A Methodology and Framework to Assist in the Architecture Design and Functional Verification of Complex Electronic Systems

DISSERTATION

submitted for the Degree of Doctor of Philosophy by

KOLDO TOMASENA ARRIAGA

under the supervision of

Igone Vélez Isasmendi and

Juan Francisco Sevillano Berasategui

San Sebastián, July 2013
To my parents and Arnaia
Acknowledgements

This work has been possible thanks to many people. First and foremost, I would like to express my deep gratitude to my research supervisors Ignone Vélez and Juan Francisco Sevillano for their dedication, guidance and continuous encouragement during the whole research work. I would like to thank Andoni Irizar for giving me the opportunity to join the Digital Signal Processing group. I feel also grateful to Ainhoa Cortés and Luis Fontán for their support since I joined CEIT.

My special thanks to Alvaro, Esti, Jose, Jose Ramón, Almir and Txaber for their friendship and invaluable help during this years at CEIT.

I would also like to express my sincere thanks to all the people of the DSP laboratory. Thanks to Marta, Leticia, Jorge, Ainara, Aritz and Marcos. It has been a privilege to work with all of you.

I feel grateful to my colleagues at the Electronics and Communications Department of CEIT for their support. My special thanks to Fernando Arizti for his wise words and advice.

Financially, I would like to acknowledge the Basque Government, Spanish Government, the European Community, Amigos de la Universidad de Navarra and CEIT.

I want to thank my family, specially my parents for their support and encouragement. I would like to express my greatest gratitude to Amaia for her unconditional support. You always have the correct words to encourage me, thank you so much. I also want to thank my friends for their interest in the evolution of this work, specially during these last months. Eskerrik asko aita, ama, Amaia eta lagunei, bihotz bihotzez. I feel so fortunate to have you all.

Koldo Tomasena, July 2013.
Summary

Modern electronic systems are increasing their complexity to meet market requirements. However, present Electronic Design Automation (EDA) tools can not handle this complexity growth efficiently. This productivity loss produces the so-called design productivity gap. New design and verification methodologies supported by modern tools are needed to handle present electronic systems and reduce the development time.

In order to address the design productivity gap, Electronic System Level (ESL) design tools and languages are demanded to enable the creation of high level executable functional descriptions of the system being designed. In this context, the SystemC language has become the de facto standard to create executable models of electronic systems. SystemC provides resources to model electronic systems at a wide range of abstraction levels; from the Register Transfer Level (RTL) up to the high abstraction level Transaction Level Modeling (TLM). TLM has been proposed for early system definition, architecture exploration, software development and platform design.

The approaches based on SystemC–TLM partially solved the design productivity gap. However, as indicated in the last reports published by the International Technology Roadmap for Semiconductors (ITRS), the design productivity gap is still an open issue. Recently, the Model-driven Design (MDD) methodology was proposed in order to increase the level of abstraction above TLM and automate the processes that produce executable specifications of electronic systems. MDD captures all the information of the system being designed in visual models.

The main objective of this research work is to propose a methodology and a framework to assist in the architecture design and functional verification of complex electronic systems described using SystemC. This work makes several contributions to the modern electronic systems design and verification processes.

First, a methodology and a framework are proposed in order to apply Assertion-based verification (ABV) techniques to the functional SystemC–
TLM system models at a high abstraction level. Furthermore, a flexible Assertion Specification Language (ASL) is provided to enable verification–engineers to write high level system assertions easily.

Next, an approach is proposed to efficiently abstract and simplify the task of integrating MATLAB algorithms in a SystemC model. This approach enables building high abstraction executable models, where the architecture is described in SystemC–TLM and algorithms are described in MATLAB.

Finally, a MDD based methodology and a modelling framework have been proposed in order to translate electronic systems described using Systems Modeling Language (SysML) to SystemC–TLM heterogeneous executable models. The main goal of the MDD approach is to define rigorous semantics for SysML State Machine diagrams and Activity diagrams using Finite State Machine (FSM) and Synchronous Data Flow (SDF) Models of Computation (MoC), respectively. The proposed approach is supported by a new tool called Automatic SysML to SystemC Translator (ASSYST) that automates the SystemC–TLM source code generation.

The proposed methodologies and frameworks have been successfully applied to the CONFIDENCE European project, which has been led and coordinated by Centro de Estudios e Investigaciones Técnicas de Gipuzkoa (CEIT). The CONFIDENCE project has been employed to validate the contributions of this thesis dissertation and demonstrate their benefits.

An integrated, powerful and flexible development environment has been created in this research work in order to simulate and verify electronic systems starting from a very high abstraction level. The system architecture design, the algorithm tuning and the functional verification can be performed efficiently early in the development process by means the proposed contributions.
Contents

List of Figures xviii
List of Tables xix
List of Listings xxii
List of Abbreviations xxiii

1 Introduction 1
  1.1 Outline of the research work .................................... 2

2 State of the Art 5
  2.1 System Level Design Languages ................................. 6
    2.1.1 SystemC Design Language ................................ 7
  2.2 Transaction Level Modelling (TLM) .............................. 8
    2.2.1 SystemC Transaction Level Modelling (TLM) ............. 11
  2.3 High Abstraction Level Electronic System Development .... 13
  2.4 Verification of TLM Executable Models ......................... 14
    2.4.1 Simulation based methods ................................. 15
    2.4.2 Formal Methods .......................................... 16
    2.4.3 Assertion-based Verification (ABV) ....................... 17
      2.4.3.1 State of the art of ABV-based approaches .......... 19
  2.5 Algorithmic Models Integration in TLM Executable models .. 21
    2.5.1 State of the art of SystemC–MATLAB integration approaches ................. 22
  2.6 MDD for High Abstraction Level Electronic System Development ............ 23
    2.6.1 High Level Model Description ................................ 24
    2.6.2 State of the art of MDD-based approaches for electronic systems development ................. 25
4.2.1.7 Class Monitor ........................................ 73
4.2.1.8 Class XMLAssertionsRecorder ...................... 74
4.2.1.9 Class XMLAssertionsBuilder .......................... 74
4.2.2 Verification Framework integration with a SystemC Executable Model ................................. 74
4.2.3 System Model Verification Process ....................... 75
  4.2.3.1 Phase 1 ............................................. 75
  4.2.3.2 Phase 2 ............................................. 77
  4.2.3.3 Phase 3 ............................................. 78
4.2.4 XML Framework Interface ............................... 79
  4.2.4.1 Assertion Description XML file ..................... 79
  4.2.4.2 AssertionsManager XML configuration file ......... 82
  4.2.4.3 AssertionsManager XML report file ................. 82
4.3 Concluding Remarks ..................................... 83

5 Embedding Matlab in SystemC-TLM ......................... 85
  5.1 Matlab’s C API ........................................... 86
  5.2 MatlabEngine++ Implementation .......................... 89
    5.2.1 MatlabEngine++ Architecture ....................... 89
      5.2.1.1 Class MatlabSession ............................ 90
      5.2.1.2 Class MatlabSessionConfig ..................... 91
      5.2.1.3 Class MatlabManager ............................ 92
      5.2.1.4 Class MatlabVar_b ................................ 93
      5.2.1.5 Class MatlabVar ................................. 95
      5.2.1.6 Class MatlabVarRow ............................. 96
      5.2.1.7 Class MatlabVarColumn ......................... 96
      5.2.1.8 Class MatlabVarMatrix ......................... 96
    5.2.2 MatlabEngine++ Operation ............................ 97
  5.3 Concluding Remarks .................................... 100

6 Behavioural semantic formalization of SysML ................. 103
  6.1 System modelling with SysML ............................ 105
    6.1.1 Structural Modelling ................................ 105
    6.1.2 Behavioural Modelling ................................ 106
  6.2 Approach to Behavioural Semantic Mapping ............... 107
  6.3 Formal Semantics for SysML State Machine Diagrams ... 108
    6.3.1 Abstract syntax of State Machine Diagrams ........ 109
    6.3.2 Semantic Domain of State Machine Diagrams ........ 111
    6.3.3 Semantic Mapping of State Machine Diagrams ........ 112
  6.4 Formal Semantics for SysML Activity Diagrams ........... 113
6.4.1 Abstract syntax of Activity Diagrams . . . . . . . . 113
6.4.2 Semantic Domain of Activity Diagrams . . . . . . . 117
6.4.3 Semantic Mapping of Activity Diagrams . . . . . . . 124
6.5 ASSYST Modelling Methodology . . . . . . . . . . . . . . 129
  6.5.1 Phase 1: Modelling Functionality with Use Cases . . 129
  6.5.2 Phase 2A: Modelling Event–Based Behaviour with State Machines . 130
  6.5.3 Phase 2B: Modelling Flow–Based Behaviour with Activities . . . . . . 130
  6.5.4 Phase 3: Modelling Structure with Blocks . . . . . . 132
6.6 Concluding Remarks . . . . . . . . . . . . . . . . . . . . . 132

7 Automatic SysML to SystemC-TLM code generation 135
7.1 Proposed ASSYST Framework . . . . . . . . . . . . . . . 137
7.2 Implementing Behavioural Semantic Mapping . . . . . . . 138
  7.2.1 ASSYST metamodels definition . . . . . . . . . . . . 140
    7.2.1.1 FSM metamodel definition . . . . . . . . . . 141
    7.2.1.2 SDF metamodel definition . . . . . . . . . . 141
  7.2.2 ASSYST ATL model transformations specification . 142
    7.2.2.1 State Machine to FSM transformation . . . 143
    7.2.2.2 Activity to SDF transformation . . . . . . 147
7.3 Implementing SystemC-TLM Code Generation . . . . . . . 154
  7.3.1 ASSYST C++ library . . . . . . . . . . . . . . . . . . 156
    7.3.1.1 Structural Class . . . . . . . . . . . . . . . . 156
    7.3.1.2 Communication Classes . . . . . . . . . . . . 157
    7.3.1.3 XML Management Classes . . . . . . . . . . 159
    7.3.1.4 Behavioural Classes . . . . . . . . . . . . . . 162
    7.3.1.5 FSM Behavioural Classes . . . . . . . . . . 163
    7.3.1.6 SDF Behavioural Classes . . . . . . . . . . 164
  7.3.2 ASSYST Acceleo code generation . . . . . . . . . . . 167
  7.3.3 ASSYST SysML code generation . . . . . . . . . . . . 169
    7.3.3.1 Actor code generation . . . . . . . . . . . . 169
    7.3.3.2 DataType code generation . . . . . . . . . . 172
    7.3.3.3 Enumeration code generation . . . . . . . . . . 174
    7.3.3.4 Transaction code generation . . . . . . . . . 174
    7.3.3.5 UseCase code generation . . . . . . . . . . . 174
    7.3.3.6 Block code generation . . . . . . . . . . . . 176
    7.3.3.7 Model code generation . . . . . . . . . . . . 178
  7.3.4 ASSYST FSM behavioural code generation . . . . . . . 179
    7.3.4.1 State code generation . . . . . . . . . . . . 180
8 Case study

8.1 Assessment methodology .............................................. 188
8.2 CONFIDENCE Project .................................................. 190
8.3 CONFIDENCE system development with SAVY ......................... 192
  8.3.1 Level 0 UML model .................................................. 193
  8.3.2 Level 1 UML model .................................................. 194
  8.3.3 Level 2 UML model .................................................. 196
  8.3.4 UML to SAVY translation procedure .............................. 197
  8.3.5 Evaluation of the SAVY methodology .............................. 200
8.4 CONFIDENCE system verification ...................................... 202
  8.4.1 Level 0 verification ................................................. 203
     8.4.1.1 Level 0 assertion specification ............................... 204
  8.4.2 Level 1 verification ................................................. 205
     8.4.2.1 Level 1 assertion specification ............................... 208
  8.4.3 Level 2 verification ................................................. 210
     8.4.3.1 Level 2 assertion specification ............................... 214
  8.4.4 Evaluation of the verification framework ........................ 215
8.5 CONFIDENCE system algorithmic design .............................. 218
  8.5.1 Level 3 algorithmic design ....................................... 219
  8.5.2 Evaluation of the Matlab framework .............................. 224
8.6 CONFIDENCE system development with ASSYST ......................... 226
  8.6.1 CONFIDENCE SysML model ....................................... 226
  8.6.2 Modelling Functionality with Use Cases .......................... 229
     8.6.2.1 User Actor ..................................................... 229
     8.6.2.2 AlarmReceiver Actor ........................................ 230
     8.6.2.3 SystemSetup Use Case ........................................ 230
     8.6.2.4 ProcessReconstruction Use Case ............................. 230
     8.6.2.5 ProcessSituation Use Case .................................. 230
     8.6.2.6 CallAlarmReceiver Use Case .................................. 231
  8.6.3 Modelling Event–Based Behaviour with State Machines ....... 231
     8.6.3.1 SystemSetup State Machine Diagram ......................... 231
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.6.3.2 CallAlarmReceiver State Machine Diagram</td>
<td>240</td>
</tr>
<tr>
<td>8.6.4 Modelling Flow-Based Behaviour with Activities</td>
<td>242</td>
</tr>
<tr>
<td>8.6.4.1 ProcessReconstruction activity diagram</td>
<td>242</td>
</tr>
<tr>
<td>8.6.4.2 ProcessSituation activity diagram</td>
<td>244</td>
</tr>
<tr>
<td>8.6.5 Modelling Structure with Blocks</td>
<td>245</td>
</tr>
<tr>
<td>8.6.5.1 Block Definition Diagram of Level 0 system model</td>
<td>245</td>
</tr>
<tr>
<td>8.6.6 Automatic SysML to SystemC-TLM code generation</td>
<td>247</td>
</tr>
<tr>
<td>8.6.7 CONFIDENCE SystemC-TLM Simulation</td>
<td>251</td>
</tr>
<tr>
<td>8.6.8 Assessment of the proposed System Development Framework</td>
<td>253</td>
</tr>
<tr>
<td>8.7 Concluding Remarks</td>
<td>255</td>
</tr>
<tr>
<td>9 Conclusions and areas for further research</td>
<td>257</td>
</tr>
<tr>
<td>9.1 Conclusions</td>
<td>258</td>
</tr>
<tr>
<td>9.2 Areas for further research</td>
<td>263</td>
</tr>
<tr>
<td>References</td>
<td>265</td>
</tr>
</tbody>
</table>

### A Foundations of Model-driven Initiatives

A.1 Model, Metamodel and Meta-metamodel                                  | 286  |
| A.1.1 Four-layer Metamodel architecture                                | 287  |
| A.2 MBE, MDE, MDD and MDA                                              | 288  |
| A.2.1 Model-based Engineering (MBE)                                   | 289  |
| A.2.2 Model-driven Engineering (MDE)                                  | 289  |
| A.2.3 Model-driven Design (MDD)                                       | 289  |
| A.2.4 Model-driven Architecture (MDA)                                 | 290  |
| A.3 Model Transformations                                              | 291  |
| A.4 Visual Modelling Languages                                         | 293  |
| A.4.1 Unified Modeling Language (UML)                                  | 293  |
| A.4.2 Systems Modelling Language (SysML)                              | 295  |
| A.5 Syntax and Semantics                                               | 298  |
| A.6 Modelling Tools                                                    | 298  |
| A.6.1 Eclipse                                                          | 298  |
| A.6.2 TOPCASED                                                         | 302  |
| A.6.3 IBM Rational Rhapsody                                            | 303  |
| A.6.4 YAKINDU                                                          | 303  |
B  SAVY Library 305
   B.1 System Behaviour Capture as an Executable Model Using SAVY ............ 306
   B.2 Entity Modelling Using SAVY .................................................. 307
      B.2.1 Entity Architecture Pattern ............................................. 307
      B.2.2 State Machine Pattern ..................................................... 309
   B.3 Actor Modelling Using SAVY .................................................. 310
      B.3.1 Actor Architecture Pattern .............................................. 310
      B.3.2 File Format for Transactions ........................................... 312

C  CONFIDENCE Project 313

D  Research Results 319
   D.1 International Journal Papers .................................................. 321
      IEEE TSE 2013 – I ................................................................. 323
      IEEE TSE 2013 – II ................................................................. 337
   D.2 International Conference Papers ............................................ 353
      CENICS 2009 ................................................................. 355
      DCIS 2009 ................................................................. 363
      FDL 2006 ................................................................. 371
   D.3 National Conference Papers .................................................... 377
      JCRA 2009 ................................................................. 379
      TECNUN 2012 ................................................................. 391
# List of Figures

## Chapter 2

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>ESL design flow.</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>TLM abstraction levels.</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>TLM design flow.</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>Executable system model refinement.</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>Simulation based environment.</td>
<td>15</td>
</tr>
<tr>
<td>2.6</td>
<td>Assertion-based verification environment.</td>
<td>17</td>
</tr>
<tr>
<td>2.7</td>
<td>Verification of the different SystemC executable models.</td>
<td>18</td>
</tr>
<tr>
<td>2.8</td>
<td>Algorithmic executable model creation.</td>
<td>21</td>
</tr>
<tr>
<td>2.9</td>
<td>Executable models generation from visual models.</td>
<td>25</td>
</tr>
</tbody>
</table>

## Chapter 4

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Integration of ABV in a SystemC–TLM Flow.</td>
<td>46</td>
</tr>
<tr>
<td>4.2</td>
<td>Transaction operations.</td>
<td>48</td>
</tr>
<tr>
<td>4.3</td>
<td>ASL language architecture.</td>
<td>50</td>
</tr>
<tr>
<td>4.4</td>
<td>ASL property.</td>
<td>52</td>
</tr>
<tr>
<td>4.5</td>
<td>Relationship between ASL architecture levels.</td>
<td>53</td>
</tr>
<tr>
<td>4.6</td>
<td>Finite-state machine associated to propositions.</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>(a) Positive proposition.</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>(b) Negative proposition.</td>
<td>55</td>
</tr>
<tr>
<td>4.7</td>
<td>Property example.</td>
<td>56</td>
</tr>
<tr>
<td>4.8</td>
<td>Several property instances.</td>
<td>57</td>
</tr>
<tr>
<td>4.9</td>
<td>Non-overlapping mode.</td>
<td>58</td>
</tr>
<tr>
<td>4.10</td>
<td>Overlapping mode steps.</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>(a) Step 1.</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>(b) Step 2.</td>
<td>59</td>
</tr>
</tbody>
</table>
(c) Step 3. .................................................. 59
(d) Step 4. .................................................. 59
4.11 CPU-Memory Example. ............................. 64
   (a) Memory write operation. ......................... 64
   (b) Memory read operation. ......................... 64
   (c) Check memory state operation. ................. 64
4.12 Verification framework application example. .... 67
4.13 Verification Framework software architecture. .... 68
4.14 Verification Framework class diagram. ........... 69
4.15 DUTEvent class. .................................... 70
4.16 Proposition class. .................................. 70
4.17 Property class. .................................... 72
4.18 Assertion class. .................................... 72
4.19 AssertionManager class. .......................... 73
4.20 Monitor class. ....................................... 74
4.21 XMLAssertionsRecorder class. .................... 74
4.22 XMLAssertionsBuilder class. ..................... 75
4.23 AssertionsManager verification process. ......... 76
4.24 Event evaluation process. .......................... 77

Chapter 5

5.1 Integration of MATLAB in a TLM Flow. ........... 87
5.2 MatlabEngine++ software architecture. ............ 89
5.3 MatlabEngine++ class diagram. ..................... 90
5.4 MatlabSession class. ................................ 91
5.5 MatlabSessionConfig class. ........................ 92
5.6 MatlabManager class. ................................ 93
5.7 MatlabVar_b class. .................................. 94
5.8 Synchronization State Machine. .................... 94
5.9 MatlabVar class. .................................... 96
5.10 SystemC–MatlabEngine++ integration example. .... 97

Chapter 6

6.1 Simplified abstract syntax of the BehaviorStateMachine package. .................. 109
6.2 Abstract syntax of the BasicActivities package. ..... 114
6.3 SDF graph example. ................................. 120
## Chapter 7

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4</td>
<td>Intuitive Activity to SDF graph mapping.</td>
<td>124</td>
</tr>
<tr>
<td>7.1</td>
<td>Proposed ASSYST workflow.</td>
<td>138</td>
</tr>
<tr>
<td>7.2</td>
<td>Model-to-model transformation.</td>
<td>139</td>
</tr>
<tr>
<td>7.3</td>
<td>FSM and SDF models generation workflow.</td>
<td>139</td>
</tr>
<tr>
<td>7.4</td>
<td>FSM metamodel diagram.</td>
<td>141</td>
</tr>
<tr>
<td>7.5</td>
<td>SDF metamodel diagram.</td>
<td>142</td>
</tr>
<tr>
<td>7.6</td>
<td>SystemC-TLM code generation workflow.</td>
<td>155</td>
</tr>
<tr>
<td>7.7</td>
<td>ASSYST C++ library class diagram.</td>
<td>156</td>
</tr>
<tr>
<td>7.8</td>
<td>Block class.</td>
<td>157</td>
</tr>
<tr>
<td>7.9</td>
<td>Transaction class.</td>
<td>157</td>
</tr>
<tr>
<td>7.10</td>
<td>TransactionsSocket class.</td>
<td>158</td>
</tr>
<tr>
<td>7.11</td>
<td>TransactionFactory class.</td>
<td>159</td>
</tr>
<tr>
<td>7.12</td>
<td>XMLTransactionReader class.</td>
<td>160</td>
</tr>
<tr>
<td>7.13</td>
<td>XMLTransactionWriter class.</td>
<td>161</td>
</tr>
<tr>
<td>7.14</td>
<td>UseCase class.</td>
<td>162</td>
</tr>
<tr>
<td>7.15</td>
<td>Behavior class.</td>
<td>162</td>
</tr>
<tr>
<td>7.16</td>
<td>State class.</td>
<td>163</td>
</tr>
<tr>
<td>7.17</td>
<td>Transition class.</td>
<td>163</td>
</tr>
<tr>
<td>7.18</td>
<td>FiniteStateMachine class.</td>
<td>164</td>
</tr>
<tr>
<td>7.19</td>
<td>Node class.</td>
<td>164</td>
</tr>
<tr>
<td>7.20</td>
<td>Arc class.</td>
<td>165</td>
</tr>
<tr>
<td>7.21</td>
<td>SynchronousDataFlow class.</td>
<td>166</td>
</tr>
<tr>
<td>7.22</td>
<td>Model-to-Text transformation.</td>
<td>168</td>
</tr>
<tr>
<td>7.23</td>
<td>Actor Acceleo generation.</td>
<td>172</td>
</tr>
<tr>
<td>7.24</td>
<td>Actor architecture.</td>
<td>172</td>
</tr>
<tr>
<td>7.25</td>
<td>DataType Acceleo generation.</td>
<td>173</td>
</tr>
<tr>
<td>7.26</td>
<td>Enumeration Acceleo generation.</td>
<td>174</td>
</tr>
<tr>
<td>7.27</td>
<td>Transaction Acceleo generation.</td>
<td>175</td>
</tr>
<tr>
<td>7.28</td>
<td>UseCase Acceleo generation.</td>
<td>175</td>
</tr>
<tr>
<td>7.29</td>
<td>Block Acceleo generation.</td>
<td>176</td>
</tr>
<tr>
<td>7.30</td>
<td>Example of block interface bindings.</td>
<td>177</td>
</tr>
<tr>
<td>7.31</td>
<td>Model Acceleo generation.</td>
<td>178</td>
</tr>
<tr>
<td>7.32</td>
<td>State Acceleo generation.</td>
<td>180</td>
</tr>
<tr>
<td>7.33</td>
<td>Transition Acceleo generation.</td>
<td>181</td>
</tr>
<tr>
<td>7.34</td>
<td>FSM Acceleo generation.</td>
<td>181</td>
</tr>
<tr>
<td>7.35</td>
<td>Arc Acceleo generation.</td>
<td>182</td>
</tr>
</tbody>
</table>
Chapter 8

8.1 Assessment methodology. ........................... 189
   (a) SAVY Framework Assessment. ................. 189
   (b) Verification Framework Assessment. .......... 189
   (c) MATLAB Framework Assessment. ............... 189
   (d) ASSYST Framework Assessment. ............... 189
8.2 CONFIDENCE system. ................................ 190
   (a) CONFIDENCE indoors. ............................ 190
   (b) CONFIDENCE outdoors. ........................... 190
8.3 SAVY system design flow. .......................... 193
8.4 CONFIDENCE Level 0 Use Case Diagram. .......... 194
8.5 CONFIDENCE Level 1 system components. ....... 195
8.6 Base Station Level 1 Use Case Diagram. ....... 196
8.7 UML–to–SAVY transformation. ..................... 198
8.8 CONFIDENCE system design flow. ................. 201
8.9 CONFIDENCE Level 0 verification. ............... 203
8.10 CONFIDENCE Level 1 verification. ............... 207
8.11 CONFIDENCE Level 2 verification. ............... 213
8.12 CONFIDENCE Level 3 algorithmic executable model. 220
8.13 SysML model creation. ............................ 227
   (a) New SysML model. ............................... 227
   (b) New SysML model options. ..................... 227
8.14 SysML model created. ............................. 228
8.15 SysML model outline. .............................. 228
8.16 CONFIDENCE use case diagram. .................. 229
8.17 SystemSetup properties. .......................... 232
8.18 CONFIDENCE SystemSetup TagInstallation state machine diagram. 234
8.19 CONFIDENCE SystemSetup CallProtocolConfiguration state machine diagram. 236
8.20 CONFIDENCE SystemSetup ConfigureAlarms state machine diagram. 239
8.21 CONFIDENCE CallAlarmReceiver state machine diagram. 241
8.22 CONFIDENCE ProcessReconstruction activity diagram. 243
8.23 CONFIDENCE ProcessSituation activity diagram. 244
List of Figures xvii

8.24 Confidence System block. 246
   (a) Confidence System block outline. 246
   (b) Confidence System SysML block. 246
8.25 CONFIDENCE Level 0 datatypes. 247
8.26 ASSYST launch. 248
8.27 ASSYST dialogs. 249
   (a) ASSYST new C++ project dialog. 249
   (b) ASSYST progress dialog. 249
8.28 ASSYST FSM and SDF models generation. 250
   (a) CONFIDENCE Level 0 FSM model. 250
   (b) CONFIDENCE Level 0 SDF model. 250
8.29 ASSYST generation process. 251
   (a) ASSYST models created. 251
   (b) ASSYST sources created. 251
8.30 ASSYST Framework Assessment. 253

Appendix A
A.1 Four-layer Metamodel architecture. 288
A.2 Relationship between model-driven acronyms. 289
A.3 MDA modelling methodology. 291
A.4 UML diagrams. 294
A.5 Overview of the SysML/UML interrelationship. 296
A.6 SysML diagrams. 297
A.7 Eclipse Modeling Project mindmap. 299

Appendix B
B.1 UML based system behaviour capture. 306
B.2 SAVY entity model architecture pattern. 308
B.3 Event identification in SAVY state machines. 310
B.4 SAVY actor model architecture pattern. 311

Appendix C
C.1 CONFIDENCE system. 315
   (a) CONFIDENCE indoors. 315
   (b) CONFIDENCE outdoors. 315
C.2 Detailed CONFIDENCE system description 317
(a) Confidence Base Station (BS). . . . . . . . . . . . . . 317
(b) Confidence Portable Device (PD). . . . . . . . . . . . . . . 317
## List of Tables

### Chapter 2

2.1 Comparison of works in the literature addressing MDD methods for electronic systems development. 32

### Chapter 7

7.1 StateMachine-to-FSM tranformation rules. 144
7.2 Activity-to-SDF tranformation rules. 148
7.3 ASSYST C++ data type generation. 173

### Chapter 8

8.1 The LoC necessaries to create TL system models from the UML system descriptions. 202
8.2 Verification Framework Integration Cost. 215
8.3 Verification Framework Performance. 218
8.4 Cost of Modifying SystemC Level 2 Model. 225
8.5 Level 3 System Model Performance Analysis. 225
8.6 Assessment of the ASSYST framework. 254
## List of Listings

### Chapter 4

| 4.1 System Assertions XML structure. | 80 |
| 4.2 XML Assertion example. | 81 |
| 4.3 Verification engine XML configuration file. | 82 |
| 4.4 XML Report file structure. | 83 |

### Chapter 5

| 5.1 *MatlabSession* management within *sc_main.cpp*. | 98 |
| 5.2 Algorithm integration within a SystemC module. | 99 |

### Chapter 7

| 7.1 ATL transformation file header for FSM generation. | 143 |
| 7.2 Model\textsubscript{to}FSMDomainModel ATL rule. | 144 |
| 7.3 StateMachine\textsubscript{to}FSM ATL rule. | 145 |
| 7.4 State\textsubscript{to}State ATL rule. | 146 |
| 7.5 Pseudostate\textsubscript{to}State ATL rule. | 146 |
| 7.6 Transition\textsubscript{to}Transition ATL rule. | 147 |
| 7.7 ATL transformation file header for SDF generation. | 147 |
| 7.8 Model\textsubscript{to}SDFDomainModel ATL rule. | 149 |
| 7.9 Activity\textsubscript{to}SDF ATL rule. | 149 |
| 7.10 ActivityNode\textsubscript{to}Node ATL rule. | 150 |
| 7.11 ControlFlow\textsubscript{to}Arc ATL rule. | 151 |
| 7.12 PinToPinObjectFlow\textsubscript{to}Arc ATL rule. | 152 |
| 7.13 APNToPinObjectFlow\textsubscript{to}Arc ATL rule. | 153 |
| 7.14 PinToAPNObjectFlow\textsubscript{to}Arc ATL rule. | 154 |
| 7.15 ASSYST input stimuli XML file. | 161 |
List of Listings

7.16 Acceleo template file “generate_actor.mtl”.......................... 170
7.17 Header file template details of “generate_actor.mtl”........... 171

