types the name that is used in import declarations differs from the actual implementation name of the type.
  – **Syntax:** ImportName:=StringLiteral
    Example A.7: For example, in VHDL specializations of std_logic_vector, such as std_logic_vector(7 downto 0), must use std_logic_vector in the import declaration rather than their actual name. This can be enforced by constraining std_logic_vector(7 downto 0) with ImportName:=std_logic_vector. The import name will be applied in all VHDL use clauses for std_logic_vector(7 downto 0).

• ImplementationAddressSpace
  – **Description:** This property defines the number of consecutive addresses that are allocated instances of the implementation type.
  – **Syntax:** ImplementationAddressSpace:=ConstantAddressRangeSpecification
    Example A.8: ImplementationAddressSpace:=(4,‘Byte’) states an implementation type to allocate four bytes in the address space of the master node.

• ImplementationArea
  – **Description:** This property defines the area of instances of the implementation type. The specified value is the area that is perceived at the architectural level, but not in the micro-architecture of the constrained behavior. This QoS-constraint is used during platform mapping as basis for the estimation of implementations.
  – **Syntax:** ImplementationArea:=AreaSpecification
    Example A.9: A QoS-constraint ImplementationArea:=(8,’Gate’) defines the area acquired by instances of the extended implementation type to be eight gate equivalents. Extensions could use other units, such as the number of RFUs.

**ImplementationOperation**

**Type:** Stereotype

**Base Type:** Operation

**Description:** This stereotype declares an operation to represent an implementation operation. Implementation operations may realize design operations. Each implementation operation must fulfill the same contract as the design operation it realizes.

**Additional Constraints:** none

**Tag Definitions:**
• ImplementationName
  – Description: This property defines the operation name that is to be used in the implementation. The implementation name is used as name of the element in its implementation and, if applicable, all usages of the element. If this tagged value is not defined the native name of the element is used.
  – Syntax: ImplementationName:=StringLiteral

ImplementationProperty

Type: Stereotype
Base Type: Property
Description: This stereotype declares a property (attribute) to represent an implementation property (attribute).
Additional Constraints: none
Tag Definitions:
- ImplementationName
  – Description: see ImplementationOperation
  – Syntax: ImplementationName:=StringLiteral

ImplementationParameter

Type: Stereotype
Base Type: Parameter
Description: This stereotype declares a parameter to represent an implementation parameter. Implementation parameters are only defined in the context of implementation operations.
Additional Constraints: none
Tag Definitions:
- ImplementationName
  – Description: see ImplementationOperation
  – Syntax: ImplementationName:=StringLiteral

ImplementationBehavior

Type: Stereotype
Base Type: Behavior
Description: This stereotype declares a behavior to represent an implementation behavior. Implementation behaviors are used to implement design behaviors. The stereotype defines additional constraints and properties being used for platform mapping and synthesis.
Additional Constraints:
- ImplementationMaxInstances
  – Description: see ImplementationType
  – Syntax: ImplementationMaxInstances:=ConstantQuantitySpecification
Tag Definitions:

- **ImplementationLanguagePattern**
  - **Description:** This property defines a pattern in terms of the used implementation language that is used to generate the implementation of the behavior (→ Section 5.3.1). Implementation language patterns link the implementation platform model into the actual platform. The actual syntax of the pattern is specific to the node generator component of the particular platform. If this tagged value is not specified a default implementation pattern, which is specific to the generator component, is used.
  - **Syntax:** ImplementationLanguagePattern := StringLiteral
    
    EXAMPLE A.10: The `ImplementationLanguagePattern := ($this+$other)` represents a pattern for an "add" operation. The actual values of $this and $other are set by generators to the name of the instance on which the "add" operation is invoked and the instance representing the second summand respectively.

- **ImplementationLatency**
  - **Description:** This constraint defines the latency of the behavior.
  - **Syntax:** ImplementationLatency := MetricTimeSpecification
    - **Semantics:** In case the latency is specified as absolute time value this directly gives the latency of the behavior. Latencies can also be defined relative to some clock cycle, i.e. when the time unit is Cycle. Then this time specification is relative to the clock cycle of the deployment location that will execute the behavior. If this value is not defined for some element, the implementation latency is considered unknown.
      
      EXAMPLE A.11: `ImplementationLatency := (6.14,'ns')` states that the behavior takes 6.14 nanoseconds to execute on some target. In contrast, `ImplementationLatency := (0.4,'Cycle')` defines the latency relatively to the clock cycle of the execution context, e.g. deployment location or implementation component instance.

- **ImplementationArea**
  - **Description:** This property defines the area of the behavior. The specified value is the area that is perceived at the architectural level, but not in the micro-architecture of the constrained behavior. This QoS-constraint is used during platform mapping as basis for the estimation of implementations.
  - **Additional Constraints:** none
  - **Tag Definitions:** none
  - **Syntax:** ImplementationArea := AreaSpecification

- **ImplementationDefault**
  - **Description:** This property defines the default behavior of an operation. This behavior is used to execute the operation whenever no other metric applies. If this constraint is not defined and no other metric applies, a random behavior is selected.
  - **Syntax:** ImplementationDefault := BooleanLiteral

**ImplementationComponent**

**Type:** Stereotype

**Base Type:** Component

**Description:** This stereotype declares a component to represent an implementation component. Implementation components must be specialized by a collection of implementation types. This stereotype is specialized to reflect the semantics of the component.

**Additional Constraints:**

- **Allocatable**
A.6. Target-Platform Profiles

- **Description:** This constraint marks an implementation component to be allocatable for implementations. Generators use only allocatable components to realize implementations of user designs. By default all components are considered to be *not* allocatable.

- **Syntax:** `Allocatable:=BooleanLiteral`

  - `ImplementationMaxInstances`

    - **Description:** see `ImplementationType`
    
    - **Syntax:** `ImplementationMaxInstances:=ConstantQuantitySpecification`

      **Example A.12:** `ImplementationMaxInstances:=(96,'Qty')` constrains an implementation component such that at most 96 instances of the resource service that is represented by the component are available for implementations.

  **Tag Definitions:** none

StorageComponent

  **Type:** Stereotype

  **Base Type:** Component

  **Description:** This stereotype declares a component to represent a storage component. An example of this type of component is the memory of some uP system, or memory blocks attached to or embedded into reconfigurable fabrics.

  **Additional Constraints:** none

  **Tag Definitions:** none

CommunicationComponent

  **Type:** Stereotype

  **Base Type:** Component

  **Description:** This stereotype declares a component to represent an implementation component the offers a communication resource service. An example of this type of component is a PCI-to-MOB bridge, as is it used in hardware designs.

  **Additional Constraints:** none

  **Tag Definitions:** none

ResetComponent

  **Type:** Stereotype

  **Base Type:** Component

  **Description:** This stereotype declares a component to represent an implementation component the offers a reset resource service. Reset components implement reset generators that are used by hardware designs to reset the logic to an initial state.

  **Additional Constraints:** none

  **Tag Definitions:** none
ClockingComponent

**Type:** Stereotype

**Base Type:** Component

**Description:** This stereotype declares a component to represent an implementation component the offers a clocking resource service. Clocking components implement clock generators that provide the clock of the user logic.

**Additional Constraints:** none

**Tag Definitions:** none

ProcessingComponent

**Type:** Stereotype

**Base Type:** Component

**Description:** This stereotype declares a component to represent an implementation component the offers a processing resource service. Processing resource services are implemented in detail by the behaviors of the implementation types realizing the processing component. In general, all implementation types that do not realize other implementation components do realize the processing component.

**Additional Constraints:** none

**Tag Definitions:** none

VirtualRoot

**Type:** Stereotype

**Base Type:** ImplementationType

**Description:** This stereotype declares an implementation type to represent a virtual root. Classifiers declared as virtual root are not considered for implementation. They are a convenient tool to simplify modeling hierarchies of implementation classifiers in type systems that do not support inheritance, e.g., C or VHDL. Multiple implementations types can specialize a virtual root type. This root type defines all features that are common to its specializations. Consequently, these features do not have to be modeled in each type again.

**Objective:** Many implementation platforms, such as C and VHDL, do not support type inheritance. To model such platforms would generally require a lot of effort, since common features must be modeled for each type individually. The virtual root stereotype enables implementation platform models to exploit the benefits of inheritance even though the modeled platform does not support this notion.

**Additional Constraints:** none

**Tag Definitions:** none

Configuration

**Type:** Stereotype

**Base Type:** Artifact

**Description:** This stereotype declares an artifact to represent a configuration context for reconfigurable fabrics. The software equivalent of this stereotype is executable in the "UML Standard Profile" [6].
**ModelCompilerComponent**

**Type:** Stereotype

**Base Type:** Component

**Description:** This stereotype declares a component to represent a model compiler component.

**Additional Constraints:** none

**Tag Definitions:**

- **ExecutionMode**
  - **Description:** The execution mode of the component. The execution mode defines when a component is invoked by other model compiler components. This mode is used to synchronize the execution of different components of the same platform. Model compiler components invoke other compiler components only if there is an association modeled between the participating components. The execution mode defines when a component can be invoked by another component over the modeled association.
  - **Syntax:** ExecutionMode:=ExecutionModeSpecification
  - **Semantics:**
    - **always** – The called component is invoked always immediately after the calling component has processed all model elements for which it is responsible. For example, an interpreter that is associated with a generator and has set its execution mode to always, is invoked by the generator immediately after the generator has finished synthesis for the particular platform.
    - **concurrent** – The called component is invoked always immediately after the calling component has processed one model elements for which it is responsible. For example, an interpreter that is associated with a generator and has set its execution mode to always, is invoked by the generator immediately after the generator has finished synthesis for the particular element. The processed element is passed to the called component.

**NodeEstimator**

**Type:** Stereotype

**Base Type:** Component

**Description:** This stereotype declares a component to represent a node-specific estimator component (→ Section 3.3.4). Node-specific estimators are used during platform mapping to compute platform-specific estimates of model element mappings of the particular platform.

**Additional Constraints:** none

**Tag Definitions:** none
Description: This stereotype declares a component to represent an node-specific generator component (→ Section 3.3.4). These generators synthesize implementations of the model elements that have been mapped to the particular platform.

Additional Constraints: none

Tag Definitions: none

NodeInterpreter

Type: Stereotype

Base Type: Component

Description: This stereotype declares a component to represent an node-specific interpreter component (→ Section 3.3.4). Interpreters represent lower level design flows. They process the output of the generator of the respective platform. In general, modeled interpreters proxy the lower levels flows. The additional tags control the interfacing between the proxy and the proxied design flow. An implementation platform can define multiple interpreter components.

Additional Constraints: none

Tag Definitions:

- WorkingDirectory
  - Description: The working directory to which the proxy shall change before executing the specified command line.
  - Syntax: WorkingDirectory:= StringLiteral The string must comply to the directory path naming rules of the underlying operating system. If not specified, the current directory is used as working directory.
    EXAMPLE A.13: WorkingDirectory:=c:/mocca/build configures a node interpreter to change to the specified directory, before invoking the actual interpreter.

- CommandLine
  - Description: The command line that is executed in the working directory. Before this command is executed, the interpreter changes into the working directory.
  - Syntax: CommandLine:=StringLiteral The detailed syntax of the command line is specific to the particular implementation platform). For instance, command line templates may be used that are instantiated and populated with the actual data by node generators.
    EXAMPLE A.14: CommandLine:=make -f %file%; %file% may be replaced by the caller of the component with some file name.

- InputType
  - Description: A textual description of the input that is accepted by the interpreter component. This is intended to select between different interpreters modeled for the same platform.
  - Syntax: InputType:=StringLiteral
  - Example: InputType:=code/vhdl

- OutputType
  - Description: A textual description of the output that is created by the interpreter component. This is intended to select between different interpreters modeled for the same platform.
  - Syntax: OutputType:=StringLiteral
  - Example: OutputType:=binary/bitstring
NodeMapper

Type: Stereotype

Base Type: Component

Description: This stereotype declares a component to represent a node-specific mapper component (→ Section 3.3.4). These mappers are used during platform mapping to compute the platform-specific breeding and model transformations.

Additional Constraints: none

Tag Definitions: none

A.6.2 C/C++ Platform Profile

The MOCCA C/C++ implementation platform profile is the foundation for modeling C/C++ implementation platforms. This profile is an extension of the generic implementation platform profile. It specializes some of the extensions of this profile. This profile is not meant to enable the modeling of C/C++ implementations. Instead, its focus is on modeling the respective platforms. The profile is rather slim, due to the proximity of UML and object-oriented software implementations. The profile may be specialized for various dialects and extensions of C and C++, thus all extensions are prefixed Cxx.
Cxx ImplementationPlatform

Type: Stereotype
Base Type: Model
Description: This stereotype declares a model to represent a C/C++ implementation platform model. This stereotype specializes ImplementationPlatform from the implementation platform profile.
Additional Constraints: none
Tag Definitions: none

Cxx ImplementationType

Type: Stereotype
Base Type: Classifier
Description: This stereotype declares a classifier to represent an implementation type of a C/C++ implementation platform model. This stereotype specializes ImplementationType from the implementation platform profile.
Additional Constraints: none
Tag Definitions: none

Cxx SystemHeader

Type: Stereotype
Base Type: Artifact
Description: This stereotype declares an artifact to represent a system header file. This stereotype specializes the file stereotype from the "UML Standard Profile" [6]. Implementations that depend on this header file import the header using angle brackets (#include <filename>). The filename property of the artifact denotes the name of the included file.
Additional Constraints: none
Tag Definitions: none

Cxx ProjectHeader

Type: Stereotype
Base Type: Artifact
Description: This stereotype declares an artifact to represent a project-specific header file. This stereotype specializes the file stereotype from the "UML Standard Profile" [6]. Implementations that depend on this header file import the header using double-quote (#include "filename"). The filename property of the artifact denotes the name of the included file.
Additional Constraints: none
Tag Definitions: none
Cxx Library

Type: Stereotype

Base Type: Artifact

Description: This stereotype declares an artifact to represent a library file. This stereotype specializes the library stereotype from the "UML Standard Profile" [6]. Implementations that depend on this library must link it to the executable during compilation. The filename property of the artifact denotes the name of the included file.

Additional Constraints: none

Tag Definitions: none

Cxx Remote

Type: Stereotype

Base Type: Classifier

Description: This stereotype declares a classifier to represent the remote type used in C/C++ implementations. The stereotyped element is used to enable the synthesis of creation/destruction and communication with remote objects. Instances of this type serve as local proxy of the remote object.

Constraints:

context "Cxx ImplementationPlatform"
inv: "Cxx Remote".allInstances()->size() <= 1

Additional Constraints: none

Tag Definitions: none

Cxx Operation

Type: Stereotype

Base Type: Classifier

Description: This stereotype declares a classifier to represent the operation concept of C/C++. The stereotyped element is used to enable the synthesis of operation calls and returns. For this, according operations, such as call and return, are modeled in the operation interface. These operations are used by generators to synthesize the according code, whereas the common synthesis constraints apply. These constraints are defined by the "Implementation Platform Profile" in Section A.6.1.

Objective: As discussed in Section 2.3, the capability to exchange structured messages is inherent to the object concept. The mechanism for message exchange is defined by the execution environment. This mechanism can be anything from native call instruction of a microprocessor to internetwork transfers. Also, the set of core operations of objects does not include core operations, such as call and return because, if we assume that a message of an object to itself is treated like any other message, their invocation requires the utilization of some meta-message exchange mechanism. The stereotype and the classifier make the message exchange mechanism underlying the C/C++ abstract machine explicit.

Constraints:

context "Cxx ImplementationPlatform"
inv: "Cxx Operation".allInstances()->size() <= 1

Additional Constraints: none

Tag Definitions: none
A. MOCCA Modeling Framework

Cxx StatementBlock

Type: Stereotype

Base Type: Classifier

Description: This stereotype declares a classifier to represent the statement block concept of C/C++. The stereotyped element is used to enable the synthesis of flow control statements between statement blocks. For this, according branch operations are modeled in the operation interface. These operations are used by generators to synthesize the according code, whereas the common synthesis constraints apply. These constraints are defined by the "Implementation Platform Profile" in Section A.6.1.

Objective: The UML actions define the control-flow and data-flow of activities in terms of token being exchanged between activity nodes along activity edges. Control-flow is realized in software implementations using calls, returns, and branches. The mechanisms should be made explicit at the model level in order to parameterize synthesis, mapping, and estimation. The mechanisms for transferring control-flow between operations are modeled using Cxx Operation. The stereotype and the classifier make the control-flow transfer mechanism of statement blocks of the underlying the C/C++ abstract machine explicit.

Constraints:

context "Cxx ImplementationPlatform"
inv: "Cxx StatementBlock".allInstances()->size() <= 1

Additional Constraints: none

Tag Definitions: none

A.6.3 VHDL Platform Profile

The MOCCA VHDL implementation platform profile is used to model VHDL implementation platforms. This profile extends the MOCCA implementation platform profile and specializes some extensions. The background of this profile is not the detailed modeling of VHDL designs. The focus is on the generation of synthesizable implementations. Thus, only a minimum of extensions are defined that are required to model VHDL platforms. This profile may be extended by related approaches to support specific model compilers and approaches for simulation, verification, and synthesis.

VHDL ImplementationPlatform

Type: Stereotype

Base Type: Model

Description: This stereotype declares a model to represent a VHDL implementation platform model. This stereotype specializes ImplementationPlatform from the implementation platform profile.

Additional Constraints: none

Tag Definitions: none

VHDL ImplementationType

Type: Stereotype

Base Type: Classifier
**Description:** This stereotype declares a classifier to represent an implementation type of a VHDL implementation platform model. This stereotype specializes `ImplementationType` from the implementation platform profile.

**Additional Constraints:** none

**Tag Definitions:** none

**VHDL Entity**

**Type:** Stereotype

**Base Type:** Interface
**Description:** This stereotype declares an interface to represent a VHDL entity declaration (→ Section 2.2.1). All features of the interface are interpreted to represent port declarations of the entity.

**Additional Constraints:** none

**Tag Definitions:** none

---

**VHDL Architecture**

**Type:** Stereotype

**Base Type:** Class

**Description:** This stereotype declares a class to represent a VHDL architecture definition (→ Section 2.2.1). All features of the interface are interpreted to represent port declarations of the entity. The architecture must implement a VHDL Entity. The interfaces of the architecture and the entity must be equal.

**Additional Constraints:** none

**Tag Definitions:** none

---

**VHDL StorageArchitecture**

**Type:** Stereotype

**Base Type:** Class

**Description:** This stereotype declares a class to represent an architecture of a storage component. The stereotype specializes VHDL Architecture.

**Constraints:** Instances of VHDL StorageArchitecture must only realize instances of StorageComponent that are part of a VHDL ImplementationPlatform instance (can not be expressed in OCL).

**Additional Constraints:** none

**Tag Definitions:**

- **ReadAccessLatency**
  - **Description:** Defines the latency of read accesses to the storage. This property is evaluated by estimators during platform mapping to assess the performance of designs using the architecture.
  - **Syntax:** ReadAccessLatency:=ConstantMetricTimeSpecification

- **WriteAccessLatency**
  - **Description:** Defines the latency of write accesses to the storage component. This property is evaluated by estimators during platform mapping to assess the performance of designs using the architecture.
  - **Syntax:** WriteAccessLatency:=ConstantMetricTimeSpecification

---

**VHDL BitType**

**Type:** Stereotype

**Base Type:** Classifier

**Description:** This stereotype declares a classifier to represent a VHDL bit type, i.e. a type that represents a single bit, such as std_logic.

**Additional Constraints:** none

**Tag Definitions:** none
VHDL BitVectorType

Type: Stereotype
Base Type: Classifier
Description: This stereotype declares a classifier to represent a VHDL bit vector type, i.e. a type that represents a vector of bits, such as std_logic_vector.
Additional Constraints: none
Tag Definitions:
- VHDL BitVector RightIndex
  - Description: Defines the right index of the VHDL bit vector array specification.
  - Syntax: VHDL BitVector RightIndex:=IntegerNumberLiteral
  EXAMPLE A.15: The right index of the bit vector type std_logic_vector (0 to 8) is eight, while it is zero for the type std_logic_vector(8 downto 0).
- VHDL BitVector LeftIndex
  - Description: Defines the left index of the VHDL bit vector array specification.
  - Syntax: VHDL BitVector LeftIndex:=IntegerNumberLiteral
  EXAMPLE A.16: The left index of the bit vector type std_logic_vector (0 to 8) is zero, while it is eight for the type std_logic_vector(8 downto 0).

VHDL Library

Type: Stereotype
Base Type: Package
Description: This stereotype declares a package to represent a VHDL library, such as work and ieee.
Additional Constraints: none
Tag Definitions: none

VHDL Package

Type: Stereotype
Base Type: Package
Description: This stereotype declares a package to represent a VHDL package, such as mocca_pkg and std_logic_1164.
Additional Constraints: none
Tag Definitions: none

VHDL DeviceConnect

Type: Stereotype
Base Type: Operation
Description: This stereotype declares an operation to represent a part of the connection of the design to the environment. All parameters of the operation are routed to the top-level design hierarchy and are made a part of the top-level interface (→ Section 5.4.2).
Additional Constraints: none
Tag Definitions: none
A.6.4 Deployment-Platform Profile

The MOCCA deployment platform modeling profile enables the definition of deployment platforms. The deployment platform model is the foundation for the installation and execution of deployment models that can be used with the model compiler.

![Deployment Platform Profile Diagram]

**Fig. A.20:** Deployment Platform Profile

**SystemMaster**

- **Type:** Stereotype
- **Base Type:** Node
- **Description:** This stereotype declares a node to be the master node of the deployment architecture.
- **Constraints:** context `SystemMaster` inv: `self.allInstances()->size() = 1`
- **Additional Constraints:** none
- **Tag Definitions:** none

**ProcessingElement**

- **Type:** Stereotype
- **Base Type:** Node
- **Description:** This stereotype declares a node of the deployment architecture to represent a processing element.
- **Additional Constraints:**
  - Allocatable
    - **Description:** Declares the processing element to be allocatable for implementation and execution. All nodes that may be used for implementation must set the constraint value true.
    - **Syntax:** Allocatable:=BooleanLiteral
Tag Definitions:

- **ClockCycle**
  - **Description:** Declares the clock cycle of the processing element. The clock cycle must be a metric time specification, which is not relative to some clock.
  - **Syntax:** `ClockCycle := ConstantMetricTimeSpecification`
  
  **Example A.17:** The specification `ClockCycle := (10, 'ns')` sets the clock cycle of the processing element to ten nanoseconds.

- **AvailableArea**
  - **Description:** Declares the area that is available on the element for implementation of user-specific designs.
  - **Syntax:** `AvailableArea := ConstantAreaSpecification`
  
  **Example A.18:** The specification `AvailableArea := (2, 'MByte')` sets the available implementation area to two mega-bytes.

- **StaticActivitySchedulingPolicy**
  - **Description:** Declares the default scheduling policy that is to be used in implementations for the processing element.
  - **Syntax:** `StaticActivitySchedulingPolicy := SchedulingPolicySpecification`
  - **Semantics:** (`→ Section 4.1.3`)
    - `asap` - Use ASAP scheduling policy.
    - `alap` - Use ALAP scheduling policy.
    - `force` - Use force-driven scheduling policy.
    - `sequential` - Use ASAP scheduling policy with one slot per time step. Introduced for convenience.

- **StaticActivitySchedulingSlots**
  - **Description:** Declares the number of slots per time step that are allowed per schedule. At most this number of actions (operations) can be scheduled to the same time step (`→ Section 4.1.3`). If the constraint is not defined an infinite number of slots is assumed.
  - **Syntax:** `StaticActivitySchedulingSlots := IntegerNumberLiteral`

- **AddressAlignment**
  - **Description:** This constraint defines the address alignment of all object interface elements that are executed by the node. If not defined the address alignment is delegated to the lower level design flow.
  - **Syntax:** `AddressAlignment := AddressAlignmentSpecification`
  - **Semantics:** (`→ Section 5.4.3`)
    - `TypeInstanceSize` - Each element is aligned to an address that is divisible by the size the element allocates in the address space. For example, an element that allocates four bytes in the address space is aligned to an address that is divisible by four.
    - `IntegerNumberLiteral` - Each element is aligned to an address that is divisible by the given integer.

- **BaseAddress**
  - **Description:** This constraint defines the base address of the constrained node in the address space of the master node. The object interface elements are mapped to addresses equal and including this address. If not defined, the base address is determined either statically or dynamically during run-time. For example, common operating systems determine the address mapping of devices dynamically throughout device enumeration.
  - **Syntax:** `BaseAddress := ConstantAddressSpecification`
  - **Semantics:** The base address is the static address in the address space of the master node to which the constrained node is mapped.
PLD

Type: Stereotype
Base Type: Node
Description: This stereotype declares a node to represent a programmable logic device.
Additional Constraints: none
Tag Definitions: none

CPLD

Type: Stereotype
Base Type: Node
Description: This stereotype declares a node to represent a complex programmable logic device.
Additional Constraints: none
Tag Definitions: none

FPGA

Type: Stereotype
Base Type: Node
Description: This stereotype declares a node to represent a field-programmable gate array (FPGA).
Additional Constraints: none
Tag Definitions: none

Microprocessor

Type: Stereotype
Base Type: Node
Description: This stereotype declares a node to represent a microprocessor (uP).
Additional Constraints: none
Tag Definitions: none

Device

Type: Stereotype
Base Type: Node
Description: This stereotype declares a node to represent a device (see UML specification [6]). Introduced for compatibility reasons between UML 1.4 and 2.0.
Additional Constraints: none
Tag Definitions: none
ExecutionEnvironment

Type: Stereotype

Base Type: Node

Description: This stereotype declares a node to represent an execution environment (see UML specification [6]). Introduced for compatibility reasons between UML 1.4 and 2.0.