Chapter 8

8.1 Monitors creation within ConfidenceLevel0.cpp .................. 204
8.2 Monitors and AssertionManager creation within sc_main .......... 205
8.3 CONFIDENCE level 0 assertion example............................... 206
8.4 Functional devices creation within sc_main ......................... 208
8.5 Monitors creation within BSLevel1.................................... 208
8.6 Monitors and AssertionManager creation within sc_main ........... 209
8.7 CONFIDENCE level 1 IsIndoor global variable definition .......... 209
8.8 CONFIDENCE level 1 assertions managing IsIndoor............... 211
8.9 CONFIDENCE level 1 assertion example............................... 212
8.10 Monitors and AssertionManager creation within sc_main .......... 214
8.11 CONFIDENCE level 2 assertion example............................... 216
8.12 CONFIDENCE level 2 assertion example............................... 217
8.13 MatlabEngine++ management within sc_main ....................... 221
8.14 Set parameters in MATLAB for Tag and BS operation............. 222
8.15 Ranging parameters in Tag............................................. 223
8.16 Ranging algorithm in Tag.............................................. 223
8.17 Positioning parameters in BS–L..................................... 224
8.18 Positioning algorithm in BS–L ....................................... 224
8.19 Simulation example...................................................... 252
List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABV</td>
<td>Assertion-based verification</td>
</tr>
<tr>
<td>ACK</td>
<td>Acknowledgement</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AR</td>
<td>Alarm Receiver</td>
</tr>
<tr>
<td>ASL</td>
<td>Assertion Specification Language</td>
</tr>
<tr>
<td>ASSYST</td>
<td>Automatic SysML to SystemC Translator</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>ATL</td>
<td>ATL Transformation Language</td>
</tr>
<tr>
<td>AWGN</td>
<td>Additive White Gaussian Noise</td>
</tr>
<tr>
<td>BDD</td>
<td>Block Definition Diagram</td>
</tr>
<tr>
<td>BS</td>
<td>Base Station</td>
</tr>
<tr>
<td>BS–I</td>
<td>BS Interpretation</td>
</tr>
<tr>
<td>BS–L</td>
<td>BS Localization</td>
</tr>
<tr>
<td>BS–R</td>
<td>BS Reconstruction</td>
</tr>
<tr>
<td>BS–SI</td>
<td>BS System Interface</td>
</tr>
<tr>
<td>CEIT</td>
<td>Centro de Estudios e Investigaciones Técnicas de Gipuzkoa</td>
</tr>
<tr>
<td>CIM</td>
<td>Computation Independent Model</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
</tbody>
</table>
**List of Abbreviations**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTL</td>
<td>Computational Tree Logic</td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model</td>
</tr>
<tr>
<td>DoW</td>
<td>Description of Work</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain-specific Language</td>
</tr>
<tr>
<td>DUT</td>
<td>Device Under Test</td>
</tr>
<tr>
<td>DUV</td>
<td>Device Under Verification</td>
</tr>
<tr>
<td>EBNF</td>
<td>Extended Backus–Naur Form</td>
</tr>
<tr>
<td>ECC</td>
<td>Event Communication Channel</td>
</tr>
<tr>
<td>EDA</td>
<td>Electronic Design Automation</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modeling Framework</td>
</tr>
<tr>
<td>EMFT</td>
<td>Eclipse Modeling Framework Technology</td>
</tr>
<tr>
<td>EMP</td>
<td>Eclipse Modeling Project</td>
</tr>
<tr>
<td>EPL</td>
<td>Eclipse Public License</td>
</tr>
<tr>
<td>ESL</td>
<td>Electronic System Level</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out</td>
</tr>
<tr>
<td>FP7</td>
<td>Seventh Framework Programme</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>GMF</td>
<td>Graphical Modeling Framework</td>
</tr>
<tr>
<td>GPS</td>
<td>Global Positioning System</td>
</tr>
<tr>
<td>GSL</td>
<td>GNU Scientific Library</td>
</tr>
<tr>
<td>HW</td>
<td>Hardware</td>
</tr>
<tr>
<td>ICT</td>
<td>Information and Communications Technology</td>
</tr>
<tr>
<td>IR</td>
<td>Impulse–Radio</td>
</tr>
<tr>
<td>IBD</td>
<td>Internal Block Diagram</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>INCOSE</td>
<td>International Council on Systems Engineering</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>ITRS</td>
<td>International Technology Roadmap for Semiconductors</td>
</tr>
<tr>
<td>LoC</td>
<td>Lines of Code</td>
</tr>
<tr>
<td>LRM</td>
<td>Language Reference Manual</td>
</tr>
<tr>
<td>LTL</td>
<td>Linear Temporal Logic</td>
</tr>
<tr>
<td>MARTE</td>
<td>Modeling and Analysis of Real-time Embedded Systems</td>
</tr>
<tr>
<td>MBE</td>
<td>Model-based Engineering</td>
</tr>
<tr>
<td>MDA</td>
<td>Model-driven Architecture</td>
</tr>
<tr>
<td>MDD</td>
<td>Model-driven Design</td>
</tr>
<tr>
<td>MDE</td>
<td>Model-driven Engineering</td>
</tr>
<tr>
<td>MDT</td>
<td>Model Development Tools</td>
</tr>
<tr>
<td>MoC</td>
<td>Model of Computation</td>
</tr>
<tr>
<td>MOF</td>
<td>Meta-Object Facility</td>
</tr>
<tr>
<td>MOFM2T</td>
<td>MOF Model to Text Transformation Language</td>
</tr>
<tr>
<td>MTL</td>
<td>Model-to-Text Language</td>
</tr>
<tr>
<td>MTP</td>
<td>Monitorizable Transactional Port</td>
</tr>
<tr>
<td>M2M</td>
<td>Model-to-Model</td>
</tr>
<tr>
<td>M2T</td>
<td>Model-to-Text</td>
</tr>
<tr>
<td>oAW</td>
<td>openArchitectureWare</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Definition</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>OSCI</td>
<td>Open SystemC Initiative</td>
</tr>
<tr>
<td>PD</td>
<td>Portable Device</td>
</tr>
<tr>
<td>PD–I</td>
<td>PD Interpretation</td>
</tr>
<tr>
<td>PD–L</td>
<td>PD Localization</td>
</tr>
<tr>
<td>PD–SI</td>
<td>PD System Interface</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform Independent Model</td>
</tr>
<tr>
<td>PN</td>
<td>Petri Net</td>
</tr>
<tr>
<td>PSL</td>
<td>Property Specification Language</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform Specific Model</td>
</tr>
<tr>
<td>QVT</td>
<td>Query/View/Transformation</td>
</tr>
<tr>
<td>RFP</td>
<td>Request For Proposal</td>
</tr>
<tr>
<td>RLS</td>
<td>Recursive Least Square</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>RTT</td>
<td>Round-Trip-Time</td>
</tr>
<tr>
<td>SAM</td>
<td>System Architectural Model</td>
</tr>
<tr>
<td>SCV</td>
<td>SystemC Verification</td>
</tr>
<tr>
<td>SDF</td>
<td>Synchronous Data Flow</td>
</tr>
<tr>
<td>SNR</td>
<td>Signal-to-noise ratio</td>
</tr>
<tr>
<td>SoC</td>
<td>System on Chip</td>
</tr>
<tr>
<td>SVA</td>
<td>System Verilog Assertions</td>
</tr>
<tr>
<td>SW</td>
<td>Software</td>
</tr>
<tr>
<td>SysML</td>
<td>Systems Modeling Language</td>
</tr>
<tr>
<td>TE</td>
<td>Transactional Event</td>
</tr>
<tr>
<td>TI</td>
<td>Traffic Injector</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>TII</td>
<td>Transactional Input Interface</td>
</tr>
<tr>
<td>TLM</td>
<td>Transaction Level Modeling</td>
</tr>
<tr>
<td>TMF</td>
<td>Textual Modeling Framework</td>
</tr>
<tr>
<td>TOI</td>
<td>Transactional Output Interface</td>
</tr>
<tr>
<td>TR</td>
<td>Traffic Receiver</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>UWB</td>
<td>Ultra-Wideband</td>
</tr>
<tr>
<td>VHDL</td>
<td>Very-high-speed integrated circuits Hardware Description Language</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Metadata Interchange</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
<tr>
<td>XTG</td>
<td>XML Transaction Generator</td>
</tr>
<tr>
<td>XTR</td>
<td>XML Transaction Recorder</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

| Contents |
|-----------------|------------------|
| 1.1 Outline of the research work | 2 |

In the last decades, the development life cycles of modern electronic systems have been shortened drastically due to the increasing market demands. In turn, product functionality and consequently the complexity of electronic systems are rapidly increasing. Modern electronic systems have arrived to a degree of complexity that makes traditional design methodologies inefficient and error-prone. This productivity loss is known as the design productivity gap, which is produced by the inability of Electronic Design Automation (EDA) tools to efficiently handle the design complexity growth. The International Technology Roadmap for Semiconductors (ITRS) suggested three foundations in order to address the design productivity challenge: increasing the level of abstraction, automation processes and tools, and producing executable specifications. In order to reduce the development time and improve the product quality, new solutions and design paradigms are needed to handle present electronic systems.

SystemC language was proposed to close the gap between system requirements specification and system design. SystemC has become the de facto standard to create executable models of electronic systems. The
Transaction Level Modeling (TLM) supported by SystemC has been proposed for the creation of high abstraction level executable models. TLM offers a new design methodology for digital electronic systems at a higher abstraction level above Register Transfer Level (RTL). These high abstraction level SystemC–TLM models are fast to build and they provide a very high speed of simulation.

In order to increase the level of abstraction above TLM and automate the processes that produce executable specifications of electronic systems, new approaches and methodologies were proposed recently: model-driven initiatives. Model-driven Design (MDD) is the most well-known approach among model-driven initiatives. MDD aims to capture in visual models such as Unified Modeling Language (UML) or Systems Modeling Language (SysML) all the information of the system being designed. MDD approaches can be used to automatically generate functional SystemC models at TLM level from visual modelling languages. As a result, the error-prone manual SystemC–TLM coding process can be improved. Thus, the development time of the device being designed is reduced significantly.

Verification is also a key process of any electronic system development work. Functional verification consumes about 75% of the resources in a typical electronic system design process. As a result, an enhancement in the verification process may cause a considerable improvement in the overall development process.

The objective of this research work is to enhance the electronic system design and verification methodologies in order to improve the productivity of the modern electronic device design and verification processes. This research work shall propose novel system methodologies and frameworks supported by software tools for functional system models design and verification.

1.1 OUTLINE OF THE RESEARCH WORK

This PhD dissertation is organized as follows:

Chapter 2: State of the Art

First, this chapter presents the state of the art of electronic systems design and verification based on SystemC–TLM executable models. This
1.1 Outline of the research work

Chapter also introduces the application of the MDD methodology for the design of modern electronic systems.

Chapter 3: Objectives

This chapter decomposes the main objective of this PhD dissertation into partial objectives and defines the scope of the research work.

Chapter 4: An ABV framework for SystemC-TLM

This chapter presents a Assertion-based verification (ABV) framework that enables the verification team to write early design assertions simply and fast, at a very high level of abstraction. Furthermore, this chapter proposes an assertion specification language to support the ABV framework. The proposed framework is very flexible and can be attached to any SystemC–TLM model.

Chapter 5: Embedding Matlab in SystemC-TLM

This chapter proposes a framework that aims to integrate MATLAB algorithms into a TLM–SystemC executable system models. The objective is to facilitate verifying that the different algorithms cooperate correctly to achieve the high–level functionalities and enable a robust design–flow with a reduced development cost. The chapter proposes an abstraction layer for the MATLAB Application Programming Interface (API) in order to simplify the interaction between the MATLAB Engine and the SystemC model. Thus, the resulting code for the SystemC–MATLAB model is clear; and the system–design team can focus on the system description.

Chapter 6: Behavioural semantic formalization of SysML

This chapter formalizes the semantics of SysML State Machine diagrams and Activity diagrams using Model of Computations (MoCs) as semantic domains. Rigorous semantic mappings are provided in order to translate the abstract syntaxes of State Machine diagrams and Activity diagrams to Finite State Machine (FSM) and Synchronous Data Flow (SDF) MoCs, respectively. Furthermore, this chapter defines a modelling methodology to create SysML models.

The behavioural semantic formalization of SysML State Machine diagrams and Activity diagrams, and the modelling methodology proposed in
Chapter 1: Introduction

This chapter establishes the theoretical background for the practical MDD framework implementation proposed in Chapter 7.

**Chapter 7: Automatic SysML to SystemC-TLM code generation**

In this chapter, a MDD approach to translate electronic systems described using SysML to SystemC-TLM heterogeneous executable models is proposed. This generation process is supported by a new tool called Automatic SysML to SystemC Translator (ASSYST). The main goal is to define rigorous semantics for SysML State Machine diagrams and Activity diagrams using FSM and SDF Models of Computation (MoCs), respectively. The proposed approach is supported by Eclipse-based plug-ins that automate the SystemC-TLM source code generation by means of Model-to-Model (M2M) transformation techniques and Model-to-Text (M2T) code generation processes.

**Chapter 8: Case study**

This chapter presents the assessment of the verification framework, the MATLAB framework, and the ASSYST framework that supports a SysML to SystemC-TLM development flow. The European CONFIDENCE project, developed within the Seventh Framework Programme (FP7), has been employed to validate the design and verification frameworks described in this thesis dissertation.

**Chapter 9: Conclusions and areas for further research**

The main conclusions and areas for further research are put forward in Chapter 9.
Modern electronic systems are increasing their complexity to meet market requirements. However, present development tools can not handle this complexity growth efficiently. This productivity loss produces the so-called design productivity gap [1], as documented by several editions of the Design Report [2] published by the International Technology Roadmap for Semiconductors (ITRS). The ITRS suggested three foundations in order to address the design productivity challenge: increasing the level of abstraction, automation processes and tools, and producing executable specifications. Therefore, in order to reduce the development time and improve
the product quality, new languages and design paradigms are needed to handle present electronic systems.

This chapter describes the state of the art of the techniques and methodologies proposed for the design and verification of modern electronic systems.

### 2.1 SYSTEM LEVEL DESIGN LANGUAGES

In the late 90s, several Electronic System Level (ESL) design languages based on C/C++ were proposed in order to increase the abstraction level from Register Transfer Level (RTL) to the system level and enable the design of executable system models. The ESL design languages close the gap between system requirements specification and RTL designs, as depicted in Figure 2.1. Following the ESL design flow, the design team creates executable system models described using ESL languages from the system requirements specification. System models described using ESL languages have higher abstraction level, are faster to build and provide faster simulation speeds than RTL models. Once the desired functionality is achieved with the executable models, the ESL model is translated manually or au-

---

**Figure 2.1:** ESL design flow.
2.1 System Level Design Languages

Automatically (using high level synthesis tools) to a RTL model. Then, the traditional RTL-to-silicon design flow is followed.

SystemC [3], Handel-C [4] and SpecC [5] are some of the most widely-known system level languages. SystemC has become the de facto standard to create executable models of electronic systems.

2.1.1 SYSTEMC DESIGN LANGUAGE

In 1999, the Open SystemC Initiative (OSCI) foundation was announced: a consortium of companies dedicated to the development of the SystemC library. The first main objective of SystemC was to promote the exchange of system level Intellectual Property (IP) models using a C++ programming environment. SystemC v1.0 was born in 2000 from this first idea as a C++ open source and free class library providing modelling structures analogous to the hardware modelling structures provided by RTL languages such as Verilog [6] or Very-high-speed integrated circuits Hardware Description Language (VHDL) [7]. C++ programming language was chosen due to its suitable abstraction level, hardware-software integration, and performance leading to an efficient and solid environment to model complex systems.

SystemC took a major upgrade in 2002 with the publication of the SystemC v2.0 library and it became widely used. In this version the library implemented several new constructs for systems design, such as communication channels and events. The popularity of SystemC led the OSCI foundation to prepare a Language Reference Manual (LRM) in order to submit it for Institute of Electrical and Electronics Engineers (IEEE) standard.

The IEEE 1666 Standard SystemC LRM [8] was approved on December 12, 2005. Thanks to the quick approval of the IEEE 1666 Standard in only 8 months, SystemC took an important step towards its establishment in the System on Chip (SoC) design industry. Proof of this is that leading edge companies from all over the world such as, AMD, ARM, Cadence, Forte Design Systems, Freescale, Intel, Mentor Graphics, NXP, Qualcomm, ST Microelectronics, Synopsys, and Texas Instruments, use and support the SystemC initiative.
The SystemC library continued evolving through several versions until the launch of the SystemC v2.2 library in 2007.

Accellera association and the OSCI joined in 2011 to form Accellera Systems Initiative [3] organization, to address the needs of the system and semiconductor designers who must find new and smarter ways to create and produce increasingly complex chips. The objective of the new organization was to evolve to create more comprehensive standards that benefit the global electronic design community. As a result of this effort, Accellera announced in 2012 the release of the SystemC v2.3 library, fully compatible with the last update of the IEEE 1666 Standard [9].

Today, SystemC provides a foundation to model hardware and software of a system based on a single language. The SystemC core language provides fundamental system modelling components, for example, modules, ports and channels. Moreover, SystemC enables the simulation of concurrent processes. Although SystemC provides language constructs to represent RTL models, the main reason for using the language is to work at higher abstraction levels than RTL [10].

2.2 TRANSACTION LEVEL MODELLING (TLM)

Transaction Level Modeling (TLM) offers a new design methodology for digital electronic systems at a higher abstraction level above RTL. TLM is gaining acceptance for electronic systems thanks to faster design and verification times [11].

In TLM, systems are composed of modules interconnected by communication channels. The behaviour of the modules is modelled by a set of concurrent processes, which can be executed in parallel. TLM modules and communication channels are bound together by means of ports in order to exchange data between them to perform the designed system behaviour.

TLM was proposed to abstract the interactions between modules. A transaction denotes the elemental information unit being exchanged between modules in a TLM model. A master or initiator is a module that initiates transactions in a system, while a slave or target is a module that receives and serves transactional requests. TLM uses function calls (read functions or write functions), rather than signals or wires, for transactional communications between modules. This lets the system designers concen-
2.2 Transaction Level Modelling (TLM)

Figure 2.2: TLM abstraction levels.

A TLM design approach supports several abstraction levels depending on the accuracy level required by the simulation of the system under development. Black and Donovan [10] defined different TLM abstractions according to the timing detail of both the functionality and the communication of the model, as depicted in Figure 2.2. The communication and functionality can be untimed (UT), approximately-timed (AT), or cycle-accurate (CA). Two corner cases are defined in Figure 2.2: System Architectural Model (SAM) and RTL model. The SAM represents a high abstraction level TLM model with untimed communication and untimed functionality. On the other hand, the RTL represents a model with cycle-accurate communication and cycle-accurate functionality.

In a TLM-based methodology executable system models are refined from high abstraction level models to RTL models, as shown in Figure 2.3. In modern complex electronic systems this gap is too large to be closed in a single step. The TLM-based methodology follows a top-bottom development approach, as described in [10].

During the first phase, the SAM is produced from the requirements specification of the system being developed. This initial executable model
Figure 2.3: TLM design flow.
is created by the system engineers and describes the system functionality, which is free of any Hardware (HW)/Software (SW) implementation details. The SAM is similar to the untimed functional model defined in [12]. Functional delays may be implemented in order to specify very basic timings for the simulation of the system simulation.

The SAM is then refined into the TLM executable models adding more detailed timing information both to the communications and the functionality. In this phase, several TLM models are produced supporting the timing abstractions proposed by Black and Donovan [10]. The primary objective of the TLM models is the SW/HW partitioning process that decides whether system functionalities described in the previous phase are designed as SW or HW. The TLM models also serve as a system platform for the SW design and development, and as a reference model (golden reference) for the hardware functional verification.

In the last phase, once the low level details are implemented in the TLM models, a RTL model is produced by means of a refinement process. Then, the system development follows the traditional RTL-to-silicon design flow previously described in Figure 2.1.

### 2.2.1 SYSTEMC TRANSACTION LEVEL MODELLING (TLM)

The OSCI developed the TLM-1.0 library [13] in order to support transaction level modelling system design using SystemC. Additionally, this group published the TLM-2.0 standard [14] that enhanced TLM-1.0.

The key element of the TLM-2.0 is the class that represents the transactions: the generic payload class. The generic payload is the class type offered by the TLM-2.0 standard for transaction objects passed through the core interfaces. It is specifically aimed at modelling memory-mapped buses and it includes some of the attributes found in typical memory-mapped bus protocols such as command, address, data, byte enables, single word transfers, burst transfers, streaming, and response status. The generic payload may also be used as the basis for modelling protocols other than memory-mapped buses since it includes an extension mechanism so that applications can add their own specialized attributes.
Chapter 2: State of the Art

The OSCI TLM-2.0 standard specifically defines two coding styles: \textit{loosely-timed} and \textit{approximately-timed}. However, \textit{untimed} style and \textit{cycle-accurate} modelling are also supported.

**Untimed coding style** The TLM-2.0 standard does not provide an explicit untimed coding style, however it may be easily represented using TLM-2.0 language constructs in order to simulate the concurrent execution and communication of multiple processes.

**Loosely-timed coding style** The loosely-timed coding style makes use of the blocking transport interface. This interface allows only two timing points to be associated with each transaction, corresponding to the call to and return from the blocking transport function. In the case of the base protocol, the first timing point marks the beginning of the request, and the second marks the beginning of the response. These two timing points could occur at the same simulation time or at different times.

**Approximately-timed coding style** This coding style is supported by the non-blocking transport interface, which is appropriate for the use cases of architectural exploration and performance analysis. The non-blocking transport interface provides for timing annotation and for multiple phases and timing points during the lifetime of a transaction.

For approximately-timed modelling, a transaction is broken down into multiple phases, with an explicit timing point marking the transition between phases. In the case of the base protocol there are exactly four timing points marking the beginning and the end of the request and the beginning and the end of the response.

**Cycle-accurate modelling** Although the cycle-accurate modelling style may be addressed by a future OSCI standard, it is currently out of the scope of TLM-2.0. However, a cycle-accurate model may be represented using language constructs from SystemC and TLM.

Each coding style can support several abstraction levels depending on functionality, timing and communication details.
2.3 HIGH ABSTRACTION LEVEL ELECTRONIC SYSTEM DEVELOPMENT

System design in general describes an iterative process going from the system requirements to the implementation level. The objective of this iterative process is to break the system development into a series of smaller steps, as the design gap between the specification and the implementation is too large to be resolved in a single step [15].

Following the SystemC–TLM methodology explained in Section 2.2, at each design step an executable system model is created representing a different level of abstraction. These executable models are very useful to specify the functionality of a large system, where different engineers work together to develop the system [16]. The models must describe both the interactions and the functionality of the system. In order to abstract the interactions, TLM [11] has been proposed.

The first objective of the system engineers is to create the SAM, that is, the high level functional model description which is independent from an implementation in either HW or SW. As described in the SystemC–TLM design flow presented in Section 2.2, this model is created from the system requirements using an untimed coding style both for the functionality and the communications.

As depicted in Figure 2.4, several model refinements are necessary to create a high abstraction level executable model describing the functional architecture of modern complex electronic systems prior to HW/SW partitioning [17]. In the initial models, the designer focuses on the functional description of the system and on the information exchanged between the system and the environment. During the first design steps, the system is divided recursively into smaller parts. The detail of the models increments to describe the functionality of the different parts and the information exchanged between them. These early system definition and functional architecture exploration models are very high abstraction models. For the functionality, the focus is on what functions shall be performed, not on how the functions shall be performed. For the communications, the focus is on what information is exchanged, not on how the information is exchanged. These models of the system are addressed by means of untimed TLM modelling techniques [14]. These high abstraction level TLM models are fast to build and they enable very high speed of simulation.
Hereinafter, this PhD dissertation will focus on the design and verification of these high abstraction level TLM models, which are created before the HW/SW partitioning of the system occurs.

2.4 VERIFICATION OF TLM EXECUTABLE MODELS

Verification is a process used to demonstrate the functional correctness of a design [18]. As opposed to device testing, verification involves analysis on a computer model of the system before it is manufactured [15]. In each step of the design process, the design team needs to check that the executable model behaves successfully and that the model matches the original intent specified in the system requirements. This task is accomplished verifying each system design model.

The verification is one of the key components of any system design effort. Functional verification consumes about 75% of the resources in a typical electronic system design process [19]. As a result, an enhancement
in the verification process may cause a considerable improvement in the overall development process.

### 2.4.1 SIMULATION BASED METHODS

Simulation based methods are widely used for electronic systems verification. A simulation based environment consists of a testbench and the system to be verified, also known as Device Under Verification (DUV) [18], as depicted in Figure 2.5. The testbench contains a simulator in order to execute the behaviour of the DUV and coordinate both the injection of stimuli and the recording of the DUV outputs. The stimuli represents the input patterns injected in the DUV in order to stimulate certain functionality of the system and try to uncover bugs. The simulator injects the input stimuli in the DUV and executes it. Ultimately, the outputs of the DUV will be updated and the testbench will record them. The objective of the testbench is to check whether the DUV outputs match the expected behaviour of the system according to the system requirements specification. The pair of the stimulus and the corresponding output is known as a test-case. The set of test-cases are usually prepared manually from the system requirements specification.

The main drawbacks of simulation based methods are that they need a lot of test-cases in order to verify the complete system and thus, the preparation of the verification process is very time consuming.
2.4.2 FORMAL METHODS

Formal methods are based on mathematical analysis techniques for the development of electronic systems. The key difference from simulation based verification is the absence of test-cases. The objective of formal methods is to guarantee that the design is free of functional errors, regardless the input stimuli applied to it. The most well-known formal verification techniques are [20, 21]:

- Model checking: this method compares a system model with its requirements specification. On one hand, system requirements are represented as mathematical properties using temporal logic formulas expressed with Linear Temporal Logic (LTL) or Computational Tree Logic (CTL) operators. On the other hand, the system under verification is represented with mathematical structures (usually finite state machines). The verification procedure (model checker) consists of a systematically exhaustive exploration of the mathematical structure representing the design model in order to check whether it satisfies the system specification regardless of the input stimuli of the system.

- Formal equivalence checking: this technique checks the equivalence of two mathematical structures (usually finite state machines) representing the system model. It is often used to compare two models representing the same system at different abstraction levels. For example, formal equivalence checking can be used to compare a TLM model with its corresponding RTL model.

- Automated theorem proving: this method proves that the system implementation satisfies a requirements specification by mathematical reasoning. Both the system implementation and specification are expressed as formulas in a formal logic. The objective of the automated theorem proving is to provide a mathematical theorem that proves the equivalence of these formulas using mathematical deduction techniques.

Formal methods have several drawbacks for practical use within complex electronic systems design. Model checking and formal equivalence checking methods suffer from the state explosion problem [20]. On one hand, their goal is a systematic exploration of all the system states. However, this process becomes infeasible in real systems as the number of states to check
is extremely large. On the other hand, as the system under verification is represented with state-based mathematical models, these methods are limited to control-based designs. In automatic theorem proving techniques, the reasoning tools can not always generate the complete proof and human guidance is often needed.

2.4.3 ASSERTION-BASED VERIFICATION (ABV)

Assertion-based verification (ABV) has become a successful method for the verification of electronic systems [22]. Using this method, system requirements are expressed by means of assertions. These assertions are intended to detect design process bugs. An assertion is a directive given to a tool to check a property on a model of the system. Specifically, an assertion is a conditional statement about a specific functional characteristic or property that is expected to hold for a design [23]. Assertions may be described using mathematical properties like in formal verification methods. However, the assertion verification process is different, as ABV relies on system simulation. As a result, ABV is sometimes considered a semi-formal verification technique [24].

An ABV environment is depicted in Figure 2.6. While the ABV simulator injects stimuli into the DUV, the ABV environment checks that the system execution satisfies the assertion set defined for the DUV. In order to carry out that task, the ABV environment receives information about
Figure 2.7: Verification of the different SystemC executable models.

input stimuli, internal test points and system outputs by means of assertion monitors.

The executable models based on TLM enable functional verification based on assertions. In a TLM model, the behaviour of the system manifests itself as a flow of transactions. In order to verify the correctness of the system, the flow of transactions in the system shall be observed. Thus, the fundamental unit of information for a ABV verification at TLM is the transaction.

As discussed in [18], a reliable design flow is to have separate design and verification teams, as shown in Figure 2.7. The design team creates design models of the system using the specification as initial input as depicted in Figure 2.7. The verification team creates assertions from the system spec-
2.4 Verification of TLM Executable Models

ifications. The redundancy introduced by working with these two teams mitigates the probability of a specification misinterpretation.

Following the SystemC–TLM methodology, at each design step an executable system model is created representing a different level of abstraction. As the transformations between abstraction levels are performed manually, ABV is used to verify the model transformations. Using ABV, system properties are expressed by means of assertions which are further checked against the system model using dynamic verification methods. The SystemC executable models can be classified according to their abstraction level of the functionality into Level 0, Level 1, etc.

In the first step of the design flow, the group responsible for the system design creates the Level 0 model, while the verification group defines the first assertion set of the system. Level 0 describes the system and its transactions with the environment at the higher level of abstraction. System designers and verification designers use the first system specifications as a reference. The Level 0 assertion list checks that the design team has correctly interpreted the initial system specification.

In a second step, the Level 0 system model is divided into functional modules yielding the Level 1 model. New requirements appear for each module and the interfaces between the devices. The Level 0 model is refined by the design team to take into account these new requirements. Additionally, the verification team adds new assertions to the ones defined in the first step in order to cover the new model details. This way, the Level 1 model is verified against the initial requirements and the new ones that take into account details of the implementation. These refinement steps can be repeated taking into account more implementation details.

2.4.3.1 State of the art of ABV-based approaches

The use of assertions at RTL using Property Specification Language (PSL) [25] or System Verilog Assertions (SVA) [26] is a popular technique [27, 28]. It has been proposed to raise the abstraction level of the assertions to the TLM [29–33]. However, PSL or SVA assertion description languages are not suitable for TLM as they are specifically tailored for RTL [34].
In [32], a TLM assertion framework is proposed. The assertions are specified using a custom language. These assertions are then provided to a custom compiler that generates a SystemC implementation of the assertions, which is merged with the SystemC model of the design. A modification of the set of assertions or a modification of the design model needs to go through the whole custom compiler flow which slows down the system development process. Moreover, the approach described in [32] has limitations when a proposition references the data in a former proposition within the same assertion, which in turn can obscure the coding of the assertion and demands a detailed knowledge of the design model by the verification team.

In [33], the TLM assertions are embedded in the code of the design model using a set of proposed functions and macros. Different macros and functions are proposed according to the level of abstraction of the model. Embedding the verification code in the design model leads to a very complex code. Furthermore, it makes it difficult for the verification team to work in parallel with the design team.

A ABV approach is presented in [35] for the verification of SystemC–TLM models. In this approach, the assertions are specified using a undetermined subset of the PSL language which was originally developed for RTL models. The authors embed the assertions within the SystemC–TLM code by means of SystemC versions of the PSL operators. Similarly to [32], a modification of the set of assertions requires the manual modification of the system model and the recompilation of the whole design. The manual modification of the system model to integrate the assertions may be an error-prone process. In early design phases where many iterations are performed to tune the architecture and to correct functional errors, the recompilation tasks may slow down noticeably the system development process.

In order to apply ABV techniques efficiently to a model driven methodology, two key elements are necessary: a clear definition of how assertions are interpreted (an assertion specification language) and a verification framework to evaluate the assertions during simulation. The ABV approaches proposed in the literature lack an effective assertion specification language that enables both to build assertions fast at TLM and to reuse assertions easily between different models. Moreover, an ABV framework that enables the design and verification teams to work in parallel is missing.
2.5 Algorithmic Models Integration in TLM Executable models

Once what the system modules shall do has been defined, the next design step is to develop the algorithms, as depicted in Figure 2.8. The algorithms devised for each module will be included in the next system model. This new TLM model is a first description of the how, and it is intended to check that the different algorithms cooperate correctly to achieve the high-level functionalities. At this stage of the design process, design iterations will be performed for architecture and algorithm tuning.

**Figure 2.8:** Algorithmic executable model creation.

### 2.5 Algorithmic Models Integration in TLM Executable Models

Once what the system modules shall do has been defined, the next design step is to develop the algorithms, as depicted in Figure 2.8. The algorithms devised for each module will be included in the next system model. This new TLM model is a first description of the how, and it is intended to check that the different algorithms cooperate correctly to achieve the high-level functionalities. At this stage of the design process, design iterations will be performed for architecture and algorithm tuning.
Algorithm development is commonly carried out using MATLAB [36] from Mathworks. The algorithm designers produce executable specifications of the algorithms as M-files or Simulink models. Translating the algorithms developed in MATLAB to C++ or SystemC is a possible approach. MATLAB Coder [37] aims to generate C/C++ code from MATLAB M-files. However, it only supports a limited set of functions from the MATLAB language. Besides, the Mathworks community expressed its concerns about the quality of the code generated by MATLAB Coder. As a result, the designers often tend to look at the generated code and then rewrite the MATLAB code in order to get the desired C++ code. These iterations increases the algorithm development time. Therefore, the benefits of a code generator vanish.

In order to reduce the development cost, it is preferable to enable both the system-design and the algorithm-design teams to work in their preferred environments: SystemC and MATLAB respectively. MATLAB provides an Application Programming Interface (API) [36] to enable access to the MATLAB Engine from C programs. This API consists of several routines to handle the MATLAB Engine sessions and the exchange of data between the C application and the MATLAB Engine. However, the direct use of these routines in the SystemC code is cumbersome; i.e.: it results in a awkward code, and it distracts the system-design team from the system description. Additionally, the MATLAB API offers a basic access to MATLAB execution sessions. This API lacks the capability to maintain the shared data between the C application and MATLAB synchronized. As a result, the design team is responsible for including specific code within the C application to keep the synchronization.

2.5.1 STATE OF THE ART OF SYSTEMC–MATLAB INTEGRATION APPROACHES

Authors in [38–40] propose similar solutions to co-simulate SystemC and MATLAB models. All these works propose to integrate SystemC models into MATLAB-Simulink models. Wrappers must be created using MATLAB S-Functions to link SystemC modules with Simulink in order to exchange data between both simulators.

Hylla et al. [41] presented an approach to use SystemC within a Simulink verification flow. Similarly to previous works, S-function wrappers must be implemented for each SystemC based testbench modules in order to inte-
grate them into Simulink. However, this proposal only addresses SystemC testbench modules within a Simulink design.

In all these works, the control of the simulation is managed by Simulink; no information is provided whether algorithms written in traditional MATLAB M-file scripts are supported. Works proposing the SystemC model as the main controller of the simulation during the integration of the MATLAB algorithms are missing in the literature. This approach would avoid the creation of S-Function wrappers and it would allow to maintain the same simulation environment used in the SystemC–TLM design and verification flow.

2.6 MDD FOR HIGH ABSTRACTION LEVEL ELECTRONIC SYSTEM DEVELOPMENT

Previous sections described the state of the art of SystemC–TLM design and verification flows. The aim of these methodologies was to increase the level of abstraction from RTL to TLM and create executable models. The approaches based on SystemC–TLM partially solved the design productivity gap. However, as indicated in the last report published by ITRS [19], the design productivity gap is still an open issue. In order to increase the level of abstraction above TLM and automate the processes that produce executable specifications of electronic systems, new approaches and methodologies were proposed recently: model-driven initiatives.

Model-driven initiatives were initially proposed as development methodologies within a software engineering context [42]. Model-driven initiatives promoted the acceptance of visual model-centric approaches over code-centric software development strategies. The foundations of model-driven initiatives are described in Appendix A.

Model-driven Design (MDD) is the most notable effort among model-driven initiatives. The first experiences with MDD date back to the late 90s [43, 44]. It has been in the last decade, when MDD has emerged as a development methodology for software engineering [45–47]. The main objective of MDD is to capture in visual models all the information of the system being designed in order to concentrate on generating implementations from models. The Unified Modeling Language (UML) [48] standard was conceived by the Object Management Group (OMG) as an object-oriented software modelling language applicable within the MDD method-
ology. UML defines a graphical notation supported by modelling diagrams that allows to create visual models of systems under development.

In a MDD approach the model plays the primary role. Thus, the design products are models instead of source code. A key point of MDD is that the transformation and generation processes can be applied to graphical models in order to create diverse products, such as, text documentation, source code, other models, etc. The automation of the source code generation from system models simplifies the work of engineers and avoids the error-prone and time-consuming manual translation process.

The MDD approach is not exclusively limited to pure software development. In order to extend the benefits of this approach to other domains, the International Council on Systems Engineering (INCOSE) in close collaboration with OMG defined Systems Modeling Language (SysML) [49] as a general purpose modelling language for systems engineering. SysML is defined as an extension of a subset of UML, and like UML, it defines a graphical notation supported by modelling diagrams. A brief overview of the diagrams provided by UML and SysML is provided in Section A.4 of the Appendix A.

MDD has been proposed to handle the complexity of the design of modern electronic systems [16, 50, 51] through visual modelling techniques using languages such as UML and SysML. MDD increases the abstraction level and automate the generation of executable system models. Besides, the use of model-driven design methodologies simplifies the work of the engineers, helping them to proceed in a simple and regular way increasing the productivity. As a result, the development time of the device being designed is reduced significantly.

2.6.1 HIGH LEVEL MODEL DESCRIPTION

MDD methodologies can be employed to develop complex electronic systems. MDD approaches can be used to close the gap between the system requirements specification and the executable models design, as shown in Figure 2.9. In this design flow, system engineers create visual system models from the system specification using modelling languages such as UML and SysML. The initial system model (Level 0) can be further refined to produce more detailed models (Level 1, Level 2, etc.). This approach enables to create visual models that represent a different level of abstrac-
2.6 MDD for High Abstraction Level Electronic System Development

System Engineers

System Requirements Specification

Visual System Models

System Model Level 0

System Model Level 1

System Model Level 2

... 

Executable System Models

Executable Model Level 0

Executable Model Level 1

Executable Model Level 2

Figure 2.9: Executable models generation from visual models.

tion. Executable models can be created from visual system models using Model-to-Text (M2T) methods.

System engineers benefit from the application of the MDD methodology; this approach eases the creation of the initial system model from the specifications, facilitates the early evaluation and analysis of the system and improves the productivity of the system engineers by means of M2T code generation techniques.

2.6.2 STATE OF THE ART OF MDD-BASED APPROACHES FOR ELECTRONIC SYSTEMS DEVELOPMENT

Many research works in the literature address the design of electronic systems using MDD approaches. Some works in the literature are focused on translating UML models into languages such as Handel-C [52],
Rebeca [53] and Simulink [54]. However, the most widely used languages for electronic design are VHDL and SystemC. The description of the works based on VHDL and SystemC has been divided in several sections. Section 2.6.2.1 describes works addressing the VHDL code generation from UML models. Section 2.6.2.2 describes works addressing the SystemC code generation from UML models. Section 2.6.2.3 describes works dealing with the SystemC code generation from SysML models. Section 2.6.2.4 presents the projects carried out within the European Seventh Framework Programme (FP7). Finally, Section 2.6.2.5 provides comments and a comparison of the works in the literature, as the works have many different characteristics.

2.6.2.1 VHDL code generation from UML

The initial efforts to apply MDD techniques for electronic systems design focused on mapping UML to VHDL code.

One of the first contribution was proposed by McUmber in 1999 [55]. This work presented a framework for deriving VHDL specifications from the UML class and state diagrams in order to capture the structure and the behaviour of embedded systems. The mapping between UML class diagrams and state machines on the one hand and VHDL constructs on the other hand were provided by means of a metamodel. The authors were aware of the lack of formal semantics of UML, however UML-to-VHDL mapping rules were defined in plain natural language.

The researchers in [56] addressed a strategy to translate UML state machine diagrams models into an intermediate language called SMDL, which is then mapped to an automata representing a state machine. The automata is then compiled into VHDL code.

A mapping from class diagrams to VHDL code is also proposed in [57]. They complement this static mapping with the definition of a hardware development process similar to the process described in [58]. In [59], the previous works based on UML mapping are extended to generate ANSI-C and CodeSimulink artifacts in addition to VHDL. Although this work focuses on an automatic generation process, the mapping proposal is neither formalised nor rigorously described, and no supported modelling tools are mentioned.
The work in [60] proposes a mechanism for mapping UML state diagrams to synthesizable VHDL code, using MDD techniques. Transformation rules are used to transform state machines conforming to UML to state machines conforming the VHDL metamodel. A VHDL metamodel is proposed to support VHDL code generation.

Vidal et al. [61] discussed the design of embedded systems using Modeling and Analysis of Real–time Embedded Systems (MARTE) and UML. The authors claim that the proposed approach is not directly connected to any implementation language, therefore, the code generation process is able to target any language such as SystemC or VHDL. Nevertheless, this paper only addresses the generation of synthesizable VHDL.

Initially, VHDL was used within MDD approaches. However, this PhD dissertation focuses on the generation of SystemC executable models.

### 2.6.2.2 SystemC code generation from UML

SystemC has become the de facto standard to create executable models of electronic systems. Many contributions in the literature are focused on electronic systems design methods involving UML and SystemC executable modelling language [50, 62–85]. The objective of these works is to generate SystemC executable code from a UML system description model.

An earlier effort is YAML [62], which only uses UML to capture the structural aspects of the system under design. No behaviour is captured and thus, only a structural skeleton of the system modelling code is generated from certain structural aspects.

An incremental modelling approach named SLOOP is addressed in [63]. This modelling methodology defines three UML models in ascending level of abstraction: the Conceptual Model, the Functional Model and the Performance Model. SLOOP adopts C++ as an implementation language for the Conceptual Model and it proposes SystemC as an implementation language for the Functional Model and the Performance Model. The methodology presented in [63] does not focus either on formalising the semantics of UML or on the automatic C++/SystemC code generation.

Many works in the literature are based on creating UML profiles to model specifically SystemC elements, and modelling system behaviours only with state machine diagrams [50, 64, 65, 67, 71–74, 86].
In [50], the authors define an UML profile to model SystemC elements. This work addresses the automatic SystemC–TLM code generation from UML models. The presented framework is based on Rhapsody and it captures both the structural and the behavioural aspects of the system through UML class diagrams and state machine diagrams. This research work is extended by [64] proposing an additional TLM to RTL refinement step. Also in [67], the authors define an UML profile to model SystemC elements. However, no behavioural aspects of UML models are translated to SystemC.

Riccobene et al. presented several contributions focusing on a MDD methodology based on UML, UML profiles and SystemC, to model embedded systems and generate executable code [65, 71, 74]. The authors proposed UML profiles to support SystemC constructs for HW/SW co-design. The behavioural aspects of the system under design are captured using only UML state machine diagrams. Moreover, the approaches proposed by Riccobene et al. do not create executable TLM models.

There are also several works that model behavioural aspects by means of state machine diagrams [72, 73, 86]. These contributions are also focused on UML-to-SystemC methodologies proposing specific UML profiles defining SystemC elements. Specifically, in [73], the authors presented a HW/SW co-design methodology based on two UML profiles: the SystemC UML profile for hardware and the C UML profile for software.

Generally, these works lack the capability to design systems with heterogeneous behaviours. That is, these works only support the design of control-based behaviours by means of UML state machine diagrams; the design of data flow behaviours is not supported. Moreover, these approaches do not define the Model of Computation (MoC) of the generated code. A MoC is an abstract representation that precisely describes the behavioural semantics of a system. The well-defined computation semantics of MoCs allow to unambiguously capture the required functionality. If no MoC is defined, the exact behaviour of the system remains unknown.

In [66], a UML profile is presented (TUT profile) to embedded systems designs. This profile defines a set of stereotypes to model both the application tasks and the platform. The presented framework uses a library to allow performance analysis and it provides an architecture space exploration tool that back annotates the UML model. Nevertheless, the proposed
design flow only allows software code generation in C, no hardware code is generated.

The work in [68] joins a UML-based SoC modelling approach with formal verification techniques. A UML profile called DIPLODOCUS has been specified in order to generate either a SystemC model or a formal LOTOS specification from UML system models.

In [69] and [70], UML model specifications are translated into Rebeca and AsmL modelling languages respectively. Then these intermediate models are mapped to SystemC executable models. In both cases, the executable model creation requires two translation steps. These works define the MoC of the generated SystemC code using Finite State Machines (FSMs) However, these approaches do not create executable TLM models.

Recently, SAVY [75] was proposed to provide clear design patterns to model the functionality of a system at high levels of abstraction. Two design patterns are defined to model the actors and the system: actor architecture pattern and entity architecture pattern. These design patterns take as input the information obtained from the UML Use Case diagrams and UML Statechart diagrams. A description of the SAVY approach is provided in Appendix B. SAVY enables to manually create high abstraction SystemC–TLM executable models from UML diagrams. However, the procedure to go from UML to SAVY is not clearly explained in [75]. Moreover, the proposal from [75] has several limitations to reuse models and to support a complete MDD methodology where several system abstraction levels are described.

2.6.2.3 SystemC code generation from SysML

SysML offers systems engineers several noteworthy improvements over UML, which tends to be software-centric [85]. In recent years several works have been proposed that focus on creating SystemC code from SysML.

Raslan et al. proposed in [76–78] one of the earliest approaches addressing SystemC code generation from SysML. SysML models specified in Rhapsody are translated into SystemC, however mapping rules are presented informally in natural language and only a few SystemC code snippets are presented. Although the proposed method supports the generation of SystemC code from SysML State Machine diagrams and Activity diagrams,
no model of computation is specified and thus, the behavioural semantics of the generated code remains unknown.

A work involving SysML is found in [79]. This work provides a study of the SystemC code generation capabilities of SysML and MARTE, but it does not present any formal semantics. An informal, example-based mapping is provided only for structural features of the system being designed using SysML or MARTE. The code generation process for behavioural features is not provided, the presented methodology only generates a SystemC code skeleton.

### 2.6.2.4 FP7 European projects

Several projects related to MDD were recently proposed within the European Seventh Framework Programme (FP7). These projects are presented in the following as examples that confirm an increasing interest in MDD-based methodologies for electronic systems development.

Several works in the literature [80–85] are related to the Saturn European project [87]. The SATURN project aims to bridge the gap between the modelling phase and the design phase in UML based designs of embedded systems that are composed of hardware and software parts. The objective of this project is to evaluate the UML profile for MARTE for its complementary application with SysML. However, the works in the literature related to SATURN only take SysML and MARTE into account independently and proposes a number of extra UML profiles for co-simulation, synthesis and code generation purposes. Additionally, no high abstraction executable models are created prior to SW/HW partitioning phase; synthesizable HW parts and embedded SW parts are modelled directly using visual modelling languages. Therefore, system engineers can not simulate a purely functional model; they must make design decisions about SW/HW partitioning in order to create the first executable model. Mueller et al. [80] presented the SATURN design flow that generates SystemC/C code from SysML models describing both HW and SW parts of the system. The work is based in the Artisan Studio development tool where SysML has been customized with UML profiles for SW/HW co-modelling. Mischkalla et al. [82, 83] presented a very similar work where the SATURN UML profiles for SW/HW co-modelling are detailed. The SATURN profile is composed of three sub-profiles: a synthesizable SystemC profile to model HW components, an Agility synthesis profile to translate SystemC models into VHDL by means
of the Agility compiler, and a C profile to model the embedded SW. These SysML based proposals model behavioural aspects by means of UML Activity diagrams, but they do not define the MoCs of the generated code and they do not address clear behavioural mappings. Besides, they are not focused on creating executable TLM models.

The aim of the COMPLEX European project is to develop an innovative, highly efficient and productive design methodology for iteratively exploring the design space of embedded HW/SW systems [88]. The COMPLEX methodology is introduced in [89, 90]. These works focus on the executable virtual prototype of an embedded HW/SW system after functional specification, partitioning and mapping to an execution platform. The scope of these works is the HW/SW co-design to study different platforms, HW/SW partition alternatives and power management strategies. They do not address the early creation of executable functional SystemC–TLM models that are independent from an HW/SW implementation. Similarly to the SATURN project, the drawback of this approach is that system engineers must make design decisions about SW/HW partitioning in order to create an executable SystemC model. Besides, the authors do not discuss the application of any MoC for the development of behavioural parts.

The MADEs European project [91] aims to develop a model-driven approach for the design, validation, simulation, and code generation of complex embedded systems. The MADEs approach focuses on the generation of platform-specific embedded software using C language and the generation of hardware descriptions using VHDL language of the modelled target architecture [92–95]. These models are expressed in the MADEs modelling language which is based in a combination of SysML and MARTE.

2.6.2.5 Comments to the works in the literature

A comparison of the works in the literature addressing MDD methods for electronic systems development is presented in Table 2.1. The meaning of the columns of Table 2.1 is explained in the following list:

- The **Graphical modelling Language** column determines the language used for the visual modelling of the system. Some works extend the pure UML, MARTE or SysML languages by means of profiles; others propose a combination of languages such as SysML/MARTE.
### Table 2.1: Comparison of works in the literature addressing MDD methods for electronic systems development.

<table>
<thead>
<tr>
<th>Authors &amp; References</th>
<th>Graphical Modelling Language</th>
<th>Behavioural Modelling Language</th>
<th>Electronic System Description Language</th>
<th>TLM</th>
<th>Model of Computation</th>
<th>Heterogeneous Behaviour</th>
<th>Mapping rigor</th>
<th>Code Generation</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>McUmber et al. [55]</td>
<td>UML</td>
<td>SMD</td>
<td>VHDL</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>1999</td>
</tr>
<tr>
<td>Bjorklund et al. [56]</td>
<td>UML</td>
<td>×</td>
<td>VHDL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2002</td>
</tr>
<tr>
<td>Damasevicius et al. [57]</td>
<td>UML</td>
<td>×</td>
<td>VHDL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2004</td>
</tr>
<tr>
<td>Reyneri et al. [59]</td>
<td>UML</td>
<td>×</td>
<td>C/VHDL/Analog</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2006</td>
</tr>
<tr>
<td>Vidal et al. [60]</td>
<td>UML/SLX</td>
<td>SMIF</td>
<td>VHDL</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2008</td>
</tr>
<tr>
<td>Wood et al. [61]</td>
<td>UML</td>
<td>×</td>
<td>VHDL</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2009</td>
</tr>
<tr>
<td>Vidal et al. [62]</td>
<td>UML</td>
<td>×</td>
<td>VHDL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2009</td>
</tr>
<tr>
<td>Schattkowsky et al. [52]</td>
<td>UML</td>
<td>×</td>
<td>VHDL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2009</td>
</tr>
<tr>
<td>Alavizada et al. [53]</td>
<td>UML + Rebeca profile</td>
<td>SD</td>
<td>Rebeca SystemC</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2007</td>
</tr>
<tr>
<td>Schattkowsky et al. [54]</td>
<td>UML</td>
<td>×</td>
<td>VHDL</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2008</td>
</tr>
<tr>
<td>Sinha et al. [62]</td>
<td>UML + profile</td>
<td>×</td>
<td>SystemC</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2005</td>
</tr>
<tr>
<td>Zhu et al. [63]</td>
<td>UML + SystemC profile</td>
<td>×</td>
<td>C++/SystemC</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2005</td>
</tr>
<tr>
<td>Nguyen et al. [50]</td>
<td>UML + SystemC profile</td>
<td>×</td>
<td>SystemC</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2004</td>
</tr>
<tr>
<td>Xi et al. [64]</td>
<td>UML + SystemC profile</td>
<td>×</td>
<td>SystemC</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2005</td>
</tr>
<tr>
<td>Riccobene et al. [65]</td>
<td>UML + SystemC profile</td>
<td>×</td>
<td>SystemC</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2005</td>
</tr>
<tr>
<td>Kukkala et al. [66]</td>
<td>UML + TUT profile</td>
<td>×</td>
<td>SystemC</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2005</td>
</tr>
<tr>
<td>Wang et al. [67]</td>
<td>UML + SystemC profile</td>
<td>×</td>
<td>SystemC</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2006</td>
</tr>
<tr>
<td>Habibi and Tahar [69]</td>
<td>UML</td>
<td>×</td>
<td>AsmL/SystemC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>formal</td>
<td>✓</td>
<td>2006</td>
</tr>
<tr>
<td>Perez et al. [75]</td>
<td>UML</td>
<td>×</td>
<td>SystemC</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2008</td>
</tr>
<tr>
<td>Pena et al. [81]</td>
<td>UML</td>
<td>×</td>
<td>KPN,CSP,SDF</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2010</td>
</tr>
<tr>
<td>Peñil et al. [84]</td>
<td>MARTE</td>
<td>CSD</td>
<td>SystemC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2011</td>
</tr>
<tr>
<td>Mischkalla et al. [83]</td>
<td>SysML + SystemC/C profile</td>
<td>×</td>
<td>SystemC</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2010</td>
</tr>
<tr>
<td>Mischkalla et al. [82]</td>
<td>SysML + SystemC/C profile</td>
<td>×</td>
<td>SystemC</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2010</td>
</tr>
<tr>
<td>Raslan et al. [78]</td>
<td>SysML</td>
<td>×</td>
<td>SystemC</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2007</td>
</tr>
<tr>
<td>Raslan et al. [77]</td>
<td>SysML</td>
<td>×</td>
<td>SystemC</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2007</td>
</tr>
<tr>
<td>Mueller et al. [80]</td>
<td>SysML + SystemC/C profiles</td>
<td>×</td>
<td>SystemC</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2010</td>
</tr>
<tr>
<td>Raslan et al. [76]</td>
<td>SysML + SystemC/C profiles</td>
<td>×</td>
<td>SystemC</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2007</td>
</tr>
<tr>
<td>Sarno et al. [85]</td>
<td>SysML + SystemC profile</td>
<td>×</td>
<td>SystemC</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2005</td>
</tr>
<tr>
<td>Audsley et al. [92]</td>
<td>SysML/MARTE + profile</td>
<td>×</td>
<td>C/VHDL/AD</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2011</td>
</tr>
<tr>
<td>Gray et al. [93]</td>
<td>SysML/MARTE + profile</td>
<td>×</td>
<td>C/VHDL/AD</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2011</td>
</tr>
<tr>
<td>Quadri et al. [94]</td>
<td>SysML/MARTE + profile</td>
<td>×</td>
<td>C/VHDL/AD</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2011</td>
</tr>
<tr>
<td>Andersson et al. [95]</td>
<td>SysML/MARTE + profile</td>
<td>×</td>
<td>C/VHDL/AD</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2011</td>
</tr>
<tr>
<td>Audsley et al. [96]</td>
<td>SysML/MARTE + profile</td>
<td>×</td>
<td>C/VHDL/AD</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2011</td>
</tr>
<tr>
<td>Audsley et al. [97]</td>
<td>SysML/MARTE + profile</td>
<td>×</td>
<td>C/VHDL/AD</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2011</td>
</tr>
<tr>
<td>Audsley et al. [98]</td>
<td>SysML/MARTE + profile</td>
<td>×</td>
<td>C/VHDL/AD</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2011</td>
</tr>
<tr>
<td>Audsley et al. [99]</td>
<td>SysML/MARTE + profile</td>
<td>×</td>
<td>C/VHDL/AD</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>informal</td>
<td>✓</td>
<td>2011</td>
</tr>
<tr>
<td>Behavioural Modelling</td>
<td>Electronic System Description Language</td>
<td>TLM</td>
<td>Model of Computation</td>
<td>Heterogeneous Behaviour</td>
<td>Behavioural Mapping</td>
<td>Code Generation</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>
<td></td>
</tr>
<tr>
<td>- SMD: State Machine diagram.</td>
<td>- AD: Activity diagram.</td>
<td>- SD: Sequence diagram.</td>
<td>- CSD: Composite Structure diagram.</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Although the works in the literature span through several research topics, the key aspects and comments of the research works are discussed in the following.
Modelling Language: The vast majority of the published works selected UML as the graphical modelling language for electronic system design [50, 52, 54, 62, 64–74, 79]. While UML has been widely criticized for being software-centric, SysML offers a promising perspective for a high abstraction level electronic systems design methodology [97]. SysML simplifies UML and provides clear advantages. SysML discards some diagrams from UML and provides new diagrams for general systems engineering. Therefore, SysML focuses on the efficient specification of complex heterogeneous systems that may include software and hardware parts. As a result, SysML is better suited for high abstraction level functional electronic system modelling and contributes to the applicability of MDD techniques for SoC design.

UML Profiling: The UML profile mechanism is flexible and powerful, however it does not provide a means for precisely defining semantics associated with extensions [98]. Research works proposing a UML profile have made their own extensions to adapt UML to their needs. Additionally, these research works tend to provide a one-to-one mapping with SystemC, that is, the proposed UML profiles generally define a stereotype for each modelling construct from SystemC, such as modules, threads, signals, ports and data types. As a result, these works make UML to “look like” SystemC, missing the high abstraction level approach provided by visual modelling languages. The modelling methodology becomes a SystemC-centric process that do not exploit the benefits of a MDD approach.

Behavioural Heterogeneity: Modern electronic systems are often composed of control-based parts and signal processing oriented parts, but the modelling requirements for control-based applications and signal processing applications are very different. Control-based parts are efficiently modelled by finite state machines while signal processing algorithms are better described using data flow style representations. This heterogeneity is supported within UML/SysML using State Machine diagrams for state-oriented control modelling and Activity diagrams for data flow behaviour models. However, the MDD approaches presented in recent years are mainly focused on generating only control-based applications using state-oriented modelling techniques by means of UML/SysML State Machine diagrams and thus, do not support any behavioural heterogeneity. The majority of the presented works lack of heterogeneous modelling support, and thus, the practical use of these tools within an electronic system development process is very restricted. Few exceptions are found in the literature
providing support for heterogeneous behaviours. On one hand, Raslan et al. proposed in [76–78] a methodology supporting the generation of SystemC code from SysML State Machine diagrams and Activity diagrams. However no MoC is specified and thus, the behavioural semantics of the generated code is not defined. On the other hand, authors in [81] and [80] claim that the proposed MDD approaches support several MoCs, but these works lack the definition of these MoCs and their applicability within a SystemC code generation process. Moreover, these works are not focused on creating TLM code.

**Formal Semantics:** The UML standards [99, 100] concentrate on syntax and do not offer a rigorous definition of the semantics. The UML standards provide explanations about the semantics in informal natural language and lack many details. SysML [49] also lacks formal semantics, as it reuses the core modelling aspects of UML. The lack of formal semantics makes it difficult the acceptance of visual modelling languages by developers and complicates the exchange of complex models among different tools [101]. Besides, automation tools for electronic system development can only be applied if models and corresponding abstraction levels are well-defined with clear and unambiguous semantics [17].

**Model of Computation:** A MoC is an abstract representation that precisely describes the behavioural semantics of a system. A MoC is the semantics of the interaction between modules or components of a system [102]. MoCs are represented in a formal manner, using typically a mathematical description (functions, notations from the set theory, etc.) that has a syntax and semantics defined by rules for the computation of the behaviour.

MoCs are used in computer programming as well as in the design of electronic systems. MoCs, together with design languages, provide the foundation for defining system behaviour throughout the design flow [17]. The well-defined computation semantics of MoCs allow to unambiguously capture the required functionality [103]. Besides, unambiguous semantics enable the application of formal design and verification techniques.