Additional Constraints: none

Tag Definitions: none

CommunicationPath

Type: Stereotype

Base Type: Association

Description: This stereotype declares an association node to represent a communication path (see UML specification [6]). Introduced for compatibility reasons between UML 1.4 and 2.0.

Additional Constraints: none

Tag Definitions: none
B. PLATFORM MODELS

B.1 Design Platform

This section gives an overview of the design platform model that is being used for the examples in this thesis. It is modeled using the "Design Platform Profile" which is presented in Section A.5.1. This platform provides the core data types and core operations that are required by MOCCA. These data types and operations have been presented already in Section A.2, so they are not discussed in detail here. This presentation of the platform models in the current section and all following sections concentrates on the most important aspects of the platforms. It is not meant to provide a complete documentation of the platforms.

B.1.1 Design Platform Types

Fig. B.1-B.3 present the design platform types diagrammatically. All core types of MOCCA are provided. Constraints, such as for the value ranges, are similar to the ones that have been recommended in Section A.2. All type constraints are defined in Section B.1.2.

**Console Input and Output Types**

With the exception of the design types `system`, `ostream`, and `istream`, all presented types have been discussed in detail in Section A.2. The `system` type enables accessing the standard input and output streams of console applications, i.e. for writing program messages (`out`), error messages (`err`), and reading input from the console (`in`). The stream types `istream` and `ostream` provide basic operations for reading and writing the other design platform types respectively.

B.1.2 Design Platform Types Constraints

Tab. B.1-B.4 define the value ranges and distance vectors for those types for which these constraints are applicable. For each integral type and floating point type a distance vector is defined (→ Section 3.3.2). Since distance vectors override the default computation entirely, the vector defined for a type should include the type itself. In general, the distance of a type to itself should be the least positive integer in the vector. Negative distances specify that instances of a type are not assignable to instances of another type. The order of specification of distances in a vector is not relevant.
### Tab. B.1: Design Platform Integral Types Constraints

<table>
<thead>
<tr>
<th>Name</th>
<th>Domain</th>
<th>Distance Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit</td>
<td>[0, 1]</td>
<td>bit=0, byte=1, short=2, int=3, long=4, float=5, double=6, object=7</td>
</tr>
<tr>
<td>byte</td>
<td>[-128, 127]</td>
<td>byte=0, short=1, int=2, long=3, float=4, double=5, object=6</td>
</tr>
<tr>
<td>short</td>
<td>[-32768, 32767]</td>
<td>short=0, int=1, long=2, float=3, double=4, object=5</td>
</tr>
<tr>
<td>int</td>
<td>[-2147483648, 2147483647]</td>
<td>int=0, long=1, float=2, double=3, object=4</td>
</tr>
<tr>
<td>long</td>
<td>[-9223372036854775808, 9223372036854775807]</td>
<td>long=0, float=1, double=2, object=3</td>
</tr>
</tbody>
</table>

### Tab. B.2: Design Platform Floating Point Types Constraints

<table>
<thead>
<tr>
<th>Name</th>
<th>Domain</th>
<th>Distance Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>[-1.40129846432481707e-45, 3.4028234663852885981170418348452e+38]</td>
<td>float=0, double=1, object=2</td>
</tr>
<tr>
<td>double</td>
<td>[-4.94065645841246544e-324, 1.79769313486231570e+308]</td>
<td>double=0, object=1</td>
</tr>
</tbody>
</table>

### Tab. B.3: Design Platform Time Type Constraints

<table>
<thead>
<tr>
<th>Name</th>
<th>Domain</th>
<th>Distance Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>[-9223372036854775808ps, 9223372036854775807ps]</td>
<td>time=0, long=1, object=2</td>
</tr>
</tbody>
</table>

### Tab. B.4: Design Platform Character Type Constraints

<table>
<thead>
<tr>
<th>Name</th>
<th>Domain</th>
<th>Distance Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>[0, 65535]</td>
<td>char=0, int=1, long=2, float=3, double=4, object=5</td>
</tr>
</tbody>
</table>
B.2 C/C++ Implementation-Platform

This section overviews the C/C++ implementation platform that is being used for the implementation of software modules in this thesis. The platform is based on the International Organization for Standardization (ISO) C++ standard [281] and includes already extensions of the forthcoming update of this standard [282]. It may serve as a starting point for the creation of new platform models. To define this platform model the "C/C++ Platform Profile", being defined in Section A.6.2, is used.

B.2.1 Packages

Fig. B.2.1 illustrates the packages that comprise the platform. A relatively rich package structure simplifies managing the various elements comprising the platform. Moreover, packages are used to reflect the packaging of the platform model elements in native directories on the platform. Thereby the package $\text{cpp}$ serves as placeholder of the root directory.

$\text{cpp} \quad$ - The root of the implementation platform. Contains all primitive implementation types and their mappings to the respective design platform types.
datamodel - Contains the data model of the RTR-Manager. The data model comprises the proxy type for remote types and all required helper elements.

datamodel/include - Contains additional includes of the data model.

datamodel/include/utility - Contains utility includes of the data model, such as for the implementation of smart-pointers.

runtime - Contains the relevant run-time elements of the RTR-Manager. The run-time comprises the actual RTR-Manager and helper elements. There are different implementations of the run-time for various languages, such as C/C++ and Java.

runtime/cpp - Contains the C/C++ implementation of the run-time.

OSLF - Contains the model of the operating system abstraction layer framework (OSLF). The OSLF provides a common light-weight interface to typical operating system services. It is used to implement active classes and the synchronization of concurrent control flows. The operating system specific implementations of the OSLF are located in sub-packages of this package.

OSLF/pthreads - Contains the implementation of the OSLF which is based on the Portable Operating System Interface for Unix (POSIX) threads standard [283].

components - Contains the model compiler components.

The dependencies between the packages are shown in Fig. B.2. All packages depend on the Cpp package since its defines the all implementation types apart from the proxy types for remote objects and the RTR-Manager. Notice, that the dependencies modeled for an element are automatically used for all its sub-elements. That is, dependencies are inherited down to containment hierarchy and may be specialized by sub-elements.

B.2.2 Implementation Types

Implementation types are the key elements of all implementation platforms. The C/C++ platform reflects the most common data types defined by the C++ standard. Fig. B.6-B.8 show the key data types of this platform. The interface of the implementation types is mostly straightforward. As will be shown later in this section, the most of the other implementation types realize core design types whose interface has already been documented in Section A.2.

The full platform definitions contains minor specializations of some types, such as array types that are specialized to store primitive types (e.g. int[], short[], et cetera). These specializations are not mandatory however, but they are used to define the other data types and operations being part of this platform.
The most of the shown implementation types are self explanatory. The implementation types `object` and `object[]` exemplify the usage of virtual roots. Both types are not native C++ types. Instead, they represent the common root of an inheritance hierarchy. The root types provide operations that are common to the virtual specializations, such as for assignment operations and implementations of the core operations of the design types `object`, `object[]` and `classifier` (→ Tab. A.6 on page 144). This approach reduces the overall modeling effort, since not all operations have to be specialized for all types. Specialization should be done whenever there are multiple specializations of an element that have different QoS or implementations.

Among the modeled implementation types there is one notable exception - the `void" type"`. Commonly, `void` is not actually a type but is used to notify that a typed model element does not actually have a type. For simplicity, `void` is treated like a primitive data type. Of course, `void` does not define any features.

In Fig. B.7 predefined classes representing the implementation of the remote object proxy (`IHwObject`), the RTR-Manager (`RTRManager`), and the OSLF (`OSLF_xyz`) are shown. Thereby specialized classes for synchronization, implementation of processes, and exceptions are known. As for the primitive types, a virtual root is used to model a common basis for all C++ classes.

The set of miscellaneous data types is shown in Fig. B.8. These types provide services for system control (`system`), basic input (`istream`) and output (`ostream`). The types `Operation` and `StatementBlock` exemplify the application of Cxx `Operation` and Cxx `StatementBlock` respectively. To better discuss these concepts, in contrast to all other implementation types, the operations of both types are shown in the diagram.

The type `Operation` provides two operations for calling operations synchronously (`call`) or asynchronously (`send`). Both operations take an array of parameters that are to be transferred to the receiver.
object. The return operation returns an object back to the caller of a previous operation invocation. For all operations the QoS-constraints are shown.

StatementBlock represents the concept of statement blocks that are executed within the context of the same behavior. The transfer of control between the blocks can be performed conditionally or unconditionally. For both schemes a respective operation is modeled. Thereby a strict object-oriented approach is taken. Statement blocks are ordinary objects that can send and receive messages which cause control transfers between the blocks. Software implementations may map these operations to unconditional and conditional branch instructions of a microprocessor respectively.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn(in IHwObject: other):IHwObject</td>
<td>Assign an instance of IHwObject to the current instance of IHwObject.</td>
</tr>
<tr>
<td>other</td>
<td>The instance that is assigned to the instance on which the assignment is invoked.</td>
</tr>
<tr>
<td>eq(in IHwObject: other):bool</td>
<td>Compare an instance of IHwObject to the current instance of IHwObject. This operation checks if both references refer to the same physical instance.</td>
</tr>
<tr>
<td>other</td>
<td>The instance for which to check if it is the same instance as the one on which the operation is invoked.</td>
</tr>
<tr>
<td>return</td>
<td>Returns true if the current instance reference and the instance denoted by other refer to the same physical instance. Otherwise, false is returned.</td>
</tr>
<tr>
<td>neq(in IHwObject: other):bool</td>
<td>Compare an instance of IHwObject to the current instance of IHwObject. This operation checks if both references do not refer to the same physical instance.</td>
</tr>
</tbody>
</table>

continued on next page
**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>other</code></td>
<td>The instance for which to check if it is not the same instance as the one on which the operation is invoked.</td>
</tr>
<tr>
<td><code>return</code></td>
<td>Returns true if the current instance reference and the instance denoted by <code>other</code> do not refer to the same physical instance. Otherwise, false is returned.</td>
</tr>
</tbody>
</table>

**create(in type_id: int):type**

Create a new instance of a remote object with a given type. If successful, an instance of the remote type is returned. In case the run-time environment was not able to create an instance null is returned.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>type_id</code></td>
<td>The identifier of the type that is to be instantiated. The type identifiers are computed by MOCCA. The association between the identifiers and types is made explicit in the hardware object model (→ Section 5.5).</td>
</tr>
</tbody>
</table>

**destroy():void**

Destroy an instance of a remote object type.

**read(in address: int):type**

Read a remote object from a specified address. The return parameter `type` must be specialized in order to enable the model compiler to determine the number of bytes and data representation of the read object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>address</code></td>
<td>The address of the object to be read. This address is relative to the start address of the remote object that encapsulates the read object in the address space of the master node. This object is proxied by the instance of <code>IHwObject</code> on which the operation is called.</td>
</tr>
<tr>
<td><code>return</code></td>
<td>Returns the object that is accessible at the given address and having the data type <code>type</code>.</td>
</tr>
</tbody>
</table>

**read(in address: int, in size: int):type[]**

Read a number of remote objects from a specified address. The base type of the return parameter `type` must be specialized in order to enable the model compiler to determine the number of bytes and data representation of the read objects. All objects are assumed having the same type.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>address</code></td>
<td>The address of the first object to be read. This address is relative to the start address of the remote object that encapsulates the read objects in the address space of the master node.</td>
</tr>
<tr>
<td><code>size</code></td>
<td>The number of objects to be read.</td>
</tr>
<tr>
<td><code>return</code></td>
<td>Returns the objects that are accessible at the given address and having the data type <code>type</code>.</td>
</tr>
</tbody>
</table>

**write(in address:int, in val: type):void**

Write an object instance to the specified relative address.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>address</code></td>
<td>The address to which the object is to be written. This address is relative to the start address of the remote object that will encapsulate the written object in the address space of the master node.</td>
</tr>
<tr>
<td><code>val</code></td>
<td>The object to be written. The type of the object <code>type</code> must be specialized in order to allow the model compiler to determined the written number of bytes and the data representation.</td>
</tr>
</tbody>
</table>

**write(in address:int, in values: type[], in size: int):void**

Write a number of object instances to the specified relative address.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>address</code></td>
<td>The address to which the first object is to be written. This address is relative to the start address of the remote object that will encapsulate the written objects in the address space of the master node.</td>
</tr>
</tbody>
</table>
### B. Platform Models

#### B.2.3 Type Mappings

Fig. B.9-B.11 define the mappings of design platform types to types of the C++ implementation platform. All design types are part of the design platform model. As one would expect, the mappings between the types are straightforward.

The realizing types must satisfy the same contract as the realized type. That is, they must have the same value ranges and must provide operations that provide the same functionality as is provided by the respective design types.

![Fig. B.9: Base Primitive Type Mappings](image)

#### B.2.4 Model Compiler Components

The employed compiler components are shown in Fig. B.12. MOCCA_Cpp_Mapper and MOCCA_Cpp_Estimator represent the platform-specific mapper and estimator components. Both of them are used by MOCCA during platform mapping. The component MOCCA_Cpp_Generator realizes the generator and is being used during synthesis to generate the software modules. These software modules are interpreted by a design flow that is proxied by the Make_Software_Modules component. This component triggers...
B.3. VHDL Implementation-Platform

The VHDL implementation platform is used to implement hardware modules for those model elements that have been mapped to reconfigurable hardware. It supports the latest VHDL standard. However, since the focus of this platform is synthesis just the synthesizable language subset is supported. As for the C/C++ implementation platform the purpose of the following sections is to give an overview of the most important concepts of this platform. Thereby the focus is on the concepts that are relevant to understand the examples in this thesis rather than on completeness of presentation. The platform utilizes the "VHDL Implementation Platform Profile" that is defined in Section A.6.3.

the execution of the lower level compilation by means of a Makefile. This approach adds flexibility, since it is more independent from the actual compiler tool chain. The presentation of the compiler component interfaces is outside the scope of this thesis, since this would it require the presentation of compiler internals. The employed component interfaces are likely to change for the same model compiler in the course of time and they will be fairly different for different model compilers, of course.

B.2.5 UML-to-C++ Mapping

Tab. B.6 overviews the mapping of UML model elements to the respective constructs of the C++ language. The mapping is straightforward and common to software implementations of UML models. The mappings are incorporated into the generator component of this implementation platform.

B.3 VHDL Implementation-Platform

Fig. B.10: Primitive Type Mappings

Fig. B.11: Complex Type Mappings

Fig. B.10: Primitive Type Mappings

Fig. B.11: Complex Type Mappings

Tab. B.6 overviews the mapping of UML model elements to the respective constructs of the C++ language. The mapping is straightforward and common to software implementations of UML models. The mappings are incorporated into the generator component of this implementation platform.

B.2.5 UML-to-C++ Mapping

Tab. B.6 overviews the mapping of UML model elements to the respective constructs of the C++ language. The mapping is straightforward and common to software implementations of UML models. The mappings are incorporated into the generator component of this implementation platform.

B.3 VHDL Implementation-Platform

The VHDL implementation platform is used to implement hardware modules for those model elements that have been mapped to reconfigurable hardware. It supports the latest VHDL standard. However, since the focus of this platform is synthesis just the synthesizable language subset is supported. As for the C/C++ implementation platform the purpose of the following sections is to give an overview of the most important concepts of this platform. Thereby the focus is on the concepts that are relevant to understand the examples in this thesis rather than on completeness of presentation. The platform utilizes the "VHDL Implementation Platform Profile" that is defined in Section A.6.3.
B.3.1 Packages

The VHDL implementation platform contains the packages shown in Fig. B.13. The package and library concept of VHDL is not directory-based as in C/C++ and most other languages. Packages are files containing VHDL declarations and definitions that can be reused in several designs. Libraries, or better design libraries, are defined in the language reference manual as a storage for previously analyzed design units. The implementation of the storage is up to the tool chain.

VHDL – The root of the VHDL implementation platform model. The package structure underneath this
### Tab. B.6: UML-to-C++ Mappings

<table>
<thead>
<tr>
<th>UML Model Element</th>
<th>C++ Construct</th>
<th>UML Model Element</th>
<th>C++ Construct</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Structural Elements</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Artifact</td>
<td>File(s)</td>
<td>Component</td>
<td>Set of Classes</td>
</tr>
<tr>
<td>Package</td>
<td>Folder</td>
<td>Class</td>
<td>Class</td>
</tr>
<tr>
<td>Interface</td>
<td>Abstract Class</td>
<td>Property (Attribute)</td>
<td>Attribute</td>
</tr>
<tr>
<td><strong>Behavioral Elements</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Activity</td>
<td>Operation Function</td>
<td>Parameter</td>
<td>Argument</td>
</tr>
<tr>
<td>ActivityGroup (basic block)</td>
<td>Basic Block</td>
<td>ActivityGroup (no basic block)</td>
<td>Statement, Expression</td>
</tr>
<tr>
<td>ConditionalNode</td>
<td>if-then-else, switch</td>
<td>LoopNode</td>
<td>for, do-while, while</td>
</tr>
<tr>
<td><strong>Relationships</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generalization</td>
<td>Class Inheritance</td>
<td>Usage Dependency</td>
<td>Inclusion of Header-File (#include)</td>
</tr>
<tr>
<td><strong>Auxiliary Model Element Properties</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VisibilityKind</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>public</td>
<td>public</td>
<td>protected</td>
<td>protected</td>
</tr>
<tr>
<td>private</td>
<td>private</td>
<td>package</td>
<td>public</td>
</tr>
<tr>
<td>Scope (Feature.isStatic)</td>
<td>true</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>ChangeabilityKind</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unrestricted</td>
<td>Variable/Attribute</td>
<td>readOnly</td>
<td>Constant (#define)</td>
</tr>
<tr>
<td>addOnly</td>
<td>not relevant</td>
<td>removeOnly</td>
<td>not relevant</td>
</tr>
<tr>
<td>ParameterDirectionKind</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>return</td>
<td>Return-Parameter</td>
<td>in</td>
<td>Argument</td>
</tr>
<tr>
<td>out</td>
<td>Pointer-Argument</td>
<td>inout</td>
<td>Pointer-Argument</td>
</tr>
</tbody>
</table>

package resembles the standard VHDL library and package structure.

IEEE - A standard resource library being defined by the Institute of Electrical and Electronics Engineers (IEEE).

IEEE/std_logic_1164 - Standard package defining the multivalue standard logic types and respective simple operators and functions.

IEEE/std_logic_arith - Standard package that complements the std_logic_1164 package by additional arithmetic operators and functions.

IEEE/numeric_std - Standard package that complements the std_logic_116 package by additional arithmetic operators and functions. The package is design toward synthesizability.

work - The standard working library. In this library all design units that have been analyzed during synthesis are stored. The library contains all user-defined design units that are not stored in dedicated resource libraries. Design units that are generated from the design model are stored in this library.

work/mocca_pkg - MOCCA-specific package that contains the data types (mBYTE, mSHORT, and mINT) specializing the standard logic types. The package contains additional operators and functions to work with these types.

components - Contains the model compiler components.
Fig. B.14 illustrates the dependencies between the packages and libraries. All design units that have been generated from design models depend on the standard IEEE libraries and packages. Also, since these units use the MOCCA-specific VHDL types, they depend on the package mocca_pkg.

All modeled dependencies are translated by the model compiler into respective VHDL library clauses and use clauses. A dependency on a package is translated into an import of all elements of the respective package, while a dependency on specific element is translated into an element-specific import. This ensures that designers can select the proper elements to be imported directly through the model.

![Package Dependencies Diagram](image)

**Fig. B.14: Package Dependencies**

### B.3.2 Implementation Types

In Fig. B.15 the most important part of the type hierarchy of the VHDL implementation platform is illustrated. At the core of this hierarchy are the standard IEEE types std_logic and std_logic_vector. Since std_logic_vector represents an array type, i.e. a bundle of instances of std_logic, it is derived from object[]. The type std_logic is a primitive type whose instances represent individual signals.

The type std_logic_vector is specialized and restricted to a fixed number of signals. This is done via the std_logic_vector<x>, whereas x is a positive integral number. The vector comprises x individual signals. The left index and the right index of the signals is set using the constraints VHDL BitVector LeftIndex and VHDL BitVector RightIndex respectively. The types mBYTE, mSHORT, and mINT simplify adding new operators and functions to the specialized standard logic vectors. These types are located in the package mocca_pkg.

Notably, the current VHDL implementation platform does not contain any floating point types. Such types may be added in a straightforward manner. However, the implementation of floating point operations using digital logic is generally expensive in terms of required chip area and latency. For example, a freely available combinational 11-Bit exponent, 52-Bit mantissa floating point multiplier requires about 25% of the overall slices of a Xilinx Virtex-II 3 million gate device [284]. Very good introductory material on the computer arithmetic hardware design can be found in [285].

As for C/C++, the "type" void has been added to the platform to simplify modeling of typed model elements. Although, void is not defined in VHDL it is introduced to explicitly express in the platform model that some typed model element does not actually have a type, i.e. that the model element is not used. The semantics of the types classifier and type is given by the respective design platform types. This is expressed in the type mappings.

All previously documented implementation types are mainly used to implement user-specific behavior. In the following the implementation types realizing implementation components of this platform are presented. These components are used to integrate user designs into the execution environment.
The implementation types implementing clock generators and the communication component are shown in Fig. B.16(a) and B.16(b). Two separate implementation types are used to model the entity and architecture part of each design unit (→ Section 2.2.1). Again, the modeled types proxy the actual VHDL implementation of the components and make them applicable on the modeling level.

**Clocking and Communication Types**

The clock generator (`clock_dcm` and `clock_dcm_struct`) refreshes the clock signal and makes it available at its output. Since clock preparation generally requires analog hardware support the implementation of clock generators is specific to some device or device family. The current implementation uses the digital clock manager (DCM) component which is available in the latest Xilinx FPGA device families [20, 21, 66].

The target platform that is used for desktop computer-based examples in this thesis contains an add-on FPGA card. This card is attached to the peripheral component interconnect (PCI) bus of the computer system using a 9080 PCI input/output (I/O) accelerator device from PLX [286]. The PCI interface of this device that is visible to the FPGA is interfaced to the MOB bus (→ Section 5.4.2). This interfacing is implemented by in a dedicated hardware design (`PLX9080_PCI_BIU` and `PLX9080_PCI_BIU_mixed`). Because the hardware designs can also be reset over the native PCI bus reset signal this design unit also implements the reset generator.
Storage Types

Two types of storage are offered by the VHDL implementation platform - registers and memory blocks. The available register types are illustrated in Fig. B.17. All registers are dual-ported, comprising an external and a local interface. Both interfaces have data width of 8 Bit and use the same clock. The external port is accessible from the environment of the hardware design. The local interface is connected to the application-specific logic. For the local interface different read/write modes are available. If the application only performs read or write accesses on the local interface a register that supports just this particular access model can be used. For the implementation of control registers a special register type exists that simplifies the reset of individual register bits on the local port.

![VHDL Storage Architecture Diagram]

Fig. B.17: Register Implementation Types

Fig. B.18(a) and B.18(b) document the implementation types that are used for the implementation of memory blocks. Like registers, all memory blocks are dual-ported. To simplify their automated integration into hardware designs memory blocks are wrapped by additional glue logic. This logic aligns the native interface of the physical memory device to the MOB.

The target hardware contains two types of memory blocks, namely BlockRAMs and zero bus turnaround time (ZBT)-RAM. BlockRAMs are memory blocks being embedded into the FPGA. Their number and size depends on the particular device. The width of the data interface is configurable. On a Xilinx Virtex-II device all BlockRAMs have size 16 KBit. The storage depth is determined from the size and the width of the data interface. The current VHDL implementation platform model comprises three different versions of BlockRAM interfaces, \texttt{bram16} \_\texttt{32xw}\_\texttt{xst} and \texttt{bram16} \_\texttt{32xw}\_\texttt{xst}\_\texttt{struct}, whereas \texttt{w} stands for the width in number of individual bits comprising the local data. The width of the external data interface is 32 Bit, which is denoted by the 32 in the name\(^1\).

Additionally, the target platform contains one ZBT-RAM component of 2 MBytes. The width of both the local and the external data interface is 32 Bit. Since the physical memory is single ported, the second port is realized in the glue logic.

Implementation Component Interface Types

Implementation types are used to realize implementation components. These components are automatically integrated into hardware designs by the model compiler. In order to simplify hardware generation and to make implementation components applicable to an automated approach, their interfaces must be standardized (→ Section 3.3.4).

\(^1\) Notice, that the name of the BlockRAMs is not interpreted by MOCCA.
In the following the standard implementation component interfaces of the VHDL implementation platform are presented. Although, for the purpose of presentation UML interfaces are used, this does not imply, however, that UML interfaces are also used for the modeling. In fact, from the model compilers point of view it is only important that a specific set of features is available, but it is not important in which context UML interface these features are defined. This gives the component designer more freedom in modeling interfaces according to the requirements of the platform rather than the restrictions of the model compiler.

Fig. B.19-B.22 illustrate the standard interface types of MOCCA. Except from processing components, for each supported type of implementation component a set of standard interfaces is defined. The interface of processing components is determined by their realizing implementation types. Processing components are not instantiated as single entity, but they are implicitly realized by the types instantiated in user designs and glue logic.

In the interface definition the IEEE standard type system for multivalued logic is used, i.e. std_logic and std_logic_vector. The model compiler is not restricted to these types however. Instead, any implementation types that realize the design platform types bit and bit[] can be used. Moreover, all instances of std_logic_vector must be specialized to reflect the actual vector size. For instance, to state that the parameter data in the interface CommLocalAccess (→ Fig. B.20) is 32 Bit, the respective parameter type must be set to std_logic_vector<32>. Recall, that the VHDL type std_logic_vector has an unrestricted size and is therefore not synthesizable.

All signals are considered active high. That is, if a signal carries a one this corresponds to the logical activity of the signal while a driven zero represents logical inactivity.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface MemBlockLocalAccess (→ Fig. B.22)</td>
<td>Operation local_access</td>
</tr>
</tbody>
</table>

Tab. B.12: Memory Block Interface Description

continued on next page
This operation must be implemented by storage components that are stereotyped StorageComponent and that represent memory blocks. The operation implements the local interface of the storage. This interface is being connected to the application-specific logic. The local interface must implement the MOB protocol.

- **i_data**: Represents the data bus of the storage component. The width of the data bus is the size of one addressed word.
- **i_enable**: The parameter determines the activity of a transfer on the bus. A transfer is active whenever the enable is active.
- **i_rw**: The parameter determines the direction of the current transfer on the MOB. It is only valid when i_enable is active. A write transfer is pending when the value carried by this parameter is active.
- **i_address**: Represents the address bus of the MOB. Each address represents a word whose width is the width of the data bus (i_data).
- **ack**: Acknowledges the success of the currently pending transfer to the logical master of the transfer. The provision of this parameter is optional. If not implemented the parameter is active by default. The implementation must ensure that no loss of data can occur. This parameter can be used to enable the communication between components operating at different transfer rates.

### Interface MemBlockExternalAccess (→ Fig. B.22)

**Operation external_access**

This operation must be implemented by storage components that are stereotyped StorageComponent and that represent memory blocks. The operation implements the external interface of the storage. This interface is being connected to the communication component. The external interface must implement the MOB protocol.

- **x_data**: Represents the external data bus of the storage component. The data bus is divided into multiple byte lanes. The number of byte lanes equals the width of the parameter be. Consequently, the width of the data bus must be divisible by eight.
- **x_enable**: The parameter determines the activity of a transfer on the bus. A transfer is active whenever the enable is active.
- **x_rw**: The parameter determines the direction of the current transfer on the MOB. It is only valid when x_enable is active. A write transfer is pending when the value carried by this parameter is active.
- **x_address**: Represents the address bus of the MOB. Each address represents a word whose width is the width of the data bus (x_data). Individual bytes within the word are addressed by a dedicated byte enable parameter (x_be).
- **x_be**: Represents the byte enable of the MOB and is logically a part of the address bus. The parameter comprises as many individual signals as there are byte lanes on the data bus. Each bit of x_be controls the activity of a byte lane. Thereby the leftmost bit controls byte lane zero and the rightmost bit controls the rightmost byte lane. The signals comprising a byte lane may only be driven if the corresponding byte enable signal is active.
- **x_ack**: Acknowledges the success of the currently pending transfer to the logical master of the transfer. The provision of this parameter is optional. If not implemented the parameter is active by default. The implementation must ensure that no loss of data can occur. This parameter can be used to enable the communication between components operating at different transfer rates.

### B.3.3 Implementation Components

Implementation components are building blocks that are used to construct hardware or software designs. On the modeling level, each implementation component is realized by one or more implementation types. The implementation types of the VHDL implementation platform have been presented previously in this
B.3. VHDL Implementation-Platform

SetClock
+setClock (in clock : std_logic) : void

GetClock
+getClock() : std_logic

SetReset
+reset ( in reset : std_logic ) : void

GetReset
+reset () : std_logic

(a) Clocking Interfaces

(b) Reset Interface

Fig. B.19: Clocking and Reset Interfaces

CommLocalAccess
+local_access (inout data : std_logic_vector, out enable : std_logic, out rw : std_logic, out address : std_logic_vector, out be : std_logic_vector, in ack : std_logic = 1) : void

RegLocalAccess
+local_access (inout data : std_logic_vector, in enable : std_logic, in rw : std_logic) : void

RegExternalAccess
+external_access (inoutx data : std_logic_vector, in x enable : std_logic, in x rw : std_logic, out jack : std_logic = 1) : void

CRegLocalAccess
+local_access (outgo : std_logic_vector) : void

+reset (in done : std_logic_vector) : void

(a) Data Register Interfaces

(b) Control Register Interfaces

Fig. B.21: Register Interfaces

Tab. B.7: Clocking Interfaces Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface SetClock (→ Fig. B.19(a))</td>
<td>Operation setClock</td>
</tr>
<tr>
<td>clock</td>
<td>This operation is provided by hardware design units that are clocked by an external clocking source. The provision of this operation by hardware designs is optional.</td>
</tr>
<tr>
<td></td>
<td>The parameter carries the clock signal.</td>
</tr>
<tr>
<td>Interface GetClock (→ Fig. B.19(a))</td>
<td>Operation getClock</td>
</tr>
<tr>
<td>return</td>
<td>This operation is provided by hardware designs providing a clock source at their output, such as clock generator components.</td>
</tr>
<tr>
<td></td>
<td>The current state of the clock is given by the return parameter.</td>
</tr>
</tbody>
</table>
Tab. B.8: Reset Interfaces Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface SetReset (→ Fig. B.19(b))</td>
<td></td>
</tr>
<tr>
<td>Operation reset</td>
<td>This operation is provided by hardware design units that have a reset input. The provision of this operation by hardware designs is optional.</td>
</tr>
<tr>
<td>reset</td>
<td>This parameter carries the reset signal.</td>
</tr>
<tr>
<td>Interface GetReset (→ Fig. B.19(b))</td>
<td></td>
</tr>
<tr>
<td>Operation reset</td>
<td>This operation is provided by hardware designs providing a reset source at their output. It must be implemented by reset generator components that are stereotyped ResetGenerator.</td>
</tr>
<tr>
<td>return</td>
<td>The current state of the reset is given by the return parameter.</td>
</tr>
</tbody>
</table>

Tab. B.9: Communication Interface Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface CommLocalAccess (→ Fig. B.20)</td>
<td></td>
</tr>
<tr>
<td>Operation local_access</td>
<td></td>
</tr>
<tr>
<td>data</td>
<td>Represents the data bus of the communication component. The data bus is divided into multiple byte lanes. The number of byte lanes equals the width of the parameter be. Consequently, the width of the data bus must be divisible by eight.</td>
</tr>
<tr>
<td>enable</td>
<td>The parameter determines the activity of a transfer on the bus. A transfer is active whenever the enable is active.</td>
</tr>
<tr>
<td>rw</td>
<td>The parameter determines the direction of the current transfer on the MOB. It is only valid when enable is active. A write transfer is pending when the value carried by this parameter is active.</td>
</tr>
<tr>
<td>address</td>
<td>Represents the address bus of the MOB. Each address represents a word whose width is the width of the data bus (data). Individual bytes within the word are addressed by a dedicated byte enable parameter (be).</td>
</tr>
<tr>
<td>be</td>
<td>Represents the byte enable of the MOB and is logically a part of the address bus. The parameter comprises as many individual signals as there are byte lanes on the data bus. Each bit of be controls the activity of a byte lane. Thereby the leftmost bit controls byte lane zero and the rightmost bit controls the rightmost byte lane. The signals comprising a byte lane may only be driven if the corresponding byte enable signal is active.</td>
</tr>
<tr>
<td>ack</td>
<td>Acknowledges the success of the currently pending transfer to the logical master of the transfer. The provision of this parameter is optional. If not implemented the parameter is active by default. The implementation must ensure that no loss of data can occur. This parameter can be used to enable the communication between components operating at different transfer rates.</td>
</tr>
</tbody>
</table>
Tab. B.10: Data Register Interfaces Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| **Interface RegLocalAccess (→ Fig. B.21(a))** | **Operation local_access**  
This operation must be implemented by all registers. It represents the local interface of the register, i.e. the interface that is accessed by the application-specific logic. |
| i_data         | Represents the data port of the register. The width of this port is user-definable. To get a good compromise between fragmentation and additionally required glue logic a width of 8Bit is recommended. The values carried by the parameter is only valid when i_enable is active. The data is driven by the register, when i_rw is inactive. The data is stored in the register at a rising clock edge (→ SetClock interface) when i_rw is active. If the register is written concurrently by multiple sources the stored value is indeterminate. Implementations of this interface may vary the direction kind of this parameter in order to implement registers that can only be read or written on the local interface. |
| i_enable       | Controls the validity of the data. If active, the data is considered valid.                                                                   |
| i_rw           | Determines the transfer direction of the data. The value carried by this parameter is only valid, if a transfer is pending. When active, the data carried by i_data is written to the register. Inactivity of this parameter signals a read transfer. That is, the data must be driven by the register. |
| **Interface RegExternalAccess (→ Fig. B.21(a))** | **Operation external_access**  
This operation must be implemented by all registers. It represents the external interface of the register, i.e. the interface that is connected to the local interface of the communication component (→ Tab. B.9). The protocol is determined by the communication component. The interface must implement the MOB protocol. |
| x_data         | Represents the data port of the register. The width should be equal to the width of a byte lane on the local interface of the communication component. The values carried by the parameter is only valid when x_enable is active. The data is driven by the register, when x_rw is inactive. The data is stored in the register at a rising clock edge (→ SetClock interface) when x_rw is active. If the register is written concurrently by multiple sources the stored value is indeterminate. |
| x_enable       | Controls the validity of the data. If active, the data is considered valid.                                                                   |
| x_rw           | Determines the transfer direction of the data. The value carried by this parameter is only valid, if a transfer is pending. When active, the data carried by x_data is written to the register. Inactivity of this parameter signals a read transfer. That is, the data must be driven by the register. |
| x_ack          | This optional parameter is used to acknowledge the success of the current transfer to the component that initiated the transfer (i.e. the transfer master). When active, the transfer is considered successful and can be finished. |
**Fig. B.22: Memory Block Interfaces**

**Tab. B.11: Control Register Interfaces Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface CRegLocalAccess (→ Fig. B.21(b))</td>
<td></td>
</tr>
<tr>
<td><strong>Operation</strong></td>
<td></td>
</tr>
<tr>
<td><strong>local_access</strong></td>
<td>This operation <em>must</em> be implemented by all control registers. It represents the local interface of the register, i.e. the interface that is accessed by the application-specific logic.</td>
</tr>
<tr>
<td>go</td>
<td>Represents the data port of the register. The width of this port is user-definable. To get a good compromise between fragmentation and additionally required glue logic a width of 8Bit is recommended. If the register is modified concurrently by multiple sources the stored value is indeterminate. The individual bits of the parameter represent GO signals. The parameter is always driven by the register.</td>
</tr>
<tr>
<td><strong>reset</strong></td>
<td>This operation <em>must</em> be implemented by all control registers. It represents the local reset interface of the register. The operation allows the application-specific logic to reset individual bits of the go parameter on the local interface.</td>
</tr>
<tr>
<td>done</td>
<td>Represents the reset vector. Each bit of this vector corresponds with the bit at the same position of the go parameter in the local_access operation. The activation of a bit resets, i.e. inactivates, the corresponding go bit. The parameter must always be driven.</td>
</tr>
</tbody>
</table>

section. Fig. B.23-B.25 define the realization of the implementation components of this platform with the implementation types. The processing component, which is not shown in the figures, is realized by all implementation types which are used to realize application-specific designs.

The mere specification of component interfaces is not sufficient for defining the principal structure of the hardware designs that can be constructed with the components. Additionally, the associations between the components in terms of provided and required interfaces must be defined. In UML this is done using interfaces and ports. For a clear presentation, the component structure is given in two separate figures Fig. B.26 and Fig. B.27, which define the clocking and data exchange between the components of the implementation platform.

The coupling between components is loosely defined. That is, most components provide interfaces but do not directly require particular interfaces to be provided by other components. For example, in Fig. B.26 the component SystemClock provides the interface GetClock, which enables other components to get the current clock. Likewise, components such as BlockRAM and RF provide the SetClock interface. None of the components implementing the SetClock interface require the existence of a component offering the GetClock interface however. This relationship is established by the model compiler.

A central clock generator component provides the clock for all other implementation components. The components themselves may use further clocks internally, however. The clock generator component provides the clock signal on a dedicated port that implements the GetClock interface. The sinks, with respect to the clock signal, provide the SetClock interface.

All data exchange among the implementation components is done only via the publicly visible interfaces. Thereby different interfaces are used in order to accomplish the different semantics of the component types. As can be seen in Fig. B.27, the reconfigurable fabric (RF) does not define a dedicated interface to
communicate with the storage components. Instead, the interface is generated by the model compiler depending on the storage component types that are actually accessed by the fabric. Conceptually, the ports and interfaces are adopted to the requirements of the current design.

**B.3.4 Type Mappings**

The type mappings of the platform are given in Fig. B.28 and Fig. B.29. Only the subset of implementation types that directly realizes design types is mapped. Thereby a VHDL implementation type can implement one or more design platform types, which is a consequence of lowering the level of abstraction. All mapped types that are used directly in user designs are realized basically using either `std_logic` or `std_logic_vector`.

**B.3.5 Model Compiler Components**

The model compiler components of the VHDL platform are illustrated in Fig. B.30. For the platform mapping and synthesis two components that are specialized in synthesizing designs that use the MOB are used. This is because the mapper implements MOB-specific transformations, such as the array access transformation (→ Tab. C.3). These transformations are embodied into actual hardware modules by the generator component. The estimator component is fairly generic since it performs estimation after transformation. Consequently, it does not require additional knowledge of the MOB.

Each MOCCA_VHDL_Generator is associated with an instance of Xflow_Script_Generator. The
B. Platform Models

latter is responsible for generating synthesis scripts for each generated hardware module. The synthesis scripts are specific to the Xilinx Synthesis Tools (XST) Xflow synthesis flow [253]. This generator is invoked repeatedly by the VHDL generator during synthesis and retrieves all necessary information to be included into the script. For instance, this generator collects for each generated hardware module the files that comprise the design of the module. Moreover, all constraints are collected and passed on to the lower level synthesis flow. At the end of system-level synthesis, when all synthesis scripts have been generated, the script generator invokes the actual synthesis process for each generated script. Notice, that the Xilinx synthesis specific compiler components are modeled in a nested XST implementation platform which is not shown here.

Fig. B.24: Register Storage Components
Fig. B.25: Memory Block Components

Fig. B.26: Clocking of Implementation Components
**Fig. B.27:** Data Exchange between Implementation Components

**Fig. B.28:** Primitive Type Mappings

**Fig. B.29:** std_logic Type Mappings
Fig. B.30: MOCCA Compiler Components
### B.3.6 UML-to-VHDL Mapping

Tab. B.13: UML-to-VHDL Mappings

<table>
<thead>
<tr>
<th>Model Element</th>
<th>VHDL Construct</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Structural Elements</strong></td>
<td></td>
</tr>
<tr>
<td>Artifact (stereotyped “Configuration”)</td>
<td>technology dependent, Bitstream (on RTR-FPGA)</td>
</tr>
<tr>
<td>Component</td>
<td>Entity/Architecture: instantiates Classes, Communication Interface, and implements Address Decoders, Register File, Message Dispatch</td>
</tr>
<tr>
<td>Package</td>
<td>Folder</td>
</tr>
<tr>
<td>Class</td>
<td>Entity/Architecture: instantiates Operations</td>
</tr>
<tr>
<td>Interface</td>
<td>not relevant</td>
</tr>
<tr>
<td>Property (Attribute)</td>
<td>Register, Storage Component (dual-ported)</td>
</tr>
<tr>
<td>Variable</td>
<td>Latch, Storage Component</td>
</tr>
<tr>
<td><strong>Behavioral Elements</strong></td>
<td></td>
</tr>
<tr>
<td>Operation</td>
<td>Entity/Architecture: contains Activity and auxiliary logic</td>
</tr>
<tr>
<td>Parameter</td>
<td>Register, Storage component (dual-ported)</td>
</tr>
<tr>
<td>Activity</td>
<td>3 Processes (FSM, data-path, sync) and auxiliary logic</td>
</tr>
<tr>
<td><strong>Relationships</strong></td>
<td></td>
</tr>
<tr>
<td>Generalization</td>
<td>multiplexed polymorphic Operations</td>
</tr>
<tr>
<td>Usage Dependency</td>
<td>Inclusion of Package or Library</td>
</tr>
<tr>
<td><strong>Auxiliary Model Element Properties</strong></td>
<td></td>
</tr>
<tr>
<td>VisibilityKind</td>
<td></td>
</tr>
<tr>
<td>all</td>
<td>public</td>
</tr>
<tr>
<td>true</td>
<td>Class Feature</td>
</tr>
<tr>
<td>false</td>
<td>Instance Feature</td>
</tr>
<tr>
<td>ChangeabilityKind</td>
<td></td>
</tr>
<tr>
<td>unrestricted</td>
<td>Variable/Attribute</td>
</tr>
<tr>
<td>readOnly</td>
<td>constant</td>
</tr>
<tr>
<td>addOnly</td>
<td>not relevant</td>
</tr>
<tr>
<td>removeOnly</td>
<td>not relevant</td>
</tr>
<tr>
<td>ParameterDirectionKind</td>
<td></td>
</tr>
<tr>
<td>return, out</td>
<td>external read-only, local write-only Register, Storage Component</td>
</tr>
<tr>
<td>in</td>
<td>external write-only, local read-only Register, Storage Component</td>
</tr>
<tr>
<td>inout</td>
<td>external and local read/write Register, Storage Component</td>
</tr>
</tbody>
</table>
This section gives an overview of the deployment platform model that is used for the examples in this thesis. The platform is a standard desktop PC which is augmented by a FPGA add-in card. This computer system was used to perform all tests regarding execution time and compilation time. It possesses the following hardware architecture properties and relevant software packages:

- Processor: Intel Pentium 4, 2400 MHz [268]
- Chipset: Intel i854PE
- Physical Memory: 1 GByte (PC2100 133MHz)
- FPGA Add-In Card: Alpha Data ADM-XRC-II PCI [287]
  - Xilinx Virtex-II (3 million gate equivalents, 96 Block-RAMs, 1152 Pins), Speedgrade 5 [21]
  - 100 MHz
  - 6 MByte ZBT RAM
  - PLX 9080 PCI bridge [286]
- Operating System: Windows 2000/XP
- C/C++ Compiler: GNU gcc C/C++ compiler 3.2.3
- Logic Synthesis: Xilinx ISE 7.1, WebPack 8.1

The hardware architecture is modeled using the "Deployment Platform Profile" which is presented in Section A.6.4. This platform provides the deployment locations and resources that are available for the execution of design implementations.

The platform comprises two nodes h0 and h1, whereas h0 acts as master node. The nodes are connected by a communication path. The constraints being defined for the nodes are given in Tab. B.14. Additional constraints, such as the package type and the speed grade may be defined in the model. Such constraints are not interpreted by MOCCA and thus they are not part of the profiles. MOCCA does, however, handle this information transparently and passes them to the lower level flow.
## Tab. B.14: Deployment Platform Nodes Constraints

<table>
<thead>
<tr>
<th>Constraint/Tagged Value (Property)</th>
<th>h0</th>
<th>h1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraints</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Allocatable</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>Tagged Values (Properties)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ClockCycle</td>
<td>0.416 ns</td>
<td>10 ns</td>
</tr>
<tr>
<td>ImplementationArea</td>
<td>1 GByte</td>
<td>3000000 Gate</td>
</tr>
<tr>
<td>StaticActivity-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SchedulingPolicy</td>
<td></td>
<td>asap</td>
</tr>
<tr>
<td>StaticActivity-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SchedulingSlots</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>AddressAlignment</td>
<td>-</td>
<td>TypeInstanceSize</td>
</tr>
<tr>
<td>BaseAddress</td>
<td>-</td>
<td>determined dynamically by the operating system</td>
</tr>
</tbody>
</table>
C. MODEL TRANSFORMATIONS

C.1 Primitive Transformations

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bind/unbind({Resource Service}, Model Element)</td>
<td>Bind/unbind a set of resource services from the element. In case of bind the element is realized with the resource services in the set.</td>
</tr>
<tr>
<td>join/split(Feature, Class)</td>
<td>Join (split) a feature, like attribute or operation, to (from) a class. In case of join the class will encapsulate the feature.</td>
</tr>
<tr>
<td>join/split(Class, Component)</td>
<td>Join (split) a class to (from) a component. In case of join the component will realize the class.</td>
</tr>
<tr>
<td>join/split(Component, Node)</td>
<td>Join (split) a component to (from) a node. In case of join the component is deployed on the node.</td>
</tr>
<tr>
<td>implement(Behavior, Operation)</td>
<td>Associates the behavior to the operation. The behavior defines the implementation of that operation. The operation provides the interface to the behavior.</td>
</tr>
<tr>
<td>implement(Behavior, Class)</td>
<td>Associates the behavior to the class. The behavior defines the states the instances of the class may be in during their life-time.</td>
</tr>
</tbody>
</table>
### C.2 Technology Independent Transformations

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pruning</td>
<td>a</td>
<td>Eliminate unused model elements from the model. This transformation works on all kinds of model elements in the hierarchy. The optimization leads to an improvement in area and power. It is enabled by local and global control-flow analysis. Operations that are never invoked and classes that are never instantiated directly or indirectly through their specializations, are excluded from platform mapping and synthesis.</td>
</tr>
<tr>
<td>common-sub-expression elimination</td>
<td>p</td>
<td>Eliminate MAL expressions evaluating to the same result. Such expressions are replaced by references to a single computation of the result [229].</td>
</tr>
<tr>
<td>dead-code elimination</td>
<td>p</td>
<td>Remove MAL expressions whose result is never used [229].</td>
</tr>
<tr>
<td>unreachable-code elimination</td>
<td>p</td>
<td>Remove MAL expressions and statements that are never evaluated [229].</td>
</tr>
<tr>
<td>copy propagation</td>
<td>a</td>
<td>Replace copies of an expression by the original expression. This optimization can enable arithmetic/logic optimizations, improve scheduling, and reduce register/memory consumption [229].</td>
</tr>
<tr>
<td>loop unrolling</td>
<td>p</td>
<td>Unroll several iterations of loops. This optimization may enable further optimizations (elimination of common sub-expressions, loop index variables, and dead code) at the cost of possible code bloat [229].</td>
</tr>
<tr>
<td>loop index variable elimination</td>
<td>p</td>
<td>Copy propagation of the index variable of a completely unrolled loop. Since the propagated copy is constant this is also an instance of constant propagation [159].</td>
</tr>
<tr>
<td>unused variable elimination</td>
<td>p</td>
<td>Remove local variables that are neither defined nor used. Local variables that are defined but not used are removed by dead-code elimination.</td>
</tr>
<tr>
<td>variable merging</td>
<td>p</td>
<td>Merge variables with the same type and mutual exclusive lifetimes into a single variable. This generally causes the reduction in registers and memory. In hardware variable merging can require the allocation of additional multiplexers [175].</td>
</tr>
<tr>
<td>arithmetic/logic simplifications</td>
<td>p</td>
<td>A group of optimizations, including operator strength reduction, elimination of algebraically or logically equivalent computations, and constant folding [229].</td>
</tr>
<tr>
<td>constant propagation</td>
<td>p</td>
<td>Replace uses variables that carry a constant value by the constant as long as the variable is not assigned a different value. Using extensive control-flow and data-flow analyses constant propagation is performed within individual behaviors and between behaviors by analyzing the message exchange in the model [229].</td>
</tr>
<tr>
<td>loop invariant code motion</td>
<td>p</td>
<td>Move expressions of a loop body that do not depend on the execution of the loop before the loop header [229].</td>
</tr>
</tbody>
</table>

*a - automatic, m - manual, p - parameterizable*
C.3 Technology Dependent Transformations

Tab. C.3: MOCCA Technology Independent Optimizations

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inlining</td>
<td>p</td>
<td>Replace the invocation of a behavior by the behavior itself. This optimization eliminates the invocation and may pave way for latency optimizations. Code explosion is a frequent negative side effect. The inlined behavior must not be inclusion polymorphic [229]. Used if a platform does not support the implementation of invocation actions.</td>
</tr>
<tr>
<td>exlining</td>
<td>m</td>
<td>Exline activity groups from their behaviors, and associate the activity group with a new behavior. The activity group is replaced by an invocation action in the original behavior. Represents the reverse operation of inlining. Used in software implementations to share instructions. Exlining is useful if an activity group is not (efficiently) implementable at some platform. This activity group can be exlined and mapped to a different platform [288, 289].</td>
</tr>
<tr>
<td>array access transformation</td>
<td>a</td>
<td>Used in hardware implementations when mapping arrays to memory blocks. All accesses to arrays are transformed such that they comply to the physical interface of the memory. Reads and writes of array elements are transformed into explicit read and write transfers from/to the memory using dedicated address-, data-, and control-signals. This is not actually an optimization, but it increases the explored part of the design space.</td>
</tr>
<tr>
<td>multi-cycling</td>
<td>a</td>
<td>Assign an operation to a sequence of time steps (→ Section 4.1.3). This optimization is used in data-path scheduling of FSMDs. It enables the utilization of functional units that are slower than one time step. Therefore, it aims at area/power reduction and can enable the implementation of data-paths using a specific target technology or library.</td>
</tr>
<tr>
<td>chaining</td>
<td>a</td>
<td>Assign control- and/or data-dependent operations to the same time step (→ Section 4.1.3). This optimization is used during data-path scheduling of FSMD-based designs. It decreases the number of registers and multiplexers, at cost of reduced sharing of functional units. Often opportunities for intra-time step optimizations are created. Standard chaining schedules only operations to the control same step whose total latency is less or equal the clock frequency. Extended chaining ignores the control step boundaries and allows chaining of operations over multiple steps. This combination of standard operation chaining and multi-cycling frequently creates additional optimization opportunities, since potentially more operations can be chained. Moreover, it tends to increase the maximum frequency of designs.</td>
</tr>
</tbody>
</table>

* a - automatic, m - manual, p - parameterizable
C. Model Transformations
D. EXPERIMENTAL RESULTS

D.1 Run-Time Reconfiguration Characteristics

The average time $t_{\text{conf}}$ (→ Eq. 2.1) required to reconfigure the FPGA $h_1$ on the employed FPGA add-in card is summarized in Tab. D.1 (→ Section B.4). The reconfiguration from a file containing the bitstream takes longer than the reconfiguration from a memory buffer into which the bitstream is loaded before the reconfiguration is started. In both scenarios DMA decreases the latency significantly. In the fastest mode - reconfiguration from a buffer using DMA - the theoretical minimum reconfiguration time ($\approx 60$ ms) of this particular FPGA is nearly reached [21]. This mode is used by the RTR-Manager which adds approximately 5 ms overhead when a FPGA is reconfigured (→ last column in Tab. D.1).