In electronic systems the timing is usually the key parameter. In [104] several MoCs are distinguished with respect to the time abstraction they use: continuous time, discrete time, synchronous and untimed MoCs. The most well-known MoCs are presented in the following list:
Finite State Machines (FSM)
Hierarchical Concurrent Finite State Machines (HCFSM)
Kahn process networks (KPN)
Bounded Kahn process networks (BKPN)
Communicating Sequential Processes (CSP)
Synchronous Data Flow (SDF)
Asynchronous Data Flow (ADF)
Dynamic Data Flow (DDF)
Petri Nets
Synchronous Reactive (SR)
Clock Synchronous Reactive (CS)

Few works in the literature define or provide a description of the MoC implemented in the generated SystemC code. Several research works that are not related to electronic system modelling are focused on the formalization of certain behavioural aspects of UML. Harel proposed the Statechart language [105–108] which defines the semantics of the core part of the UML State Machine diagrams.

On the other hand, the UML 2.0 standard affirmed that Activity diagrams have a Petri Net (PN) [109–111] like flow semantics to better support modelling of activity flows but no formal semantics are provided [112]. The authors in [113] pointed out several problems that may occur when applying Activities diagrams in practice due to problems in the syntax and semantics of the UML 2.0 language. Since then, several studies have been carried out both to analyse the affirmation of the UML standard and to provide formal semantics for UML activity diagrams. Several studies focused on formalizing the semantics of UML 2.0 Activity diagrams using PN [112, 114–117] or stochastic PN [118]. However, translating UML activities into PNs creates new problems [119]. The PN diagrams are more complex, they contain more nodes and edges, and they are unsuitable for visualization by developers or customers.
2.7 Concluding Remarks

Code Generation Rules: The proposals in the literature do not provide clear transformation rules to translate UML/SysML models to SystemC; they provide example-based SystemC code generation mapping rules. The absence of formal transformation rules prevents the efficient tooling support for automatic code generation. On the other hand, these works do not apply the proposed methodologies to a complex system design.

TLM code generation: Few works in the literature address the generation of SystemC–TLM executable code. MDD methodologies generating TLM models are required to fill the gap between the system specifications and the TLM abstraction level.

Complete solution: The research works found in the literature only focused on some parts of the MDD approach for electronic systems design. None of the works proposed a complete solution solving all the issues discussed previously in this section. A MDD approach to generate SystemC–TLM executable code from SysML, with clear generation rules supporting formal semantics, behavioural heterogeneity and clear MoCs is missing.

2.7 CONCLUDING REMARKS

The research in electronic systems design and verification methodologies has been a widely discussed topic during the last decades. The ITRS expressed its concerns about the design productivity challenge and suggested three foundations in order to minimize the design gap: producing executable system specifications, increasing the level of abstraction, and automating development processes and tools.

SystemC has become the de facto standard to create executable models of electronic systems. Recently, TLM has been proposed for the creation of high abstraction level executable models. The OSCI developed a TLM library for SystemC in order to support TLM. The main objective of TLM is to decouple the functional aspects and the communication aspects of a system. Initially, TLM untimed functional models are created from the system requirements. These high abstraction level TLM models are fast to build and they provide a very high speed of simulation. Additionally, these models enable early functional architecture analysis, early functional verification and early algorithm integration.
ABV techniques have been proposed in the literature to verify SystemC–TLM models. However, they lack a flexible and high abstraction level environment for assertions definition. PSL and SVA assertion description languages are used for RTL models but they are not suitable for TLM. Several works embed the verification code in the design model. This leads to a very complex code and makes it difficult for the verification team to work in parallel with the design team. Furthermore, a modification of the set of assertions requires a modification of the design model which may be an error-prone task. This approach needs to go through the whole compilation flow. In early design phases where many iterations are performed to tune the architecture and to correct functional errors, the recompilation tasks may slow down noticeably the system development process. The ABV approaches proposed in the literature lack an effective assertion specification language that enables both to build assertions fast at TLM and to reuse assertions easily between different models. Moreover, an ABV framework that enables the design and verification teams to work in parallel is missing.

Eventually, once the high abstraction level executable model describing the architecture has been created, algorithms are developed and integrated in the TLM model. The objective of this integration is to check that the different algorithms cooperate correctly to achieve the high-level functionalities. Algorithm development is commonly carried out using MATLAB. The MATLAB Engine API offers an option to integrate MATLAB scripts with SystemC. However, new tools are necessary, as the MATLAB Engine API lacks the capability to automatically maintain the shared data between the SystemC application and MATLAB synchronized, and the direct use of the routines of the API results in a awkward code. In the works found in the literature, SystemC models are wrapped by S-Function wrappers. Thus, the control of the simulation is managed by Simulink. Moreover, no information is provided whether algorithms written in traditional MATLAB M-file scripts are supported. Works proposing the SystemC model as the main controller of the simulation during the integration of the MATLAB algorithms are missing in the literature. This approach would avoid the creation of S-Function wrappers and it would allow to maintain the same simulation environment used in the SystemC–TLM design and verification flow.

Recently, the MDD methodology has been proposed as a promising methodology for complex electronic systems modelling and design. The MDD methodology is supported by visual modelling languages such as
UML or SysML. These visual modelling languages provide a high abstraction level modelling environment and enable automatic code generation processes using M2T techniques. However, several issues are still open that prevent a successful application of MDD in a SysML-to-SystemC design flow.

In the literature, works mainly selected UML as the graphical modelling language. However, the UML standards concentrate on syntax and do not offer a formal definition of the semantics. Although SysML is better suited to model electronic devices at system level, it also lacks formal semantics, as its reuses the core modelling constructs of UML. The lack of formal semantics makes difficult the acceptance of visual modelling languages by developers and complicates the exchange of complex models among different tools. This issue is specially significant in behavioural aspects. Moreover, the research works found in the literature do not define the MoC of the generated SystemC code. The well-defined computation semantics of MoCs allow to unambiguously capture the required functionality and enable the application of formal design and verification techniques. The automation tools for electronic system development can only be applied if the behavioural semantics and the corresponding MoCs are well-defined.

Additionally, the approaches presented in recent years are focused on generating only control-based applications using state-oriented modelling techniques by means of UML State Machine diagrams. Modern electronic systems are often composed of control-based parts and signal processing oriented parts, but the modelling requirements for control-based applications and signal processing applications are very different. Control-based parts are efficiently modelled by finite state machines while signal processing algorithms are better described using data flow style representations. This heterogeneity may be supported within UML and SysML using State Machine diagrams for state-oriented control modelling and Activity diagrams for data flow behaviour models. In general, the presented research works lack of heterogeneous modelling support, and thus, the practical use of these tools within an electronic system development process is very restricted.

Finally, M2T techniques can be applied to the MDD methodology for the automatic generation of SystemC–TLM executable models from visual system models. However, few works in the literature address the generation of SystemC–TLM executable models. MDD methodologies generating
TLM models are required to fill the gap between the system specifications and the TLM abstraction level.

Concluding, the research works found in the literature only focused on some parts of the MDD approach for electronic systems design. None of the works proposed a complete MDD solution to generate SystemC–TLM executable code from SysML. An MDD approach with clear generation rules supporting formal semantics, behavioural heterogeneity and clear MoCs is required.
Current Electronic Design Automation (EDA) tools and frameworks are unable to cope with the design productivity gap. This gap represents the inability of EDA tools to efficiently handle the complexity growth of the modern electronic systems. As a result, the research in electronic systems design and verification methodologies has been widely discussed during the last decades.

The main objective of this research work is to propose a methodology and a framework to assist in the architecture design and functional verification of complex electronic systems described using SystemC. This research work also aims to enhance the electronic system design and verification methodologies in order to improve the productivity of the modern electronic device development process. This research work shall propose novel system methodologies supported by software tools for functional system models design and verification. The CONFIDENCE [120] project will be used to evaluate the proposed development methodologies and EDA tools.
3.1 PARTIAL OBJECTIVES

Chapter 2 introduced the state of the art of the techniques and methodologies proposed for the development of electronic systems, and it described the issues that are still open. Chapter 2 focused in three open issues: First, an effective Assertion-based verification (ABV) approach that enables the verification of SystemC models at Transaction Level Modeling (TLM) level is missing in the literature. Secondly, new tools are necessary to easily integrate MATLAB algorithms within a SystemC–TLM design flow. Finally, a complete Model-driven Design (MDD) based methodology for SystemC–TLM code generation from Systems Modeling Language (SysML) models is still missing.

The global objective of this dissertation has been split into the following partial objectives:

- **Propose a TLM model verification flow with integrated ABV capabilities supported by an assertions specification language and a verification framework.**
  The TLM verification framework shall enable the verification team to write early design assertions simply and fast, at a very high level of abstraction. It shall enable the verification team to write assertions with increased detail and it shall be prepared to support a SystemC–TLM design flow. Furthermore, the assertion specification shall be decoupled from the design model. This enables the verification team to work in parallel with the design team, reducing the development time. The proposed framework shall be very flexible and it shall enable to reuse assertions between different TLM models.
  This partial objective is addressed in Chapter 4.

- **Integrate the algorithm simulation capabilities of Matlab into SystemC–TLM executable models.**
  During the initial iterations to tune the functional architecture and the algorithms, the SystemC–TLM model of the system shall execute the MATLAB description of the algorithms provided by the algorithm-design team. The proposed approach shall facilitate verifying that the different MATLAB algorithms cooperate correctly to achieve the high-level functionalities and it shall enable a robust design-flow with a reduced development cost. Both Simulink models and traditional
3.1 Partial Objectives

MATLAB M-file scripts shall be supported for simulation. The SystemC model shall be the main controller of the simulation during the integration of the MATLAB algorithms. This approach would allow to maintain the same simulation environment used in the SystemC-TLM design flow. Additionally, this approach shall allow to use ABV techniques to verify a SystemC-TLM model with integrated MATLAB algorithms.

This partial objective is addressed in Chapter 5.

- **Formalize the behavioural semantics of SysML’s State Machine diagrams and Activity diagrams.**

Rigorous semantics shall be defined for SysML State Machine diagrams and Activity diagrams by means of formally defined Model of Computations (MoCs). The syntax of the State Machine diagrams and Activity diagrams that supported by the formalisation shall be clearly defined. The proposed semantic formalisation shall support modelling modern electronic systems composed of control-based parts and signal processing oriented parts using SysML State Machine diagrams and Activity diagrams, respectively. This partial objective shall establish the modelling methodology and the foundations required to develop effective MDD methodologies supported by practical modelling frameworks.

This partial objective is addressed in Chapter 6.

- **Provide a MDD based approach to model complex electronic systems at a high abstraction level.**

The MDD based approach shall support the modelling methodology and the behavioural formalisation defined previously. The proposed MDD framework shall implement the formalisation of the behavioural semantics of SysML State Machine diagrams and Activity diagrams. Moreover, the framework shall translate electronic systems described using SysML into SystemC-TLM executable models supporting control-based behaviours and data flow behaviours (behavioural heterogeneity).

This partial objective is addressed in Chapter 7.

- **Assessment of the proposed methodology and tools.**
CONFIDENCE project shall be used to evaluate the proposed development methodologies and frameworks.

This partial objective is addressed in Chapter 8.
Chapter 4

An ABV framework for SystemC-TLM

Contents

4.1 Proposed Assertion Specification Language . . . . . . . . 47
4.2 Verification Framework Implementation . . . . . . . . . . 67
4.3 Concluding Remarks . . . . . . . . . . . . . . . . . . . . . 83

Following the SystemC–Transaction Level Modeling (TLM) methodology described in Section 2.3, at each design step an executable system model is created representing a different level of abstraction. The SystemC–TLM executable models are classified according to the level of detail of the functionality. The employed design methodology is an iterative process, where new design details are added in each iteration. As the transformations between abstraction levels are performed manually, Assertion-based verification (ABV) techniques have been proposed in the literature to verify SystemC–TLM models. Within ABV, system properties are expressed by means of assertions which are further checked against the system model using dynamic verification methods.

In the first step of the design flow, system designers and verification designers use the system specifications as a reference. The design group proposes the first executable model and the verification group defines the
corresponding assertion set. The initial assertion list checks that the design group has correctly interpreted the system specification. In successive steps, the design group applies refinement steps to produce new executable models based on the previous SystemC–TLM models, while the verification group proposes new assertion sets.

This chapter presents an ABV framework for SystemC–TLM executable models. Figure 4.1 describes how the proposed ABV framework is integrated into a TLM development flow. The ABV framework enables the verification team to write early design assertions simply and fast, at a very high level of abstraction. It enables the verification team to write asser-
tions with increased detail and it is prepared to support a TLM–to–Register Transfer Level (RTL) model driven design flow. Furthermore, the assertion specification is decoupled from the design model. The proposed verification framework is supported by a C++ library built on top of SystemC–TLM. The assertion set is specified by means of an eXtensible Markup Language (XML) file. This enables the verification team to work in parallel with the design team, reducing the development time. The assertion specification XML file is read during the elaboration of the design model and the appropriate elements for the verification are built. Thus, a modification in the assertion specification does not need access to the design model code or recompiling. The proposed framework is very flexible and can be attached to any SystemC–TLM model.

The chapter is organized as follows. Section 4.1 presents the proposed Assertion Specification Language (ASL), its semantics and its architecture. In this section, the assertion evaluation process is explained. Additionally, a XML syntax is proposed for ASL. Section 4.2 describes the implementation of the proposed verification framework. Finally, section 4.3 summarizes the conclusions of this chapter.

4.1 PROPOSED ASSERTION SPECIFICATION LANGUAGE

The ASL is a language valid to describe the operation of electronic systems designed using transactional level modelling techniques. Nevertheless, ASL is not limited to the initial stage of the design. Throughout the System on Chip (SoC) design cycle, it serves as the unique reference across different teams for three strategic activities: early software development, architecture analysis, and functional verification.

There are languages to describe the desired behaviour of an application. However, they are mainly focused on low abstraction levels designs. Known the current complexity of the current SoC designs, it is not feasible to restrict the verification process to the final stages of the project. The detection of unwanted behaviours in the early stages is a primary objective in modern electronic systems design, resulting in greater margin of reaction and reduced development cost.

The current SoC design processes use transaction level modelling methods to abstract the implementation details and describe the behaviour of the system at a high abstraction level.
4.1.1 TRANSACTION LEVEL MODELS

A transactional model describes a system at a high abstraction level. This modelling approach emphasizes the separation between communication from computation within a system.

SystemC provides a TLM library. In SystemC-TLM, system components are modelled as modules with a collection of concurrent processes that represent their functionality. These modules abstract the communication as a transaction exchange through dedicated channels where write or read operations may be executed as depicted in Figure 4.2. TLM interfaces are implemented within this channels to encapsulate communication protocols. To establish communication, a process simply needs to access these interfaces through module ports. A master or initiator is a port that initiates transactions in a system, while a slave or target is a port that receives transactional requests. TLM defines a transaction as the data transfer between two ports at a specific instant.

4.1.2 ASL CONCEPTS

ASL is oriented to describe the functional aspects of a transaction-based system model. Thus, the concepts of ASL are defined around the fundamental element of TLM: the transaction.

If a TLM system model is conceived as a black-box model, the relationship with the outside world is reduced to the processing of transactions. If the model abstraction is extended to that of gray-box, the perception increases and the internal state of the system are exposed. ASL allows to describe transaction-based behaviours as well as the internal states of a module. ASL is based on the observation of events, therefore both transaction operations and changes in the state of a module can be described.
Following, the key concepts of ASL are discussed:

- **Events**: In a TLM system an event can be considered as the occurrence of an inter-system or intra-system transaction.
- **Time**: In an electronic system model the timing is a determinant feature to describe functional behaviors.
- **Data**: In ASL it is possible to access the data related to a transaction. A transaction describes an operation on a port, which exchanges data.
- **Composition**: Using ASL compositions, several transaction transfers can be described in order to identify behaviours and complex contexts in the system.
- **Expressions**: The description of a system at a functional level requires not only to observe independent transactions, but also to establish relationships between the transactions.
- **Statements**: The behaviour of a system is specified in terms of simple and compound statements in a declarative form. The objective is to avoid ambiguities and misinterpretations.

Let us analyse an example. We have a system with a Central Processing Unit (CPU) and a memory. For this system, we have the following functional behaviour expressed in natural language:

*If a read operation is performed on memory requesting the value of address 0, then the memory shall return the value requested before 2 milliseconds.*

The ASL concepts presented previously are identified below:

- **Events**: Two transactions are identified in the previous postulate: a read transaction and a write transaction.
- **Time**: A maximum time delay of 2 milliseconds is specified between the read and write transactions.
4.1.3 ASL ARCHITECTURE

In this section the architecture of the ASL language is presented. ASL shows a hierarchical top-down architecture as shown in Figure 4.3. Each of the levels presented in Figure 4.3 is described in the sections below.

4.1.3.1 Event Layer

An event identifies the occurrence of an operation or a perturbation in a port within a system model. The fundamental unit of information for TLM executable model is the transaction. We define a Transactional Event (TE) as the occurrence of a transaction. The events of ASL contain the following attributes:

- Event identifier.
- Port: determines which port is the TE allocated to.


- **Operation**: name of the operation associated to the event. In the case of CPU-Memory example, *read* operations and *write* operations are defined.

- **Data**: defines several parameters associated to the operation or perturbation. It is not a predetermined attribute; it is possible to discover the contained fields in runtime.

### 4.1.3.2 Proposition Layer

A proposition defines a occurrence. In the case of the CPU-Memory example, a read operation in address 0 is a proposition, and another proposition is that the write operation is generated in 2 milliseconds.

The proposition level contains the concepts below:

- **Event**: An event is an operation or a perturbation in a port.

- **Time**: It defines the relative time in which the event shall occur. Within a sequence of events, the time is specified with respect to the previous event.

- **Sentence**: A sentence is a set of boolean and arithmetic expressions that shall be fulfilled in order to declare the property satisfied. Data from several transactions may be related. The default value for the sentence is *true*, that is, satisfied.

There are two types of propositions:

- **Positive**: A positive proposition is satisfied when the indicated event occurs in the specified time and the associated sentence is true.

- **Negative**: A negative proposition is satisfied when the indicated event does not occur, or the event does not occur in the specified time, or the associated sentence is not true, before the next negative proposition is satisfied or the simulation is finished.

### 4.1.3.3 Sequence Layer

A sequence identifies a set of consecutive propositions. A sequence of propositions specifies TEs occurring in a defined time order.
4.1.3.4 Property Layer

An ASL property contains an antecedent, an implication and a consequence. Both the antecedent and the consequence are sequences, while the implication is an operator.

Let us analyse another example based on the previous CPU-Memory example. The following functional behaviour is specified in natural language:

If the value $V$ is written in the memory address $A$ and the memory address $A$ is read afterwards, then the value obtained is equal to $V$.

The analysis of the former functional specification is shown in Figure 4.4. The behavioural specification is composed of three propositions. The sentence If the value $V$ is written in the memory address $A$ and the memory address $A$ is read afterwards is the antecedent and the sentence the value obtained is equal to $V$ forms the consequence. The antecedent and the consequence are joined by the implication then.

4.1.3.5 Assertion Layer

The assertion layer represents the highest level in the architecture of the ASL language. The aim of the assertion level is to formalize the assertion management. In an assertion several concepts are defined: the property to be verified, both the actions to be performed if the property is satisfied or not satisfied, and the management of the property statistics and its impact on the system under verification. This layer does not add any expressive power to the ASL language. The assertion layer manages the
4.1 Proposed Assertion Specification Language

relation between the system under verification, the collection of properties and the verification context. Figure 4.5 depicts the relationship between ASL architecture levels.

The fundamental elements of an assertion are discussed below:

- **Property**: An assertion includes a property which specifies a behavioural pattern of the system under verification.

- **Statistics**: The assertion manages the property statistics, such as:
  - How many times is the antecedent of the property satisfied.
  - How many times is the consequence of the property satisfied.
  - The time elapsed from the activation of the property until it is fully evaluated (successfully or not).

- **Associated actions**: actions are associated to the status of the property evaluation
  - Warning report: on screen or in a log file.
  - Error report: on screen or in a log file.
  - Abort the verification process.

Figure 4.5: Relationship between ASL architecture levels.
4.1.4 ASL ASSERTION EVALUATION FOR TLM MODELS

In a TLM model, the behaviour of the Device Under Verification (DUV) manifests itself as a flow of transactions. In order to verify the correctness of the DUV, we observe the flow of transactions in the DUV. Thus, the fundamental unit of information for TLM verification is the TE defined in 4.1.3.1.

An ASL property is evaluated checking all of its propositions. Propositions are evaluated one by one starting from the first proposition of the antecedent. A proposition can not be evaluated if its previous proposition has not been evaluated.

For assertion evaluation, three concepts are described for propositions:

- **trigger**: It specifies the identities of the initiator or the target ports involved in the TEs; the type of transaction; and the sequential relationship between the time instants of the TEs involved in the evaluation of the property.

- **time-out**: It describes a time-out condition.

- **expression**: It is only evaluated when the trigger is true. It describes a relationship between the information transmitted in the TEs involved in the evaluation of the property.

4.1.4.1 State of a proposition

We introduce the concept of state of a proposition. At a certain instant of simulation, the propositions can be in three states:

- **PENDING**
- **FETCHED**
- **UNACHIEVABLE**

According to the behaviour of the proposition we can distinguish two types of propositions, as described in Section 4.1.3.2: positive propositions and negative propositions. In a positive proposition, we state that the TE
4.1 Proposed Assertion Specification Language

The behaviour of a positive and negative proposition is depicted in figure 4.6. In a positive proposition, either a TE that meets the trigger condition or a time-out can produce a state transition. In a negative proposition, a state transition can also be produced by the state transition of a latter proposition within the property.

When a positive proposition is triggered, the proposition goes from the PENDING state to the FETCHED state if the proposition is true. When it yields false, the positive proposition remains in the PENDING state. A positive proposition only goes to the UNACHIEVABLE state when the time-out is true.

On the other hand, when a negative proposition is triggered, the proposition remains in the PENDING state if the proposition yields false. If it yields true, it goes to the UNACHIEVABLE state. A negative proposition will only go to the FETCHED state when the time-out proposition is true or when one of the next propositions moves away from the PENDING state.

4.1.4.2 State of a property

Initially, all the propositions within the property are in the PENDING state. As the simulation of the DUV produces TEs, the propositions of the property change their state. When a proposition changes its state, a new
instance of the property can be built to allow an overlapping evaluation of the property. The new instance of the property is a clone of the former instance before the change of state. According to the state of the propositions within an instance of a property, the instance can be said to be in four states:

- The *PENDING* state means that all the propositions in that instance are in the *pending* state.
- The *FETCHED* state means that all the propositions in that instance of the property are in the *FETCHED* state.
- The *UNACHIEVABLE* state means that at least one of the propositions in that instance is in the *UNACHIEVABLE* state. A property will be false if any of its instances is in the *UNACHIEVABLE* state.
- The *PARTIALLY FETCHED* state means that there are propositions in the *FETCHED* and in the *PENDING* state.

ASL properties are written as implications, where two parts are distinguished: the antecedent and the consequence. Whenever the antecedent is true, the consequence must be true for the property to hold. The implication meaning does not affect directly the evaluations of the property, but it makes sense from the coverage point of view.

Let's propose the property example depicted in Figure 4.7. This property specifies that if the proposition $B$ is satisfied after the proposition $A$, then propositions $C$ and $D$ shall not occur before proposition $E$ is satisfied.

The evaluation of a property is guided by an evaluation pointer. Initially, the evaluation pointer is in the first proposition. In case of the example presented in Figure 4.7, the evaluation pointer is initially in the proposition $A$. A new TE fires the evaluation of the proposition $A$. If the proposition is satisfied, the evaluation pointer jumps to proposition $B$ and waits for a new TE.
The proposition $B$ is evaluated upon arrival of a new TE. Analogously to proposition $A$, if the proposition $B$ is satisfied, the evaluation pointer jumps to proposition $C$ and waits new TEs.

In the next step, when a new TE is received, the proposition $!C$ is evaluated. In this example, if $C$ is satisfied (i.e. $!C$ is not satisfied) the proposition $!C$ goes to the UNACHIEVABLE state and the property will be false. On the other hand, if $C$ is not satisfied (i.e. $!C$ is satisfied) the evaluation pointer jumps to the proposition $!D$. The proposition $!D$ is evaluated immediately with the same TE the proposition $!C$ checked. The proposition $!D$ is evaluated analogously to the proposition $!C$. If $D$ is satisfied the property will be false. However, if $D$ is not satisfied the evaluation pointer is updated to the proposition $E$. If the proposition $E$ is not satisfied by the received TE the evaluation pointer is reset to the proposition $!C$. Else, if the proposition $E$ is satisfied, the propositions $!C$, $!D$ and $E$ become FETCHED. Then, property goes to the FETCHED state and it becomes true.

4.1.4.3 ASL Property Interpretation

An assertion is defined by a property. A property identifies a behaviour that may not by punctual but composed by a sequential events over the time, and thus, an assertion may have several active property instances overlapped over the time. For example, the property described in Figure 4.8 specifies that a proposition $C$ shall be satisfied if propositions $A$ and $B$ occur.

In the following, the non-overlapping interpretation mode and the overlapping interpretation mode are explained based on the example of the Figure 4.8.


4.1.4.4 Non-overlapping mode

The non-overlapping mode is the simplest implication mode where no property overlapping occurs. In the non-overlapping mode the assertion contains a unique property instance.

Let us analyse an example presented in Figure 4.9: \( A, B \Rightarrow C \). When the simulation starts, a property instance is created and the evaluation process is launched. If \( A \) occurs, then the proposition \( A \) is evaluated and the assertion evaluator focuses on the proposition \( B \). While waiting for \( B \) to occur, all the possible occurrences of \( A \) are ignored. If \( B \) occurs, then the proposition \( B \) is evaluated and the assertion evaluator focuses on the proposition \( C \). Finally, if \( C \) occurs, then the proposition \( C \) is evaluated and the assertion evaluator restarts the assertion evaluation waiting for \( A \) events.

4.1.4.5 Overlapping mode

This evaluation mode allows the existence of several property instances related to an assertion. When a overlapping-enabled proposition reaches the FETCHED state, a new property instance is created. The new property instance is a clone of the original except that the evaluation pointer directs to the last satisfied proposition.

Let us analyse the example found in Figure 4.10: \( A, B \Rightarrow C \). In this example the overlapping mode is only enabled for propositions \( A \) and \( B \).

**Step 1:** In Figure 4.10(a) the simulation is launched and the default instance (I1) of the property is created. All the propositions of the new instance are in \textit{PENDING} state.
4.1 Proposed Assertion Specification Language

Step 2: In Figure 4.10(b) a new event satisfying the proposition $A$ is received. As $A$ is an overlapping proposition, a new property instance (I2) is cloned from I1. Then, the proposition $A$ of I1 is marked as FETCHED and its evaluation pointer is established in proposition $B$.

Step 3: In Figure 4.10(c) a new event satisfying the proposition $B$ is received. As $B$ is an overlapping proposition, a new property instance (I3) is cloned from I1. Then, the proposition $B$ of I1 is marked as FETCHED and its evaluation pointer is established in proposition $C$.

Step 4: In Figure 4.10(d) a new event satisfying the proposition $C$ is received. The first instance (I1) is satisfied and thus, it is removed. As $C$ is non-overlapping, no new instance is created.
4.1.5  ASL SYNTAX

This section describes the syntax of the proposed ASL. The organization of this section is based on the ASL architecture presented in Section 4.1.3.

4.1.5.1  Event Layer

An event is characterized by an identifier, a port, a transactional operation, and the name of the transaction. The concrete syntax of an event is defined as:

\[ @Event_ID\{Port.Operation.Transaction\} \]

The syntax below specifies a transactional event in which a port Port1 sends a writing transaction transac1:

\[ @1\{Port1.write.transac1\} \]

The operator ‘@’ identifies unequivocally a transactional event within a proposition. In this example, the event is determined by the number 1. If no identifier is included, ASL will use the next unique number within an integer number set.

4.1.5.2  Proposition Layer

The proposition level includes the event, time, sentence, and proposition mode concepts. The concrete syntax of a proposition is defined as:

\[ Mode\ Type\ Event[Time]\{Sentence\} \]

The mode represents whether the proposition is overlapping or non-overlapping. The type indicates whether the proposition is positive or negative. The time in ASL is always relative to the previous event. In the case of the first event, the time is determined with respect to the start of the simulation.

The syntax below specifies a TE in which a port Port1 sends a writing transaction transac1 within 2 milliseconds (0 to 2 time interval):

\[ @1\{Port1.write.transac1\}[0:2(ms)] \]
4.1 Proposed Assertion Specification Language

The operator \([t_1 : t_2]\) identifies the time interval in which the event shall occur, that is, the event shall occur not before time instant \(t_1\) and not after \(t_2\). If no identifier is included, ASL will assume an infinite interval \([0 : \infty]\).

The sentence extends the event concept to include boolean and arithmetic expressions. The result of a sentence is a boolean value; it shall be either true or false. The default value of a sentence is true. If no sentence is specified, ASL assumes the default true value.