<table>
<thead>
<tr>
<th></th>
<th>File</th>
<th>File (DMA)</th>
<th>Buffer</th>
<th>Buffer (DMA)</th>
<th>RTR-Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{\text{conf}}$ [ms]</td>
<td>660,36</td>
<td>312,05</td>
<td>422,17</td>
<td>65,2</td>
<td>69,24</td>
</tr>
<tr>
<td>$\sigma$ [ms]</td>
<td>18,49</td>
<td>5,82</td>
<td>4,28</td>
<td>6,25</td>
<td>5,92</td>
</tr>
</tbody>
</table>

In addition to the FPGA reconfiguration latency the RTR-Manager requires time for the creation ($t_{\text{create}}$) and destruction ($t_{\text{destroy}}$) of hardware objects and their proxies. This overhead is caused mostly by the management and search of the data structures. It is summarized in Tab. D.2. The figures only quantify the average effort. If the creation of a hardware object necessitates the reconfiguration of a FPGA this causes additional overhead (→ Tab. D.1).

<table>
<thead>
<tr>
<th>Object Creation</th>
<th>Object Destruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{\text{create}}$ [ns]</td>
<td>$\sigma$ [ns]</td>
</tr>
<tr>
<td>$t_{\text{destroy}}$ [ns]</td>
<td>$\sigma$ [ns]</td>
</tr>
<tr>
<td>3851,49</td>
<td>653,28</td>
</tr>
<tr>
<td>2969,57</td>
<td>67,27</td>
</tr>
</tbody>
</table>

Tab. D.3 summarizes the average communication overhead $t_{\text{comm}}$ (→ Eq. 2.1) for read and write data transfers between nodes $h_0$ and $h_1$. Thereby a read transfer is caused by a read access of $h_0$ to some data element in the address space being allocated by $h_1$. Respectively, a write transfer is caused by $h_0$ to write a data element to $h_1$. Both nodes are connected through a PCI-bus running at 33 MHz, whereas $h_0$ operates as bus master and $h_1$ solely operates as bus slave. All remote data transfers are handled by a proxy of a hardware object. The dereferencing of the proxy and the calculation of the accessed address incur additional overhead. The time for a single transfer can be derived from the block transfers, because the overhead caused by the proxy is distributed over many transfers. Then a single integer write transfer takes approximately 4 cycles on the PCI-bus ($\frac{33\text{MHz}}{5\text{ns}} \cdot 5\text{ns} \approx 4$) and a read transfer 23 PCI cycles. Burst transfers are not supported by the PCI-MOB bridge. In case of single transfers, the difference between writes and reads is significant. Also there is a large variance in the data. The reason for this behavior could not be found until now¹. However, buffering in the southbridge and side effects of the operating system are likely.

¹ Measurements in the PCI-MOB bridge have not shown a significant difference between the transfer types. Read and write transfers finished in about 3.4 PCI cycles.
D. Experimental Results

Tab. D.3: Remote Communication Overhead on the PC-based Platform

<table>
<thead>
<tr>
<th>Transfer Size [Bit]</th>
<th>$t_{\text{comm}}$ [ns]</th>
<th>$\sigma$ [ns]</th>
<th>$t_{\text{comm}}$ [ns]</th>
<th>$\sigma$ [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Transfers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>78,54</td>
<td>13,98</td>
<td>1613,41</td>
<td>13,48</td>
</tr>
<tr>
<td>16</td>
<td>74,71</td>
<td>6,86</td>
<td>1606,92</td>
<td>12,05</td>
</tr>
<tr>
<td>32</td>
<td>77,04</td>
<td>16,21</td>
<td>1617,74</td>
<td>21,68</td>
</tr>
<tr>
<td>Block Transfers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100 · 16</td>
<td>3584,59</td>
<td>3769,83</td>
<td>77530,09</td>
<td>217,59</td>
</tr>
<tr>
<td>200 · 16</td>
<td>18433,29</td>
<td>6516,28</td>
<td>153826,0</td>
<td>105,50</td>
</tr>
<tr>
<td>300 · 16</td>
<td>32699,60</td>
<td>5607,95</td>
<td>252622,82</td>
<td>64471,20</td>
</tr>
<tr>
<td>400 · 16</td>
<td>48533,56</td>
<td>5862,05</td>
<td>308427,23</td>
<td>5277,64</td>
</tr>
<tr>
<td>500 · 16</td>
<td>65125,47</td>
<td>5905,95</td>
<td>383715,74</td>
<td>1725,83</td>
</tr>
</tbody>
</table>

D.2 Boolean Neural Network

D.2.1 Description of BNN Tests

To evaluate the presented approach multiple designs of a BNN (BNN0..BNN14) were modeled. All designs have the same functionality, which is described in different ways. This is to test the effects of different designs styles on the implementation characteristics. The functionality of the Boolean network was already described in Example 7.1 on page 117. The designs are characterized as follows:

BNN0 - This design calculates the output vector and the k-functions of the network from scalar attributes representing the inputs ($x_0..x_2$) and the outcomes of the k-functions ($k_01..k_04$) and the output layer ($y_00..y_09$). Before the calculation is started, the input vector is transferred to the BNN as array. The scalar attributes are extracted from the array by a dedicated operation (→ Listing D.2). After computation, the scalar values comprising the output vector are copied into the array $y$ (→ Listing D.3) which is communicated back to the caller. The computation of the output layer and the hidden layer is modeled using one operation per neuron ($k_1()..k_4()$, $y_1()..y_9()$). The operations are invoked synchronously by the calculate(...) operation, which represents the entire network.

Listing D.1: Design of calculate(...) of BNN0

```
1 k01=k1(); k02=k2(); k03=k3(); k04=k4(); y00=y0();
2 y01=y1(); y02=y2(); y03=y3(); y04=y4(); y05=y5();
3 y06=y6(); y07=y7(); y08=y8(); y09=y9();
4 return true;
```

Listing D.2: Design of init_x(...) of BNN0

```
x0=inputs[0]; x1=inputs[1]; x2=inputs[2];
```

Listing D.3: Design of get_y(...) of BNN0

```
```

BNN1 - Like BNN0, but the input vector and the output vector are encoded into a 32 Bit integer value whose individual bits represent an input or output value of the network respectively. Encoding and decoding is performed by two dedicated operations (get_y(...), init_x(...)) (→ Listings D.5, D.4).
Listing D.4: Design of init_x(...) of BNN1

```c
for (int i=0; i<3; i++) {
    switch(i) {
    case 0:
        if ((x & 1) == 1) x0 = true;
        else x0 = false; break;
    case 1:
        if ((x & 1) == 1) x1 = true;
        else x1 = false; break;
    case 2:
        if ((x & 1) == 1) x2 = true;
        else x2 = false; break;
    }
    x = x >> 1;
}
```

Listing D.5: Design of get_y(...) of BNN1

```c
int y=0;
for (int i=0; i<10; i++) {
    switch (i) {
    case 0: y00; break;
    case 1: y01; break;
    case 2: y02; break;
    case 3: y03; break;
    case 4: y04; break;
    case 5: y05; break;
    case 6: y06; break;
    case 7: y07; break;
    case 8: y08; break;
    case 9: y09; break;
    }
    y = y << 1;
}
return y;
```

BNN2 - Like BNN0, but the explicit extraction/packing of the values from/into an array is omitted. Each input and output values is transferred individually.

BNN3 - Another modification of BNN0 in that the packing of the array elements into the output array is performed using a loop (→ Listing D.6). The objective of this design is to decrease design complexity stemming from loads of individually described array accesses. In contrast to Listing D.5, the output vector is transferred back to the caller using an output parameter rather than the return parameter.

Listing D.6: Design of get_y(...) of BNN3

```c
for (int i=0; i<10; i++) {
    boolean result = false;
    switch (i) {
    case 0: result=y00; break;
    case 1: result=y01; break;
    case 2: result=y02; break;
    case 3: result=y03; break;
    case 4: result=y04; break;
    case 5: result=y05; break;
    case 6: result=y06; break;
    case 7: result=y07; break;
    case 8: result=y08; break;
    case 9: result=y09; break;
    }
```
BNN4 - A modification of BNN2 that avoids using an individual operation per neuron. Instead the computations of all neurons are flattened into the `calculate(...)` operation (→ Listing D.7). This design avoids message receiver inlining (→ Tab. C.3) and can therefore be used to test negative effects of design decomposition using operations.

Listing D.7: Design of `calculate(...)` of BNN4

```plaintext
k01 = (!x0&&!x2 || x0&&!x1&&x2 || x0&&x1&&!x2);
k02 = (x0&&!x1&&x2); k03 = (!x0&&!x1);
k04 = (!x0&&x1&&!x2 || x0&&x1);
y00 = k01; y01 = k01 || k02; y02 = k03 || k04;
y03 = k01 || k03; y04 = k02 || k04; y05 = k01 || k04;
y06 = k01 || k02 || k03; y07 = k02 || k03;
y08 = k02 || k03 || k04; y09 = k04;
return true;
```

BNN5 - Like BNN4, but additionally common sub-expressions are eliminated manually (→ Listing D.8). The objective of this design is to test the effectiveness of automatic common sub-expression elimination (→ Tab. C.3).

Listing D.8: Design of `calculate(...)` of BNN5

```plaintext
boolean nx0 = !x0;
boolean nx1 = !x1;
boolean nx2 = !x2;
boolean nx0_a_nx2 = nx0&&!nx2;
boolean nx0_a_nx1 = x0&&!nx1;
k01 = (nx0_a_nx2 || x0_a_nx1&&c || a&&b&&!nx2);
k02 = (x0_a_nx1&&!nx2); k03 = (nx0&&nx1);
k04 = (nx0_a_nx2&&a&&c);
y00 = k01; y01 = k01 || k02; y02 = k03 || k04;
y03 = k01 || k03; y04 = k02 || k04; y05 = k01 || k04;
y06 = k01 || k02 || k03; y07 = k02 || k03;
y08 = k02 || k03 || k04; y09 = k04;
return true;
```

BNN6 - This design avoids using attributes and explicit operations to transfer the input and output vectors of the network. This design style makes the network simpler to use, because all data is transferred when the `calculate(...)` operation is called. In this design `calculate(...)` incorporates the functionality of `init_x(...)` and `get_y(...)` of Listings D.2 and D.3. The other objective of this design is to test if the use of attributes affects the optimality of implementations. In BNN6 the input and output vectors are transferred using arrays. The computation is done directly from the array, i.e. there is no explicit data extraction. The effect should be the sequentialization of all computations since only one array element can be accessed at any point in time. As in BNN0 the neurons are modeled using individual operations.

BNN7 - Like BNN6, however the computation of the output vector values and their writing into the output array is done using a loop. Further, the elements of the input vector are copied into local scalar variables (`x0`...`x2`) before the actual computation is started. The objective of this design is to test the effects of loops and conditional executions.

Listing D.9: Design of `calculate(...)` of BNN7

```plaintext
boolean x0=x[0]; boolean x1=x[1]; boolean x2=x[2];
for (int i = 0; i < 10; i++) {
    boolean result=false;
    switch(i){
```
D.2. Boolean Neural Network

BNN8 - Like BNN7, but the explicit copying of the input vector elements into local variables is omitted. The result are more array accesses, which should make the implementation more complex.

Listing D.10: Design of calculate(...) of BNN8

```cpp
for (int i = 0; i < 10; i++)
{
    boolean result = false;
    switch (i)
    {
        case 0: result = y0(x[0], x[1], x[2]); break;
        case 1: result = y1(x[0], x[1], x[2]); break;
        case 2: result = y2(x[0], x[1], x[2]); break;
        case 3: result = y3(x[0], x[1], x[2]); break;
        case 4: result = y4(x[0], x[1], x[2]); break;
        case 5: result = y5(x[0], x[1], x[2]); break;
        case 6: result = y6(x[0], x[1], x[2]); break;
        case 7: result = y7(x[0], x[1], x[2]); break;
        case 8: result = y8(x[0], x[1], x[2]); break;
        case 9: result = y9(x[0], x[1], x[2]); break;
    }
    y[i] = result;
}
return true;
```

BNN9 - Like BNN1, but using parameters instead of attributes.

BNN10 - Like BNN2, but using parameters instead of attributes.

BNN11 - Like BNN4, but using parameters instead of attributes.

BNN12 - Like BNN5, but using parameters instead of attributes.

BNN13 - This design uses array typed attributes to store the input vector, the values of the hidden layer (k-functions), and the output vector. Each array element is accessed as often as required. Operations are used to model the functionality of the neurons.

Listing D.11: Design of calculate(...) of BNN13

```cpp
return true;
```

BNN14 - Modification of BNN13 that computes the values of the hidden layer and the output layer using loops and conditional execution.
The optimization levels are defined as follows, using the optimizations given in Tab. C.2 and C.3:

**L0** - variable merging, unreachable code elimination, in hardware additionally message receiver inlining, array access transformation, and multi-cycling.

**L1** - L0 plus arithmetic/logic simplifications and copy propagation.

**L2** - L1 plus common sub-expression elimination.

**L3** - L2 plus dead code elimination and local constant propagation.

**L4** - L3 plus loop invariant code motion.

**L5** - L4 plus loop unrolling with maximal 8 iterations (performed after loop invariant code motion) and loop index variable elimination.

**L6** - L5 plus global constant propagation.

**L7** - L6 plus standard operation chaining.

**L8** - L7 plus extended operation chaining.

**L9** - L8 plus pruning.
## D.2. Boolean Neural Network

### D.2.2 Hardware Implementation of the BNNs

Tab. D.4: FPGA Communication Latencies of the BNNs (L9)

<table>
<thead>
<tr>
<th>Design</th>
<th>( t_{\text{write, x}} ) [ns]</th>
<th>( \sigma ) [ns]</th>
<th>( t_{\text{read, y}} ) [ns]</th>
<th>( \sigma ) [ns]</th>
<th>( t_{\text{comm}} ) [ns]</th>
<th>( \sigma ) [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNN0</td>
<td>200.01</td>
<td>39.72</td>
<td>7019.42</td>
<td>117.91</td>
<td>7219.43</td>
<td>124.08</td>
</tr>
<tr>
<td>BNN1</td>
<td>146.43</td>
<td>24.00</td>
<td>1600.44</td>
<td>14.38</td>
<td>1746.87</td>
<td>36.02</td>
</tr>
<tr>
<td>BNN2</td>
<td>293.53</td>
<td>7.21</td>
<td>15663.90</td>
<td>17.36</td>
<td>15957.43</td>
<td>14.43</td>
</tr>
<tr>
<td>BNN3</td>
<td>174.72</td>
<td>21.79</td>
<td>7019.42</td>
<td>117.91</td>
<td>7194.14</td>
<td>124.06</td>
</tr>
<tr>
<td>BNN4</td>
<td>315.16</td>
<td>5.59</td>
<td>15691.85</td>
<td>59.74</td>
<td>16007.01</td>
<td>64.55</td>
</tr>
<tr>
<td>BNN5</td>
<td>392.04</td>
<td>21.43</td>
<td>16023.65</td>
<td>127.49</td>
<td>16415.69</td>
<td>117.13</td>
</tr>
<tr>
<td>BNN6</td>
<td>115.15</td>
<td>40.32</td>
<td>6719.90</td>
<td>63.33</td>
<td>6835.05</td>
<td>91.98</td>
</tr>
<tr>
<td>BNN7</td>
<td>117.81</td>
<td>40.18</td>
<td>6802.76</td>
<td>167.90</td>
<td>6920.58</td>
<td>198.58</td>
</tr>
<tr>
<td>BNN8</td>
<td>112.15</td>
<td>48.01</td>
<td>6730.88</td>
<td>109.02</td>
<td>6843.03</td>
<td>127.13</td>
</tr>
<tr>
<td>BNN9</td>
<td>96.84</td>
<td>23.22</td>
<td>1597.77</td>
<td>11.61</td>
<td>1694.62</td>
<td>18.88</td>
</tr>
<tr>
<td>BNN10</td>
<td>287.21</td>
<td>9.75</td>
<td>15719.14</td>
<td>44.46</td>
<td>16006.35</td>
<td>35.68</td>
</tr>
<tr>
<td>BNN11</td>
<td>279.22</td>
<td>7.93</td>
<td>15714.15</td>
<td>56.02</td>
<td>15993.37</td>
<td>49.00</td>
</tr>
<tr>
<td>BNN12</td>
<td>278.55</td>
<td>8.03</td>
<td>15707.49</td>
<td>54.81</td>
<td>15986.05</td>
<td>46.96</td>
</tr>
<tr>
<td>BNN13</td>
<td>220.98</td>
<td>16.82</td>
<td>7029.40</td>
<td>250.89</td>
<td>7250.38</td>
<td>259.38</td>
</tr>
<tr>
<td>BNN14</td>
<td>227.97</td>
<td>6.66</td>
<td>7039.05</td>
<td>51.08</td>
<td>7267.02</td>
<td>52.76</td>
</tr>
</tbody>
</table>
### Table D.5: FPGA Execution Latencies of the BNNs (L9)

<table>
<thead>
<tr>
<th>Design</th>
<th>$t_{exec,init,x}$ [ns]</th>
<th>$\sigma$ [ns]</th>
<th>$t_{exec,calculate}$ [ns]</th>
<th>$\sigma$ [ns]</th>
<th>$t_{exec, get_y}$ [ns]</th>
<th>$\sigma$ [ns]</th>
<th>$t_{exec}$ [ns]</th>
<th>$\sigma$ [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNN0</td>
<td>2233.75</td>
<td>98.06</td>
<td>2261.71</td>
<td>58.30</td>
<td>2131.25</td>
<td>74.15</td>
<td>6626.71</td>
<td>159.99</td>
</tr>
<tr>
<td>BNN1</td>
<td>2810.50</td>
<td>847.85</td>
<td>2244.40</td>
<td>82.83</td>
<td>3718.71</td>
<td>71.49</td>
<td>8773.61</td>
<td>826.30</td>
</tr>
<tr>
<td>BNN2</td>
<td>0.00</td>
<td>0.00</td>
<td>2278.35</td>
<td>85.44</td>
<td>0.00</td>
<td>0.00</td>
<td>2278.35</td>
<td>85.44</td>
</tr>
<tr>
<td>BNN3</td>
<td>2525.29</td>
<td>718.59</td>
<td>2208.79</td>
<td>95.36</td>
<td>3759.64</td>
<td>77.73</td>
<td>8493.72</td>
<td>716.36</td>
</tr>
<tr>
<td>BNN4</td>
<td>0.00</td>
<td>0.00</td>
<td>2318.28</td>
<td>99.13</td>
<td>0.00</td>
<td>0.00</td>
<td>2318.28</td>
<td>99.13</td>
</tr>
<tr>
<td>BNN5</td>
<td>0.00</td>
<td>0.00</td>
<td>2292.99</td>
<td>93.22</td>
<td>0.00</td>
<td>0.00</td>
<td>2292.99</td>
<td>93.22</td>
</tr>
<tr>
<td>BNN6</td>
<td>0.00</td>
<td>0.00</td>
<td>3739.01</td>
<td>66.95</td>
<td>0.00</td>
<td>0.00</td>
<td>3739.01</td>
<td>66.95</td>
</tr>
<tr>
<td>BNN7</td>
<td>0.00</td>
<td>0.00</td>
<td>3742.00</td>
<td>93.17</td>
<td>0.00</td>
<td>0.00</td>
<td>3742.00</td>
<td>93.17</td>
</tr>
<tr>
<td>BNN8</td>
<td>0.00</td>
<td>0.00</td>
<td>3738.68</td>
<td>108.60</td>
<td>0.00</td>
<td>0.00</td>
<td>3738.68</td>
<td>108.60</td>
</tr>
<tr>
<td>BNN9</td>
<td>0.00</td>
<td>0.00</td>
<td>3758.98</td>
<td>98.60</td>
<td>0.00</td>
<td>0.00</td>
<td>3758.98</td>
<td>98.60</td>
</tr>
<tr>
<td>BNN10</td>
<td>0.00</td>
<td>0.00</td>
<td>2289.66</td>
<td>22.75</td>
<td>0.00</td>
<td>0.00</td>
<td>2289.66</td>
<td>22.75</td>
</tr>
<tr>
<td>BNN11</td>
<td>0.00</td>
<td>0.00</td>
<td>2214.12</td>
<td>100.05</td>
<td>0.00</td>
<td>0.00</td>
<td>2214.12</td>
<td>100.05</td>
</tr>
<tr>
<td>BNN12</td>
<td>0.00</td>
<td>0.00</td>
<td>2280.01</td>
<td>12.53</td>
<td>0.00</td>
<td>0.00</td>
<td>2280.01</td>
<td>12.53</td>
</tr>
<tr>
<td>BNN13</td>
<td>0.00</td>
<td>0.00</td>
<td>3798.91</td>
<td>83.89</td>
<td>0.00</td>
<td>0.00</td>
<td>3798.91</td>
<td>83.89</td>
</tr>
<tr>
<td>BNN14</td>
<td>0.00</td>
<td>0.00</td>
<td>3769.96</td>
<td>113.97</td>
<td>0.00</td>
<td>0.00</td>
<td>3769.96</td>
<td>113.97</td>
</tr>
<tr>
<td>Design</td>
<td>$t_{t_{\text{exec,calculate}}}^i$ [ns]</td>
<td>$t_{t_{\text{exec,calculate}}}^i$ [ns]</td>
<td>$t_{err}$ [ns]</td>
<td>$t_{err}$ [%]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>---------------------</td>
<td>---------------------</td>
<td>-------------</td>
<td>-----------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNN0</td>
<td>60</td>
<td>50</td>
<td>-10</td>
<td>-16,67</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNN1</td>
<td>50</td>
<td>50</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNN2</td>
<td>60</td>
<td>50</td>
<td>-10</td>
<td>-16,67</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNN3</td>
<td>50</td>
<td>50</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNN4</td>
<td>50</td>
<td>50</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNN5</td>
<td>50</td>
<td>50</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNN6</td>
<td>510</td>
<td>720</td>
<td>210</td>
<td>41,18</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNN7</td>
<td>1300</td>
<td>1310</td>
<td>10</td>
<td>0,77</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNN8</td>
<td>1305</td>
<td>1310</td>
<td>5</td>
<td>0,38</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNN9</td>
<td>570</td>
<td>510</td>
<td>-60</td>
<td>-10,53</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNN10</td>
<td>50</td>
<td>50</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNN11</td>
<td>50</td>
<td>50</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNN12</td>
<td>50</td>
<td>50</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNN13</td>
<td>870</td>
<td>1140</td>
<td>270</td>
<td>31,03</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNN14</td>
<td>1920</td>
<td>1970</td>
<td>50</td>
<td>2,6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Tab. D.7: FPGA Implementation Characteristics Component BNN0

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>$A$ [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{max}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>94</td>
<td>162816</td>
<td>512</td>
<td>1048</td>
<td>2</td>
<td>145</td>
</tr>
<tr>
<td>L1</td>
<td>94</td>
<td>162507</td>
<td>476</td>
<td>1018</td>
<td>2</td>
<td>145</td>
</tr>
<tr>
<td>L2</td>
<td>94</td>
<td>162508</td>
<td>457</td>
<td>1042</td>
<td>2</td>
<td>138</td>
</tr>
<tr>
<td>L3</td>
<td>93</td>
<td>162274</td>
<td>457</td>
<td>1016</td>
<td>2</td>
<td>138</td>
</tr>
<tr>
<td>L4</td>
<td>93</td>
<td>162262</td>
<td>457</td>
<td>1014</td>
<td>2</td>
<td>138</td>
</tr>
<tr>
<td>L5</td>
<td>93</td>
<td>162391</td>
<td>457</td>
<td>1026</td>
<td>2</td>
<td>138</td>
</tr>
<tr>
<td>L6</td>
<td>93</td>
<td>162274</td>
<td>457</td>
<td>1016</td>
<td>2</td>
<td>138</td>
</tr>
<tr>
<td>L7</td>
<td>93</td>
<td>162823</td>
<td>499</td>
<td>1043</td>
<td>2</td>
<td>138</td>
</tr>
<tr>
<td>L8</td>
<td>88</td>
<td>162351</td>
<td>455</td>
<td>1022</td>
<td>2</td>
<td>138</td>
</tr>
<tr>
<td>L9</td>
<td>51</td>
<td>159265</td>
<td>307</td>
<td>706</td>
<td>2</td>
<td>147</td>
</tr>
</tbody>
</table>

### Tab. D.8: FPGA Implementation Characteristics Class Bnn BNN0

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>$A$ [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{max}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>94</td>
<td>2460</td>
<td>141</td>
<td>155</td>
<td>0</td>
<td>238</td>
</tr>
<tr>
<td>L1</td>
<td>94</td>
<td>2046</td>
<td>84</td>
<td>116</td>
<td>0</td>
<td>238</td>
</tr>
<tr>
<td>L2</td>
<td>94</td>
<td>2046</td>
<td>84</td>
<td>116</td>
<td>0</td>
<td>238</td>
</tr>
<tr>
<td>L3</td>
<td>93</td>
<td>2031</td>
<td>84</td>
<td>113</td>
<td>0</td>
<td>238</td>
</tr>
<tr>
<td>L4</td>
<td>93</td>
<td>2031</td>
<td>86</td>
<td>113</td>
<td>0</td>
<td>238</td>
</tr>
<tr>
<td>L5</td>
<td>93</td>
<td>2031</td>
<td>85</td>
<td>113</td>
<td>0</td>
<td>238</td>
</tr>
<tr>
<td>L6</td>
<td>93</td>
<td>2031</td>
<td>86</td>
<td>113</td>
<td>0</td>
<td>238</td>
</tr>
<tr>
<td>L7</td>
<td>93</td>
<td>2467</td>
<td>132</td>
<td>127</td>
<td>0</td>
<td>238</td>
</tr>
<tr>
<td>L8</td>
<td>88</td>
<td>1191</td>
<td>82</td>
<td>109</td>
<td>0</td>
<td>238</td>
</tr>
<tr>
<td>L9</td>
<td>51</td>
<td>1441</td>
<td>60</td>
<td>83</td>
<td>0</td>
<td>238</td>
</tr>
</tbody>
</table>

### Tab. D.9: FPGA Implementation Characteristics Bnn::calculate(...) BNN0

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>$A$ [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{max}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>8</td>
<td>452</td>
<td>26</td>
<td>22</td>
<td>0</td>
<td>468</td>
</tr>
<tr>
<td>L1</td>
<td>8</td>
<td>302</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L2</td>
<td>8</td>
<td>302</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L3</td>
<td>8</td>
<td>302</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L4</td>
<td>8</td>
<td>302</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L5</td>
<td>8</td>
<td>302</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L6</td>
<td>8</td>
<td>302</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L7</td>
<td>8</td>
<td>452</td>
<td>26</td>
<td>22</td>
<td>0</td>
<td>472</td>
</tr>
<tr>
<td>L8</td>
<td>6</td>
<td>290</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>354</td>
</tr>
<tr>
<td>L9</td>
<td>6</td>
<td>290</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>354</td>
</tr>
</tbody>
</table>
Tab. D.10: FPGA Implementation Characteristics Component BNN1

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>$A$ [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{max}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>102</td>
<td>33911</td>
<td>682</td>
<td>1130</td>
<td>0</td>
<td>163</td>
</tr>
<tr>
<td>L1</td>
<td>102</td>
<td>33511</td>
<td>638</td>
<td>1121</td>
<td>0</td>
<td>163</td>
</tr>
<tr>
<td>L2</td>
<td>102</td>
<td>34409</td>
<td>700</td>
<td>1188</td>
<td>0</td>
<td>163</td>
</tr>
<tr>
<td>L3</td>
<td>102</td>
<td>34409</td>
<td>700</td>
<td>1188</td>
<td>0</td>
<td>163</td>
</tr>
<tr>
<td>L4</td>
<td>102</td>
<td>34409</td>
<td>700</td>
<td>1188</td>
<td>0</td>
<td>163</td>
</tr>
<tr>
<td>L5</td>
<td>102</td>
<td>34347</td>
<td>699</td>
<td>1179</td>
<td>0</td>
<td>163</td>
</tr>
<tr>
<td>L6</td>
<td>102</td>
<td>34409</td>
<td>700</td>
<td>1188</td>
<td>0</td>
<td>163</td>
</tr>
<tr>
<td>L7</td>
<td>102</td>
<td>34809</td>
<td>744</td>
<td>1197</td>
<td>0</td>
<td>163</td>
</tr>
<tr>
<td>L8</td>
<td>97</td>
<td>34319</td>
<td>697</td>
<td>1177</td>
<td>0</td>
<td>163</td>
</tr>
<tr>
<td>L9</td>
<td>60</td>
<td>31513</td>
<td>551</td>
<td>905</td>
<td>0</td>
<td>171</td>
</tr>
</tbody>
</table>

Tab. D.11: FPGA Implementation Characteristics Class Bnn BNN1

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>$A$ [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{max}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>102</td>
<td>5186</td>
<td>262</td>
<td>262</td>
<td>0</td>
<td>171</td>
</tr>
<tr>
<td>L1</td>
<td>102</td>
<td>4726</td>
<td>212</td>
<td>248</td>
<td>0</td>
<td>171</td>
</tr>
<tr>
<td>L2</td>
<td>102</td>
<td>4696</td>
<td>212</td>
<td>243</td>
<td>0</td>
<td>171</td>
</tr>
<tr>
<td>L3</td>
<td>102</td>
<td>4726</td>
<td>212</td>
<td>248</td>
<td>0</td>
<td>171</td>
</tr>
<tr>
<td>L4</td>
<td>102</td>
<td>4726</td>
<td>212</td>
<td>248</td>
<td>0</td>
<td>171</td>
</tr>
<tr>
<td>L5</td>
<td>102</td>
<td>4726</td>
<td>212</td>
<td>248</td>
<td>0</td>
<td>171</td>
</tr>
<tr>
<td>L6</td>
<td>102</td>
<td>4726</td>
<td>212</td>
<td>248</td>
<td>0</td>
<td>171</td>
</tr>
<tr>
<td>L7</td>
<td>102</td>
<td>4686</td>
<td>210</td>
<td>244</td>
<td>0</td>
<td>171</td>
</tr>
<tr>
<td>L8</td>
<td>60</td>
<td>4136</td>
<td>178</td>
<td>218</td>
<td>0</td>
<td>171</td>
</tr>
</tbody>
</table>

Tab. D.12: FPGA Implementation Characteristics Bnn::calculate(...) BNN1

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>$A$ [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{max}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>8</td>
<td>452</td>
<td>26</td>
<td>22</td>
<td>0</td>
<td>468</td>
</tr>
<tr>
<td>L1</td>
<td>8</td>
<td>302</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L2</td>
<td>8</td>
<td>302</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L3</td>
<td>8</td>
<td>302</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L4</td>
<td>8</td>
<td>302</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L5</td>
<td>8</td>
<td>302</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L6</td>
<td>8</td>
<td>302</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L7</td>
<td>8</td>
<td>452</td>
<td>26</td>
<td>22</td>
<td>0</td>
<td>472</td>
</tr>
<tr>
<td>L8</td>
<td>6</td>
<td>290</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>354</td>
</tr>
<tr>
<td>L9</td>
<td>6</td>
<td>290</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>354</td>
</tr>
</tbody>
</table>
### Tab. D.13: FPGA Implementation Characteristics Component BNN2

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>A [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{max}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>52</td>
<td>29200</td>
<td>413</td>
<td>776</td>
<td>0</td>
<td>207</td>
</tr>
<tr>
<td>L1</td>
<td>52</td>
<td>28692</td>
<td>369</td>
<td>750</td>
<td>0</td>
<td>208</td>
</tr>
<tr>
<td>L2</td>
<td>52</td>
<td>28692</td>
<td>369</td>
<td>750</td>
<td>0</td>
<td>207</td>
</tr>
<tr>
<td>L3</td>
<td>52</td>
<td>28692</td>
<td>369</td>
<td>750</td>
<td>0</td>
<td>207</td>
</tr>
<tr>
<td>L4</td>
<td>52</td>
<td>28692</td>
<td>369</td>
<td>750</td>
<td>0</td>
<td>208</td>
</tr>
<tr>
<td>L5</td>
<td>52</td>
<td>28692</td>
<td>369</td>
<td>750</td>
<td>0</td>
<td>207</td>
</tr>
<tr>
<td>L6</td>
<td>52</td>
<td>28692</td>
<td>369</td>
<td>750</td>
<td>0</td>
<td>208</td>
</tr>
<tr>
<td>L7</td>
<td>52</td>
<td>29200</td>
<td>413</td>
<td>776</td>
<td>0</td>
<td>207</td>
</tr>
<tr>
<td>L8</td>
<td>47</td>
<td>28664</td>
<td>367</td>
<td>748</td>
<td>0</td>
<td>209</td>
</tr>
<tr>
<td>L9</td>
<td>10</td>
<td>23892</td>
<td>217</td>
<td>486</td>
<td>0</td>
<td>231</td>
</tr>
</tbody>
</table>

### Tab. D.14: FPGA Implementation Characteristics Class Bnn BNN2

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>A [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{max}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>52</td>
<td>1449</td>
<td>76</td>
<td>77</td>
<td>0</td>
<td>408</td>
</tr>
<tr>
<td>L1</td>
<td>52</td>
<td>1019</td>
<td>32</td>
<td>64</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L2</td>
<td>52</td>
<td>1019</td>
<td>32</td>
<td>64</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L3</td>
<td>52</td>
<td>1019</td>
<td>32</td>
<td>64</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L4</td>
<td>52</td>
<td>1019</td>
<td>32</td>
<td>64</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L5</td>
<td>52</td>
<td>1019</td>
<td>32</td>
<td>64</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L6</td>
<td>52</td>
<td>1019</td>
<td>32</td>
<td>64</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L7</td>
<td>52</td>
<td>1449</td>
<td>76</td>
<td>77</td>
<td>0</td>
<td>408</td>
</tr>
<tr>
<td>L8</td>
<td>47</td>
<td>979</td>
<td>30</td>
<td>60</td>
<td>0</td>
<td>354</td>
</tr>
<tr>
<td>L9</td>
<td>10</td>
<td>315</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>354</td>
</tr>
</tbody>
</table>

### Tab. D.15: FPGA Implementation Characteristics Bnn::calculate(...) BNN2

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>A [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{max}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>8</td>
<td>461</td>
<td>26</td>
<td>22</td>
<td>0</td>
<td>472</td>
</tr>
<tr>
<td>L1</td>
<td>8</td>
<td>311</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L2</td>
<td>8</td>
<td>311</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L3</td>
<td>8</td>
<td>311</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L4</td>
<td>8</td>
<td>311</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L5</td>
<td>8</td>
<td>311</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>249</td>
</tr>
<tr>
<td>L6</td>
<td>8</td>
<td>311</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L7</td>
<td>8</td>
<td>461</td>
<td>26</td>
<td>22</td>
<td>0</td>
<td>472</td>
</tr>
<tr>
<td>L8</td>
<td>6</td>
<td>299</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>354</td>
</tr>
<tr>
<td>L9</td>
<td>6</td>
<td>299</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>354</td>
</tr>
</tbody>
</table>
### Tab. D.16: FPGA Implementation Characteristics Component BNN3

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>$A$ [GE]</th>
<th>$#$FF</th>
<th>$#$LUT</th>
<th>$#$BRAM</th>
<th>$f_{\text{max}}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>83</td>
<td>163655</td>
<td>543</td>
<td>1089</td>
<td>2</td>
<td>144</td>
</tr>
<tr>
<td>L1</td>
<td>83</td>
<td>163289</td>
<td>501</td>
<td>1079</td>
<td>2</td>
<td>144</td>
</tr>
<tr>
<td>L2</td>
<td>83</td>
<td>163285</td>
<td>502</td>
<td>1077</td>
<td>2</td>
<td>144</td>
</tr>
<tr>
<td>L3</td>
<td>82</td>
<td>163162</td>
<td>502</td>
<td>1060</td>
<td>2</td>
<td>144</td>
</tr>
<tr>
<td>L4</td>
<td>82</td>
<td>163174</td>
<td>502</td>
<td>1062</td>
<td>2</td>
<td>144</td>
</tr>
<tr>
<td>L5</td>
<td>82</td>
<td>163148</td>
<td>501</td>
<td>1059</td>
<td>2</td>
<td>144</td>
</tr>
<tr>
<td>L6</td>
<td>82</td>
<td>163180</td>
<td>502</td>
<td>1063</td>
<td>2</td>
<td>144</td>
</tr>
<tr>
<td>L7</td>
<td>82</td>
<td>163624</td>
<td>544</td>
<td>1082</td>
<td>2</td>
<td>144</td>
</tr>
<tr>
<td>L8</td>
<td>77</td>
<td>163333</td>
<td>499</td>
<td>1100</td>
<td>2</td>
<td>145</td>
</tr>
<tr>
<td>L9</td>
<td>30</td>
<td>160306</td>
<td>352</td>
<td>785</td>
<td>2</td>
<td>147</td>
</tr>
</tbody>
</table>

### Tab. D.17: FPGA Implementation Characteristics Class Bnn BNN3

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>$A$ [GE]</th>
<th>$#$FF</th>
<th>$#$LUT</th>
<th>$#$BRAM</th>
<th>$f_{\text{max}}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>83</td>
<td>3263</td>
<td>141</td>
<td>169</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L1</td>
<td>83</td>
<td>2827</td>
<td>94</td>
<td>155</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L2</td>
<td>83</td>
<td>2827</td>
<td>94</td>
<td>155</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L3</td>
<td>82</td>
<td>2812</td>
<td>94</td>
<td>152</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L4</td>
<td>82</td>
<td>2812</td>
<td>94</td>
<td>152</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L5</td>
<td>82</td>
<td>2812</td>
<td>94</td>
<td>152</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L6</td>
<td>82</td>
<td>2812</td>
<td>94</td>
<td>152</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L7</td>
<td>82</td>
<td>2812</td>
<td>94</td>
<td>152</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L8</td>
<td>77</td>
<td>2772</td>
<td>92</td>
<td>148</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L9</td>
<td>30</td>
<td>2222</td>
<td>69</td>
<td>122</td>
<td>0</td>
<td>170</td>
</tr>
</tbody>
</table>

### Tab. D.18: FPGA Implementation Characteristics Bnn::calculate(...) BNN3

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>$A$ [GE]</th>
<th>$#$FF</th>
<th>$#$LUT</th>
<th>$#$BRAM</th>
<th>$f_{\text{max}}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>8</td>
<td>452</td>
<td>26</td>
<td>22</td>
<td>0</td>
<td>472</td>
</tr>
<tr>
<td>L1</td>
<td>8</td>
<td>302</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L2</td>
<td>8</td>
<td>302</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L3</td>
<td>8</td>
<td>302</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L4</td>
<td>8</td>
<td>302</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L5</td>
<td>8</td>
<td>302</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L6</td>
<td>8</td>
<td>302</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L7</td>
<td>8</td>
<td>452</td>
<td>26</td>
<td>22</td>
<td>0</td>
<td>472</td>
</tr>
<tr>
<td>L8</td>
<td>6</td>
<td>290</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>354</td>
</tr>
<tr>
<td>L9</td>
<td>6</td>
<td>290</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>354</td>
</tr>
</tbody>
</table>
### Tab. D.19: FPGA Implementation Characteristics Component BNN4

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>$A$ [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{\text{max}}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>12</td>
<td>25690</td>
<td>215</td>
<td>455</td>
<td>0</td>
<td>213</td>
</tr>
<tr>
<td>L1</td>
<td>12</td>
<td>25452</td>
<td>195</td>
<td>442</td>
<td>0</td>
<td>213</td>
</tr>
<tr>
<td>L2</td>
<td>12</td>
<td>25452</td>
<td>195</td>
<td>442</td>
<td>0</td>
<td>213</td>
</tr>
<tr>
<td>L3</td>
<td>12</td>
<td>25452</td>
<td>195</td>
<td>442</td>
<td>0</td>
<td>213</td>
</tr>
<tr>
<td>L4</td>
<td>12</td>
<td>25446</td>
<td>195</td>
<td>441</td>
<td>0</td>
<td>213</td>
</tr>
<tr>
<td>L5</td>
<td>12</td>
<td>25452</td>
<td>195</td>
<td>442</td>
<td>0</td>
<td>213</td>
</tr>
<tr>
<td>L6</td>
<td>12</td>
<td>25452</td>
<td>195</td>
<td>442</td>
<td>0</td>
<td>213</td>
</tr>
<tr>
<td>L7</td>
<td>12</td>
<td>25684</td>
<td>215</td>
<td>454</td>
<td>0</td>
<td>213</td>
</tr>
<tr>
<td>L8</td>
<td>10</td>
<td>25548</td>
<td>195</td>
<td>458</td>
<td>0</td>
<td>231</td>
</tr>
<tr>
<td>L9</td>
<td>10</td>
<td>25548</td>
<td>195</td>
<td>458</td>
<td>0</td>
<td>231</td>
</tr>
</tbody>
</table>

### Tab. D.20: FPGA Implementation Characteristics Class Bnn BNN4

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>$A$ [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{\text{max}}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>12</td>
<td>477</td>
<td>26</td>
<td>22</td>
<td>0</td>
<td>472</td>
</tr>
<tr>
<td>L1</td>
<td>12</td>
<td>327</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L2</td>
<td>12</td>
<td>327</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L3</td>
<td>12</td>
<td>327</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L4</td>
<td>12</td>
<td>327</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L5</td>
<td>12</td>
<td>327</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L6</td>
<td>12</td>
<td>477</td>
<td>26</td>
<td>22</td>
<td>0</td>
<td>472</td>
</tr>
<tr>
<td>L7</td>
<td>10</td>
<td>315</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>354</td>
</tr>
<tr>
<td>L8</td>
<td>10</td>
<td>315</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>354</td>
</tr>
<tr>
<td>L9</td>
<td>10</td>
<td>315</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>354</td>
</tr>
</tbody>
</table>

### Tab. D.21: FPGA Implementation Characteristics Bnn::calculate(...) BNN4

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>$A$ [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{\text{max}}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>8</td>
<td>461</td>
<td>26</td>
<td>22</td>
<td>0</td>
<td>472</td>
</tr>
<tr>
<td>L1</td>
<td>8</td>
<td>311</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L2</td>
<td>8</td>
<td>311</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L3</td>
<td>8</td>
<td>311</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L4</td>
<td>8</td>
<td>311</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L5</td>
<td>8</td>
<td>311</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L6</td>
<td>8</td>
<td>311</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L7</td>
<td>8</td>
<td>461</td>
<td>26</td>
<td>22</td>
<td>0</td>
<td>472</td>
</tr>
<tr>
<td>L8</td>
<td>6</td>
<td>299</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>354</td>
</tr>
<tr>
<td>L9</td>
<td>6</td>
<td>299</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>354</td>
</tr>
</tbody>
</table>
Tab. D.22: FPGA Implementation Characteristics Component BNN5

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>(A) [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>(f_{\text{max}}) [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>12</td>
<td>25686</td>
<td>216</td>
<td>453</td>
<td>0</td>
<td>213</td>
</tr>
<tr>
<td>L1</td>
<td>12</td>
<td>25446</td>
<td>195</td>
<td>441</td>
<td>0</td>
<td>213</td>
</tr>
<tr>
<td>L2</td>
<td>12</td>
<td>25446</td>
<td>195</td>
<td>441</td>
<td>0</td>
<td>213</td>
</tr>
<tr>
<td>L3</td>
<td>12</td>
<td>25446</td>
<td>195</td>
<td>441</td>
<td>0</td>
<td>213</td>
</tr>
<tr>
<td>L4</td>
<td>12</td>
<td>25446</td>
<td>195</td>
<td>441</td>
<td>0</td>
<td>213</td>
</tr>
<tr>
<td>L5</td>
<td>12</td>
<td>25446</td>
<td>195</td>
<td>441</td>
<td>0</td>
<td>213</td>
</tr>
<tr>
<td>L6</td>
<td>12</td>
<td>25446</td>
<td>195</td>
<td>441</td>
<td>0</td>
<td>213</td>
</tr>
<tr>
<td>L7</td>
<td>12</td>
<td>25698</td>
<td>216</td>
<td>455</td>
<td>0</td>
<td>213</td>
</tr>
<tr>
<td>L8</td>
<td>10</td>
<td>25554</td>
<td>195</td>
<td>459</td>
<td>0</td>
<td>231</td>
</tr>
<tr>
<td>L9</td>
<td>10</td>
<td>25554</td>
<td>195</td>
<td>459</td>
<td>0</td>
<td>231</td>
</tr>
</tbody>
</table>

Tab. D.23: FPGA Implementation Characteristics Class Bnn BNN5

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>(A) [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>(f_{\text{max}}) [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>12</td>
<td>463</td>
<td>25</td>
<td>21</td>
<td>0</td>
<td>472</td>
</tr>
<tr>
<td>L1</td>
<td>12</td>
<td>327</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L2</td>
<td>12</td>
<td>327</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L3</td>
<td>12</td>
<td>327</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L4</td>
<td>12</td>
<td>327</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L5</td>
<td>12</td>
<td>327</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L6</td>
<td>12</td>
<td>327</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L7</td>
<td>12</td>
<td>463</td>
<td>25</td>
<td>21</td>
<td>0</td>
<td>472</td>
</tr>
<tr>
<td>L8</td>
<td>10</td>
<td>315</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>354</td>
</tr>
<tr>
<td>L9</td>
<td>10</td>
<td>315</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>354</td>
</tr>
</tbody>
</table>

Tab. D.24: FPGA Implementation Characteristics Bnn::calculate(...) BNN5

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>(A) [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>(f_{\text{max}}) [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>8</td>
<td>447</td>
<td>25</td>
<td>21</td>
<td>0</td>
<td>472</td>
</tr>
<tr>
<td>L1</td>
<td>8</td>
<td>311</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L2</td>
<td>8</td>
<td>311</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L3</td>
<td>8</td>
<td>311</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L4</td>
<td>8</td>
<td>311</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L5</td>
<td>8</td>
<td>311</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L6</td>
<td>8</td>
<td>311</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>349</td>
</tr>
<tr>
<td>L7</td>
<td>8</td>
<td>447</td>
<td>25</td>
<td>21</td>
<td>0</td>
<td>472</td>
</tr>
<tr>
<td>L8</td>
<td>6</td>
<td>299</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>354</td>
</tr>
<tr>
<td>L9</td>
<td>6</td>
<td>299</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>354</td>
</tr>
</tbody>
</table>
### Tab. D.25: FPGA Implementation Characteristics Component BNN6

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>A [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{max}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>409</td>
<td>173310</td>
<td>1385</td>
<td>1609</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L1</td>
<td>463</td>
<td>171718</td>
<td>1237</td>
<td>1544</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L2</td>
<td>132</td>
<td>165012</td>
<td>674</td>
<td>1174</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L3</td>
<td>132</td>
<td>165122</td>
<td>684</td>
<td>1179</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L4</td>
<td>132</td>
<td>164958</td>
<td>674</td>
<td>1168</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L5</td>
<td>132</td>
<td>164958</td>
<td>674</td>
<td>1168</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L6</td>
<td>132</td>
<td>165048</td>
<td>677</td>
<td>1176</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L7</td>
<td>132</td>
<td>166660</td>
<td>832</td>
<td>1241</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L8</td>
<td>115</td>
<td>164900</td>
<td>666</td>
<td>1166</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L9</td>
<td>50</td>
<td>156036</td>
<td>161</td>
<td>365</td>
<td>2</td>
<td>146</td>
</tr>
</tbody>
</table>

### Tab. D.26: FPGA Implementation Characteristics Class Bnn BNN6

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>A [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{max}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>409</td>
<td>10487</td>
<td>837</td>
<td>544</td>
<td>0</td>
<td>173</td>
</tr>
<tr>
<td>L1</td>
<td>463</td>
<td>8977</td>
<td>689</td>
<td>507</td>
<td>0</td>
<td>180</td>
</tr>
<tr>
<td>L2</td>
<td>132</td>
<td>2211</td>
<td>126</td>
<td>130</td>
<td>0</td>
<td>231</td>
</tr>
<tr>
<td>L3</td>
<td>132</td>
<td>2321</td>
<td>136</td>
<td>135</td>
<td>0</td>
<td>233</td>
</tr>
<tr>
<td>L4</td>
<td>132</td>
<td>2211</td>
<td>126</td>
<td>130</td>
<td>0</td>
<td>231</td>
</tr>
<tr>
<td>L5</td>
<td>132</td>
<td>2211</td>
<td>126</td>
<td>130</td>
<td>0</td>
<td>231</td>
</tr>
<tr>
<td>L6</td>
<td>132</td>
<td>2247</td>
<td>129</td>
<td>132</td>
<td>0</td>
<td>233</td>
</tr>
<tr>
<td>L7</td>
<td>132</td>
<td>3873</td>
<td>284</td>
<td>179</td>
<td>0</td>
<td>231</td>
</tr>
<tr>
<td>L8</td>
<td>115</td>
<td>2099</td>
<td>118</td>
<td>122</td>
<td>0</td>
<td>233</td>
</tr>
<tr>
<td>L9</td>
<td>50</td>
<td>1239</td>
<td>74</td>
<td>63</td>
<td>0</td>
<td>231</td>
</tr>
</tbody>
</table>

### Tab. D.27: FPGA Implementation Characteristics Bnn::calculate(...) BNN6

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>A [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{max}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>324</td>
<td>8003</td>
<td>641</td>
<td>437</td>
<td>0</td>
<td>173</td>
</tr>
<tr>
<td>L1</td>
<td>378</td>
<td>8005</td>
<td>639</td>
<td>440</td>
<td>0</td>
<td>180</td>
</tr>
<tr>
<td>L2</td>
<td>47</td>
<td>1239</td>
<td>76</td>
<td>63</td>
<td>0</td>
<td>231</td>
</tr>
<tr>
<td>L3</td>
<td>47</td>
<td>1349</td>
<td>86</td>
<td>68</td>
<td>0</td>
<td>233</td>
</tr>
<tr>
<td>L4</td>
<td>47</td>
<td>1239</td>
<td>76</td>
<td>63</td>
<td>0</td>
<td>231</td>
</tr>
<tr>
<td>L5</td>
<td>47</td>
<td>1239</td>
<td>76</td>
<td>63</td>
<td>0</td>
<td>231</td>
</tr>
<tr>
<td>L6</td>
<td>47</td>
<td>1275</td>
<td>79</td>
<td>65</td>
<td>0</td>
<td>233</td>
</tr>
<tr>
<td>L7</td>
<td>47</td>
<td>1389</td>
<td>88</td>
<td>72</td>
<td>0</td>
<td>231</td>
</tr>
<tr>
<td>L8</td>
<td>47</td>
<td>1259</td>
<td>77</td>
<td>65</td>
<td>0</td>
<td>233</td>
</tr>
<tr>
<td>L9</td>
<td>46</td>
<td>1223</td>
<td>74</td>
<td>63</td>
<td>0</td>
<td>231</td>
</tr>
</tbody>
</table>
### Tab. D.28: FPGA Implementation Characteristics Component BNN7