The syntax below specifies a transactional event in which a port \(\text{Port1}\) sends a writing transaction \(\text{transac1}\) whose attribute \(\text{attr1}\) is 0. The transaction is sent within 2 milliseconds (0 to 2 time interval).

\[
\text{@1\{Port1.write.transac1\}[0:2\text{(ms)}](\text{@1\{transac1.attr1\}==0})}
\]

There are two types of propositions: positive propositions and negative propositions. The operator `!` identifies a negative proposition. If no type operator is included, ASL will assume a positive proposition. The following negative proposition is satisfied when the defined event does not occur, or the event does not occur in the specified time interval, or the specified sentence is false.

\[
!(\text{@1\{Port1.write.transac1\}[0:2\text{(ms)}](\text{@1\{transac1.attr1\}==0}))
\]

As described in Section 4.1.4.3, the interpretation and the evaluation of a property changes if the overlapping mode is assumed in one or more of its propositions. The overlapping mode of a proposition is indicated using the operator `||`. If no proposition mode operator is included, ASL will assume a non-overlapping proposition mode. The following syntax defines an overlapping proposition:

\[
|| \text{@1\{Port1.write.transac1\}[0:2\text{(ms)}](\text{@1\{transac1.attr1\}==0})
\]

ASL allows verification designers to define global variables in order to help and make easier the assertion description. The usage of global variables in ASL is analogous to variables in other programming languages. First, global variables are declared assigning them a name; then, their value can be read and written within propositions. Global variables are not associated to either entities or operations; they are accessible to all properties specified for a system.
For example, the following syntax describes a global variable var1 taking the value ‘true’:

\[ G(var1) = \text{true} \]

### 4.1.5.3 Sequence Layer

A sequence is a succession of several propositions separated by the operator ‘;’. A sequence is satisfied if all the specified propositions are satisfied in the defined order within the sequence.

\[
\text{Proposition}_1; \\
\text{Proposition}_2; \\
\cdots \\
\text{Proposition}_N
\]

The syntax below specifies that the port Port1 sends a writing transaction transac1 and that it does not send a reading transaction transac2 within 10 milliseconds:

\[ @1\{\text{Port1.write.transac1}\}; \non\{@2\{\text{Port1.read.transac2}\}[0:10\text{ms}]) \]

### 4.1.5.4 Property Layer

A property relates an antecedent and a consequence by means of an implication. The antecedent and the sequence are sequences and the implication is the operator ‘=>’. The concrete syntax of a property is:

\[
\text{Antecedent} \Rightarrow \text{Consequence}
\]

The following specification determines that if the port Port1 sends a writing transaction transac1 and that it does not send a reading transaction transac2 within 10 milliseconds, then the port Port2 shall send a writing transaction transac3:

\[ @1\{\text{Port1.write.transac1}\}; \non\{@2\{\text{Port1.read.transac2}\}[0:10\text{ms}]) \Rightarrow @3\{\text{Port2.write.transac3}\} \]
4.1 Proposed Assertion Specification Language

If one or more propositions of a property are defined using the overlapping mode, the operator ‘\(\Rightarrow\)’ shall define the maximum number of property instances allowed during the verification process. This is defined as ‘\(\Rightarrow(n)\)’, where ‘\(n\)’ specifies the maximum number of property instances allowed. If no maximum number of property instances is defined, ASL will assume that only one property instance will be available during the system verification, thus disabling all overlapping operators included in the property.

For example, the previous property syntax example is extended activating the overlapping mode in the first proposition. The implication operator allows a maximum of 5 property instances during the system verification. The syntax of the specified property is:

\[
|| @1\{\text{Port1.write.transac1}\};
!(@2\{\text{Port1.read.transac2}[0:10\text{(ms)}]\})
\Rightarrow\{5\}
@3\{\text{Port2.write.transac3}\}
\]

As presented in Section 4.1.5.2, ASL allows verification designers to define global variables to make easier the assertion description.

Propositions can be used to change the value of global variables. For example, the following listing specifies that the value of the variable \(\text{var1}\) is set to true whenever the \(\text{Port1}\) writes the message \(\text{transac1}\):

\[
@1\{\text{Port1.write.transac1}\}
\Rightarrow
\@2(G\{\text{var1}\}=true)
\]

In the following example, a global variable is used in the expression of a proposition:

\[
@1\{\text{Port1.write.transac1}\}[G\{\text{var1}\}=true]
\Rightarrow
@2\{\text{Port1.write.transac2}\}
\]

4.1.5.5 CPU-Memory Example Application

Let us suppose a simple TLM system model consisting of a CPU and a memory component. The functionality of the CPU-Memory system is divided into three use cases as presented in Figure 4.11. The first use case is depicted in Figure 4.11(a) and defines how a new value is written in the memory. The CPU sends the \(WriteData\) transaction to the memory
to write the value specified by the \textit{Data} attribute in the memory position determined by the \textit{Address} attribute. The memory sends the \textit{ConfirmationWriteData} transaction to the CPU to indicate the result of a previous write request on the memory. The \textit{WriteOK} attribute of the \textit{ConfirmationWriteData} transaction specifies whether the writing process was successful or not. \textit{Data} and \textit{Address} attributes indicate the value and the memory position that the CPU tried to write.

The second use case defines how a value is read from the memory. This use case is shown in Figure 4.11(b). The CPU sends the \textit{ReadData} transaction to the memory to request the value stored in the memory position determined by the \textit{Address} attribute. The memory sends the \textit{SendData}
transaction to the CPU to indicate the result of a previous read operation on the memory. The *Data* and *Address* attributes indicate the value and the memory position that the CPU tried to request.

The third use case is depicted in Figure 4.11(c) and defines how the memory component notifies the CPU that it is full. The memory component sends the *MemoryState* transaction to the CPU to notify the status of the memory. The *Full* attribute of *MemoryState* determines whether the memory is full or not. The CPU sends the *ConfirmationMemoryState* transaction to the memory to indicate that the *MemoryState* transaction was correctly received.

First, three functional requirements had been specified in natural language for the CPU-Memory system. Then, these functional requirements have been unambiguously specified using ASL.

Requirement 1:

*The memory component shall send an acknowledgement message before 5 milliseconds whenever the CPU requests a new value to be written. If the memory is full it shall not perform the writing process and shall notify the CPU with a negative acknowledgement.*

This requirement is specified by two ASL assertions. The first assertion specifies that the CPU shall receive the *ConfirmationWriteData* transaction before 5 milliseconds when it sends the transaction *WriteData* to a memory with empty positions. The global variable *MemFull* helps to determine when the memory is full. Additionally, the *WriteOK* attribute of the transaction *ConfirmationWriteData* shall be true and the attributes *Data* and *Address* shall be equal to the attributes *Data* and *Address* of the request transaction.

\[
\begin{align*}
&\text{||} \ 01\{\text{cpu\_out.write.WriteData}\}(G\{\text{MemFull}\}==\text{false}) \\
&\quad \Rightarrow \{10\} \\
&\quad \text{02}\{\text{cpu\_in.read.ConfirmationWriteData}\}[0:5\text{ms}]_1 \{ \\
&\quad \quad \text{02}\{\text{ConfirmationWriteData.WriteOK}\}==\text{true \ &} \\
&\quad \quad \text{02}\{\text{ConfirmationWriteData.Data}\}==\text{01}\{\text{WriteData.Data}\} \ & \\
&\quad \quad \text{02}\{\text{ConfirmationWriteData.Address}\}==\text{01}\{\text{WriteData.Address}\} \}
\end{align*}
\]

The CPU shall also receive the transaction *ConfirmationWriteData* before 5 milliseconds when it sends the transaction *WriteData* to a full mem-
ory. However, in this case the attribute WriteOK shall be false to notify that the writing request was unsuccessful. In this case there shall be also a correspondence between the data and address values.

\[\begin{array}{l}
\text{| I } @1\{\text{cpu\_out.write.WriteData}\}\{\text{G\{MemFull\}}=\text{true}\} \\
\text{ } @2\{\text{cpu\_in.read.ConfirmationWriteData}\}\{0:5\{\text{ms}\}\}\{\text{WriteOK}=\text{false} \text{ } \&\& \text{ } @2\{\text{ConfirmationWriteData.Data}\}=\@1\{\text{WriteData.Data}\} \text{ } \&\& \text{ } @2\{\text{ConfirmationWriteData.Address}\}=\@1\{\text{WriteData.Address}\}\}
\end{array}\]

Both assertions specifying the writing request of the CPU contain an initial overlapping proposition. Only the first proposition accepts an overlapping mode; the proposition forming the consequence is non-overlapping. The maximum number of property instances is limited to 10.

Requirement 2:

*Whenever the CPU request to read a value, the memory shall send the data before 3 milliseconds.*

The assertion below specifies that whenever the CPU sends a ReadData transaction, it shall receive a SendData transaction with the Address attribute equal to the requested memory address before 3 milliseconds.

\[\begin{array}{l}
@1\{\text{cpu\_out.write.ReadData}\} \\
\text{ } @2\{\text{cpu\_in.read.SendData}\}\{0:3\{\text{ms}\}\}\{\text{SendData.Address}=\@1\{\text{ReadData.Address}\}\}
\end{array}\]

Requirement 3:

*When the memory is full it shall notify the CPU to prevent more writing requests. The CPU shall confirm the notification from the memory not later than 1 milliseconds. Moreover, the memory shall notify the CPU when a memory position becomes free.*

This requirement is specified by two ASL assertions. The first assertion determines that whenever the memory sends a MemoryState transaction
4.2 Verification Framework Implementation

The proposed framework runs in parallel to the simulation engine, evaluating the assertions and collecting statistical data of the verification. Figure 4.12 illustrates how the verification framework is used with a SystemC model. The system model is made up of three SystemC modules: Source, DUV and Sink. These modules are interconnected using monitorizable

![Diagram of Verification Framework](image)

Figure 4.12: Verification framework application example.

notifying that the memory is full (Full==true), it shall receive a ConfirmationMemoryState transaction from the CPU no later than 1 millisecond. Additionally, the global variable MemFull is set to true in order to make easier the definition of other assertions.

@1{mem_out.write.MemoryState}(@1{MemoryState.Full==true}) =>
@2{mem_in.read.ConfirmationMemoryState}[0:1(ms)];
@3(G{MemFull}=true)

Also, whenever the memory sends a MemoryState transaction notifying that the memory has empty space (Full==false), it shall receive a ConfirmationMemoryState transaction from the CPU no later than 1 millisecond, as expressed in the following assertion. In this case the the global variable MemFull is set to false.

@1{mem_out.write.MemoryState}(@1{MemoryState.Full==false}) =>
@2{mem_in.read.ConfirmationMemoryState}[0:1(ms)];
@3(G{MemFull}=false)
4.2.1 FRAMEWORK ARCHITECTURE

The software architecture of the verification framework is depicted in Figure 4.13. The verification framework has been developed using C++, SystemC [6], SystemC Verification Library [121], Boost C++ libraries [122], Xerces [123] and Xalan [124] libraries. Figure 4.14 shows a class diagram of the most relevant parts of the framework. The main features of the elements shown in the figure are described below.

4.2.1.1 Class DUTEvent

In the proposed framework, TEs are implemented by class DUTEvent described in Figure 4.15. It has the following attributes:

TLM based transactional interfaces provided by the Verification Framework. The verification framework obtains information about TEs occurring in the interfaces by means of the monitor elements attached to the former interfaces.

The verification code is not embedded within the system model code, providing independence to the design and verification teams. The assertions are specified by means of XML files, which can be easily understood by human beings. Additionally, there are many tools available to manage and modify XML files.

The description of the proposed framework is organized in three parts. Firstly, the architecture of the framework is presented. Secondly, we describe how the framework integrates within the SystemC simulation and performs the assertion evaluation. Finally, the XML interface with the framework is described.
4.2 Verification Framework Implementation

![Figure 4.14: Verification Framework class diagram.](image)

- **Source**: It is the source interface element where the TE was produced. In TLM, the available interfaces are the transactional ports.

- **Operation**: It defines the operation carried out by the interface; i.e.: read, write, etc.

- **Data**: It contains the data associated to the TE. In TLM, it contains the transaction sent or received through the interface.

- **Time**: It is the instant of time when the TE happened.

### 4.2.1.2 Class Proposition

Class *Proposition*, described in Figure 4.16, implements the concept of proposition presented in section 4.1.4. The *trigger* is specified by the following attributes:
Chapter 4: An ABV framework for SystemC-TLM

Figure 4.15: DUTEvent class.

```
DUTEvent

+ Source : string
+ Operation : string
+ Time : sc_time
+ Data : DataElement*

+ DUTEvent(source : string, operation : string, data : DataElement*)
+ ~DUTEvent()
+ print() : void
```

Figure 4.16: Proposition class.

```
Proposition

+ Source : string
+ Operation : string
+ StartTime : sc_time
+ EndTime : sc_time
+ Data : DataElement*
+ Expression : PropositionExpression
+ Mode : PropositionMode
+ State : PropositionState

+ Proposition()
+ ~Proposition()
+ eval(event : DUTEvent*) : PropositionState

<<enum>>

PropositionMode

OVERLAPPING
NON_OVERLAPPING

PropositionState

PENDING
FETCHED
UNACHIEVABLE

PositiveProposition

+ PositiveProposition()
+ ~PositiveProposition()
+ eval(event : DUTEvent*) : PropositionState

NegativeProposition

+ NegativeProposition()
+ ~NegativeProposition()
+ eval(event : DUTEvent*) : PropositionState
```
4.2 Verification Framework Implementation

- **Source**: It identifies the transactional port the TE must be associated with.
- **Operation**: It defines the operation to be performed by the transactional port.
- **Data**: It specifies which must be the transaction in the TE.

The time-out condition is defined by a range of time in which the TE must happen. This range of time is specified taking as reference the time instant of the previous TE. Two attributes are used:

- **StartTime**: It defines the beginning of the time-out interval.
- **EndTime**: It defines the end of the time-out interval.

The attribute *Expression* implements the expression part of the proposition. It can be any arithmetic or boolean relationship between the information in the TEs. Class *Proposition* implements resources to check the *Expression* part of the propositions. It has been implemented using the *Regex* library from *Boost C++ Libraries* [122].

The attribute *PropositionMode* specifies whether the overlapping evaluation of the proposition is permitted or not.

The last attribute is *PropositionState*, which stores the state of the proposition. The possible values of the state attribute are:

- **PENDING_TO_FETCH**: It corresponds with the *pending* state of section 4.1.4.
- **FETCHED**: It corresponds with the *fetched* state of section 4.1.4.
- **UNACHIEVABLE**: It corresponds with the *unachievable* state of section 4.1.4.

4.2.1.3 Class Property

Properties are implemented using the class *Property* presented in Figure 4.17. It has the following attributes:
### Property

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>propositions</td>
<td>Proposition*</td>
</tr>
<tr>
<td>numPropositionsInAntecedent</td>
<td>int</td>
</tr>
<tr>
<td>Name</td>
<td>string</td>
</tr>
</tbody>
</table>

```java
+ Property(propertyLiteral : string)
  + Property(name : string, propositions : Proposition*, numPropositionsInAntecedent : int)
  + ~Property()
  + getProposition(index : unsigned) : Proposition*
  + getCurrentProposition() : Proposition*
  + eval(event : DUTEvent*) : void
```

**Figure 4.17:** Property class.

### Assertion

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>concurrentInstancesNumber</td>
<td>unsigned</td>
</tr>
<tr>
<td>currentInstanceIndex</td>
<td>unsigned</td>
</tr>
<tr>
<td>propertyInstances</td>
<td>PropertyInstances*</td>
</tr>
<tr>
<td>Id</td>
<td>string</td>
</tr>
</tbody>
</table>

```java
+ Assertion(identifier : string, property : Property*, concurrentInstNumber : unsigned)
  + ~Assertion()
  + processEvent(event : DUTEvent*) : void
  + updateStatistics() : void
  + recountStatistics() : void
  + getMaxConcurrentInstances() : unsigned
```

**Figure 4.18:** Assertion class.

- NumPropositionsInAntecedent: This attribute specifies the number of propositions that form the antecedent part of the property.
- Propositions: It is a list with objects of type *Propositions*.

#### 4.2.1.4 Class Assertion

Class *Assertion* implements a container class that stores all the instances of the properties during the simulation of the DUV. It is responsible for creating, managing and deleting the property instances. The declaration of *Assertion* is presented in Figure 4.18.

#### 4.2.1.5 Class AssertionsManager

Class *AssertionsManager* is the responsible for the coordination and resource management of the system verification process. *AssertionsManager*
### 4.2 Verification Framework Implementation

#### 4.2.1.6 Classes for Monitorizable Interfaces

The verification framework provides the classes `TransactionalInterfaceIn` and `TransactionalInterfaceOut`, which are wrappers for TLM based transactional interfaces. These Monitorizable Transactional Ports (MTPs) are the system model access point for the verification framework. MTPs are needed to capture the transactions occurring in the transactional ports. From the system simulation point of view, the MTPs provided by the verification framework act as TLM transactional ports. From the point of view of the verification process, MTPs send information about the transactions passing through the monitorizable port to the verification framework monitors. They are responsible for creating a `DUTEvent` object each time a TE is produced in the interface.

#### 4.2.1.7 Class Monitor

Objects of class `Monitor` collect the TEs occurring in the interfaces of the DUV. Each monitor holds an assertion list. The monitor passes the collected TEs to the assertions in the list for evaluation. Monitors communicate with property instances by means of class `Assertion`. The declaration of the class `Monitor` is presented in Figure 4.20.

```plaintext
<table>
<thead>
<tr>
<th>AssertionManager</th>
</tr>
</thead>
<tbody>
<tr>
<td>assertionsIdentifier : string</td>
</tr>
<tr>
<td>monitors : MonitorsCollection</td>
</tr>
<tr>
<td>assertions : AssertionsCollection</td>
</tr>
</tbody>
</table>

+ AssertionManager(monitor : MonitorsCollection, assertionsFile : string)
+ ¬ AssertionManager()
+ bindAssertionsToMonitors() : void
+ recountStatistics() : void
+ SaveStatistics(reportId : string, reportsFileName : string) : void
```

Figure 4.19: `AssertionManager` class.

contains two lists to store the `Assertions` and the `Monitors` as presented in Figure 4.19.
4.2.1.8 Class XMLAssertionsRecorder

This class is responsible for creating the XML verification report when the simulation of the DUV is finished. XMLAssertionsRecorder is described in Figure 4.21.

4.2.1.9 Class XMLAssertionsBuilder

This class is responsible for reading the assertion input XML file and extracting the information. It also creates the assertion list for Assertion-Manager. XMLAssertionsBuilder is described in Figure 4.22.

4.2.2 VERIFICATION FRAMEWORK INTEGRATION WITH A SYSTEMC EXECUTABLE MODEL

The centralized architecture of the proposed verification framework facilitates its use in any SystemC based TLM system model. The following actions are needed in order to integrate the proposed verification framework:

1. Create an AssertionsManager object inside the system testbench code in order to enable all the verification framework capabilities.
4.2 Verification Framework Implementation

### 4.2.3 SYSTEM MODEL VERIFICATION PROCESS

Figure 4.23 shows the operations performed by the verification framework in each of the three phases of the SystemC simulation. These operations are described in detail below.

#### 4.2.3.1 Phase 1

This first phase takes place before the SystemC simulation starts. Once, the system modules are elaborated and the module hierarchy of the system is set up, the AssertionsManager uses the XMLAssertionsBuilder to parse the XML file where the assertions have been specified. With the information

---

**Figure 4.22: XMLAssertionsBuilder class.**

2. Register MTPs in the AssertionsManager. This code is implemented in the system testbench.

Another action is needed in case the system design team does not use MTPs right from the first model of the system under development. In this case, after an AssertionsManager object is created, all the transactional ports shall be replaced with the MTPs provided by the proposed verification framework.
Figure 4.23: AssertionsManager verification process.
extracted from the XML file, the list of Assertion objects is generated inside the AssertionsManager.

Next, one transactional monitor is created for each monitorizable interface. Valid monitorizable points are input or output transactional ports. All the monitors are passed to the AssertionsManager. In the example shown in Figure 4.12, the verification monitors both the transactional interface communicating Source and DUV, and the transactional interface communicating DUV and Sink. The main task of the monitorizable interfaces is to detect TEs occurring in the interfaces and pass the information to the attached monitor.

At this point of the execution, the AssertionsManager contains two lists representing all the assertions and all the monitors of the DUV. In the final step of this phase, the AssertionsManager binds the assertions with the appropriate monitors. Thus, each transactional monitor stores an assertion list specifying which assertions the monitor must report TEs to.

### 4.2.3.2 Phase 2

The system simulation and so, the evaluation of the properties, takes place after the SystemC `sc_start()` function is called. During the system simulation, the monitorizable interfaces capture the occurrence of transactions on-the-fly and translates each transaction occurrence into DUTEvent objects as depicted in Figure 4.24. The monitorizable interface calls the `raiseEvent()` function to pass the DUTEvent to the attached Monitor. The Monitor calls the `processEvent()` function for each assertion in its list of
assertions. Thus, the DUTEvent is distributed to all the assertions bound
to the monitor for evaluation.

The Assertion class contains a Property list. Each element in this list
represents a property instance of the Assertion. An Assertion can con-
tain one or several instances during the system simulation to support the
overlapping and non-overlapping evaluation modes.

The Assertion uses the TE information received from the verification
monitors to evaluate the collection of property instances. Each Property
object keeps track of the property evaluation process using an index pointer
inside the proposition list. This pointer points to a proposition pending to
evaluate inside the property. The property evaluation is a sequential process
where the propositions are evaluated one after the other.

4.2.3.3 Phase 3

Once the simulation of the DUV is finished, the Assertion Manager
collects the statistical information from each assertion. Then, the Asser-
tionManager generates an XML report file by means of the XMLAssertions-
Builder class. This XML report file contains all the statistical information
of the verification process.

At the end of the simulation, the property instances can be in four pos-
sible states that match the four property states described in section 4.1.4:

1. PENDING_TO_FETCH: It corresponds with state pending. It means
that no TE was produced that initiated the evaluation of that prop-
erty instance.

2. FETCHED: It corresponds with state fetched. It means that the TEs
involved in that property instance satisfy the property.

3. UNACHEIVABLE: It corresponds with state unachievable. It means
that the TEs involved in that property instance did not satisfy the
property.

4. PARTIALLY_FETCHED: It corresponds with state partially-fetched.
It means that TEs were produced that initiated the evaluation of that
property instance, but the simulation did not last enough to arrive to
a result.
4.2 Verification Framework Implementation

4.2.4 XML FRAMEWORK INTERFACE

The XML verification framework interface consists of three files: an assertion description file, an assertion manager configuration file, and a report file. The first two files are written by the verification team. The configuration file is used to specify which actions the AssertionsManager has to perform when a property instance is found to be satisfied or unachievable. The third file, the report file, is generated by the AssertionsManager when the simulation of the DUV is finished.

4.2.4.1 Assertion Description XML file

The basic structure of the XML file that describes the assertions is shown in Listing 4.1. The main node of the assertion descriptions file is the AssertionsManager where the global variables and the assertions are defined. Listing 4.1 also shows the structure of an assertion where a system property is specified. The Assertion node is specified by an assertion name (Name) and the MaxOverlappingInstances attribute. The Name field must be unique in the system simulation context.

The XML syntax of a proposition is the implementation of the attributes described in Section 4.1.5.2. The structure of a Proposition XML node is shown in Listing 4.1. If the attributes StartTime and EndTime are not specified the default [0:∞] time interval is applied to the proposition. The Expression attribute is a string, and it supports the C++ condition syntax to describe boolean and arithmetic expressions. If not specified, the Expression is assumed true. Transaction attributes can be referenced in expressions by means of the following syntax:

@id{transaction_name.attribute_name}

The implemented ExpressionEvaluator uses the DataIntrospection capability of the Verification Framework to retrieve the attribute values of the system transactions. This provides flexibility and frees the verification team from the details of the model implementation.

The verification team can define global variables to facilitate the description of the properties; e.g.: to describe internal states of the system model. Global variables have three attributes as shown in Listing 4.1: Name, Type and Value, that specify the name of the variable, its type and
Chapter 4: An ABV framework for SystemC-TLM

Listing 4.1: System Assertions XML structure.

```
<AssertionsManager>
  <GlobalVariables>
    <Variable Name="" Type="" Value="" />
    <Variable Name="" Type="" Value="" />
  </GlobalVariables>
  <Assertions>
    <Assertion Name="" MaxOverlappingInstances="">
      <Property>
        <Antecedent>
          <Proposition Overlap="..." Type="..." Id="..."
            Interface="..." Operation="..." Data="..."
            StartTime="..." EndTime="..." Expression="..."/>
        </Antecedent>
        <Consequence>
          <Proposition .../>
          <Proposition .../>
        </Consequence>
      </Property>
    </Assertion>
    ...  
  </Assertions>
</AssertionsManager>
```

the default value respectively. Integer, double precision, string and boolean data types are supported for global variables. Their value can be modified by performing an assignment within the Expression attribute. The global variables are referenced within the Expression attribute of a proposition using the following syntax:

```
G(global_variable_name)
```

Let us suppose again the CPU-Memory system model presented in Section 4.1.5.5. For this system, we have the following property expressed in natural language:

*If a write operation is performed on memory, no more write operations are performed at the same memory address, and the*
4.2 Verification Framework Implementation

Listing 4.2: XML Assertion example.

```xml
<AssertionsManager>
  <Assertions>
    <Assertion Name="CPU_Mem_Read" MaxOverlappingInstances="0">
      <Property>
        <Antecedent>
          <Proposition Id="1" Overlap="DISABLED" Type="POSITIVE">
            Interface="CPU_out_port" Operation="WRITE"
            Data="WriteData"/>
          <Proposition Id="2" Overlap="DISABLED" Type="NEGATIVE">
            Interface="CPU_out_port" Operation="WRITE"
            Data="WriteData" Expression="(@2{WriteData.Address} == @1{WriteData.Address})"/>
          <Proposition Id="3" Overlap="DISABLED" Type="POSITIVE">
            Interface="CPU_out_port" Operation="WRITE"
            Data="RequestData" Expression="(@3{RequestData.Address} == @1{WriteData.Address})"/>
          </Antecedent>
        </Consequence>
        <Consequence>
          <Proposition Id="4" Overlap="DISABLED" Type="POSITIVE">
            Interface="CPU_in_port" Operation="READ"
            Data="ReadData" StartTime="0" EndTime="3"
            Expression="((@4{ReadData.Data} == @1{WriteData.Data}) && (@4{ReadData.Address} == @1{WriteData.Address}))"/>
        </Consequence>
      </Property>
    </Assertion>
  </Assertions>
</AssertionsManager>
```

memory is requested to read that address; then, the memory shall return the value formerly written before 3 ms.

This property can be split into four propositions, where each proposition is focused on one event. Listing 4.2 shows how this property can be described using the proposed XML syntax. In this example, the expression of the second proposition specifies that the Address attribute of the WriteData transaction that belongs to the proposition with Id=2 is equal to the Address attribute of the WriteData transaction that belongs to the proposition with Id=1.
Listing 4.3: Verification engine XML configuration file.

```
<AssertionsManagerConfig>
  <Actions>
    <Case Event="ON_FETCHED" Action="CONSOLE_OUTPUT"/>
    <Case Event="ON_UNACHEIEVABLE" Action="STOP"/>
  </Actions>
</AssertionsManagerConfig>
```

4.2.4.2 AssertionsManager XML configuration file

The verification team can specify which actions the AssertionsManager has to carry out when a property instance becomes either fetched or unachievable. By default, “CONSOLE_OUTPUT” and “STOP” actions are available. The former sends a message to the console output, while the latter stops the system simulation. The flexible and layered structure of the AssertionsManager makes it possible to easily add new user defined actions. Linking an action to a property instance state is not mandatory and no action is performed by default.

Listing 4.3 is presented as an example. In this case, fetched property instances are reported on screen and unachievable property instances stop the system simulation immediately.

4.2.4.3 AssertionsManager XML report file

The AssertionsManager generates an assertions report file when the system simulation is finished. In each Assertion node the report file specifies the statistical information of the antecedent and consequence sentences. The number of maximum concurrent, fetched, unachievable and partially fetched instances is reported. Due to the flexible and layered structure of the verification framework, adding new user defined statistical data reports is straightforward.

The example shown in Listing 4.4 is the XML report of the assertion described in Listing 4.2. In this case, the MaxConcurrentInstancesNumber attribute is one because all the propositions of the assertion are defined as non-overlapping. As show in the example, five antecedent instances have been fetched, however only four consequence instances have been satisfied.
4.3 Concluding Remarks

Listing 4.4: XML Report file structure.

```xml
<AssertionsReport>
  <Assertions>
    <Assertion Name="CPU_Mem_Read">
      <Statistics MaxConcurrentInstancesNumber="1">
        <Antecedent>
          <Fetched NumInstances="5"/>
          <Unachievable NumInstances="0"/>
          <PartiallyFetched NumInstances="0"/>
        </Antecedent>
        <Consequence>
          <Fetched NumInstances="4"/>
          <Unachievable NumInstances="0"/>
          <PartiallyFetched NumInstances="0"/>
        </Consequence>
      </Statistics>
    </Assertion>
    ...
    <Assertion Name="CPU_Mem_Write">
      ...
    </Assertion>
  </Assertions>
</AssertionsReport>
```

4.3 CONCLUDING REMARKS

The ABV approaches proposed in the literature lack an effective assertion specification language that enables both to build assertions fast at TLM and to reuse assertions easily between different models. Several works propose to embed the verification code in the design model which produces complex executable code and makes it difficult for the verification team to work in parallel with the design team. An ABV framework that enables the design and verification teams to work independently is missing in the literature.

The ASL language has been proposed in this chapter for assertion based transaction level system specification. ASL enables the verification team to write assertions simply and fast, with increased detail within a SystemC–TLM modelling environment. Thus, unwanted behaviours are detected in the early stages of the modern electronic systems design flow, resulting in greater margin of reaction and reduced development cost. ASL enhances antecedent–consequence style properties with temporal constructs, and thus, assertions can be created for both timed and untimed SystemC–
TLM coding styles. As a result, ASL is not only valid in the initial stages of the design; it is prepared to support a TLM–to–RTL model driven design flow.

A novel ABV verification framework supported by a library built on top of SystemC has also been presented in this chapter. The application of the proposed TLM verification framework to a TLM SystemC model is very simple.

The proposed TLM framework helps to reduce the development time in design flows with team redundancy (independent design and verification teams) from the initial natural language specification. This is achieved thanks to the decoupling of the assertion creation. This decoupling is provided by several features of the proposed TLM verification framework. Firstly, the assertions are not embedded in the design model code. Assertions are described in user friendly XML files following the ASL syntax. Secondly, a modification in the assertion descriptions does not require any system model recompiling. The proposed verification framework loads dynamically the XML assertions file during the start-up phase of the system model execution. Modifications to the input assertion set are applied to the ABV process the next time the system model is executed. Thus, an integrated, powerful and flexible framework has been created in order to simulate and verify complex systems starting from a very high abstraction level.

This chapter has been the precursor for the paper [125].
Chapter 5

Embedding Matlab in SystemC-TLM

The electronic systems design approach based on Transaction Level Modeling (TLM) creates executables models describing the system to be built, as discussed in Section 2.3. SystemC has become the de facto standard to create executable models of electronic systems. These executable models are software models of the system that can be analysed and executed on a workstation. At each design step, these models become an executable specification of the system. These models can be used for verification of correctness of each design step.

In the initial models, the designer focuses on the functional description of the system and on the information exchanged between the system and the environment. During the first design steps, the system is divided recursively into smaller parts. The detail of the models increments to describe the functionality of the different parts and the information exchanged between
them. These early system definition and architecture exploration models are very high abstract models. For the functionality, the focus is on what functions shall be performed, not on how the functions shall be performed. For the communications, the focus is on what information is exchanged, not on how the information is exchanged.

Once what the system modules shall do has been defined, the next design step is to develop the algorithms. The algorithms devised for each module, will be included in the next system model. This new TLM model is a first description of the how, and it is intended to check that the different algorithms cooperate correctly to achieve the high–level functionalities. At this stage of the design process, design iterations will be performed for architecture and algorithm tuning.

This chapter proposes to integrate a MATLAB Engine into SystemC–TLM executable models, as depicted in Figure 5.1. During the initial iterations to tune the architecture and the algorithms, the SystemC–TLM model of the system will execute the MATLAB description of the algorithms provided by the algorithm–design team. This way, the system–design team can work in SystemC; the algorithm–design team can work in Matlab; and the TLM SystemC–MATLAB model can reuse the verification resources developed for previous SystemC models. Furthermore, the chapter proposes an abstraction layer for the MATLAB Application Programming Interface (API). This abstraction layer simplifies the interaction with the MATLAB Engine from the SystemC model. Thus, the resulting code for the SystemC–MATLAB model is clear; and the system–design team can focus on the system description. The proposed approach facilitates verifying that the different algorithms cooperate correctly to achieve the high–level functionalities. Therefore, the proposed approach enables a robust design–flow with a reduced development cost.

The chapter is organized as follows. Section 5.1 analyzes the MATLAB C API. Section 5.2 describes the proposed abstraction layer and its application in an example. Finally, section 5.3 summarizes the conclusions of this chapter.

5.1 MATLAB’S C API

MATLAB provides an API to enable access to the MATLAB Engine from C programs. This API consists of several routines to handle the MATLAB
5.1 Matlab’s C API

Engine sessions and the exchange of data between the C application and the MATLAB Engine. However, the direct use of these routines in the SystemC code is cumbersome; i.e.: it results in a awkward code, and it distracts the system–design team from the system description.

In order to access the MATLAB Engine, the MATLAB’s C API offers routines to open a MATLAB session as a separate process. When a MATLAB session is opened, two pipes are created between the C application process and the MATLAB Engine process. Opening and closing a MATLAB session is time consuming. Therefore, for optimization of the simulation time, MATLAB sessions should be opened and closed only once per simulation whenever it is possible. Additionally, in order to reduce the number of
MATLAB Engines used and, thus, the cost of the simulation setup, the same MATLAB Engine should be shared between the SystemC modules whenever it is possible. The SystemC model developer is responsible for handling all the session issues and making accessible the appropriate MATLAB Engine session pointer to the different SystemC modules.

The exchange of data between the C application and the MATLAB algorithms is carried out using variables of type `mxArray` and a set of functions provided by MATLAB Engine. The traditional procedure to transfer data stored in a C variable to a variable belonging to the MATLAB Engine can be summarized as follows:

1. Create a `mxArray` object that matches the C variable type.
2. Fill the newly created `mxArray` appropriately with the data stored in the C variable. This task can require handling of pointers at low level, which results in heavy code and is error prone.
3. Send the `mxArray` object to the MATLAB Engine.
4. Delete the `mxArray`.

And the procedure to transfer data stored in a MATLAB variable of the MATLAB Engine workspace to a C variable is as follows:

1. Get the `mxArray` from the MATLAB workspace corresponding to the desired MATLAB variable.
2. Fill the C variable with the data in the `mxArray`. This task can also require tedious and error prone low level pointer handling.
3. Delete the `mxArray`.

If we want to process data stored in a SystemC variable using an algorithm specified in MATLAB by means of an M-file or a Simulink model, and retrieve the result of this processing, the SystemC model developer needs to implement the above tedious data exchange procedure. Furthermore, the synchronization between the representations of the data as SystemC variables and as MATLAB Engine workspace variables has to be implemented by the SystemC model developer.
5.2 MATLABENGINE++ IMPLEMENTATION

The direct use of the MATLAB’s C API in a SystemC executable model is a cumbersome and awkward task. There are many issues that the system–design team is demanded to implement and look after. Moreover, these issues are exclusively related to the SystemC–MATLAB interaction; they are error prone; and they are not related to the system design. In order to be able to use the algorithm M–file (or Simulink) specifications in a SystemC model efficiently, an abstraction layer is needed to free the system–design team from the SystemC–MATLAB low level issues. This way, the system–design team avoids tedious and error prone tasks; they can concentrate on the design of the system; and the development cost is reduced. Section 5.2 describes the proposed abstraction layer.

5.2 MATLABENGINE++ ARCHITECTURE

MatlabEngine++ is built on top of the MATLAB’s C API [36] and the C++ standard library, as shown in Figure 5.2. The software architecture of
the MatlabEngine++ library is depicted in Figure 5.3 by means of a Unified Modeling Language (UML) class diagram. In order to have a flexible and scalable solution, the architecture of the proposed approach relies on four main classes: MatlabSession, which implements the concept of a MATLAB session; MatlabSessionConfig, which stores the configuration of a MATLAB session; MatlabVar_b, which handles the data exchange between the C++ variables and the MATLAB session variables; and MatlabManager, which implements the resources to handle the different sessions from the C++ application. Class MatlabVar_b can be specialized to handle specific data types. The features of these four classes are described below.

5.2.1.1 Class MatlabSession

Class MatlabSession shown in Figure 5.4 represents a MATLAB Engine workspace. This class provides functions to easily manage a MATLAB session: open a session; close a session; send commands to a session; and retrieve the result from a session. Commands are sent to MATLAB sessions by calling the session’s function EvalString(string eval_str), where eval_str is a string representing the command as if it were written in the MATLAB console.
MatlabSession

- name : string
- engine : Engine*
- engine.opened : bool
- session.config : MatlabSessionConfig*
- engine.buffer : char*
- var.vector : MatlabVar b[*]

+ MatlabSession(session.name : const char*)
+ MatlabSession(session.name : string)
+ MatlabSession(session.name : const char*, session.config : MatlabSessionConfig*)
+ MatlabSession(session.name : string, session.config : MatlabSessionConfig*)
+ ~MatlabSession()
+ GetSessionConfig() : MatlabSessionConfig*
+ GetEngine() : Engine*
+ OpenEngine() : bool
+ CloseEngine() : bool
+ EvalString(eval.str : string) : bool
+ RegisterVariable(matlab.data : MatlabVar b*) : void
+ GetBuffer() : string
+ GetName() : string
+ DumpMatlabSessionInfo() : void
+ MatlabSession(session : MatlabSession&)
+ operator=(session : MatlabSession&) : MatlabSession&
+ CheckError(eval.str : string) : bool

Figure 5.4: MatlabSession class.

From the functional point of view MatlabSession is the access point to the MATLAB workspace. When a MatlabSession is opened it creates a MATLAB process. This process runs in the background as a separate process from the main C++ program and executes all the commands requested through EvalString.

5.2.1.2 Class MatlabSessionConfig

Class MatlabSessionConfig is depicted in Figure 5.5. This class encapsulates the configuration of a MATLAB session. A MATLAB session has attributes such as: Host, which specifies the name of the machine where the MATLAB session will be opened; Dir, which is the path to the MATLAB executable within the machine specified as Host; and Options, which specifies additional MATLAB options. The Host attribute enables to configure a MATLAB session in the same machine where the SystemC simulation runs or in another machine accessible through the local network. The class provides functions to set and get the values of these attributes.

When a MatlabSession object is created, a MatlabSessionConfig object is passed as a constructor argument. The MatlabSessionConfig object is attached to the MatlabSession object to store its configuration. Moreover, the
same `MatlabSessionConfig` object can be attached to several `MatlabSession` objects within a multi-session application to handle several identical MATLAB sessions. A `MatlabSession` object gets the configuration to properly open a MATLAB session from the attached `MatlabSessionConfig` object’s attributes.

### 5.2.1.3 Class MatlabManager

Class `MatlabManager` is the root element of the MatlabEngine++ library. This class is responsible for the resource management and provides routines to easily create and access `MatlabSession` objects and `MatlabSessionConfig` objects. `MatlabManager` supports the management of a multi-session and multi-configuration MATLAB environment. `MatlabManager` contains two lists to store `MatlabSession` and `MatlabSessionConfig` objects allocated during the C++ application execution. Moreover, `MatlabManager` automatically manages the memory of the created `MatlabSession` and `MatlabSessionConfig` objects; i.e.: the C++ developer does not need to free any memory associated to the MATLAB interaction at the end of the application.
A key point of `MatlabManager` is its implementation using a *singleton* design pattern [126]. The objective of a singleton is to ensure a class only has one instance, and provide a global point of access to it. Thanks to the singleton design pattern, `MatlabManager` is unique in the C++ application and we access the same `MatlabManager` object from anywhere in the C++ application. This feature presents many benefits in a SystemC model where several algorithms need to be executed from different system model modules. With the proposed approach, the same MATLAB session can be requested from different places of the model to execute different algorithms.

### 5.2.1.4 Class MatlabVar_b

Class `MatlabVar_b` is depicted in Figure 5.7. This is the base class for the variable types supported by `MatlabEngine++`. Specialized classes for specific data types are derived from class `MatlabVar_b`. The constructor of class `MatlabVar_b` has two arguments of type string: `name`, which is the name of the variable; and `session`, which specifies the `MatlabSession` object the variable is attached to. `MatlabVar_b` provides virtual functions to enable variable transfers and updates between the C++ application and MATLAB. These routines are implemented in the specific data type classes. Each specialized `MatlabVar_b` based class has its own specific C++ data type and the corresponding `mxArray` implemented as member attributes.
Class MatlabVar\_b also provides routines to automatically manage the synchronization between the representations of data as C++ variables and the representation as mxArray for MATLAB session interaction. The synchronization is handled by means of the state machine depicted in Figure 5.8. This state machine automates the data exchange between the C++ application and the MATLAB session attached to the MatlabVar\_b object.

Three states are possible in the state machine of Figure 5.8:
5.2 MatlabEngine++ Implementation

- *SYNCHRONIZED*: Both the C++ variable and the MATLAB session variable store the same values.

- *CPP_CHANGED*: The C++ variable has changed, but the value of the MATLAB session variable has not been updated.

- *MATLAB_CHANGED*: The MATLAB variable has changed, but the C++ variable has not been updated.

Class *MatlabVar_b* implements the automatically mechanisms to detect if a variable is accessed for reading or for writing in the C++ program. When a *MatlabVar_b* based variable is accessed for writing in the C++ program, its synchronization state is set to *CPP_CHANGED*, regardless its original state. If a variable is accessed to read its value, its synchronization state is checked. If the state is *SYNCHRONIZED* or *CPP_CHANGED*, the variable returns its value. Nevertheless, if the state is *MATLAB_CHANGED*, the variable is updated from the MATLAB session and its state is set to *SYNCHRONIZED* before returning its value. If a command is sent to a MATLAB session, the corresponding *MatlabSession* object updates all the attached variables that are in state *CPP_CHANGED*, and changes their state to *SYNCHRONIZED* before executing the command. The variables attached to the MATLAB session are set to state *MATLAB_CHANGED* after executing the command. The proposed synchronization scheme abstracts efficiently the steps needed to exchange data between the C++ application and MATLAB, and thus, leads to a very intuitive and clean C++ code.

In the following, specific data types supported by *MatlabEngine++* are presented: class *MatlabVar*, class *MatlabVarRow*, class *MatlabVarColumn* and class *MatlabVarMatrix*.

### 5.2.1.5 Class MatlabVar

*MatlabVar* implements a double precision scalar data type. This class stores a *double* type member attribute. The *mxArray* is created using *mxCreateDoubleScalar(…)* function from the MATLAB’s C API. The *MatlabVar* constructor syntax is:

```cpp
MatlabVar(string name, MatlabSession* session)
```
5.2.1.6 Class MatlabVarRow

This class implements a double precision row vector data type. The element size of the row is specified in the constructor of the class. This class stores a `double*` type member attribute.

The `mxArray` is created using `mxCreateDoubleMatrix(1, size,...)` function from the MATLAB’s C API. `MatlabVarRow` constructor syntax is:

```
MatlabVarRow(string name, int size, MatlabSession* session)
```

5.2.1.7 Class MatlabVarColumn

`MatlabVarColumn` implements a double precision column vector data type. The element size of the column is specified in the constructor of the class. This class stores a `double*` type member attribute.

The `mxArray` is created using `mxCreateDoubleMatrix(size, 1,...)` function from the MATLAB’s C API. `MatlabVarColumn` constructor syntax is:

```
MatlabVarColumn(string name, int size, MatlabSession* session)
```

5.2.1.8 Class MatlabVarMatrix

Class `MatlabVarMatrix` implements a double precision matrix data type. This class adds two constructor arguments to specify the matrix dimensions. This class stores a `double**` type member attribute.
The `mxArray` is created using `mxCreateDoubleMatrix(rows, columns,...)` function from the Matlab’s C API. `MatlabVarMatrix` constructor syntax is:

```
MatlabVarMatrix(string name, int rows, int columns, MatlabSession* session)
```

### 5.2.2 MATLABENGINE++ OPERATION

Once the functional modules are verified and the corresponding algorithms are developed, the next design step is to integrate algorithms written in MATLAB into the SystemC system model in order to verify the correct cooperation of the different algorithms. The proposed approach is to replace functional modules with algorithmic modules. The interfaces and the structure of the functional modules based on SystemC implemented in previous abstraction levels are reused in an algorithmic module. However, the data processing is carried out by MATLAB by means of `MatlabEngine++`. Thanks to the fact that the interfaces of the modules are reused, it is straightforward to interchange algorithmic modules and the functional modules within the system model.

The system model depicted in Figure 5.10 is used as an example to illustrate the operation of the MatlabEngine++ library within a SystemC model. `Source` and `Sink` are simple SystemC `sc_module`s connected to the Device Under Test (DUT). `Source` generates the input data for the DUT; whereas `Sink` monitors the output data of the DUT. The DUT consists of...
Listing 5.1: MatlabSession management within sc_main.cpp.

```cpp
#include <systemc>
#include "MatlabEngine++.h"
using namespace sc_core;
using namespace sc_dt;

int sc_main(int argc, char **argv){
    MatlabManager* m=MatlabManager::GetHandler();
    MatlabSessionConfig* c = m->CreateSessionConfig();
    c->SetHost("PC1");
    c->SetDir("/opt/matlab/bin");
    MatlabSession* s=m->CreateSession("Session",c);
    s.Open();
    //Instantiate SystemC modules A and B
    //A and B port binding
    sc_start();
    s.Close();
}
```

two SystemC modules: A and B, which shall perform some data processing. Let us assume that the algorithms performed by A and B are specified in the MATLAB M–files ‘A_algorithm.m’ and ‘B_algorithm.m’ respectively.

Listing 5.1 shows the relevant parts of sc_main.cpp, where the MATLAB sessions are setup. First of all, a pointer to the MatlabManager is obtained by means of function GetHandler() of class MatlabManager. This function always return the same pointer, because MatlabManager is unique in the application thanks to the singleton pattern. A MatlabSessionConfig is created by calling MatlabManager’s function CreateSessionConfig(). Once the MatlabSessionConfig object is configured, a MATLAB session is created by means of MatlabManager’s function CreateSession(...). After the MATLAB session has been created and configured, it can be opened by calling the session’s function Open(). As a result, a new MATLAB process is created as if command /opt/matlab/bin/matlab -nojvm -nosplash were executed at host "PC1". When the SystemC simulation is finished, the MATLAB process is killed and the session is closed by calling the function Close().

Listing 5.2 shows the SystemC code for module A. Let us assume that the MATLAB M–file ‘A_algorithm.m’ takes as input a scalar variable (V_i)
and returns a 2-by-1 column variable (C_o). Within the constructor, the *MatlabSession* is retrieved by means of the *MatlabManager*. Additionally, *MatlabEngine++* variables (V_i and C_o) are defined as member attributes of the modules. These variables are created within the constructor of the algorithmic modules and they are attached to the *MatlabSession* retrieved in the previous step. Finally, the MatlabEngine++ variables are automatically deleted by the *MatlabManager* when the *MatlabSession* is closed.

During the system simulation, the *SC_MODULE* executes the function `EvalString("C_o=A_algorithm(V_i);")` whenever it wants the M-file ‘A_algorithm.m’ to process the data. The user works with variables V_i and C_o as if they were standard C++ variables. Their values are automatically synchronized with the MATLAB session by *MatlabEngine++*. 
Thanks to the singleton pattern, MatlabManager is unique throughout the SystemC model. It is not necessary to pass the MATLAB session as a constructor argument to the SystemC module, and thus, the software interface of the model does not need to be modified. As a result, pure functional modules and modules executing algorithms in a MATLAB Engine are fully interchangeable within the SystemC model. Furthermore, the singleton pattern enables access to the same MatlabManager from any module within the SystemC system model. Therefore, in the example, both SystemC modules A and B can request in their constructors the same MATLAB session. Then, both MATLAB M-files ‘A_algorithm.m’ and ‘B_algorithm.m’ are executed in the same MATLAB workspace. This has several benefits: less processes running; avoid consuming licenses unnecessarily; and MATLAB workspace variables can be shared.

5.3 CONCLUDING REMARKS

The algorithm designers produce executable specifications of the algorithms using MATLAB’s M-files or Simulink models. MATLAB provides an API to handle the MATLAB Engine sessions from C programs and exchange data between them. However, the direct use of this API in the SystemC code is cumbersome. Moreover, this API lacks the capability to maintain the shared data between the C application and the MATLAB session synchronized. As a result, the design team is responsible for including specific code within the C application to keep the synchronization.

In this chapter, the MATLAB API has been abstracted using the proposed library MatlabEngine++. This abstraction simplifies the task of integrating a MATLAB Engine in a SystemC model. Additionally, the MatlabEngine++ framework is responsible for automatically keeping the variables of the MATLAB session synchronized with the SystemC application.

In all the works found in the literature, the control of the system simulation is managed by Simulink. In this case, S-Function wrappers must be created in MATLAB for each SystemC module integrated into Simulink. The proposed methodology defines the SystemC model as the main controller of the simulation during the integration of the MATLAB algorithms. The benefit of this approach is twofold: it avoids the the creation of S-Function wrappers in MATLAB and it allows to maintain the same simulation environment used in the SystemC-TLM design and verification flow.
5.3 Concluding Remarks

The inclusion of a MATLAB Engine in a SystemC model enables building TLM models, where architecture (modules and transactions) is described in SystemC and algorithms are described in MATLAB. Thanks to the proposed approach, the cost of building these models is very small and the resulting model code is clear. The SystemC–MATLAB models enable the verification of correct algorithm cooperation to achieve system functionality, with negligible additional costs in development. Thus, architecture and algorithm tuning can be performed efficiently early in the design, with the system–design and the algorithm–design teams working in their preferred environments. Additionally, this approach allows to use Assertion-based verification (ABV) techniques to verify a SystemC–TLM with integrated MATLAB algorithms.

This chapter has been the precursor for the paper [127].
In the last decade, Model-driven Design (MDD) has emerged as a development methodology for software engineering. The main proposition of MDD is to capture in visual models all the information of the system being designed. In MDD, the design products are visual models instead of source code. The purpose is to promote intuitive visual design patterns that raise the abstraction to a higher level and improve the communication between engineer teams developing the system. Moreover, the use of standard modelling languages guarantees the compatibility between different modelling frameworks. Thus, the Unified Modeling Language (UML) standard was conceived by the Object Management Group (OMG) as an object-oriented software modelling language applicable within the MDD methodology.
The MDD approach is not exclusively limited to pure software development. In order to extend the benefits of this approach to other domains, the International Council on Systems Engineering (INCOSE) in close collaboration with OMG defined a general purpose modelling language for systems engineering: Systems Modeling Language (SysML). SysML is defined as an extension of a subset of UML, and like UML, it defines a graphical notation supported by modelling diagrams that allows to create visual models of systems under development.

In recent years, MDD has been proposed as a promising methodology for electronic systems modelling and design. Model-to-Model (M2M) and Model-to-Text (M2T) techniques can be used to create SystemC–Transaction Level Modeling (TLM) code from visual modelling languages such as UML or SysML. However, some requirements shall be fulfilled in order to effectively use these techniques. On one hand, visual modelling languages with clear syntax and formal semantics are required. The lack of formal semantics makes difficult the acceptance of visual modelling languages by developers and complicates the exchange of complex models among different tools. On the other hand, the Model of Computation (MoC) of the generated executable model shall be clearly defined. MoCs, together with design languages, provide the foundation for defining system behaviour. The well-defined computation semantics of MoCs allow to unambiguously capture the required functionality and enable the application of formal design and verification techniques. The automation tools for electronic system development can only be applied if the behavioural semantics and the corresponding MoCs are well-defined.

The main goal of this chapter is to formalize the semantics of SysML State Machine diagrams and Activity diagrams using MoCs as semantic domains. Rigorous semantic mappings are provided in order to translate the abstract syntaxes of State Machine diagrams and Activity diagrams to Finite State Machine (FSM) and Synchronous Data Flow (SDF) MoCs, respectively. Furthermore, this chapter defines a modelling methodology to create SysML models. This modelling methodology comprises a set of modelling phases aimed to describe both the structural and behavioural features of the system under development at the highest level of abstraction.

The behavioural semantic formalization of SysML State Machine diagrams and Activity diagrams, and the modelling methodology proposed in this chapter establish the theoretical background for a practical MDD
framework implementation. The Chapter 7 proposes a MDD framework that implements the semantic mappings defined in this chapter and addresses the automatic generation of SystemC–TLM source code from a system model described using SysML.

The rest of the chapter is organized as follows. Section 6.1 introduces SysML. Section 6.2 presents the preliminaries of the proposed approach. Then, Section 6.3 and Section 6.4 describe the formal semantics of SysML State Machines and Activities by means of FSM and SDF models of computation, respectively. Section 6.5 Finally, Section 6.6 summarizes the conclusion of this chapter.

6.1 SYSTEM MODELLING WITH SYSML

The Systems Modeling Language (SysML) [49] is a modelling language that supports the specification, analysis, design, verification and validation of complex systems with both hardware and software components, personnel, procedures, etc. [79, 116]. SysML represents systems and product architectures, as well as their behaviour and structure.

SysML is a profile-based customization of the UML [48] for system engineering. SysML reuses a subset of UML and offers new extensions for system engineering applications. Although SysML and UML share several diagrams, SysML proposes new diagrams not found in UML, such as the Requirement diagram and the Parametric diagram. A brief overview of the diagrams provided by UML and SysML is provided in Section A.4 of the Appendix A.

6.1.1 STRUCTURAL MODELLING

In SysML, the structural constructs are modelled using structure diagrams, such as, Block Definition Diagram (BDD) and Internal Block Diagram (IBD). The structural diagrams define the relationship between the elements that compose the system.

The BDD defines blocks and relationships between blocks such as associations, generalizations, and dependencies. The block definition diagram is the most widely used diagram in SysML. It captures the definition of blocks in terms of properties and operations, and relationships such as a system hierarchy.
The IBD captures the internal structure of a block in terms of subblocks (parts) and connections between them. A block includes properties to specify its values, parts, and references to other blocks. Ports are a special class of property used to specify the interactions between blocks.

### 6.1.2 BEHAVIOURAL MODELLING

The main SysML behavioural diagrams include the Use Case diagram, State Machine diagram, and Activity diagram.

Use Case Diagrams describe the functionality of a system in relation to how the users use that system. Users are actors in use cases, they represent any human, organization or external system that participates in the system functionality.

The Use Case Diagrams describe the functionality of a system. However, a more detailed description of the use case shall be modelled using Activity Diagrams or State Machine Diagrams. In general, the choice of behavioural formalism follows these guidelines:

- State machines are useful when the interaction between the actors and the subject includes considerable control logic, not easily represented by an ordered sequence of events.

- Activities are useful where the scenario includes flow of inputs and outputs, and/or algorithms that transform data.

State Machine Diagrams define a set of concepts that can be used to model discrete behaviour through finite state transition systems. The state machine represents a event-driven behaviour. SysML restricts the StateMachine concept defined in UML; while UML supports both BehaviorStateMachine and ProtocolStateMachine types, SysML restricts the UML’s state machine abstract syntax to BehaviorStateMachine level excluding the protocol state machine concept [49].

State Machines typically describe the behaviour of blocks. A state machine models discrete event-based behaviour by means of states and transitions connecting states. External events received by blocks control the State Machine execution.
Activity Diagrams represent flow-based behaviour through the execution of a sequence of actions. Activities define the transformation of input data into output data. SysML extended the data-flow concepts of UML adding new features, such as, continuous flow behaviours and probability in activity edges [49]. Activity diagrams are similar to the well-known functional flow diagrams.

Activities can describe the behaviour of blocks or parts. An activity may be specified as the main behaviour of a block that describes how input data of the block are transformed into outputs. The activity can also be specified as the method for an operation of the block. Alternatively, when a behaviour is described using state machines, activities can specify the effects of transitions or the entry/exit behaviours of a state.

6.2 APPROACH TO BEHAVIOURAL SEMANTIC MAPPING

Using the modelling language concept introduced by [128] and [129], the SysML modelling language, $L$, is assumed as a 5-tuple

$$L = \langle A, C, S, M_C, M_S \rangle$$

The SysML language, $L$, consists of an abstract syntax ($A$), a concrete syntax ($C$), a semantic domain ($S$), a syntactic mapping ($M_C$), and a semantic mapping ($M_S$). The abstract syntax $A$ defines the concepts of the SysML language and their relationships. The concrete syntax $C$ defines the physical appearance of the language, that is, its notation. The syntactic mapping

$$M_C : C \rightarrow A$$

maps language notation elements to abstract syntax elements. The semantics describe the meaning of a model specified in SysML language. The semantics of the modelling language is given in terms of semantic domain $S$. The semantic mapping

$$M_S : A \rightarrow S$$

assigns abstract syntax elements to elements of the semantic domain so that each syntactic construct is mapped to its meaning.

The OMG’s SysML standard specification [49] is defined as an extension of the OMG’s UML v2 superstructure specification [48]. The concrete syntax ($C$) and the abstract syntax ($A$) are defined using graphical notations
of the language concepts and metamodelling techniques, respectively. The syntactic mapping \((M_C)\) relating the graphical notation and the elements of the abstract syntax is clearly described. However, those documents only provide an informal description of the semantic domain \((S)\) in natural language.

Section 6.3 and Section 6.4 focus on the definition of both formal semantic domains and semantic mappings of SysML’s behavioural constructs; specifically, State Machine diagrams and Activity diagrams.

Section 6.3 deals with the formalization of the SysML State Machine diagrams using FSM MoC as the semantic domain. An abstract syntax for State Machine diagrams is denoted as \(A_{\text{StateMachine}}\) and its semantic domain is denoted as \(S_{FSM}\). The following semantic mapping is proposed in Section 6.3:

\[
M_{FSM} : A_{\text{StateMachine}} \rightarrow S_{FSM}
\]

On the other hand, Section 6.4 presents the formalization of the SysML Activity diagrams using SDF MoC as the semantic domain. An abstract syntax for Activity diagrams is denoted as \(A_{\text{Activity}}\) and its semantic domain is denoted as \(S_{SDF}\). The following semantic mapping is proposed in Section 6.4:

\[
M_{SDF} : A_{\text{Activity}} \rightarrow S_{SDF}
\]

### 6.3 Formal Semantics for SYSML State Machine Diagrams

State Machine Diagrams define a set of concepts that can be used to model discrete behaviour through finite state transition systems. The state machine represents an event-driven behaviour. SysML restricts the StateMachine concept defined in UML; while UML supports both \(\text{BehaviorStateMachines}\) and \(\text{ProtocolStateMachines}\) types, SysML restricts the UML’s state machine abstract syntax to \(\text{BehaviorStateMachines}\). In this section, the formal semantics of the State Machine diagrams in SysML is discussed.
6.3 Formal Semantics for SysML State Machine Diagrams

6.3.1 ABSTRACT SYNTAX OF STATE MACHINE DIAGRAMS

The SysML standard [49] reuses the abstract syntax defined for the BehaviorStateMachine package in the UML standard [48]. A simplified version of the abstract syntax of the BehaviorStateMachine package is shown in Figure 6.1. A StateMachine element is composed of one or more Region elements. Regions are orthonormal behavioural parts that contain Vertex elements and Transitions connecting vertexes.

A Vertex is a generalization of State and Pseudostate elements. States are also allowed to contain Regions. This feature leads to several kinds of states: simple states, composite states, and submachine states. A simple state does not have any regions and it does not contain any references to a substate machine. On the contrary, a composite state contains one or more regions. A submachine state includes a reference to a substate machine. A FinalState is a special state that represents the termination point of the enclosing Region. Pseudostate elements model transient vertexes such as, the initial vertex of a StateMachine.