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>$A$ [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{\text{max}}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>158</td>
<td>168813</td>
<td>992</td>
<td>1346</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L1</td>
<td>158</td>
<td>166123</td>
<td>748</td>
<td>1223</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L2</td>
<td>158</td>
<td>166123</td>
<td>748</td>
<td>1223</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L3</td>
<td>158</td>
<td>166123</td>
<td>748</td>
<td>1223</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L4</td>
<td>121</td>
<td>165945</td>
<td>722</td>
<td>1228</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L5</td>
<td>121</td>
<td>165945</td>
<td>722</td>
<td>1228</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L6</td>
<td>121</td>
<td>165949</td>
<td>721</td>
<td>1231</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L7</td>
<td>121</td>
<td>167833</td>
<td>880</td>
<td>1332</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L8</td>
<td>102</td>
<td>165747</td>
<td>710</td>
<td>1215</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L9</td>
<td>38</td>
<td>157247</td>
<td>213</td>
<td>457</td>
<td>2</td>
<td>146</td>
</tr>
</tbody>
</table>

### Tab. D.29: FPGA Implementation Characteristics Class Bnn BNN7

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>$A$ [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{\text{max}}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>158</td>
<td>5880</td>
<td>408</td>
<td>269</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L1</td>
<td>158</td>
<td>3232</td>
<td>165</td>
<td>169</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L2</td>
<td>158</td>
<td>3232</td>
<td>165</td>
<td>169</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L3</td>
<td>158</td>
<td>3238</td>
<td>165</td>
<td>170</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L4</td>
<td>121</td>
<td>3068</td>
<td>140</td>
<td>175</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L5</td>
<td>121</td>
<td>3074</td>
<td>140</td>
<td>176</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L6</td>
<td>121</td>
<td>3066</td>
<td>139</td>
<td>176</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L7</td>
<td>121</td>
<td>4740</td>
<td>297</td>
<td>227</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L8</td>
<td>102</td>
<td>2904</td>
<td>127</td>
<td>165</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L9</td>
<td>38</td>
<td>2086</td>
<td>86</td>
<td>109</td>
<td>0</td>
<td>170</td>
</tr>
</tbody>
</table>

### Tab. D.30: FPGA Implementation Characteristics Bnn::calculate(...) BNN7

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>$A$ [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{\text{max}}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>73</td>
<td>3408</td>
<td>212</td>
<td>614</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L1</td>
<td>73</td>
<td>2278</td>
<td>115</td>
<td>105</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L2</td>
<td>73</td>
<td>2278</td>
<td>115</td>
<td>105</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L3</td>
<td>73</td>
<td>2278</td>
<td>115</td>
<td>105</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L4</td>
<td>36</td>
<td>2108</td>
<td>90</td>
<td>110</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L5</td>
<td>36</td>
<td>2108</td>
<td>90</td>
<td>110</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L6</td>
<td>36</td>
<td>2100</td>
<td>89</td>
<td>110</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L7</td>
<td>36</td>
<td>2262</td>
<td>101</td>
<td>121</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L8</td>
<td>34</td>
<td>2070</td>
<td>86</td>
<td>109</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L9</td>
<td>34</td>
<td>2076</td>
<td>86</td>
<td>110</td>
<td>0</td>
<td>170</td>
</tr>
</tbody>
</table>
### Table D.31: FPGA Implementation Characteristics Component BNN8

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>A [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{max}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>432</td>
<td>174493</td>
<td>1465</td>
<td>1663</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L1</td>
<td>430</td>
<td>172125</td>
<td>1226</td>
<td>1587</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L2</td>
<td>244</td>
<td>167940</td>
<td>869</td>
<td>1365</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L3</td>
<td>239</td>
<td>167990</td>
<td>870</td>
<td>1372</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L4</td>
<td>121</td>
<td>165949</td>
<td>721</td>
<td>1231</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L5</td>
<td>121</td>
<td>166004</td>
<td>723</td>
<td>1236</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L6</td>
<td>121</td>
<td>165959</td>
<td>723</td>
<td>1230</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L7</td>
<td>121</td>
<td>167657</td>
<td>882</td>
<td>1301</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L8</td>
<td>102</td>
<td>165803</td>
<td>711</td>
<td>1220</td>
<td>2</td>
<td>137</td>
</tr>
<tr>
<td>L9</td>
<td>38</td>
<td>157243</td>
<td>214</td>
<td>456</td>
<td>2</td>
<td>146</td>
</tr>
</tbody>
</table>

### Table D.32: FPGA Implementation Characteristics Class Bnn BNN8

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>A [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{max}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>432</td>
<td>11520</td>
<td>882</td>
<td>577</td>
<td>0</td>
<td>171</td>
</tr>
<tr>
<td>L1</td>
<td>430</td>
<td>9018</td>
<td>643</td>
<td>496</td>
<td>0</td>
<td>171</td>
</tr>
<tr>
<td>L2</td>
<td>244</td>
<td>4913</td>
<td>284</td>
<td>290</td>
<td>0</td>
<td>171</td>
</tr>
<tr>
<td>L3</td>
<td>239</td>
<td>4955</td>
<td>284</td>
<td>297</td>
<td>0</td>
<td>171</td>
</tr>
<tr>
<td>L4</td>
<td>121</td>
<td>3060</td>
<td>139</td>
<td>175</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L5</td>
<td>121</td>
<td>3121</td>
<td>141</td>
<td>181</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L6</td>
<td>121</td>
<td>3060</td>
<td>139</td>
<td>175</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L7</td>
<td>121</td>
<td>4742</td>
<td>298</td>
<td>226</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L8</td>
<td>102</td>
<td>2912</td>
<td>128</td>
<td>165</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L9</td>
<td>38</td>
<td>2088</td>
<td>87</td>
<td>108</td>
<td>0</td>
<td>170</td>
</tr>
</tbody>
</table>

### Table D.33: FPGA Implementation Characteristics Bnn::calculate(...) BNN8

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>A [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{max}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>347</td>
<td>9054</td>
<td>686</td>
<td>473</td>
<td>0</td>
<td>171</td>
</tr>
<tr>
<td>L1</td>
<td>345</td>
<td>8058</td>
<td>593</td>
<td>431</td>
<td>0</td>
<td>171</td>
</tr>
<tr>
<td>L2</td>
<td>159</td>
<td>3953</td>
<td>234</td>
<td>225</td>
<td>0</td>
<td>171</td>
</tr>
<tr>
<td>L3</td>
<td>154</td>
<td>4007</td>
<td>234</td>
<td>234</td>
<td>0</td>
<td>171</td>
</tr>
<tr>
<td>L4</td>
<td>36</td>
<td>2100</td>
<td>89</td>
<td>110</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L5</td>
<td>36</td>
<td>2153</td>
<td>90</td>
<td>116</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L6</td>
<td>36</td>
<td>2094</td>
<td>89</td>
<td>109</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L7</td>
<td>36</td>
<td>2270</td>
<td>102</td>
<td>121</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L8</td>
<td>34</td>
<td>2078</td>
<td>87</td>
<td>109</td>
<td>0</td>
<td>170</td>
</tr>
<tr>
<td>L9</td>
<td>34</td>
<td>2078</td>
<td>87</td>
<td>109</td>
<td>0</td>
<td>170</td>
</tr>
</tbody>
</table>
### Tab. D.34: FPGA Implementation Characteristics Component BNN9

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>$A$ [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{max}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>162</td>
<td>37518</td>
<td>1014</td>
<td>1361</td>
<td>0</td>
<td>173</td>
</tr>
<tr>
<td>L1</td>
<td>162</td>
<td>35350</td>
<td>821</td>
<td>1257</td>
<td>0</td>
<td>174</td>
</tr>
<tr>
<td>L2</td>
<td>162</td>
<td>35484</td>
<td>825</td>
<td>1274</td>
<td>0</td>
<td>173</td>
</tr>
<tr>
<td>L3</td>
<td>162</td>
<td>35446</td>
<td>824</td>
<td>1269</td>
<td>0</td>
<td>174</td>
</tr>
<tr>
<td>L4</td>
<td>162</td>
<td>35448</td>
<td>825</td>
<td>1268</td>
<td>0</td>
<td>174</td>
</tr>
<tr>
<td>L5</td>
<td>162</td>
<td>35448</td>
<td>825</td>
<td>1268</td>
<td>0</td>
<td>174</td>
</tr>
<tr>
<td>L6</td>
<td>162</td>
<td>35244</td>
<td>825</td>
<td>1234</td>
<td>0</td>
<td>192</td>
</tr>
<tr>
<td>L7</td>
<td>162</td>
<td>37539</td>
<td>1008</td>
<td>1372</td>
<td>0</td>
<td>173</td>
</tr>
<tr>
<td>L8</td>
<td>130</td>
<td>35202</td>
<td>801</td>
<td>1259</td>
<td>0</td>
<td>174</td>
</tr>
<tr>
<td>L9</td>
<td>66</td>
<td>26362</td>
<td>296</td>
<td>459</td>
<td>0</td>
<td>195</td>
</tr>
</tbody>
</table>

### Tab. D.35: FPGA Implementation Characteristics Class Bnn BNN9

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>$A$ [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{max}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>162</td>
<td>4281</td>
<td>331</td>
<td>194</td>
<td>0</td>
<td>256</td>
</tr>
<tr>
<td>L1</td>
<td>162</td>
<td>2243</td>
<td>138</td>
<td>129</td>
<td>0</td>
<td>256</td>
</tr>
<tr>
<td>L2</td>
<td>162</td>
<td>2389</td>
<td>142</td>
<td>148</td>
<td>0</td>
<td>253</td>
</tr>
<tr>
<td>L3</td>
<td>162</td>
<td>2387</td>
<td>141</td>
<td>149</td>
<td>0</td>
<td>256</td>
</tr>
<tr>
<td>L4</td>
<td>162</td>
<td>2389</td>
<td>142</td>
<td>148</td>
<td>0</td>
<td>253</td>
</tr>
<tr>
<td>L5</td>
<td>162</td>
<td>2389</td>
<td>142</td>
<td>148</td>
<td>0</td>
<td>253</td>
</tr>
<tr>
<td>L6</td>
<td>162</td>
<td>2373</td>
<td>140</td>
<td>148</td>
<td>0</td>
<td>253</td>
</tr>
<tr>
<td>L7</td>
<td>162</td>
<td>4365</td>
<td>325</td>
<td>216</td>
<td>0</td>
<td>256</td>
</tr>
<tr>
<td>L8</td>
<td>130</td>
<td>2137</td>
<td>118</td>
<td>138</td>
<td>0</td>
<td>253</td>
</tr>
<tr>
<td>L9</td>
<td>66</td>
<td>1297</td>
<td>75</td>
<td>81</td>
<td>0</td>
<td>253</td>
</tr>
</tbody>
</table>

### Tab. D.36: FPGA Implementation Characteristics Bnn::calculate(...) BNN9

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>$A$ [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{max}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>77</td>
<td>1797</td>
<td>135</td>
<td>87</td>
<td>0</td>
<td>256</td>
</tr>
<tr>
<td>L1</td>
<td>77</td>
<td>1271</td>
<td>88</td>
<td>62</td>
<td>0</td>
<td>256</td>
</tr>
<tr>
<td>L2</td>
<td>77</td>
<td>1417</td>
<td>92</td>
<td>81</td>
<td>0</td>
<td>253</td>
</tr>
<tr>
<td>L3</td>
<td>77</td>
<td>1409</td>
<td>91</td>
<td>81</td>
<td>0</td>
<td>253</td>
</tr>
<tr>
<td>L4</td>
<td>77</td>
<td>1423</td>
<td>92</td>
<td>82</td>
<td>0</td>
<td>256</td>
</tr>
<tr>
<td>L5</td>
<td>77</td>
<td>1423</td>
<td>92</td>
<td>82</td>
<td>0</td>
<td>256</td>
</tr>
<tr>
<td>L6</td>
<td>77</td>
<td>1407</td>
<td>90</td>
<td>82</td>
<td>0</td>
<td>256</td>
</tr>
<tr>
<td>L7</td>
<td>77</td>
<td>1881</td>
<td>129</td>
<td>109</td>
<td>0</td>
<td>256</td>
</tr>
<tr>
<td>L8</td>
<td>62</td>
<td>1303</td>
<td>77</td>
<td>82</td>
<td>0</td>
<td>256</td>
</tr>
<tr>
<td>L9</td>
<td>62</td>
<td>1281</td>
<td>77</td>
<td>82</td>
<td>0</td>
<td>256</td>
</tr>
</tbody>
</table>
Tab. D.37: FPGA Implementation Characteristics Component BNN10

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>A [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{max}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>93</td>
<td>36004</td>
<td>869</td>
<td>1302</td>
<td>0</td>
<td>177</td>
</tr>
<tr>
<td>L1</td>
<td>93</td>
<td>33952</td>
<td>707</td>
<td>1176</td>
<td>0</td>
<td>190</td>
</tr>
<tr>
<td>L2</td>
<td>93</td>
<td>33952</td>
<td>707</td>
<td>1176</td>
<td>0</td>
<td>190</td>
</tr>
<tr>
<td>L3</td>
<td>93</td>
<td>33952</td>
<td>707</td>
<td>1176</td>
<td>0</td>
<td>190</td>
</tr>
<tr>
<td>L4</td>
<td>93</td>
<td>33952</td>
<td>707</td>
<td>1176</td>
<td>0</td>
<td>190</td>
</tr>
<tr>
<td>L5</td>
<td>94</td>
<td>33952</td>
<td>707</td>
<td>1176</td>
<td>0</td>
<td>190</td>
</tr>
<tr>
<td>L6</td>
<td>93</td>
<td>33952</td>
<td>707</td>
<td>1176</td>
<td>0</td>
<td>190</td>
</tr>
<tr>
<td>L7</td>
<td>93</td>
<td>36004</td>
<td>869</td>
<td>1302</td>
<td>0</td>
<td>177</td>
</tr>
<tr>
<td>L8</td>
<td>74</td>
<td>33802</td>
<td>689</td>
<td>1163</td>
<td>0</td>
<td>190</td>
</tr>
<tr>
<td>L9</td>
<td>10</td>
<td>25118</td>
<td>193</td>
<td>389</td>
<td>0</td>
<td>231</td>
</tr>
</tbody>
</table>

Tab. D.38: FPGA Implementation Characteristics Class Bnn BNN10

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>A [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{max}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>93</td>
<td>2924</td>
<td>222</td>
<td>129</td>
<td>0</td>
<td>414</td>
</tr>
<tr>
<td>L1</td>
<td>93</td>
<td>1218</td>
<td>54</td>
<td>82</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L2</td>
<td>93</td>
<td>1218</td>
<td>54</td>
<td>82</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L3</td>
<td>93</td>
<td>1218</td>
<td>54</td>
<td>82</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L4</td>
<td>93</td>
<td>1218</td>
<td>54</td>
<td>82</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L5</td>
<td>94</td>
<td>1218</td>
<td>54</td>
<td>82</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L6</td>
<td>93</td>
<td>1218</td>
<td>54</td>
<td>82</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L7</td>
<td>93</td>
<td>2924</td>
<td>222</td>
<td>129</td>
<td>0</td>
<td>414</td>
</tr>
<tr>
<td>L8</td>
<td>74</td>
<td>1074</td>
<td>45</td>
<td>70</td>
<td>0</td>
<td>363</td>
</tr>
<tr>
<td>L9</td>
<td>10</td>
<td>250</td>
<td>4</td>
<td>13</td>
<td>0</td>
<td>363</td>
</tr>
</tbody>
</table>

Tab. D.39: FPGA Implementation Characteristics Bnn::calculate(...) BNN10

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>A [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{max}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>8</td>
<td>440</td>
<td>26</td>
<td>22</td>
<td>0</td>
<td>472</td>
</tr>
<tr>
<td>L1</td>
<td>8</td>
<td>246</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L2</td>
<td>8</td>
<td>246</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L3</td>
<td>8</td>
<td>246</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L4</td>
<td>8</td>
<td>246</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L5</td>
<td>9</td>
<td>246</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L6</td>
<td>8</td>
<td>246</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L7</td>
<td>8</td>
<td>440</td>
<td>26</td>
<td>22</td>
<td>0</td>
<td>472</td>
</tr>
<tr>
<td>L8</td>
<td>6</td>
<td>234</td>
<td>4</td>
<td>13</td>
<td>0</td>
<td>363</td>
</tr>
<tr>
<td>L9</td>
<td>6</td>
<td>234</td>
<td>4</td>
<td>13</td>
<td>0</td>
<td>363</td>
</tr>
</tbody>
</table>
### Tab. D.40: FPGA Implementation Characteristics Component BNN11

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>$A$ [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{\text{max}}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>12</td>
<td>25104</td>
<td>189</td>
<td>392</td>
<td>0</td>
<td>221</td>
</tr>
<tr>
<td>L1</td>
<td>12</td>
<td>24858</td>
<td>171</td>
<td>375</td>
<td>0</td>
<td>218</td>
</tr>
<tr>
<td>L2</td>
<td>12</td>
<td>24858</td>
<td>171</td>
<td>375</td>
<td>0</td>
<td>218</td>
</tr>
<tr>
<td>L3</td>
<td>12</td>
<td>24858</td>
<td>171</td>
<td>375</td>
<td>0</td>
<td>218</td>
</tr>
<tr>
<td>L4</td>
<td>12</td>
<td>24858</td>
<td>171</td>
<td>375</td>
<td>0</td>
<td>218</td>
</tr>
<tr>
<td>L5</td>
<td>12</td>
<td>24858</td>
<td>171</td>
<td>375</td>
<td>0</td>
<td>218</td>
</tr>
<tr>
<td>L6</td>
<td>12</td>
<td>24858</td>
<td>171</td>
<td>375</td>
<td>0</td>
<td>218</td>
</tr>
<tr>
<td>L7</td>
<td>12</td>
<td>24858</td>
<td>171</td>
<td>375</td>
<td>0</td>
<td>218</td>
</tr>
<tr>
<td>L8</td>
<td>10</td>
<td>24888</td>
<td>171</td>
<td>380</td>
<td>0</td>
<td>222</td>
</tr>
<tr>
<td>L9</td>
<td>10</td>
<td>24888</td>
<td>171</td>
<td>380</td>
<td>0</td>
<td>222</td>
</tr>
</tbody>
</table>

### Tab. D.41: FPGA Implementation Characteristics Class Bnn BNN11

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>$A$ [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{\text{max}}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>12</td>
<td>456</td>
<td>26</td>
<td>22</td>
<td>0</td>
<td>477</td>
</tr>
<tr>
<td>L1</td>
<td>12</td>
<td>262</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L2</td>
<td>12</td>
<td>262</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L3</td>
<td>12</td>
<td>262</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L4</td>
<td>12</td>
<td>262</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L5</td>
<td>12</td>
<td>262</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L6</td>
<td>12</td>
<td>262</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L7</td>
<td>12</td>
<td>25104</td>
<td>189</td>
<td>392</td>
<td>0</td>
<td>222</td>
</tr>
<tr>
<td>L8</td>
<td>10</td>
<td>24888</td>
<td>171</td>
<td>380</td>
<td>0</td>
<td>222</td>
</tr>
<tr>
<td>L9</td>
<td>10</td>
<td>24888</td>
<td>171</td>
<td>380</td>
<td>0</td>
<td>222</td>
</tr>
</tbody>
</table>

### Tab. D.42: FPGA Implementation Characteristics Bnn::calculate(...) BNN11

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>$A$ [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{\text{max}}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>8</td>
<td>440</td>
<td>26</td>
<td>22</td>
<td>0</td>
<td>477</td>
</tr>
<tr>
<td>L1</td>
<td>8</td>
<td>246</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L2</td>
<td>8</td>
<td>246</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L3</td>
<td>8</td>
<td>246</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L4</td>
<td>8</td>
<td>246</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L5</td>
<td>8</td>
<td>246</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L6</td>
<td>8</td>
<td>246</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L7</td>
<td>8</td>
<td>440</td>
<td>26</td>
<td>22</td>
<td>0</td>
<td>472</td>
</tr>
<tr>
<td>L8</td>
<td>6</td>
<td>234</td>
<td>4</td>
<td>13</td>
<td>0</td>
<td>363</td>
</tr>
<tr>
<td>L9</td>
<td>6</td>
<td>234</td>
<td>4</td>
<td>13</td>
<td>0</td>
<td>363</td>
</tr>
</tbody>
</table>
### Table D.43: FPGA Implementation Characteristics Component BNN12

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>(A) [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>(f_{max}) [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>12</td>
<td>25090</td>
<td>188</td>
<td>391</td>
<td>0</td>
<td>221</td>
</tr>
<tr>
<td>L1</td>
<td>12</td>
<td>24858</td>
<td>171</td>
<td>375</td>
<td>0</td>
<td>218</td>
</tr>
<tr>
<td>L2</td>
<td>12</td>
<td>24858</td>
<td>171</td>
<td>375</td>
<td>0</td>
<td>218</td>
</tr>
<tr>
<td>L3</td>
<td>12</td>
<td>24858</td>
<td>171</td>
<td>375</td>
<td>0</td>
<td>218</td>
</tr>
<tr>
<td>L4</td>
<td>12</td>
<td>24858</td>
<td>171</td>
<td>375</td>
<td>0</td>
<td>218</td>
</tr>
<tr>
<td>L5</td>
<td>12</td>
<td>24858</td>
<td>171</td>
<td>375</td>
<td>0</td>
<td>218</td>
</tr>
<tr>
<td>L6</td>
<td>12</td>
<td>24858</td>
<td>171</td>
<td>375</td>
<td>0</td>
<td>218</td>
</tr>
<tr>
<td>L7</td>
<td>12</td>
<td>25090</td>
<td>188</td>
<td>391</td>
<td>0</td>
<td>221</td>
</tr>
<tr>
<td>L8</td>
<td>10</td>
<td>24888</td>
<td>171</td>
<td>380</td>
<td>0</td>
<td>221</td>
</tr>
<tr>
<td>L9</td>
<td>10</td>
<td>24888</td>
<td>171</td>
<td>380</td>
<td>0</td>
<td>221</td>
</tr>
</tbody>
</table>

### Table D.44: FPGA Implementation Characteristics Class Bnn BNN12

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>(A) [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>(f_{max}) [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>12</td>
<td>442</td>
<td>25</td>
<td>21</td>
<td>0</td>
<td>477</td>
</tr>
<tr>
<td>L1</td>
<td>12</td>
<td>262</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L2</td>
<td>12</td>
<td>262</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L3</td>
<td>12</td>
<td>262</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L4</td>
<td>12</td>
<td>262</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L5</td>
<td>12</td>
<td>262</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L6</td>
<td>12</td>
<td>262</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L7</td>
<td>12</td>
<td>442</td>
<td>25</td>
<td>21</td>
<td>0</td>
<td>477</td>
</tr>
<tr>
<td>L8</td>
<td>10</td>
<td>250</td>
<td>4</td>
<td>13</td>
<td>0</td>
<td>363</td>
</tr>
<tr>
<td>L9</td>
<td>10</td>
<td>250</td>
<td>4</td>
<td>13</td>
<td>0</td>
<td>363</td>
</tr>
</tbody>
</table>

### Table D.45: FPGA Implementation Characteristics Bnn::calculate(...) BNN12

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>(A) [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>(f_{max}) [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>8</td>
<td>426</td>
<td>25</td>
<td>21</td>
<td>0</td>
<td>477</td>
</tr>
<tr>
<td>L1</td>
<td>8</td>
<td>246</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L2</td>
<td>8</td>
<td>246</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L3</td>
<td>8</td>
<td>246</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L4</td>
<td>8</td>
<td>246</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L5</td>
<td>8</td>
<td>246</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L6</td>
<td>8</td>
<td>246</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>358</td>
</tr>
<tr>
<td>L7</td>
<td>8</td>
<td>426</td>
<td>25</td>
<td>21</td>
<td>0</td>
<td>477</td>
</tr>
<tr>
<td>L8</td>
<td>6</td>
<td>234</td>
<td>4</td>
<td>13</td>
<td>0</td>
<td>363</td>
</tr>
<tr>
<td>L9</td>
<td>6</td>
<td>234</td>
<td>4</td>
<td>13</td>
<td>0</td>
<td>363</td>
</tr>
</tbody>
</table>
### Tab. D.46: FPGA Implementation Characteristics Component BNN13

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>$A$ [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{max}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>342</td>
<td>231313</td>
<td>752</td>
<td>1182</td>
<td>3</td>
<td>113</td>
</tr>
<tr>
<td>L1</td>
<td>346</td>
<td>231144</td>
<td>735</td>
<td>1174</td>
<td>3</td>
<td>117</td>
</tr>
<tr>
<td>L2</td>
<td>221</td>
<td>228537</td>
<td>504</td>
<td>1046</td>
<td>3</td>
<td>115</td>
</tr>
<tr>
<td>L3</td>
<td>211</td>
<td>228433</td>
<td>503</td>
<td>1028</td>
<td>3</td>
<td>113</td>
</tr>
<tr>
<td>L4</td>
<td>214</td>
<td>228438</td>
<td>504</td>
<td>1028</td>
<td>3</td>
<td>104</td>
</tr>
<tr>
<td>L5</td>
<td>211</td>
<td>228405</td>
<td>504</td>
<td>1022</td>
<td>3</td>
<td>104</td>
</tr>
<tr>
<td>L6</td>
<td>214</td>
<td>228394</td>
<td>503</td>
<td>1022</td>
<td>3</td>
<td>111</td>
</tr>
<tr>
<td>L7</td>
<td>211</td>
<td>228711</td>
<td>525</td>
<td>1045</td>
<td>3</td>
<td>104</td>
</tr>
<tr>
<td>L8</td>
<td>211</td>
<td>228405</td>
<td>504</td>
<td>1020</td>
<td>3</td>
<td>104</td>
</tr>
<tr>
<td>L9</td>
<td>76</td>
<td>222791</td>
<td>211</td>
<td>491</td>
<td>3</td>
<td>132</td>
</tr>
</tbody>
</table>

### Tab. D.47: FPGA Implementation Characteristics Class Bnn BNN13

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>$A$ [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{max}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>342</td>
<td>8234</td>
<td>531</td>
<td>563</td>
<td>0</td>
<td>202</td>
</tr>
<tr>
<td>L1</td>
<td>346</td>
<td>8036</td>
<td>512</td>
<td>556</td>
<td>0</td>
<td>199</td>
</tr>
<tr>
<td>L2</td>
<td>221</td>
<td>5451</td>
<td>289</td>
<td>422</td>
<td>0</td>
<td>233</td>
</tr>
<tr>
<td>L3</td>
<td>211</td>
<td>5258</td>
<td>281</td>
<td>398</td>
<td>0</td>
<td>233</td>
</tr>
<tr>
<td>L4</td>
<td>214</td>
<td>5325</td>
<td>286</td>
<td>403</td>
<td>0</td>
<td>233</td>
</tr>
<tr>
<td>L5</td>
<td>211</td>
<td>5272</td>
<td>282</td>
<td>399</td>
<td>0</td>
<td>232</td>
</tr>
<tr>
<td>L6</td>
<td>214</td>
<td>5260</td>
<td>282</td>
<td>197</td>
<td>0</td>
<td>233</td>
</tr>
<tr>
<td>L7</td>
<td>211</td>
<td>5610</td>
<td>307</td>
<td>422</td>
<td>0</td>
<td>231</td>
</tr>
<tr>
<td>L8</td>
<td>211</td>
<td>5311</td>
<td>285</td>
<td>400</td>
<td>0</td>
<td>232</td>
</tr>
<tr>
<td>L9</td>
<td>76</td>
<td>1958</td>
<td>119</td>
<td>104</td>
<td>0</td>
<td>232</td>
</tr>
</tbody>
</table>

### Tab. D.48: FPGA Implementation Characteristics Bnn::calculate(...) BNN13

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>$A$ [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>$f_{max}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>173</td>
<td>3992</td>
<td>295</td>
<td>211</td>
<td>0</td>
<td>207</td>
</tr>
<tr>
<td>L1</td>
<td>176</td>
<td>3973</td>
<td>290</td>
<td>214</td>
<td>0</td>
<td>207</td>
</tr>
<tr>
<td>L2</td>
<td>75</td>
<td>1958</td>
<td>121</td>
<td>104</td>
<td>0</td>
<td>233</td>
</tr>
<tr>
<td>L3</td>
<td>74</td>
<td>1950</td>
<td>120</td>
<td>104</td>
<td>0</td>
<td>233</td>
</tr>
<tr>
<td>L4</td>
<td>74</td>
<td>1950</td>
<td>120</td>
<td>104</td>
<td>0</td>
<td>233</td>
</tr>
<tr>
<td>L5</td>
<td>74</td>
<td>1950</td>
<td>120</td>
<td>104</td>
<td>0</td>
<td>232</td>
</tr>
<tr>
<td>L6</td>
<td>74</td>
<td>1950</td>
<td>120</td>
<td>104</td>
<td>0</td>
<td>233</td>
</tr>
<tr>
<td>L7</td>
<td>74</td>
<td>2032</td>
<td>125</td>
<td>111</td>
<td>0</td>
<td>232</td>
</tr>
<tr>
<td>L8</td>
<td>73</td>
<td>1942</td>
<td>119</td>
<td>104</td>
<td>0</td>
<td>232</td>
</tr>
<tr>
<td>L9</td>
<td>72</td>
<td>1942</td>
<td>119</td>
<td>104</td>
<td>0</td>
<td>232</td>
</tr>
</tbody>
</table>
### Tab. D.49: FPGA Implementation Characteristics Component BNN14

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>(A) [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>(f_{\text{max}}) [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>333</td>
<td>233113</td>
<td>824</td>
<td>1344</td>
<td>3</td>
<td>105</td>
</tr>
<tr>
<td>L1</td>
<td>333</td>
<td>232781</td>
<td>793</td>
<td>1330</td>
<td>3</td>
<td>105</td>
</tr>
<tr>
<td>L2</td>
<td>290</td>
<td>231825</td>
<td>699</td>
<td>1290</td>
<td>3</td>
<td>102</td>
</tr>
<tr>
<td>L3</td>
<td>278</td>
<td>231748</td>
<td>695</td>
<td>1284</td>
<td>3</td>
<td>101</td>
</tr>
<tr>
<td>L4</td>
<td>199</td>
<td>230333</td>
<td>874</td>
<td>1206</td>
<td>3</td>
<td>100</td>
</tr>
<tr>
<td>L5</td>
<td>198</td>
<td>230266</td>
<td>872</td>
<td>1197</td>
<td>3</td>
<td>97</td>
</tr>
<tr>
<td>L6</td>
<td>198</td>
<td>230373</td>
<td>876</td>
<td>1211</td>
<td>3</td>
<td>102</td>
</tr>
<tr>
<td>L7</td>
<td>198</td>
<td>230692</td>
<td>599</td>
<td>1234</td>
<td>3</td>
<td>102</td>
</tr>
<tr>
<td>L8</td>
<td>194</td>
<td>230285</td>
<td>571</td>
<td>1201</td>
<td>3</td>
<td>97</td>
</tr>
<tr>
<td>L9</td>
<td>64</td>
<td>224806</td>
<td>281</td>
<td>690</td>
<td>3</td>
<td>132</td>
</tr>
</tbody>
</table>

### Tab. D.50: FPGA Implementation Characteristics Class Bnn BNN14

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>(A) [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>(f_{\text{max}}) [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>333</td>
<td>9650</td>
<td>568</td>
<td>665</td>
<td>0</td>
<td>159</td>
</tr>
<tr>
<td>L1</td>
<td>333</td>
<td>9304</td>
<td>535</td>
<td>654</td>
<td>0</td>
<td>161</td>
</tr>
<tr>
<td>L2</td>
<td>290</td>
<td>8298</td>
<td>437</td>
<td>611</td>
<td>0</td>
<td>168</td>
</tr>
<tr>
<td>L3</td>
<td>278</td>
<td>8247</td>
<td>437</td>
<td>604</td>
<td>0</td>
<td>168</td>
</tr>
<tr>
<td>L4</td>
<td>199</td>
<td>6840</td>
<td>317</td>
<td>527</td>
<td>0</td>
<td>168</td>
</tr>
<tr>
<td>L5</td>
<td>198</td>
<td>6806</td>
<td>315</td>
<td>523</td>
<td>0</td>
<td>168</td>
</tr>
<tr>
<td>L6</td>
<td>198</td>
<td>6884</td>
<td>318</td>
<td>532</td>
<td>0</td>
<td>168</td>
</tr>
<tr>
<td>L7</td>
<td>198</td>
<td>7215</td>
<td>341</td>
<td>557</td>
<td>0</td>
<td>161</td>
</tr>
<tr>
<td>L8</td>
<td>194</td>
<td>6788</td>
<td>312</td>
<td>524</td>
<td>0</td>
<td>167</td>
</tr>
<tr>
<td>L9</td>
<td>64</td>
<td>3344</td>
<td>154</td>
<td>203</td>
<td>0</td>
<td>168</td>
</tr>
</tbody>
</table>

### Tab. D.51: FPGA Implementation Characteristics Bnn::calculate(...) BNN14

<table>
<thead>
<tr>
<th>Level</th>
<th>#FSM States</th>
<th>(A) [GE]</th>
<th>#FF</th>
<th>#LUT</th>
<th>#BRAM</th>
<th>(f_{\text{max}}) [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>165</td>
<td>5338</td>
<td>333</td>
<td>301</td>
<td>0</td>
<td>159</td>
</tr>
<tr>
<td>L1</td>
<td>165</td>
<td>5168</td>
<td>317</td>
<td>294</td>
<td>0</td>
<td>159</td>
</tr>
<tr>
<td>L2</td>
<td>143</td>
<td>4724</td>
<td>278</td>
<td>271</td>
<td>0</td>
<td>168</td>
</tr>
<tr>
<td>L3</td>
<td>141</td>
<td>4736</td>
<td>275</td>
<td>278</td>
<td>0</td>
<td>159</td>
</tr>
<tr>
<td>L4</td>
<td>62</td>
<td>3323</td>
<td>156</td>
<td>200</td>
<td>0</td>
<td>168</td>
</tr>
<tr>
<td>L5</td>
<td>61</td>
<td>3315</td>
<td>155</td>
<td>199</td>
<td>0</td>
<td>168</td>
</tr>
<tr>
<td>L6</td>
<td>61</td>
<td>3312</td>
<td>155</td>
<td>198</td>
<td>0</td>
<td>168</td>
</tr>
<tr>
<td>L7</td>
<td>61</td>
<td>3415</td>
<td>163</td>
<td>205</td>
<td>0</td>
<td>168</td>
</tr>
<tr>
<td>L8</td>
<td>60</td>
<td>3319</td>
<td>154</td>
<td>202</td>
<td>0</td>
<td>168</td>
</tr>
<tr>
<td>L9</td>
<td>60</td>
<td>3334</td>
<td>154</td>
<td>204</td>
<td>0</td>
<td>168</td>
</tr>
</tbody>
</table>
### Tab. D.52: FPGA Implementation Area Estimation Component BNN0

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>162816</td>
<td>171240</td>
<td>8424</td>
<td>5.17</td>
</tr>
<tr>
<td>L1</td>
<td>162507</td>
<td>171144</td>
<td>8637</td>
<td>5.31</td>
</tr>
<tr>
<td>L2</td>
<td>162508</td>
<td>170898</td>
<td>8390</td>
<td>5.16</td>
</tr>
<tr>
<td>L3</td>
<td>162274</td>
<td>171013</td>
<td>8739</td>
<td>5.39</td>
</tr>
<tr>
<td>L4</td>
<td>162262</td>
<td>171013</td>
<td>8751</td>
<td>5.39</td>
</tr>
<tr>
<td>L5</td>
<td>162391</td>
<td>170869</td>
<td>8478</td>
<td>5.22</td>
</tr>
<tr>
<td>L6</td>
<td>162274</td>
<td>170917</td>
<td>8643</td>
<td>5.33</td>
</tr>
<tr>
<td>L7</td>
<td>162823</td>
<td>170917</td>
<td>8094</td>
<td>4.97</td>
</tr>
<tr>
<td>L8</td>
<td>162351</td>
<td>170693</td>
<td>8342</td>
<td>5.14</td>
</tr>
<tr>
<td>L9</td>
<td>159265</td>
<td>170741</td>
<td>11476</td>
<td>7.21</td>
</tr>
</tbody>
</table>

### Tab. D.53: FPGA Implementation Area Estimation Component BNN1

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>33911</td>
<td>50007</td>
<td>16096</td>
<td>47.47</td>
</tr>
<tr>
<td>L1</td>
<td>33511</td>
<td>50007</td>
<td>16496</td>
<td>49.23</td>
</tr>
<tr>
<td>L2</td>
<td>34409</td>
<td>49809</td>
<td>15400</td>
<td>44.76</td>
</tr>
<tr>
<td>L3</td>
<td>34409</td>
<td>49809</td>
<td>15400</td>
<td>44.76</td>
</tr>
<tr>
<td>L4</td>
<td>34409</td>
<td>49809</td>
<td>15400</td>
<td>44.76</td>
</tr>
<tr>
<td>L5</td>
<td>34347</td>
<td>49809</td>
<td>15462</td>
<td>45.02</td>
</tr>
<tr>
<td>L6</td>
<td>34409</td>
<td>49809</td>
<td>15400</td>
<td>44.76</td>
</tr>
<tr>
<td>L7</td>
<td>34809</td>
<td>49809</td>
<td>15000</td>
<td>43.09</td>
</tr>
<tr>
<td>L8</td>
<td>34319</td>
<td>48990</td>
<td>14671</td>
<td>42.75</td>
</tr>
<tr>
<td>L9</td>
<td>31513</td>
<td>48990</td>
<td>17477</td>
<td>55.46</td>
</tr>
</tbody>
</table>

### Tab. D.54: FPGA Implementation Area Estimation Component BNN2

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>29200</td>
<td>35668</td>
<td>6468</td>
<td>22.15</td>
</tr>
<tr>
<td>L1</td>
<td>28692</td>
<td>35668</td>
<td>6976</td>
<td>24.31</td>
</tr>
<tr>
<td>L2</td>
<td>28692</td>
<td>35470</td>
<td>6778</td>
<td>23.62</td>
</tr>
<tr>
<td>L3</td>
<td>28692</td>
<td>35470</td>
<td>6778</td>
<td>23.62</td>
</tr>
<tr>
<td>L4</td>
<td>28692</td>
<td>35470</td>
<td>6778</td>
<td>23.62</td>
</tr>
<tr>
<td>L5</td>
<td>28692</td>
<td>35470</td>
<td>6778</td>
<td>23.62</td>
</tr>
<tr>
<td>L6</td>
<td>28692</td>
<td>35470</td>
<td>6778</td>
<td>23.62</td>
</tr>
<tr>
<td>L7</td>
<td>29200</td>
<td>35470</td>
<td>6270</td>
<td>21.47</td>
</tr>
<tr>
<td>L8</td>
<td>28664</td>
<td>35294</td>
<td>6630</td>
<td>23.13</td>
</tr>
<tr>
<td>L9</td>
<td>25892</td>
<td>35294</td>
<td>9402</td>
<td>36.31</td>
</tr>
</tbody>
</table>
### Tab. D.55: FPGA Implementation Area Estimation Component BNN3

<table>
<thead>
<tr>
<th>Level</th>
<th>( A ) [GE]</th>
<th>( \hat{A} ) [GE]</th>
<th>( \Delta_{err} ) [GE]</th>
<th>( \Delta_{err} ) [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>163655</td>
<td>172023</td>
<td>8368</td>
<td>5.11</td>
</tr>
<tr>
<td>L1</td>
<td>163289</td>
<td>172071</td>
<td>8782</td>
<td>5.38</td>
</tr>
<tr>
<td>L2</td>
<td>163285</td>
<td>171825</td>
<td>8540</td>
<td>5.23</td>
</tr>
<tr>
<td>L3</td>
<td>163162</td>
<td>171796</td>
<td>8634</td>
<td>5.29</td>
</tr>
<tr>
<td>L4</td>
<td>163174</td>
<td>171796</td>
<td>8622</td>
<td>5.28</td>
</tr>
<tr>
<td>L5</td>
<td>163148</td>
<td>171796</td>
<td>8648</td>
<td>5.30</td>
</tr>
<tr>
<td>L6</td>
<td>163180</td>
<td>171796</td>
<td>8616</td>
<td>5.28</td>
</tr>
<tr>
<td>L7</td>
<td>163624</td>
<td>171844</td>
<td>8220</td>
<td>5.02</td>
</tr>
<tr>
<td>L8</td>
<td>163333</td>
<td>171764</td>
<td>8431</td>
<td>5.16</td>
</tr>
<tr>
<td>L9</td>
<td>160306</td>
<td>171668</td>
<td>11362</td>
<td>7.09</td>
</tr>
</tbody>
</table>

### Tab. D.56: FPGA Implementation Area Estimation Component BNN4

<table>
<thead>
<tr>
<th>Level</th>
<th>( A ) [GE]</th>
<th>( \hat{A} ) [GE]</th>
<th>( \Delta_{err} ) [GE]</th>
<th>( \Delta_{err} ) [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>25690</td>
<td>30284</td>
<td>4594</td>
<td>17.88</td>
</tr>
<tr>
<td>L1</td>
<td>25452</td>
<td>30284</td>
<td>4832</td>
<td>18.98</td>
</tr>
<tr>
<td>L2</td>
<td>25452</td>
<td>30110</td>
<td>4658</td>
<td>18.30</td>
</tr>
<tr>
<td>L3</td>
<td>25452</td>
<td>30110</td>
<td>4658</td>
<td>18.30</td>
</tr>
<tr>
<td>L4</td>
<td>25446</td>
<td>30110</td>
<td>4664</td>
<td>18.33</td>
</tr>
<tr>
<td>L5</td>
<td>25452</td>
<td>30110</td>
<td>4658</td>
<td>18.30</td>
</tr>
<tr>
<td>L6</td>
<td>25452</td>
<td>30110</td>
<td>4658</td>
<td>18.30</td>
</tr>
<tr>
<td>L7</td>
<td>25684</td>
<td>30110</td>
<td>4426</td>
<td>17.23</td>
</tr>
<tr>
<td>L8</td>
<td>25548</td>
<td>30055</td>
<td>4507</td>
<td>17.64</td>
</tr>
<tr>
<td>L9</td>
<td>25548</td>
<td>30055</td>
<td>4507</td>
<td>17.64</td>
</tr>
</tbody>
</table>

### Tab. D.57: FPGA Implementation Area Estimation Component BNN5

<table>
<thead>
<tr>
<th>Level</th>
<th>( A ) [GE]</th>
<th>( \hat{A} ) [GE]</th>
<th>( \Delta_{err} ) [GE]</th>
<th>( \Delta_{err} ) [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>25686</td>
<td>30112</td>
<td>4426</td>
<td>17.23</td>
</tr>
<tr>
<td>L1</td>
<td>25446</td>
<td>30112</td>
<td>4666</td>
<td>18.34</td>
</tr>
<tr>
<td>L2</td>
<td>25446</td>
<td>30086</td>
<td>4640</td>
<td>18.23</td>
</tr>
<tr>
<td>L3</td>
<td>25446</td>
<td>30086</td>
<td>4640</td>
<td>18.23</td>
</tr>
<tr>
<td>L4</td>
<td>25446</td>
<td>30086</td>
<td>4640</td>
<td>18.23</td>
</tr>
<tr>
<td>L5</td>
<td>25446</td>
<td>30086</td>
<td>4640</td>
<td>18.23</td>
</tr>
<tr>
<td>L6</td>
<td>25446</td>
<td>30086</td>
<td>4640</td>
<td>18.23</td>
</tr>
<tr>
<td>L7</td>
<td>25698</td>
<td>30086</td>
<td>4388</td>
<td>17.08</td>
</tr>
<tr>
<td>L8</td>
<td>25554</td>
<td>30042</td>
<td>4488</td>
<td>17.56</td>
</tr>
<tr>
<td>L9</td>
<td>25554</td>
<td>30042</td>
<td>4488</td>
<td>17.56</td>
</tr>
</tbody>
</table>
Tab. D.58: FPGA Implementation Area Estimation Component BNN6

<table>
<thead>
<tr>
<th>Level</th>
<th>$A$ [GE]</th>
<th>$\hat{A}$ [GE]</th>
<th>$A_{err}$ [GE]</th>
<th>$A_{err}$ [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>173310</td>
<td>188544</td>
<td>15234</td>
<td>8.79</td>
</tr>
<tr>
<td>L1</td>
<td>171718</td>
<td>189736</td>
<td>18018</td>
<td>10.49</td>
</tr>
<tr>
<td>L2</td>
<td>165012</td>
<td>181226</td>
<td>16214</td>
<td>9.83</td>
</tr>
<tr>
<td>L3</td>
<td>165122</td>
<td>181197</td>
<td>16075</td>
<td>9.74</td>
</tr>
<tr>
<td>L4</td>
<td>164958</td>
<td>181149</td>
<td>16191</td>
<td>9.82</td>
</tr>
<tr>
<td>L5</td>
<td>164958</td>
<td>181182</td>
<td>16224</td>
<td>9.84</td>
</tr>
<tr>
<td>L6</td>
<td>165048</td>
<td>181160</td>
<td>16112</td>
<td>9.76</td>
</tr>
<tr>
<td>L7</td>
<td>166660</td>
<td>181252</td>
<td>14592</td>
<td>8.76</td>
</tr>
<tr>
<td>L8</td>
<td>164900</td>
<td>180831</td>
<td>15931</td>
<td>9.66</td>
</tr>
<tr>
<td>L9</td>
<td>156036</td>
<td>180779</td>
<td>24743</td>
<td>15.86</td>
</tr>
</tbody>
</table>

Tab. D.59: FPGA Implementation Area Estimation Component BNN7

<table>
<thead>
<tr>
<th>Level</th>
<th>$A$ [GE]</th>
<th>$\hat{A}$ [GE]</th>
<th>$A_{err}$ [GE]</th>
<th>$A_{err}$ [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>168813</td>
<td>184016</td>
<td>15203</td>
<td>9.01</td>
</tr>
<tr>
<td>L1</td>
<td>166123</td>
<td>183025</td>
<td>17082</td>
<td>10.28</td>
</tr>
<tr>
<td>L2</td>
<td>166123</td>
<td>183181</td>
<td>17058</td>
<td>10.27</td>
</tr>
<tr>
<td>L3</td>
<td>165945</td>
<td>182189</td>
<td>16244</td>
<td>9.79</td>
</tr>
<tr>
<td>L4</td>
<td>165945</td>
<td>182045</td>
<td>16100</td>
<td>9.70</td>
</tr>
<tr>
<td>L5</td>
<td>165949</td>
<td>182189</td>
<td>16240</td>
<td>9.79</td>
</tr>
<tr>
<td>L6</td>
<td>167833</td>
<td>182032</td>
<td>14199</td>
<td>8.46</td>
</tr>
<tr>
<td>L7</td>
<td>165747</td>
<td>181738</td>
<td>15991</td>
<td>9.65</td>
</tr>
<tr>
<td>L8</td>
<td>157247</td>
<td>181738</td>
<td>24491</td>
<td>15.57</td>
</tr>
</tbody>
</table>

Tab. D.60: FPGA Implementation Area Estimation Component BNN8

<table>
<thead>
<tr>
<th>Level</th>
<th>$A$ [GE]</th>
<th>$\hat{A}$ [GE]</th>
<th>$A_{err}$ [GE]</th>
<th>$A_{err}$ [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>174493</td>
<td>189908</td>
<td>15415</td>
<td>8.83</td>
</tr>
<tr>
<td>L1</td>
<td>172125</td>
<td>189967</td>
<td>17842</td>
<td>10.37</td>
</tr>
<tr>
<td>L2</td>
<td>167940</td>
<td>185210</td>
<td>17270</td>
<td>10.28</td>
</tr>
<tr>
<td>L3</td>
<td>167990</td>
<td>184941</td>
<td>16951</td>
<td>10.09</td>
</tr>
<tr>
<td>L4</td>
<td>165949</td>
<td>182032</td>
<td>16083</td>
<td>9.69</td>
</tr>
<tr>
<td>L5</td>
<td>166004</td>
<td>182176</td>
<td>16172</td>
<td>9.74</td>
</tr>
<tr>
<td>L6</td>
<td>165959</td>
<td>182045</td>
<td>16086</td>
<td>9.69</td>
</tr>
<tr>
<td>L7</td>
<td>167657</td>
<td>182045</td>
<td>14388</td>
<td>8.58</td>
</tr>
<tr>
<td>L8</td>
<td>165803</td>
<td>181640</td>
<td>15837</td>
<td>9.55</td>
</tr>
<tr>
<td>L9</td>
<td>157243</td>
<td>181594</td>
<td>24351</td>
<td>15.49</td>
</tr>
</tbody>
</table>
### Tab. D.61: FPGA Implementation Area Estimation Component BNN9

<table>
<thead>
<tr>
<th>Level</th>
<th>$A$ [GE]</th>
<th>$\hat{A}$ [GE]</th>
<th>$A_{err}$ [GE]</th>
<th>$A_{err}$ [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>37518</td>
<td>59733</td>
<td>22115</td>
<td>59.21</td>
</tr>
<tr>
<td>L1</td>
<td>35350</td>
<td>59733</td>
<td>24383</td>
<td>68.98</td>
</tr>
<tr>
<td>L2</td>
<td>35484</td>
<td>58682</td>
<td>23198</td>
<td>65.38</td>
</tr>
<tr>
<td>L3</td>
<td>35446</td>
<td>58693</td>
<td>23247</td>
<td>65.58</td>
</tr>
<tr>
<td>L4</td>
<td>35448</td>
<td>58682</td>
<td>23234</td>
<td>65.54</td>
</tr>
<tr>
<td>L5</td>
<td>35448</td>
<td>58676</td>
<td>23228</td>
<td>65.53</td>
</tr>
<tr>
<td>L6</td>
<td>35244</td>
<td>58676</td>
<td>23432</td>
<td>66.49</td>
</tr>
<tr>
<td>L7</td>
<td>37539</td>
<td>58676</td>
<td>21137</td>
<td>56.31</td>
</tr>
<tr>
<td>L8</td>
<td>35202</td>
<td>57215</td>
<td>22013</td>
<td>62.53</td>
</tr>
<tr>
<td>L9</td>
<td>26362</td>
<td>57215</td>
<td>30853</td>
<td>117.04</td>
</tr>
</tbody>
</table>