The Vertex elements of a StateMachine are connected using Transitions. A StateMachine traverses its vertexes based on the input events detected.
Transitions are fired by events. The Trigger attribute of a transition specifies which event can fire the transition between two Vertex elements.

Each region of the state machine can only have one initial pseudostate vertex. The initial vertex is connected to the default state of the State Machine through a single transition. This transition is not allowed to have a Trigger.

During a StateMachine execution, a State can be active or inactive. States have incoming transitions and outgoing transitions. When an outgoing transition is fired, the state is exited and it becomes inactive. When an incoming transition is fired, the state is entered and it becomes active. As Transitions are connections between two states, whenever a transition is fired the source state becomes inactive and the target state becomes active.

There can be at most one active State in a Region at any instant during the execution of the StateMachine. When the StateMachine is executed for the first time the default state becomes the active state.

A mathematical representation of a State Machine Diagram is assumed using a graph theory notation. The proposed approach is focused on basic state machines with a single region and no composite states.

**Definition 6.1.** The abstract syntax of a State Machine \( A_{StateMachine} \) is an ordered pair of sets
\[
A_{StateMachine} = (V, T)
\] (6.6)
where \( V \) is a finite set of state machine Vertex elements, and \( T \) is a set of Transitions. The use of Regions is obviated as the proposed approach is focusing on state machines with a single region. Thus, StateMachines consists of Vertex and Transition elements.

**Definition 6.2.** Vertex are partitioned into States and Pseudostates
\[
V = S \cup PS
\] (6.7)
where \( S \) is a finite set of States, and \( PS \) is a finite set of Pseudostates.

**Definition 6.3.** A vertex \( v \in V \) has a set of incoming transitions and a set of outgoing transitions
\[
\forall v \in V, incoming(v) \subseteq T
\] (6.8a)
\[
\forall v \in V, outgoing(v) \subseteq T
\] (6.8b)
6.3 Formal Semantics for SysML State Machine Diagrams

**Definition 6.4.** The function $\text{kind}(ps)$ determines the precise type ($\text{PseudostateKind}$) of the Pseudostate $ps \in \text{PS}$.

**Definition 6.5.** A Transition $t \in T$ has a trigger $\text{trg} \in \text{TRG}$

$$
\text{trigger} : T \rightarrow \text{TRG}
$$

(6.9)

**Definition 6.6.** A transition $t \in T$ has a source vertex and a target vertex

$$
\text{source} : T \rightarrow V
$$

(6.10a)

$$
\text{target} : T \rightarrow V
$$

(6.10b)

### 6.3.2 SEMANTIC DOMAIN OF STATE MACHINE DIAGRAMS

The proposed approach uses the Finite State Machine (FSM) MoC as the semantic domain $S_{\text{FSM}}$ for the State Machine diagrams of SysML. An FSM is an event-based MoC suitable to describe state-oriented models, such as, sequential control logics.

An FSM consists of an input event alphabet, a set of states and a set of transitions between these states. During its execution, a FSM receives input events and processes them while traversing its states.

The relation between states and transitions is specified by a transition function. The transition function describes the allowed transitions between states. During the execution of a FSM, it processes input events in order to fire transitions and traverse states.

**Definition 6.7.** A FSM is a 4-tuple

$$
S_{\text{FSM}} = (Q, q_0, \Psi, \Delta)
$$

(6.11)

where

- $Q$ is a finite set of symbols denoting states.
- $q_0$ is a initial state of the FSM, $q_0 \in Q$.
- $\Psi$ is a set of symbols denoting the possible input events (input alphabet).
- $\Delta$ is a transition function $\Delta : Q \times \Psi \rightarrow Q$. 
The execution of a FSM starts at the state $q_0$. A FSM processes the incoming events $\psi \in \Psi$ one at a time in order to determine in which order the states will be traversed. Each event is processed in a runtime step. A runtime step can be launched only if the previous runtime step has been successfully finished. First, a runtime step selects an enabled transition. A transition $\langle q, \psi, q' \rangle$ can be enabled in the current state $q$ only if $\psi$ is the event processed in the current runtime step. In order to guarantee the determinism of the FSM execution, there may be at most one enabled transition. If there is no enabled transition, the runtime step terminates. If an enabled transition is found, the transition’s target state $q'$ is activated, and the runtime step is terminated.

**Definition 6.8.** The execution of a FSM is a sequence of states

$$q_0 \xrightarrow{\delta_0} q_1 \xrightarrow{\delta_1} q_2 \xrightarrow{\delta_2} \cdots$$

(6.12)

where $q_0$ is the initial state of the FSM, $q_k \in Q$ and $\delta_k$ is a transition ($\delta_k \in \Delta$) that goes from $q_k$ to $q_{k+1}$.

**Definition 6.9.** A transition $q_k \xrightarrow{\delta_k} q_{k+1}$ is enabled if the following condition is satisfied

$$\langle q_k, \psi_k, q_{k+1} \rangle \in \Delta$$

(6.13)

where $q_k$ is the current state and $\psi_k$ is the incoming event processed in the current runtime step.

### 6.3.3 SEMANTIC MAPPING OF STATE MACHINE DIAGRAMS

The semantic mapping from a State Machine diagram to a FSM is straightforward. State ($S$) elements from SysML State Machines are translated to $Q$ elements of the FSM model of computation. A StateMachine can only have one initial pseudostate vertex which is translated to the initial state $q_0$ of a FSM graph. The set of input events $\Psi$ is populated with the events that trigger the Transitions ($T$) of a StateMachine. The Transitions ($T$) of a StateMachine are translated into the FSM transition function $\Delta$. The triggers and guards of Transitions are directly related to the sets $\Psi$ and $\Sigma$ of FSMs.

**Definition 6.10.** The transformation from a State Machine diagram to a FSM is a function $M_{FSM}$ where

$$M_{FSM} : A_{StateMachine} \rightarrow S_{FSM}$$

(6.14)
where

\[ Q = S \]  \hspace{1cm} (6.15a)\\
\[ q_0 = \{ ps \mid \text{kind}(ps) = \text{InitialState}, \ ps \in PS \} \]  \hspace{1cm} (6.15b)\\
\[ \Psi = \{ \text{trigger}(t) \mid t \in T \} \]  \hspace{1cm} (6.15c)\\
\[ \Delta = \{ \langle \text{source}(t), \text{trigger}(t), \text{target}(t) \rangle \mid t \in T \} \]  \hspace{1cm} (6.15d)

### 6.4 FORMAL SEMANTICS FOR SYSML ACTIVITY DIAGRAMS

Activity Diagrams represent flow-based behaviour through the execution of a sequence of actions. Activities define the transformation of input data into output data. Activities can describe the behaviour of blocks or parts. An activity may be specified as the main behaviour of a block that describes how input data of the block are transformed into outputs. In this section, the formal semantics of the Activity diagrams in SysML is discussed.

#### 6.4.1 ABSTRACT SYNTAX OF ACTIVITY DIAGRAMS

The SysML standard [49] reuses the abstract syntax defined for the BasicActivities package in the UML standard [48]. Activity diagrams consist of two primary elements: ActivityNodes and ActivityEdges, as shown in the abstract syntax of the BasicActivities package depicted in Figure 6.2. An Activity represents a data-flow behaviour where ActivityNodes are connected using ActivityEdges. An Activity specifies a data transformation function; input data is transformed into output data. This data transformation function is performed by the coordinated execution of ActivityNodes which are basic data transformation units. Data communication in an Activity diagram is token-based. Tokens are elemental information units controlled by ActivityEdges and exchanged between ActivityNodes.

The ActivityNode element is categorized into three types: Actions, ObjectNodes and ControlNodes. There are two types of ControlNodes: InitialNodes and ActivityFinalNodes. ObjectNodes has two subtypes: Pins and ActivityParameterNodes. In turn, ActivityEdge elements consist of ControlFlows and ObjectFlows as depicted in Figure 6.2. Actions are the main executable nodes and they can be connected through ControlFlows or ObjectFlows.
ControlFlow is used to establish the execution order of Action nodes and ControlNodes through the communication of control tokens. The InitialNode sets the starting point of the execution while the ActivityFinalNode is the last node of the control flow. Tokens offered by the source node are all offered to the target node.

On the other hand, ObjectFlows describe the flow of data between ObjectNodes. Pins are attached to Actions in order to receive or transmit data tokens. Pins are interfaces to allow Actions to interchange data tokens with other Actions or ActivityParameterNodes. There are input and output Pins depending on whether they are connected to incoming or outgoing ObjectFlows. An Action consumes the data tokens placed on its input Pins, processes them, and places the outgoing data tokens on its output Pins.

An Action is executed when all its ObjectFlow and ControlFlow requirements have been satisfied. An ControlFlow requirement is satisfied when each of the flows is offered one token. An ObjectFlow requirement is satisfied when all of the input Pins are offered all necessary tokens, as specified...
by their minimum multiplicity, and accept as many tokens as specified by their maximum multiplicity. When an Action is completed, it offers any object tokens that have been placed on its output Pins and control tokens on all its outgoing ControlFlows.

ActivityParameterNodes are used to specify input and output Activity parameters. Using ActivityParameterNodes the designer can model a scenario where an external agent transmits data into or gets data from an Activity.

Similarly to State Machines, the abstract syntax of an Activity is described using a graph theory notation as an ordered pair of sets. A modelling process based on the BasicActivities level of Activity Diagrams is proposed. The BasicActivities package supports control sequencing and data flow between actions.

**Definition 6.11.** The abstract syntax of an Activity, $A_{Activity}$, is an ordered pair of sets

$$A_{Activity} = \langle AN, AE \rangle$$  \hspace{1cm} (6.16)

where $AN$ is a finite set of ActivityNodes and $AE$ is a finite set of ActivityEdges.

**Definition 6.12.** ActivityNodes are partitioned into Actions, ObjectNodes and ControlNodes

$$AN = AC \cup ON \cup CN$$  \hspace{1cm} (6.17)

where $AC$ is a finite set of Actions, $ON$ is a finite set of ObjectNodes, and $CN$ is a finite set of ControlNodes.

**Definition 6.13.** The set of ActivityEdges is partitioned into ControlFlow and ObjectFlow sets

$$AE = CF \cup OF$$  \hspace{1cm} (6.18)

where $CF$ is a finite set of ControlFlow, and $OF$ is a finite set of ObjectFlow.

**Definition 6.14.** The set of ObjectNodes is partitioned into ActivityParameterNodes and Pins sets

$$ON = APN \cup PIN$$  \hspace{1cm} (6.19)

where $APN$ is a finite set of ActivityParameterNodes, and $PIN$ is a finite set of Pins.
Definition 6.15. Pin elements have an upperBound function. This function is interpreted as the token capacity of the respective Pin. The upperBound function is defined as

\[ \text{upperBound} : \text{PIN} \rightarrow \mathbb{N} \quad (6.20) \]

where \( \mathbb{N} \) refers to the set of natural numbers.

Definition 6.16. The set of ControlNodes is partitioned into InitialNode and ActivityFinalNode sets

\[ \text{CN} = \text{IN} \cup \text{AFN} \quad (6.21) \]

where \( \text{IN} \) is a finite set of InitialNode, and \( \text{AFN} \) is a finite set of ActivityFinalNodes.

Definition 6.17. An ActivityNode element \( \text{an} \in \text{AN} \) has a set of incoming ActivityEdges and a set of outgoing ActivityEdges.

\[ \forall \text{an} \in \text{AN}, \text{incoming} (\text{an}) \subseteq \text{AE} \quad (6.22a) \]
\[ \forall \text{an} \in \text{AN}, \text{outgoing} (\text{an}) \subseteq \text{AE} \quad (6.22b) \]

The incoming function returns the ActivityEdge elements reaching an ActivityNode. On the other hand, the outgoing function returns the ActivityEdge elements leaving an ActivityNode. An ActivityNode may have several incoming and/or outgoing edges.

In the following, the incoming and outgoing functions of ObjectNodes, Actions and ControlNodes are specified.

Definition 6.18. ObjectNodes only accept ObjectFlows as incoming or outgoing elements.

\[ \forall n \in \text{ON}, \text{incoming}(n) \subseteq \text{OF} \quad (6.23a) \]
\[ \forall n \in \text{ON}, \text{outgoing}(n) \subseteq \text{OF} \quad (6.23b) \]

Definition 6.19. ActionNodes and ControlNodes only accept ControlFlows as incoming or outgoing elements.

\[ \forall n \in (\text{AC} \cup \text{CN}), \text{incoming}(n) \subseteq \text{CF} \quad (6.24a) \]
\[ \forall n \in (\text{AC} \cup \text{CN}), \text{outgoing}(n) \subseteq \text{CF} \quad (6.24b) \]
Definition 6.20. Both ActionNodes and ControlNodes has a set of input Pins and a set of output Pins.

\[ \forall n \in (AC \cup CN), \text{input}(n) \subseteq PIN \quad (6.25a) \]
\[ \forall n \in (AC \cup CN), \text{output}(n) \subseteq PIN \quad (6.25b) \]

Definition 6.21. An ActivityEdge element \( ae \in AE \) has a target ActivityNode and a source ActivityNode.

\[ \text{target} : AE \rightarrow AN \quad (6.26a) \]
\[ \text{source} : AE \rightarrow AN \quad (6.26b) \]

The \text{source} function returns the ActivityNode from which the ActivityEdge consumes tokens. The \text{target} function returns the ActivityNode to which the ActivityEdge produces tokens. An ActivityEdge has only one source node and one target node.

Particularly, ControlFlow elements produce tokens to and consumes tokens from Actions and ControlNode elements.

\[ \forall c \in CF, \text{target}(c) \in (AC \cup CN) \quad (6.27a) \]
\[ \forall c \in CF, \text{source}(c) \in (AC \cup CN) \quad (6.27b) \]

However, ObjectFlow elements connects together ObjectNodes only.

\[ \forall o \in OF, \text{target}(c) \in ON \quad (6.28a) \]
\[ \forall o \in OF, \text{source}(c) \in ON \quad (6.28b) \]

6.4.2 SEMANTIC DOMAIN OF ACTIVITY DIAGRAMS

The SDF MoC is proposed as the semantic domain \( S_{SDF} \) for the Activity diagrams of SysML. The SDF model of computation is a special case of data flow in which the number of data samples produced or consumed by each node on each firing is fixed in the design process [130].

A SDF graph consist of a set of vertex (nodes) connected through a set of directed edges (arcs). The SDF graph is a digraph (directed graph) as defined in graph theory [131]. The nodes are connected defining arcs between them; arcs are communication channels. Each node has a set of
input arcs and a set of output arcs. Each arc has a source node and a target node.

A SDF node defines a token (data) rate in each of its input/output arc. A token rate is specified using a positive integer number. A token rate defined for an output arc determines how many data tokens the node produces on that arc each time the node is fired. On the other hand, a token rate defined for an input arc determines how many data tokens the node consumes from that arc each time the node is fired.

The token rates are defined for all the arcs connected to a node. When a node fires, it consumes tokens from its input arcs and produces tokens in its output arcs. In order to a node to be fired, its input arcs must have, at least, as many tokens as specified by the input token rates. If that condition is satisfied the node is ready to be fired.

**Definition 6.22.** An SDF graph \((S_{SDF})\) is a five-tuple

\[
S_{SDF} = \langle N, A, P, C, \mu_0 \rangle \quad (6.29)
\]

where

- \(N\) is the set of nodes.
- \(A \subseteq N \times N\) is the set of arcs.
- \(P : N \times A \rightarrow \mathbb{N}_0\) represents data tokens produced at node \(n \in N\) and to be carried by arc \(a \in A\).
- \(C : N \times A \rightarrow \mathbb{N}_0\) represents data tokens carried by arc \(a \in A\) and consumed in node \(n \in N\).
- \(\mu_0 : A \rightarrow \mathbb{N}_0\) is the initial marking. \(\mu_0(a)\) represents initial data tokens in arc \(a \in A\).

\(\mathbb{N}_0\) is the set of all the natural numbers \((\mathbb{N})\) including zero:

\[
\mathbb{N}_0 = \{0, 1, 2, ...\}
\]

The set of nodes \((N)\) and the set of arcs \((A)\) describe the structural aspects of a SDF graph. On the other hand, the production rate \((P)\), the consumption rate \((C)\), and the initial marking \((\mu_0)\) describe the behavioural aspects of the SDF graph.
The fixed token generation and consumption rates may seem a severe restriction, however, this restriction provides benefits regarding execution scheduling and simulation speed. First of all, these fixed execution rates are defined in the design process, and thus, they are known before the execution of the SDF-based system model. This is a valuable information that can be used to create a static scheduling of the node firings [132]. This means that the scheduling of a SDF may be done statically before runtime, thus optimizing the execution time of the system simulation. A static scheduling considerably reduces the overhead in the execution time due to context switching in system’s node firing. Several studies are found in the literature that focus on creating a static scheduling from a SDF model [130, 132–138].

On the other hand, a model described in SDF supports the analysis of the memory consumption required for a static scheduling. This enables to check the fulfillment of resource allocation requirements during the initial design stages.

The execution semantics of a SDF graph are described in terms of its marking.

**Definition 6.23.** The marking (µ) of a SDF graph describes how many tokens each arc has in a certain instant of the execution. Similarly to Petri Nets, a marking function can be defined for SDF as

\[ \mu : A \rightarrow N_0 \]  

The marking in the initial instant of the SDF graph is represented by \( \mu_0 \).

**Definition 6.24.** A node \( n \in N \) can fire if

\[ \forall a \in A, \mu(a) \geq C(n, a) \]  

When a node fires, the marking of a SDF graph is modified.

**Definition 6.25.** The execution of a SDF is a sequence of marking steps

\[ \mu_0 \xrightarrow{n_0} \mu_1 \xrightarrow{n_1} \mu_2 \xrightarrow{\delta_2} \cdots \]  

where \( \mu_0 \) is the initial marking of the SDF, \( \mu_k \) is the SDF marking at step \( k \) and \( n_k \) is a firing node (\( n_k \in N \)) that goes from \( \mu_k \) to \( \mu_{k+1} \).
**Definition 6.26.** The node execution function $n \xrightarrow{}$ is defined for any firing node $n_k \in N$ as

$$\mu_k \xrightarrow{n_k} \mu_{k+1}$$

where $\mu_{k+1}$ is computed as

$$\forall a \in A, \mu_{k+1}(a) = \mu_k(a) - C(n_k,a) + P(n_k,a)$$

The proposed SDF formal semantics can be related to the matrix representation proposed in [130]. In [130], Lee and Messerschmitt introduced a topology matrix (denoted as $\Gamma$) in order to characterize a SDF graph. The topology matrix is similar to the incidence matrix associated with directed graphs in graph theory. The relation between the proposed SDF formal semantics and the topology matrix proposed in [130] is

$$\Gamma = P - C$$

where $P$ and $C$ are the proposed production and consumption functions expressed as $|N| \times |A|$ sized matrices. $|N|$ is the number of nodes and $|A|$ is the number of arcs. As a result, the topology matrix $\Gamma$ also has size $|N| \times |A|$. The $(i,j)$ entry in the production matrix is the amount of data produced by node $j$ on arc $i$ each time it is invoked. Similarly, The $(i,j)$ entry in the consumption matrix is the amount of data consumed by node $j$ from arc $i$ each time it is invoked.

Figure 6.3 presents an example of a SDF graph. The SDF of the example consists of the nodes $U$, $V$, $W$, $X$, $Y$ and $Z$, and the arcs $a$, $b$, $c$, $d$, $e$ and $f$. Each node indicates how many tokens are produced or consumed in its arcs. For example, $V$ produces 2 tokens both in arc $b$ and arc $d$, and it consumes
1 token from arc $a$. The production ($P$) and consumption ($C$) functions of the SDF example are defined as matrices in the following expressions:

$$
P = \begin{pmatrix}
U & V & W & X & Y & Z \\
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 2 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 2 & 0 \\
0 & 2 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 2 & 0 & 0
\end{pmatrix}
$$

$$
C = \begin{pmatrix}
U & V & W & X & Y & Z \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 2 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0
\end{pmatrix}
$$

$$
\Gamma = P - C = \begin{pmatrix}
1 & -1 & 0 & 0 & 0 & 0 \\
0 & 2 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 2 & -1 \\
0 & 2 & -1 & 0 & 0 & 0 \\
0 & 0 & 1 & -2 & 0 & 0 \\
0 & 0 & 0 & 2 & -1 & 0
\end{pmatrix}
$$

An admissible static scheduling can be obtained analysing $\Gamma$. An admissible sequential schedule is a non-empty ordered list of nodes such that if the nodes are executed in the sequence given by list, the amount of tokens in the arcs will remain non-negative and bounded [130]. Each node must appear in the list at least once. As described in [132] by Lee and Messerschmitt, a necessary condition for the existence of a periodic sequential schedule with bounded memory requirements is that $\text{rank}(\Gamma) = |N| - 1$, where $|N|$ is the number of nodes. They prove that a topology matrix $\Gamma$ satisfying that condition has a strictly positive integer vector $q$ in its right nullspace, that is, $\Gamma q = 0$. Moreover, the elements of $q$ specify the number of times that the corresponding node should be invoked in one cycle of a periodic schedule.
In a cycle of a periodic schedule, nodes are invoked as many times as specified in \( q \), and the amount of tokens left in each arc ends up equal to the amount before invocations (expressed in the initial marking \( \mu_0 \)). Hence, the schedule can be repeated infinitely often with finite memory.

In the example described in Figure 6.3, the rank of the topology matrix is \( \text{rank}(\Gamma) = 5 \), which satisfies the condition for a periodic sequential schedule (\( \text{rank}(\Gamma) = |N| - 1 \)), as \( |N| - 1 = 5 \). The right nullspace of the example topology matrix is

\[
q = \begin{pmatrix}
U & 1 \\
V & 1 \\
W & 2 \\
X & 1 \\
Y & 2 \\
Z & 4 \\
\end{pmatrix}
\]

The nodes \( U, V \) and \( X \) shall be invoked once in one cycle of a periodic schedule, \( W \) and \( Y \) two times, and \( Z \) four times.

Given \( q \), Lee and Messerschmitt proposed a simple algorithm in order to calculate an admissible sequential schedule [132]:

1. Solve \( \Gamma q = 0 \) for the smallest positive integer vector \( q \).

2. For each node \( n \in N \), schedule \( n \) if it is runnable, trying each node once. A node is said to be runnable at a given time if it has not been run \( q_n \) times and running it will not cause the amount of tokens in an arc (\( \mu(a) \)) to become negative.

3. If each node \( n \in N \) has been scheduled \( q_n \) times, STOP.

4. If no node can be scheduled, indicate a deadlock (an error in the graph).

5. Else, go to 2 and repeat.

For the example described in Figure 6.3, a possible admissible sequential schedule is

\[
\]
If one cycle of the sequential schedule defined in Eq. (6.36) is executed within the SDF graph described in the Figure 6.3, the marking of the SDF \( (\mu) \) will evolve as described in Definitions 6.25 and 6.26:

\[
\begin{align*}
\mu_0 &= \begin{pmatrix}
0 \\
0 \\
0 \\
0
\end{pmatrix}, \\
U \rightarrow \mu_1 &= \begin{pmatrix}
1 \\
0 \\
0 \\
0
\end{pmatrix}, \\
V \rightarrow \mu_2 &= \begin{pmatrix}
2 \\
0 \\
0 \\
0
\end{pmatrix}, \\
W \rightarrow \mu_3 &= \begin{pmatrix}
2 \\
0 \\
1 \\
0
\end{pmatrix}, \\
W \rightarrow \mu_4 &= \begin{pmatrix}
2 \\
0 \\
1 \\
0
\end{pmatrix}, \\
X \rightarrow \mu_5 &= \begin{pmatrix}
2 \\
0 \\
0 \\
2
\end{pmatrix}, \\
Y \rightarrow \mu_6 &= \begin{pmatrix}
1 \\
2 \\
0 \\
1
\end{pmatrix}, \\
Y \rightarrow \mu_7 &= \begin{pmatrix}
0 \\
4 \\
0 \\
0
\end{pmatrix}, \\
Z \rightarrow \mu_7 &= \begin{pmatrix}
0 \\
0 \\
3 \\
0
\end{pmatrix}, \\
Z \rightarrow \mu_8 &= \begin{pmatrix}
0 \\
2 \\
0 \\
0
\end{pmatrix}, \\
Z \rightarrow \mu_9 &= \begin{pmatrix}
0 \\
2 \\
0 \\
0
\end{pmatrix}, \\
Z \rightarrow \mu_9 &= \begin{pmatrix}
0 \\
1 \\
0 \\
0
\end{pmatrix}
\end{align*}
\]

where \( \mu_0 \) represents the initial marking, that is, the amount of tokens within the arcs in the initial step.

\[
\mu_0 = \begin{pmatrix}
a & 0 \\
b & 0 \\
c & 0 \\
d & 0 \\
e & 0 \\
f & 0
\end{pmatrix}
\]

The example demonstrates that the amount of tokens left in each arc ends up equal to the amount before invocations, that is, the initial and finals markings of a periodic sequential schedule are equal. As a result, a cycle of the sequential schedule can be repeated infinitely often.
6.4.3 SEMANTIC MAPPING OF ACTIVITY DIAGRAMS

In this section, the semantic mapping $M_{SDF}$ is described. $M_{SDF}$ assigns elements from the abstract syntax of SysML Activities to elements of the SDF semantic domain; that is, the formal semantics mapping function takes an $A_{Activity} = \langle AN, AE \rangle$ as input and generates a $S_{SDF} = \langle N, A, P, C, \mu_0 \rangle$.

Definition 6.27. The semantic mapping function $M_{SDF}$ is defined as

$$M_{SDF} : A_{Activity} \rightarrow S_{SDF} \tag{6.37}$$

First, an intuitive outline of the semantic mapping process is described. Then, a detailed mapping specification is provided. An intuitive mapping relation between the elements from the abstract syntax of SysML Activities and the elements of the SDF semantic domain is depicted in Figure 6.4.

*ActivityFinalNodes, InitialNodes, ActivityParameterNodes and Actions* of Activity diagrams are translated into SDF *Nodes.*
ControlFlow elements are directly mapped into SDF Arcs. ControlFlow elements communicate control tokens that establish the execution order of Action nodes, InitialNode and ActivityFinalNode. When the source element finishes its execution it puts one token of the ControlFlow element. Then, the ControlFlow sends that token to the target element. Both the token production rate and the token consumption rate of the Arcs mapped from ControlFlow elements are equal to 1, as shown in Figure 6.4.

ObjectFlow elements carry data tokens between ObjectNodes (ActivityParameterNodes or Pins). In the last three rows of Figure 6.4 all the possible connections of ObjectFlows are represented. ObjectFlow elements are mapped into SDF Arcs. The token production rate of these Arcs are related to the number of tokens that the target ObjectNode gets. Analogously, the token consumption rate depends on the number of tokens that the source ObjectNode puts on the ObjectFlow.

The detailed mapping specification is based on the intuitive mapping provided previously.

ActivityFinalNodes (AFN), InitialNodes (IN), ActivityParameterNodes (APN) and Actions (AC) of Activity diagrams become SDF nodes (N) as described in the following equation

\[ N = AFN \cup IN \cup APN \cup AC \] (6.38)

Both ControlFlow (CF) elements and ObjectFlow (OF) elements are mapped into SDF Arcs (A). The semantic mapping of ObjectFlow (OF) elements is split in 3 cases, one for each supported connection configuration previously shown in Figure 6.4:

- Case OF1: A ObjectFlow element connects two Action nodes through their Pins.


Thus, the semantic mapping to translate ControlFlow (CF) elements and ObjectFlow (OF) elements into SDF arcs (A) can be given by

\[ A = A_{CF} \cup A_{OF1} \cup A_{OF2} \cup A_{OF3} \] (6.39)

where \( A_{CF} \), \( A_{OF1} \), \( A_{OF2} \) and \( A_{OF3} \) denote the mapping of ControlFlow (CF) elements and ObjectFlow (OF) elements (Case OF1, Case OF2 and Case OF3) into SDF Arcs (A), respectively.