### Tab. D.62: FPGA Implementation Area Estimation Component BNN10

<table>
<thead>
<tr>
<th>Level</th>
<th>$A$ [GE]</th>
<th>$\hat{A}$ [GE]</th>
<th>$A_{err}$ [GE]</th>
<th>$A_{err}$ [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>36004</td>
<td>52790</td>
<td>16786</td>
<td>46.62</td>
</tr>
<tr>
<td>L1</td>
<td>33952</td>
<td>52790</td>
<td>18838</td>
<td>55.48</td>
</tr>
<tr>
<td>L2</td>
<td>33952</td>
<td>50759</td>
<td>16807</td>
<td>49.50</td>
</tr>
<tr>
<td>L3</td>
<td>33952</td>
<td>50792</td>
<td>16840</td>
<td>49.60</td>
</tr>
<tr>
<td>L4</td>
<td>33952</td>
<td>50781</td>
<td>16829</td>
<td>49.57</td>
</tr>
<tr>
<td>L5</td>
<td>33952</td>
<td>50781</td>
<td>16829</td>
<td>49.57</td>
</tr>
<tr>
<td>L6</td>
<td>33952</td>
<td>50792</td>
<td>16840</td>
<td>49.60</td>
</tr>
<tr>
<td>L7</td>
<td>36004</td>
<td>50792</td>
<td>14788</td>
<td>41.07</td>
</tr>
<tr>
<td>L8</td>
<td>33802</td>
<td>50286</td>
<td>16484</td>
<td>48.77</td>
</tr>
<tr>
<td>L9</td>
<td>25118</td>
<td>50319</td>
<td>25201</td>
<td>100.33</td>
</tr>
</tbody>
</table>

### Tab. D.63: FPGA Implementation Area Estimation Component BNN11

<table>
<thead>
<tr>
<th>Level</th>
<th>$A$ [GE]</th>
<th>$\hat{A}$ [GE]</th>
<th>$A_{err}$ [GE]</th>
<th>$A_{err}$ [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>25104</td>
<td>28799</td>
<td>3695</td>
<td>14.72</td>
</tr>
<tr>
<td>L1</td>
<td>24858</td>
<td>28971</td>
<td>4113</td>
<td>16.55</td>
</tr>
<tr>
<td>L2</td>
<td>24858</td>
<td>28797</td>
<td>3939</td>
<td>15.85</td>
</tr>
<tr>
<td>L3</td>
<td>24858</td>
<td>28797</td>
<td>3939</td>
<td>15.85</td>
</tr>
<tr>
<td>L4</td>
<td>24858</td>
<td>28797</td>
<td>3939</td>
<td>15.85</td>
</tr>
<tr>
<td>L5</td>
<td>24858</td>
<td>28797</td>
<td>3939</td>
<td>15.85</td>
</tr>
<tr>
<td>L6</td>
<td>24858</td>
<td>28797</td>
<td>3939</td>
<td>15.85</td>
</tr>
<tr>
<td>L7</td>
<td>25104</td>
<td>28797</td>
<td>3693</td>
<td>14.71</td>
</tr>
<tr>
<td>L8</td>
<td>24888</td>
<td>28742</td>
<td>3854</td>
<td>15.49</td>
</tr>
<tr>
<td>L9</td>
<td>24888</td>
<td>28742</td>
<td>3854</td>
<td>15.49</td>
</tr>
</tbody>
</table>
### Tab. D.64: FPGA Implementation Area Estimation Component BNN12

<table>
<thead>
<tr>
<th>Level</th>
<th>$A$ [GE]</th>
<th>$A_{err}$ [GE]</th>
<th>$A_{err}$ [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>25090</td>
<td>3709</td>
<td>14.78</td>
</tr>
<tr>
<td>L1</td>
<td>24858</td>
<td>3941</td>
<td>15.85</td>
</tr>
<tr>
<td>L2</td>
<td>24858</td>
<td>3915</td>
<td>15.75</td>
</tr>
<tr>
<td>L3</td>
<td>24858</td>
<td>3915</td>
<td>15.75</td>
</tr>
<tr>
<td>L4</td>
<td>24858</td>
<td>3915</td>
<td>15.75</td>
</tr>
<tr>
<td>L5</td>
<td>24858</td>
<td>3915</td>
<td>15.75</td>
</tr>
<tr>
<td>L6</td>
<td>24858</td>
<td>3915</td>
<td>15.75</td>
</tr>
<tr>
<td>L7</td>
<td>25090</td>
<td>3683</td>
<td>14.68</td>
</tr>
<tr>
<td>L8</td>
<td>24888</td>
<td>3841</td>
<td>15.43</td>
</tr>
<tr>
<td>L9</td>
<td>24888</td>
<td>3841</td>
<td>15.43</td>
</tr>
</tbody>
</table>

### Tab. D.65: FPGA Implementation Area Estimation Component BNN13

<table>
<thead>
<tr>
<th>Level</th>
<th>$A$ [GE]</th>
<th>$A_{err}$ [GE]</th>
<th>$A_{err}$ [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>231313</td>
<td>14553</td>
<td>6.29</td>
</tr>
<tr>
<td>L1</td>
<td>231144</td>
<td>14445</td>
<td>6.25</td>
</tr>
<tr>
<td>L2</td>
<td>228537</td>
<td>14454</td>
<td>6.32</td>
</tr>
<tr>
<td>L3</td>
<td>228433</td>
<td>14293</td>
<td>6.26</td>
</tr>
<tr>
<td>L4</td>
<td>228438</td>
<td>14349</td>
<td>6.28</td>
</tr>
<tr>
<td>L5</td>
<td>228405</td>
<td>14321</td>
<td>6.27</td>
</tr>
<tr>
<td>L6</td>
<td>228394</td>
<td>14234</td>
<td>6.23</td>
</tr>
<tr>
<td>L7</td>
<td>228711</td>
<td>14034</td>
<td>6.14</td>
</tr>
<tr>
<td>L8</td>
<td>228405</td>
<td>14124</td>
<td>6.18</td>
</tr>
<tr>
<td>L9</td>
<td>222791</td>
<td>21719</td>
<td>9.75</td>
</tr>
</tbody>
</table>

### Tab. D.66: FPGA Implementation Area Estimation Component BNN14

<table>
<thead>
<tr>
<th>Level</th>
<th>$A$ [GE]</th>
<th>$A_{err}$ [GE]</th>
<th>$A_{err}$ [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>233113</td>
<td>14538</td>
<td>6.24</td>
</tr>
<tr>
<td>L1</td>
<td>232781</td>
<td>14870</td>
<td>6.39</td>
</tr>
<tr>
<td>L2</td>
<td>231825</td>
<td>15174</td>
<td>6.55</td>
</tr>
<tr>
<td>L3</td>
<td>231748</td>
<td>14645</td>
<td>6.32</td>
</tr>
<tr>
<td>L4</td>
<td>230333</td>
<td>14314</td>
<td>6.21</td>
</tr>
<tr>
<td>L5</td>
<td>230266</td>
<td>14453</td>
<td>6.28</td>
</tr>
<tr>
<td>L6</td>
<td>230373</td>
<td>14346</td>
<td>6.23</td>
</tr>
<tr>
<td>L7</td>
<td>230692</td>
<td>13966</td>
<td>6.05</td>
</tr>
<tr>
<td>L8</td>
<td>230285</td>
<td>14230</td>
<td>6.18</td>
</tr>
<tr>
<td>L9</td>
<td>224806</td>
<td>18503</td>
<td>8.23</td>
</tr>
</tbody>
</table>
### Tab. D.67: Average Compilation Times of the FPGA Implementation of BNN Designs

<table>
<thead>
<tr>
<th>Design</th>
<th>$t_{opt}$ [ms]</th>
<th>$t_{map}$ [ms]</th>
<th>$t_{syn}$ [ms]</th>
<th>$t_{sum}$ [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNN0</td>
<td>7756</td>
<td>3178</td>
<td>2731</td>
<td>13666</td>
</tr>
<tr>
<td>BNN1</td>
<td>10186</td>
<td>3672</td>
<td>3364</td>
<td>17222</td>
</tr>
<tr>
<td>BNN2</td>
<td>8680</td>
<td>3092</td>
<td>2508</td>
<td>14280</td>
</tr>
<tr>
<td>BNN3</td>
<td>10325</td>
<td>3992</td>
<td>3092</td>
<td>17410</td>
</tr>
<tr>
<td>BNN4</td>
<td>4028</td>
<td>2447</td>
<td>1789</td>
<td>8264</td>
</tr>
<tr>
<td>BNN5</td>
<td>3806</td>
<td>2675</td>
<td>1795</td>
<td>8276</td>
</tr>
<tr>
<td>BNN6</td>
<td>272345</td>
<td>701076</td>
<td>169685</td>
<td>1143106</td>
</tr>
<tr>
<td>BNN7</td>
<td>24392</td>
<td>8372</td>
<td>4050</td>
<td>36814</td>
</tr>
<tr>
<td>BNN8</td>
<td>38385</td>
<td>28256</td>
<td>12710</td>
<td>79350</td>
</tr>
<tr>
<td>BNN9</td>
<td>17124</td>
<td>5098</td>
<td>4458</td>
<td>26680</td>
</tr>
<tr>
<td>BNN10</td>
<td>80516</td>
<td>4980</td>
<td>3792</td>
<td>89287</td>
</tr>
<tr>
<td>BNN11</td>
<td>3911</td>
<td>2666</td>
<td>1697</td>
<td>8273</td>
</tr>
<tr>
<td>BNN12</td>
<td>3563</td>
<td>2634</td>
<td>1608</td>
<td>7805</td>
</tr>
<tr>
<td>BNN13</td>
<td>17287</td>
<td>3917</td>
<td>13406</td>
<td>34611</td>
</tr>
<tr>
<td>BNN14</td>
<td>7105</td>
<td>4061</td>
<td>4711</td>
<td>15877</td>
</tr>
</tbody>
</table>
### D.2.3 Software Implementation of the BNNs

Tab. D.68: Software Communication Latencies of the BNNs (L9)

<table>
<thead>
<tr>
<th>$t_{\text{write}}$ [ns]</th>
<th>$\sigma$ [ns]</th>
<th>$t_{\text{read}}$ [ns]</th>
<th>$\sigma$ [ns]</th>
<th>$t_{\text{comm}}$ [ns]</th>
<th>$\sigma$ [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
</tr>
<tr>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
</tr>
<tr>
<td>47,26</td>
<td>0,91</td>
<td>52,23</td>
<td>1,49</td>
<td>99,51</td>
<td>1,39</td>
</tr>
<tr>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
</tr>
<tr>
<td>46,92</td>
<td>0,74</td>
<td>51,58</td>
<td>0,00</td>
<td>98,51</td>
<td>0,74</td>
</tr>
<tr>
<td>47,59</td>
<td>0,91</td>
<td>51,92</td>
<td>0,74</td>
<td>99,51</td>
<td>1,39</td>
</tr>
<tr>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
</tr>
<tr>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
</tr>
<tr>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
</tr>
<tr>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
</tr>
<tr>
<td>0,00</td>
<td>0,00</td>
<td>51,92</td>
<td>0,74</td>
<td>51,92</td>
<td>0,74</td>
</tr>
<tr>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
</tr>
<tr>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
</tr>
</tbody>
</table>
### Tab. D.69: Software Execution Latencies of the BNNs (L9)

<table>
<thead>
<tr>
<th>$t_{\text{exec,init}}$ [ns]</th>
<th>$\sigma$ [ns]</th>
<th>$t_{\text{exec,calculate}}$ [ns]</th>
<th>$\sigma$ [ns]</th>
<th>$t_{\text{exec,get,y}}$ [ns]</th>
<th>$\sigma$ [ns]</th>
<th>$t_{\text{exec}}$ [ns]</th>
<th>$\sigma$ [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>82.20</td>
<td>62.02</td>
<td>393.04</td>
<td>113.21</td>
<td>70.22</td>
<td>4.91</td>
<td>545.46</td>
<td>176.39</td>
</tr>
<tr>
<td>216.99</td>
<td>174.18</td>
<td>325.81</td>
<td>259.67</td>
<td>511.85</td>
<td>126.64</td>
<td>1054.64</td>
<td>556.26</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>327.14</td>
<td>152.34</td>
<td>0.00</td>
<td>0.00</td>
<td>327.14</td>
<td>152.34</td>
</tr>
<tr>
<td>98.18</td>
<td>56.02</td>
<td>378.73</td>
<td>126.54</td>
<td>504.19</td>
<td>156.29</td>
<td>981.09</td>
<td>331.45</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>271.90</td>
<td>67.27</td>
<td>0.00</td>
<td>0.00</td>
<td>271.90</td>
<td>67.27</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>223.97</td>
<td>46.18</td>
<td>0.00</td>
<td>0.00</td>
<td>223.97</td>
<td>46.18</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>261.25</td>
<td>67.03</td>
<td>0.00</td>
<td>0.00</td>
<td>261.25</td>
<td>67.03</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>469.91</td>
<td>138.05</td>
<td>0.00</td>
<td>0.00</td>
<td>469.91</td>
<td>138.05</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>442.29</td>
<td>80.93</td>
<td>0.00</td>
<td>0.00</td>
<td>442.29</td>
<td>80.93</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>270.23</td>
<td>100.69</td>
<td>0.00</td>
<td>0.00</td>
<td>270.23</td>
<td>100.69</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>239.28</td>
<td>59.13</td>
<td>0.00</td>
<td>0.00</td>
<td>239.28</td>
<td>59.13</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>211.33</td>
<td>54.47</td>
<td>0.00</td>
<td>0.00</td>
<td>227.97</td>
<td>54.47</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>227.97</td>
<td>26.07</td>
<td>0.00</td>
<td>0.00</td>
<td>227.97</td>
<td>26.07</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>444.95</td>
<td>187.75</td>
<td>0.00</td>
<td>0.00</td>
<td>444.95</td>
<td>187.75</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>984.42</td>
<td>183.51</td>
<td>0.00</td>
<td>0.00</td>
<td>984.42</td>
<td>183.51</td>
</tr>
</tbody>
</table>
Tab. D.70: Average Compilation Times of the Software Implementation of BNN Designs

<table>
<thead>
<tr>
<th>Design</th>
<th>$t_{opt}$ [ms]</th>
<th>$t_{map}$ [ms]</th>
<th>$t_{syn}$ [ms]</th>
<th>$t_{sum}$ [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNN0</td>
<td>3931</td>
<td>2281</td>
<td>366</td>
<td>6578</td>
</tr>
<tr>
<td>BNN1</td>
<td>4797</td>
<td>3835</td>
<td>396</td>
<td>9027</td>
</tr>
<tr>
<td>BNN2</td>
<td>4066</td>
<td>2420</td>
<td>356</td>
<td>6842</td>
</tr>
<tr>
<td>BNN3</td>
<td>4063</td>
<td>2792</td>
<td>374</td>
<td>7228</td>
</tr>
<tr>
<td>BNN4</td>
<td>2772</td>
<td>1305</td>
<td>309</td>
<td>4386</td>
</tr>
<tr>
<td>BNN5</td>
<td>2653</td>
<td>1287</td>
<td>305</td>
<td>4245</td>
</tr>
<tr>
<td>BNN6</td>
<td>102241</td>
<td>5030</td>
<td>503</td>
<td>107773</td>
</tr>
<tr>
<td>BNN7</td>
<td>11275</td>
<td>5466</td>
<td>455</td>
<td>17196</td>
</tr>
<tr>
<td>BNN8</td>
<td>19352</td>
<td>5662</td>
<td>481</td>
<td>25495</td>
</tr>
<tr>
<td>BNN9</td>
<td>7616</td>
<td>4675</td>
<td>519</td>
<td>12809</td>
</tr>
<tr>
<td>BNN10</td>
<td>32861</td>
<td>4469</td>
<td>459</td>
<td>37789</td>
</tr>
<tr>
<td>BNN11</td>
<td>2731</td>
<td>1303</td>
<td>299</td>
<td>4333</td>
</tr>
<tr>
<td>BNN12</td>
<td>2625</td>
<td>1290</td>
<td>302</td>
<td>4217</td>
</tr>
<tr>
<td>BNN13</td>
<td>9555</td>
<td>3677</td>
<td>411</td>
<td>13642</td>
</tr>
<tr>
<td>BNN14</td>
<td>5617</td>
<td>4325</td>
<td>416</td>
<td>10358</td>
</tr>
</tbody>
</table>
D.3 Online Compression of Audio Streams

D.3.1 Description of the Audio Server

Fig. 7.14 summarizes the design of the audio server. The server application comprises the class Main and several classes that implement algorithms for the encoding of audio frames for the purpose of compression. A relatively rich class hierarchy is used to model the encoders. The abstract base class of the hierarchy LPEncoder defines the common interface and data for encoders using linear prediction of waveforms. The class GolombEncoder adds features specific to Golomb encoding [274]. Particular encoding algorithms are defined by the classes AudioPaKEncoder and FLACEncoder.

The behavior of the operations performing the actual encoding in the class AudioPaKEncoder is given in Listing D.13. The FLACEncoder uses a further predictor, which necessitates extensions in the intra-channel decorrelation and the encoding algorithm. For the purpose of this thesis the FLACEncoder is not used, since it does not support the full FLAC algorithm yet and is thus very similar to AudioPaK.

The operation main(...) in class Main first instantiates one or more encoder objects. In the core loop it reads input samples from the audio sources and communicates them to the encoder object. The actual encoding is performed asynchronously, whereas several coders can work concurrently. When a coder has finished the compressed data is packed into an audio frame and is sent to the audio clients. The core loop of the main(...) operation of class Main is given in Listing D.14. This design shows an advantage of object-oriented modeling. The overall system functionality is decomposed into several reusable and extensible classes.

Listing D.13: Behavior of encode()

```c
/* intra--channel decorrelation */
short pdiff1 = 0, pdiff2 = 0;
short abs_err0 = 0, abs_err1 = 0, abs_err2 = 0, abs_err3 = 0;
finished_d = false;
short psample = isamples_d[ 0 ];
for (short i = 1; i < isize_d; i++) {
    short csample = isamples_d[ i ];
    abs_err0 += csample.abs(); // P0
    abs_err1 += (diff1 - pdiff1).abs(); // P1
    abs_err2 += (diff2 - pdiff2).abs(); pdiff1 = diff1; // P2
    abs_err3 += (diff3 - pdiff2).abs(); pdiff2 = diff2; // P3
    psample = csample;
}
/* select predictor with least error */
predictor_d = STATIC_PRED_S0;
if (abs_err1 < abs_err0) { // use P1?
predictor_d = STATIC_PRED_S1;
    abs_err0 = abs_err1;
}
if (abs_err2 < abs_err0) { // use P2?
predictor_d = STATIC_PRED_S2;
    abs_err0 = abs_err2;
}
if (abs_err3 < abs_err0) { // use P3?
predictor_d = STATIC_PRED_S3;
    abs_err0 = abs_err3;
}
/* compute minimum number of bits for Golomb code */
short err = isize_d;
for (golomb_d = (byte)0, twopowk_d = (byte)1;
    err < abs_err0;
```

golomb_d++, twopowk_d <<= 1, err <<= (short)1
{ /* do nothing */ }
/* encode samples */
short psample_m3 = isamples_d[0];
short psample_m2 = isamples_d[1];
short psample_m1 = isamples_d[2];
short osamples_idx = 0;
for(short isamples_idx=predictor_d; isamples_idx<isize_d;)
    isamples_idx++
    int mask = (int) twopowk_d >> 1;
    short csample = isamples_d[isamples_idx];
    int residue = 0;
    switch (predictor_d) {
        case STATIC_PRED_S0: residue = csample; break;
        case STATIC_PRED_S1:
            residue = (int) (csample - psample_m1); break;
        case STATIC_PRED_S2:
            residue = (int) csample - (2 * psample_m1 - psample_m2); break;
        case STATIC_PRED_S3:
            residue = (int) csample - (3 * psample_m1 - 3 * psample_m2 + psample_m3);
            break;
        default: ;
    }
    /* mapping of negative values */
    if( residue < 0 ) {
        residue = (+residue << 1) + 1;
    } else {
        residue = residue << 1;
    }
    /* write preceding bits -> scale */
    if( golomb_d > 0 ) {
        while( residue > golomb_d ) {
            residue = residue - golomb_d;
            osamples_d[osamples_idx] = (bit)1;
            osamples_idx++;
        }
    }
    /* write stop bit */
    osamples_d[osamples_idx] = (bit)0;
    /* write remaining bits */
    while( mask > 0 ) {
        osamples_d[osamples_idx] = (bit) residue & mask;
        osamples_idx++;
        mask = mask >> 1;
    }
    psample_m3 = psample_m2;
    psample_m2 = psample_m1;
    psample_m1 = csample;
}
osize_d = osamples_idx;
finished_d = true;
Listing D.14: Core Loop of main(...)

```c
while( working ) {
    for( int i=0; i<NUMBER_OF_CODERS; i++ ) {
        /* get next samples from input */
        ...
        /* get previously encoded frame */
        if( coders[i].finished_d ) {
            if( coders[i].osize_d > 0 ) {
                osamples = (short[])coders[i].osamples_d;
                osize = coders[i].osize_d;
                predictor = coders[i].predictor_d;
                /* build audio frame and copy samples to network */
                ...
            }
            /* encode samples */
            coders[i].isamples_d = isamples;
            coders[i]. isize_d = FRAME_SIZE;
            async coders[i].encode();
        }
    }
}
```
D. Experimental Results

D.3.2 Implementation of the Audio Server

Tab. D.71: Communication Timing of the Audio Server (L9)

<table>
<thead>
<tr>
<th>#Samples</th>
<th>$t_{\text{read}}$ [ns]</th>
<th>$\sigma$ [ns]</th>
<th>$t_{\text{write}}$ [ns]</th>
<th>$\sigma$ [ns]</th>
<th>$t_{\text{comm}}$ [ns]</th>
<th>$\sigma$ [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>192</td>
<td>147948,24</td>
<td>472,10</td>
<td>4101,43</td>
<td>105,60</td>
<td>152049,66</td>
<td>576,16</td>
</tr>
<tr>
<td>384</td>
<td>295027,53</td>
<td>670,48</td>
<td>9154,83</td>
<td>89,53</td>
<td>304182,36</td>
<td>755,41</td>
</tr>
<tr>
<td>576</td>
<td>442052,25</td>
<td>646,71</td>
<td>17890,50</td>
<td>9754,95</td>
<td>459942,75</td>
<td>9860,62</td>
</tr>
<tr>
<td>768</td>
<td>592608,14</td>
<td>12326,46</td>
<td>42600,23</td>
<td>5803,16</td>
<td>635208,37</td>
<td>15372,7</td>
</tr>
<tr>
<td>960</td>
<td>735717,80</td>
<td>787,56</td>
<td>70192,51</td>
<td>1938,18</td>
<td>805910,31</td>
<td>803264,38</td>
</tr>
<tr>
<td>1152</td>
<td>925127,59</td>
<td>102560,41</td>
<td>91080,54</td>
<td>1681,49</td>
<td>1016208,13</td>
<td>103534,79</td>
</tr>
</tbody>
</table>

Tab. D.72: Execution Timing of the Audio Server (L9)

<table>
<thead>
<tr>
<th>#Samples</th>
<th>$t_{\text{exec}}$ [ns]</th>
<th>$\sigma$ [ns]</th>
<th>$t_{\text{sum}}$ [ns]</th>
<th>$\sigma$ [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>192</td>
<td>207014,75</td>
<td>6019,83</td>
<td>359064,41</td>
<td>5986,03</td>
</tr>
<tr>
<td>384</td>
<td>442773,59</td>
<td>15624,39</td>
<td>746955,96</td>
<td>15734,76</td>
</tr>
<tr>
<td>576</td>
<td>650667,93</td>
<td>36031,01</td>
<td>1110610,68</td>
<td>30076,86</td>
</tr>
<tr>
<td>768</td>
<td>854566,00</td>
<td>36556,61</td>
<td>1489774,37</td>
<td>42489,04</td>
</tr>
<tr>
<td>960</td>
<td>1079957,96</td>
<td>32507,42</td>
<td>1885868,28</td>
<td>32539,99</td>
</tr>
<tr>
<td>1152</td>
<td>1182947,08</td>
<td>33282,36</td>
<td>2199155,21</td>
<td>111410,68</td>
</tr>
</tbody>
</table>

Tab. D.73: FPGA Implementation Characteristics of the Audio Server Component (L9)

<table>
<thead>
<tr>
<th>#FSMStates</th>
<th>$A$ [GE]</th>
<th>$#FF$</th>
<th>$#LUT$</th>
<th>$#BRAM$</th>
<th>$f_{\text{max}}$ [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>93</td>
<td>328435</td>
<td>705</td>
<td>1990</td>
<td>4</td>
<td>45</td>
</tr>
</tbody>
</table>

Tab. D.74: FPGA Implementation Area Estimation of Audio Server Component (L9)

<table>
<thead>
<tr>
<th>$A$ [GE]</th>
<th>$\hat{A}$ [GE]</th>
<th>$A_{\text{err}}$ [GE]</th>
<th>$A_{\text{err}}$ [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>328435</td>
<td>306809</td>
<td>-21626</td>
<td>-7.05</td>
</tr>
</tbody>
</table>

Tab. D.75: Compilation Times of the FPGA Implementation of the Audio Server (L9)

<table>
<thead>
<tr>
<th>$t_{\text{opt}}$ [ms]</th>
<th>$t_{\text{map}}$ [ms]</th>
<th>$t_{\text{syn}}$ [ms]</th>
<th>$t_{\text{sum}}$ [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>12781</td>
<td>3438</td>
<td>4500</td>
<td>20719</td>
</tr>
</tbody>
</table>
BIBLIOGRAPHY