Similarly, the mapping of the production (P) and consumption (C) rates can be decomposed as:

\[ P = P_{CF} \cup P_{OF1} \cup P_{OF2} \cup P_{OF3} \] (6.40)
\[ C = C_{CF} \cup C_{OF1} \cup C_{OF2} \cup C_{OF3} \] (6.41)

On one hand, all the CF elements are translated into SDF Arcs.

\[ A_{CF} = \{(\text{source}(cf), \text{target}(cf)) | cf \in CF\} \] (6.42)

On the other hand, for a ControlFlow \( cf \in CF \), the number of tokens produced at \( \text{source}(cf) \) and to be carried by \( cf \) is fixed to 1. The number of tokens to be carried by \( cf \) and consumed at \( \text{target}(cf) \) is also 1. Hence

\[ P_{CF} = \{(\text{source}(cf), (\text{source}(cf), \text{target}(cf)), 1) | cf \in CF\} \] (6.43)
\[ C_{CF} = \{(\text{source}(cf), (\text{source}(cf), \text{target}(cf)), 1) | cf \in CF\} \] (6.44)

The token production and consumption rates related to SDF Arcs created from ObjectFlow elements depend on the upperBound attribute of the ObjectNodes connected to the ObjectFlow. The upperBound attribute from SysML is interpreted as the number of tokens that an ObjectNode puts to or gets from an ObjectFlow element. Specifically, the token production rate related to an SDF Arc created from an ObjectFlow of \( of \in OF \) is equal to the upperBound value of the source ObjectNode \( \text{source}(af) \). Similarly, the token consumption rate related to an SDF Arc created from an ObjectFlow of \( of \in OF \) is equal to the upperBound value of the target ObjectNode \( \text{target}(af) \).
6.4 Formal Semantics for SysML Activity Diagrams

ObjectFlow OF elements connecting two Action nodes AC (Case OF1) are mapped as

\begin{align*}
A_{OF1} &= \{\langle s, t \rangle \mid \{s, t\} \subset AC, of \in OF, source(of) \in output(s), \\
& \quad \quad \quad \quad target(of) \in input(t)\}\tag{6.45} \\
P_{OF1} &= \{\langle s, \langle s, t \rangle, upperBound(op) \rangle \mid \{s, t\} \subset AC, of \in OF, \\
& \quad \quad \quad \quad op \in PIN, source(of) = op, op \in output(s), \\
& \quad \quad \quad \quad target(of) \in input(t)\}\tag{6.46} \\
C_{OF1} &= \{\langle t, \langle s, t \rangle, upperBound(ip) \rangle \mid \{s, t\} \subset AC, of \in OF, \\
& \quad \quad \quad \quad ip \in PIN, source(of) \in output(s), target(of) = ip, \\
& \quad \quad \quad \quad ip \in input(t)\}\tag{6.47}
\end{align*}

ObjectFlow OF elements connecting an ActivityParameterNode APN to an Action AC (Case OF2) are mapped as

\begin{align*}
A_{OF2} &= \{\langle p, t \rangle \mid p \in APN, o \in OF, t \in AC, \\
& \quad \quad \quad \quad source(of) = p, target(of) \in input(t)\}\tag{6.48} \\
P_{OF2} &= \{\langle p, \langle p, t \rangle, 1 \rangle \mid p \in APN, of \in OF, t \in AC, \\
& \quad \quad \quad \quad source(of) = p, target(of) \in input(t)\}\tag{6.49} \\
C_{OF2} &= \{\langle t, \langle p, t \rangle, upperBound(ip) \rangle \mid p \in APN, of \in OF, \\
& \quad \quad \quad \quad ip \in PIN, t \in AC, source(of) = p, target(of) = ip, \\
& \quad \quad \quad \quad ip \in input(t)\}\tag{6.50}
\end{align*}

Finally, ObjectFlow OF elements connecting an Action AC to an ActivityParameterNode APN (Case OF3) are mapped as

\begin{align*}
A_{OF3} &= \{\langle s, p \rangle \mid s \in AC, of \in OF, p \in APN, \\
& \quad \quad \quad \quad source(of) \in output(s), target(of) = p\}\tag{6.51} \\
P_{OF3} &= \{\langle s, \langle s, p \rangle, upperBound(op) \rangle \mid s \in AC, op \in PIN, \\
& \quad \quad \quad \quad of \in OF, p \in APN, source(of) = op, target(of) = p, \\
& \quad \quad \quad \quad op \in output(s)\}\tag{6.52} \\
C_{OF3} &= \{\langle p, \langle s, p \rangle, 1 \rangle \mid s \in AC, o \in OF, p \in APN, \\
& \quad \quad \quad \quad source(of) \in output(s), target(of) = p\}\tag{6.53}
\end{align*}

Concluding, the detailed MSDF specification is described from Eq. (6.54a) to Eq. (6.54d).
\[ N = AFN \cup IN \cup AC \cup APN \quad (6.54a) \]

\[ A = ACF \cup AO_{F1} \cup AO_{F2} \cup AO_{F3} \]
\[ = \{ (\text{source}(cf), \text{target}(cf)) \mid cf \in CF \} \cup \]
\[ \{ (s, t) \mid \{ s, t \} \subset AC, \, of \in OF, \, \text{source}(of) \in \text{output}(s), \, \text{target}(of) \in \text{input}(t) \} \cup \]
\[ \{ (p, t) \mid p \in APN, \, o \in OF, \, t \in AC, \, \text{source}(of) = p, \, \text{target}(of) \in \text{input}(t) \} \cup \]
\[ \{ (s, p) \mid s \in AC, \, af \in OF, \, p \in APN, \, \text{source}(of) \in \text{output}(s), \, \text{target}(of) = p \} \quad (6.54b) \]

\[ P = P_{CF} \cup P_{OF1} \cup P_{OF2} \cup P_{OF3} \]
\[ = \{ (\text{source}(cf), (\text{source}(cf), \text{target}(cf)), 1) \mid cf \in CF \} \cup \]
\[ \{ (s, (s, t), \text{lower}(op)) \mid \{ s, t \} \subset AC, \, of \in OF, \, op \in PIN, \, \text{source}(of) = op, \, op \in \text{output}(s), \, \text{target}(of) \in \text{input}(t) \} \cup \]
\[ \{ (p, (p, t), 1) \mid p \in APN, \, of \in OF, \, t \in AC, \, \text{source}(of) = p, \, \text{target}(of) \in \text{input}(t) \} \cup \]
\[ \{ (s, (s, p), \text{lower}(op)) \mid s \in AC, \, of \in OF, \, p \in APN, \, \text{source}(of) = op, \, \text{target}(of) = p, \, op \in \text{output}(s) \} \quad (6.54c) \]

\[ C = C_{CF} \cup C_{OF1} \cup C_{OF2} \cup C_{OF3} \]
\[ = \{ (\text{source}(cf), (\text{source}(cf), \text{target}(cf)), 1) \mid cf \in CF \} \cup \]
\[ \{ (t, (s, t), \text{lower}(ip)) \mid \{ s, t \} \subset AC, \, of \in OF, \, ip \in PIN, \, \text{source}(of) \in \text{output}(s), \, \text{target}(of) = ip, \, ip \in \text{input}(t) \} \cup \]
\[ \{ (t, (p, t), \text{lower}(ip)) \mid p \in APN, \, of \in OF, \, ip \in PIN, \, t \in AC, \, \text{source}(of) = p, \, \text{target}(of) = ip, \, ip \in \text{input}(t) \} \cup \]
\[ \{ (p, (s, p), 1) \mid s \in AC, \, o \in OF, \, p \in APN, \, \text{source}(of) \in \text{output}(s), \, \text{target}(of) = p \} \quad (6.54d) \]
6.5 ASSYST MODELLING METHODOLOGY

This section defines a modelling methodology to create SysML models. This modelling methodology comprises a set of modelling phases aimed to describe both the structural and behavioural features of the system under development at the highest level of abstraction. This methodology addresses the creation of SysML State Machine diagrams and Activity diagrams conforming to the abstract syntaxes defined in Section 6.3 and Section 6.4, respectively.

The SysML model generated by this methodology represents the input to the MDD modelling approach called Automatic SysML to SystemC Translator (ASSYST) presented in Chapter 7. Chapter 7 proposes a MDD approach to translate electronic systems described using SysML to SystemC-TLM heterogeneous executable models.

The design team shall use the Topcased toolkit [139] and the Eclipse Integrated Development Environment (IDE) [140] to create the SysML model that describes the system under development. In the following sections, a set of modelling phases are described in order to guide the design team to obtain a SysML model ready to be processed by the ASSYST framework.

6.5.1 PHASE 1: MODELLING FUNCTIONALITY WITH USE CASES

The design team shall accomplish the following steps in this phase:

1. Create, at least, one Use Case Diagram to describe the functionality of the system.

2. Define the use cases of the system.

3. Define the actors that interact with the system.

4. Associate actors with use cases. An association represents that an actor interacts with a particular use case.

A more detailed description of the use case shall be modelled using Activity Diagrams or State Machine Diagrams.
6.5.2 PHASE 2A: MODELLING EVENT–BASED BEHAVIOUR WITH STATE MACHINES

The design team shall fulfill the following steps to model state machines:

1. Create a State Machine Diagram for each use case describing a control–based behaviour.

2. Associate each state machine with its corresponding use case:
   
   (a) Assign the associated use case to the Use Case attribute of the state machine.
   
   (b) Assign the state machine to the Classifier Behaviour attribute of the associated use case. This attribute specifies which is the behaviour of the use case. As a result, the behaviour of the use case is described by the state machine assigned to the Classifier Behaviour attribute.

   
   (a) The state machine shall contain one Region.
   
   (b) The design team can use simple states, Initial states, Final states and local transitions to describe the behaviour.
   
   (c) A Trigger element may be defined for transitions. The Trigger attribute shall be defined using the SysML Signal Event element. No more than one Trigger element shall be defined for each transition.
   
   (d) Associate Signal elements to each Signal Event. A Signal represents a information sent between objects. The data carried by a Signal are represented as attributes of type boolean, integer, real or string. A Signal Event may cause a state machine to trigger a transition every time the associated Signal is received.

6.5.3 PHASE 2B: MODELLING FLOW–BASED BEHAVIOUR WITH ACTIVITIES

The design team shall fulfill the following steps to model activities:
1. Create an Activity Diagram for each use case describing a data–flow oriented behaviour.

2. Associate each activity with its corresponding use case:
   (a) Assign the associated use case to the *Use Case* attribute of the activity.
   (b) Assign the activity to the *Classifier Behaviour* attribute of the associated use case. This attribute specifies which is the behaviour of the use case. As a result, the behaviour of the use case is described by the activity assigned to the *Classifier Behaviour* attribute.

3. Model the data–flow oriented behaviour using activity nodes and activity edges.
   (b) Activity actions shall be modelled using *Opaque Action* SysML elements.
   (c) The input and output parameters of an *Opaque Action* element shall be specified including *Pins*. The type attribute of a *Pin* shall be *boolean*, *integer*, *real*, *string* or *Signal*. An *Opaque Action* element can receive and send *boolean*, *integer*, *real*, *string* or *Signal* elements.
   (d) The input and output parameters of an Activity Diagram shall be specified including *Activity Parameter* elements. The *Type* attribute of an *Activity Parameter* shall be *Signal*. An Activity Diagram can only receive or send *Signal* elements.
   (e) *Initial* nodes, *Activity Final* nodes and *Opaque Action* elements shall be connected using *Control Flows*.
   (f) *Activity Parameter* nodes and *Pins* shall be connected using *Object Flows*. Connected *Activity Parameter* nodes and *Pins* shall contain equal *Type* attributes.
6.5.4 PHASE 3: MODELLING STRUCTURE WITH BLOCKS

In SysML, the structural constructs are modelled using structure diagrams, such as, BDD and IBD. The structural diagrams define the relationship between the elements that compose the system.

ASSYST supports the structural modelling using BDDs and IBDs. The design team shall fulfil the following steps to model the structure of the system under development:

1. Create BDDs and IBDs describing the structure of the system.
2. Associate each Block with the use case it implements. As a result, the Block implements the state machine or the activity specified in the use case. A Block may be associated to more than one use case. However, a use case shall only be associated to one Block.
3. Model the the structure of the system using Blocks, Data Types and Enumerations.
   
   (a) A SysML Block defines a collection of features to describe a system.
   
   (b) A SysML Data Type defines value types that may be used within a model. Data Type elements can represent complex data types by means of attributes and operations.
   
   (c) Designers shall describe the interfaces of a Block adding input and output Flow Ports.

6.6 CONCLUDING REMARKS

Many research works address the generation of SystemC–TLM executable models from systems described using UML or SysML. However, they do not solve a well known issue: the lack of formally defined behavioural semantics of State Machine diagrams and Activity diagrams. Moreover, very few works in the literature define the MoCs to be implemented within the generated executable system model.

Initially, this chapter defines the abstract syntax supported for SysML State Machine diagrams and Activity diagrams. Then, the semantic domains for SysML State Machine diagrams and Activity diagrams have been
defined using FSM and SDF MoCs, respectively. Finally, the semantics mapping from the abstract syntax of SysML State Machine diagrams and Activity diagrams to FSM and SDF models of computation has been formalized. As a result, clear semantic transformation rules have been defined for State Machine diagrams and Activity diagrams.

The proposal of this chapter enables to effectively implement a MDD based framework to create SystemC–TLM executable code with heterogeneous behavioural MoCs for early electronic system definition. The implementation of a framework for the formal semantic mappings presented are addressed in Chapter 7. Chapter 7 also describes a code generation process to produce SystemC–TLM code from a SysML model. Besides, Chapter 7 presents a practical case study where the proposed approach has been applied to a complex electronic system development process in order to illustrate its benefits.

This chapter has been the precursor for the paper [141].
## Chapter 7

### Automatic SysML to SystemC-TLM code generation: ASSYST Framework

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Proposed ASSYST Framework</td>
<td>137</td>
</tr>
<tr>
<td>7.2 Implementing Behavioural Semantic Mapping</td>
<td>138</td>
</tr>
<tr>
<td>7.3 Implementing SystemC-TLM Code Generation</td>
<td>154</td>
</tr>
<tr>
<td>7.4 Concluding Remarks</td>
<td>184</td>
</tr>
</tbody>
</table>

The modern electronic systems are increasing their complexity to meet market requirements. However, present Electronic Design Automation (EDA) tools can not handle this complexity growth efficiently. This productivity loss produces the so-called *design productivity gap*, as documented by several editions of the *Design Report* [19] published by the International Technology Roadmap for Semiconductors (ITRS). The ITRS suggested three foundations in order to address the design productivity challenge: increasing the level of abstraction, automation processes and tools, and producing executable specifications. Therefore, in order to reduce the de-
development time and improve the product quality, new languages and design paradigms are needed to handle present electronic systems.

SystemC has become the de facto standard to create executable models of electronic systems. Besides, the Transaction Level Modeling (TLM) supported by SystemC has been proposed for the creation of high abstraction level executable models. These high abstraction level SystemC–TLM models are fast to build and they provide a very high speed of simulation.

Recently, the Model-driven Design (MDD) methodology has emerged as a promising methodology for complex electronic systems modelling and design. MDD enables the modelling and the automatic code generation of functional SystemC–TLM system models at a high abstraction level from visual modelling languages such as Unified Modeling Language (UML) and Systems Modeling Language (SysML). SysML offers systems engineers several noteworthy improvements for electronic systems modelling compared with UML, which tends to be software-centric. In recent years several works have been proposed that focus on creating SystemC code from SysML.

The objective of this paper is to automatically translate SysML graphical models into heterogeneous SystemC–TLM executable code. These executable models enable functional architecture analysis and functional simulation early in the development process, before the SW/HW partitioning occurs. Besides, the automation of the SystemC–TLM executable model generation improves the error-prone manual coding process and it reduces significantly the development time of the device being designed. However, the automation tools for electronic system development can only be applied if the behavioural semantics and the corresponding Model of Computations (MoCs) are well-defined. The well-defined computation semantics of MoCs allow to unambiguously capture the required functionality and enable the application of formal design and verification techniques.

Chapter 6 clearly defined the semantic domains of State Machine diagrams and Activity diagrams using Finite State Machine (FSM) and Synchronous Data Flow (SDF) MoCs, respectively. Chapter 6 proposed the following two formal semantic mappings:

\[ M_{FSM} : A_{StateMachine} \rightarrow S_{FSM} \]
\[ M_{SDF} : A_{Activity} \rightarrow S_{SDF} \]
7.1 Proposed ASSYST Framework

$M_{FSM}$ maps the abstract syntax of SysML State Machine diagrams ($A_{State\text{Machine}}$) to a semantic domain described by the FSM MoC ($S_{FSM}$). $M_{SDF}$ maps the abstract syntax of SysML Activity diagrams ($A_{Activity}$) to a semantic domain described by the SDF MoC ($S_{SDF}$).

In this chapter, a MDD approach to model and design electronic systems using a framework called Automatic SysML to SystemC Translator (ASSYST) is proposed. ASSYST implements the behavioural semantic mappings, $M_{FSM}$ and $M_{SDF}$, defined in Chapter 6. The objective of ASSYST is to automatically translate SysML graphical models into SystemC–TLM executable code with heterogeneous MoCs.

The paper is organized as follows. Section 7.1 introduces the workflow and the modelling steps proposed by the ASSYST framework. Section 7.2 presents the implementation of the behavioural semantic mapping described in the Chapter 6 using Model-to-Model (M2M) transformation techniques. Section 7.3 describes the ASSYST code generation approach based on Model-to-Text (M2T) principles. Finally, Section 7.4 presents the concluding remarks of this chapter.

7.1 PROPOSED ASSYST FRAMEWORK

The ASSYST framework is implemented as a set of Eclipse plug-ins that integrate into the Eclipse Integrated Development Environment (IDE). Additionally, ASSYST proposes a C++ library in order to support the code generation process. The ASSYST C++ library is built on top of SystemC and TLM, and defines base classes for the description of heterogeneous MoCs.

The SystemC–TLM code generation process is depicted in Figure 7.1. ASSYST takes as input a SysML model described using the Topcased modelling platform and produces a SystemC–TLM executable model. The translation of SysML models into SystemC–TLM source code is automatically and seamlessly performed by the ASSYST framework.

The generation of the executable models follows two steps. In the first step, the SysML behavioural parts are mapped to formal MoC using M2M transformation techniques. ASSYST supports heterogeneous behavioural constructs. SysML State Machine diagrams are used to model control-based system functionalities. SysML Activity diagrams are used to model system
functionalities based on data-flow. SysML State Machines are mapped to FSMs and SysML Activities are mapped to SDFs following the formal semantic mappings, $M_{FSM}$ and $M_{SDF}$, proposed in Chapter 6. As a result, the behavioural aspects of the system are translated into FSM and SDF MoCs while the structural information is kept in the Block Definition Diagrams (BDDs) and Internal Block Diagrams (IBDs) of the SysML model. During the second step, a M2T process produces a SystemC-TLM executable code from the FSM, SDFs and SysML models describing the structure of the system being designed.

### 7.2 IMPLEMENTING BEHAVIOURAL SEMANTIC MAPPING

In general, a M2M transformation describes how a source model $M_s$ conforming to a metamodel $MM_s$ is transformed into a target model $M_t$ that conforms to a metamodel $MM_t$ as depicted in Figure 7.2. Both metamodel $MM_s$ and metamodel $MM_t$ conform to a common metametamodel $MMM$. In the M2M transformations proposed in this chapter, the source model is the SysML system model which conforms to the SysML metamodel available in the Topcased Modeling toolkit, as shown in Figure 7.3.
7.2 Implementing Behavioural Semantic Mapping

Figure 7.2: Model-to-model transformation.

Figure 7.3: FSM and SDF models generation workflow.
The targets of the M2M transformations are the SDF and the FSM models conforming to the SDF and FSM metamodels, respectively. The FSM and SDF metamodel definitions are derived from the formal semantics proposed in Chapter 6. The Eclipse Ecore tool has been used for describing FSM and SDF metamodels.

ASSYST uses the ATL Transformation Language (ATL) [142] platform to implement the formal semantic mappings defined in Section 6.3.3 and Section 6.4.3 as shown in Figure 7.3. Developed on top of the Eclipse IDE, the ATL is a M2M transformation language. ATL provides a platform to produce a set of target models from a set of source models. An ATL transformation program is described by rules that specify how source model elements are translated into target model elements.

In order to achieve the model transformations, the ATL execution engine needs the following elements:

- A source metamodel for SysML. The SysML metamodel is provided by the Topcased toolkit.
- A source system model conforming the SysML metamodel. This model describes the system being developed.
- A target metamodel for FSM.
- A target model conforming to the FSM metamodel.
- A target metamodel for SDF.
- A target model conforming to the SDF metamodel.
- An ATL specification to transform SysML State Machines into FSMs.
- An ATL specification to transform SysML Activities to SDFs.

7.2.1 ASSYST METAMODELS DEFINITION

First, the FSM and SDF metamodel definitions are derived from the formal semantics proposed in Chapter 6. The Eclipse Ecore tool has been used for describing FSM and SDF metamodels. These metamodels are used by ASSYST to perform the ATL transformations.
7.2 Implementing Behavioural Semantic Mapping

7.2.1.1 FSM metamodel definition

The FSM metamodel specification is derived from the formal semantics of Section 6.3.2. The derived FSM metamodel is represented in the Ecore diagram depicted in Figure 7.4.

The \textit{FSMDomainModel} element is the root of the FSM metamodel. \textit{FSMDomainModel} includes the FSMs of the system being designed. A FSM is composed of a set of States (\(Q\)), a set of Transitions (\(\Delta\)), and a reference to an initial State (\(q_0\)). The source and target attributes of a Transition specify which States are connected by the transition. Transitions also contain a trigger attribute of type string representing the possible input events (\(\Psi\)) of the FSM. The FSM, State, and Transition elements have a unique name attribute. These attributes ease the code generation process, however, they do not modify the semantics of the MoC.

7.2.1.2 SDF metamodel definition

Similarly to FSM metamodel case, the SDF metamodel specification is derived from the formal semantics of Section 6.4.2. The derived SDF metamodel is represented in Figure 7.5. The root of the SDF metamodel is the \textit{SDFDomainModel} element. \textit{SDFDomainModel} contains all the SDF graphs of the system.
A **SDF** is composed of a set of **Nodes** \((N)\) and a set of **Arcs** \((A)\). The **outgoing** and **incoming** attributes of a **Node** element refer to the output and input **Arcs**. Each **Arc** connects a **source** node and a **target** node. Additionally, the token production rate \((P)\), the token consumption rate \((C)\), and the initial marking \((\mu_0)\) have been integrated as attributes into the **Arcs** element in order to facilitate the implementation of the model transformation process. Therefore, an arc specifies three integer type attributes concerning the reception and transmission of tokens: **incoming_tokens**, which specifies how many tokens does the arc receive from the source node; **outgoing_tokens**, which specifies how many tokens does the arc send to the target node; and **initial_tokens**, which denotes how many tokens are initially in the arc.

All the **SDFs**, **Nodes** and **Arcs** contain an unique **name** attribute of type string. **Arc** elements include an attribute called **token_type**. This attribute specifies the type of the token managed by an **Arc**. These attributes are added to make the subsequent code generation process easier; but they do not have any effect on the semantics of the SDF MoC.

### 7.2.2 ASSYST ATL MODEL TRANSFORMATIONS SPECIFICATION

ATL produces FSM and SDF target models applying model transformation techniques to a SysML source model. An ATL specification includes
7.2 Implementing Behavioural Semantic Mapping

Listing 7.1: ATL transformation file header for FSM generation.

``` ATL
def module statemachine2fsm;
def create OUT : MMFSM from IN : MMUML, IN1 : MMSYSML;
```

a set of rules that define how SysML elements are transformed into FSM and SDF elements.

7.2.2.1 State Machine to FSM transformation

The transformation of State Machines into FSMs is specified in the ATL file “statemachine2fsm.atl”. This file contains an ATL module called statemachine2fsm. Listing 7.1 presents the header section of the ATL file “statemachine2fsm.atl”. The header section imports to ATL variables the necessary metamodels to specify the ATL specification to transform State Machines into FSMs.

- **MMFSM** represents the FSM metamodel previously defined with Ecore in Section 7.2.1.1.
- **MMSYSML** represents the SysML metamodel specified by Topcased.
- **MMUML** represents the UML metamodel specified by Topcased.

**TOPCASED** describes SysML as a metamodel and not as a profile. The SysML metamodel is inherited mainly from the UML metamodel and provides requirements diagrams and block diagrams. The UML elements directly reused in SysML are not redefined but imported from the UML metamodel into the SysML metamodel. UML **BehaviorStateMachines** are imported into SysML. As a result, the State Machine elements of a Topcased SysML model conform to the UML metamodel (variable MMUML).

The module statemachine2fsm is composed of 5 rules as shown in Table 7.1. These rules implement the semantics mapping described in Section 6.3.3.
Table 7.1: StateMachine-to-FSM transformation rules.

<table>
<thead>
<tr>
<th>SysML element</th>
<th>FSM element</th>
<th>Rule Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>FSMDomainModel</td>
<td>Model_to_FSMDomainModel</td>
</tr>
<tr>
<td>StateMachine</td>
<td>FSM</td>
<td>StateMachine_to_FSM</td>
</tr>
<tr>
<td>State</td>
<td>State</td>
<td>State_to_State</td>
</tr>
<tr>
<td>Pseudostate</td>
<td>State</td>
<td>Pseudostate_to_State</td>
</tr>
<tr>
<td>Transition</td>
<td>Transition</td>
<td>Transition_to_Transition</td>
</tr>
</tbody>
</table>

Listing 7.2: Model_to_FSMDomainModel ATL rule.

```plaintext
rule Model_to_FSMDomainModel {
  from
  m : MMUML!Model
  to
  s : MMFSM!FSMDomainModel {
    fsm <- m.allOwnedElements() ->
    select(a | a.oclIsTypeOf(MMUML!StateMachine))
  }
}
```

**Rule 1:** The rule Model_to_FSMDomainModel maps the SysML model’s root element Model to the FSMDomainModel root element as specified in Listing 7.2.

The variable `m` is an object of type `Model` from the `MMUML` metamodel, as specified by `m : MMUML!Model`. On the other hand, the variable `s` is an object of type `FSMDomainModel` from the `MMFSM` metamodel, as specified by `s : MMFSM!FSMDomainModel`. The rule specifies that `s` object are generated from `m` object.

Elements of type `StateMachine` are collected from object `m` and assigned to the attribute `fsm` of type `FSM` of object `s`. The ATL engine resolves the value of the attribute `fsm` applying the rule StateMachine_to_FSM. Only one FSMDomainModel element will be created, as the `Model` element within a SysML model is unique. The `FSMDomainModel` will include all the `FSMs` of the system being designed.

**Rule 2:** The rule StateMachine_to_FSM describes how StateMachine elements conforming to metamodel `MMUML` is transformed into a FSM ele-
7.2 Implementing Behavioural Semantic Mapping

Listing 7.3: StateMachine_to_FSM ATL rule.

```
rule StateMachine_to_FSM {
  from s : MMUML!StateMachine
  to f : MMFSM!FSM {
    name <- s.name,
    states <- s.region->
      select(n | n.oclIsKindOf(MMSYSML!State))
    or n.oclIsKindOf(MMSYSML!Pseudostate)),
    transitions <- s.region->
      select(n | n.oclIsKindOf(MMSYSML!Transition)),
    initial_state <- s.region->
      select(i | i.oclIsKindOf(MMSYSML!Pseudostate))-
      select(i | i.kind = #initial)
  }
}
```

A StateMachine conforming to metamodel MMFSM. The rule StateMachine_to_FSM in Listing 7.3 implements the semantic mapping \( M_{FSM} \) described in Section 6.3.3:

\[
M_{FSM} : A_{StateMachine} \rightarrow S_{FSM}
\]

In the first attribute binding declaration, the name of a StateMachine is assigned to the name attribute of a FSM \( f \).

Elements of type State and Pseudostate are collected from the region of the StateMachine \( s \). These elements will be transformed by the ATL Rule 3 (State_to_State) and the Rule 4 (Pseudostate_to_State), respectively, and then assigned to the attribute states of the FSM \( f \).

Similarly, Transition elements and initial State elements are gathered and assigned to the attributes transitions and initial_state of the FSM \( f \), respectively. The transformations to transitions and initial_state are created by the Rule 5 Transition_to_Transition and the Rule 4 Pseudostate_to_State, respectively.

**Rule 3:** State elements \( (S) \) from SysML models are transformed into FSM State elements \( (Q) \) by the rule State_to_State. This rule is presented in Listing 7.4 and it implements the semantic mapping transformation defined in Eq. (6.15):

\[
Q = S
\]
Chapter 7: Automatic SysML to SystemC-TLM code generation

Listing 7.4: State_to_State ATL rule.

```atl
rule State_to_State {
from
  s : MMUML!State
to
  t : MMFSM!State {
    name <- s.name
  }
}
```

Listing 7.5: Pseudostate_to_State ATL rule.

```atl
rule Pseudostate_to_State {
from
  ps : MMUML!Pseudostate {
    ps.kind = #initial
  }
to
  t : MMFSM!State {
    name <- ps.name
  }
}
```

The attribute name of a State s is assigned to the name attribute of a FSM State f.

Rule 4: The rule Pseudostate_to_State transforms each initial element Pseudostate (PS) of the SysML model into a State element ($q_0 \in Q$) conforming the metamodel MMFSM, as shown in Listing 7.5. The rule Pseudostate_to_State implements the semantic mapping transformation defined in Eq. (6.16):

$$q_0 = \{ ps | \text{kind}(ps) = \text{InitialState}, ps \in PS \}$$

The attributes of the FSM State t are created similarly to the ones defined in the rule State_to_State.

Rule 5: The rule Transition_to_Transition transforms Transition elements (T) of SysML into FSM Transition elements ($\Delta$), as shown in Listing 7.6. This rule also creates the triggers of the FSM Transitions. These triggers denote the input events ($\Psi$) of the FSM. This rule imple-
7.2 Implementing Behavioural Semantic Mapping

Listing 7.6: Transition_to_Transition ATL rule.

```atl
rule Transition_to_Transition {
  from t : MMUML!Transition
to
  n : MMFSM!Transition {
    name <- t.name,
    source <- t.source,
    target <- t.target,
    trigger <- t.trigger.name
  }
}
```

Listing 7.7: ATL transformation file header for SDF generation.

```atl
-- @nsURI MMSYSML=http://www.topcased.org/2.0/sysML
-- @nsURI MMUML=http://www.eclipse.org/uml2/3.0.0/UML
-- @path MMSDF=http://assyst/model/sdf/SDF
module activity2sdf;
create OUT : MMSDF from IN : MMUML, IN1 : MMSYSML;
```

ments the following semantic mapping transformations

\[ \Psi = \{\text{trigger}(t) \mid t \in T\} \]
\[ \Delta = \{(\text{source}(t), \text{trigger}(t), \text{target}(t)) \mid t \in T\} \]

Source and the target states are assigned to the attributes source and target. Moreover, this rule assigns the names of the trigger, guard and effect parameter of the Transition elements of SysML model to the attributes trigger, guard and effect of the Transition \( t \), respectively.

7.2.2.2 Activity to SDF transformation

The transformation of Activities into SDFs is specified in the ATL file “activity2sdf.atl”. This file contains an ATL module called activity2sdf. Listing 7.7 presents the header section of the file “activity2sdf.atl”. The header section imports to the ATL module the necessary metamodels to specify the ATL specification to transform Activities into SDFs.
Table 7.2: Activity-to-SDF transformation rules.

<table>
<thead>
<tr>
<th>SysML element</th>
<th>SDF element</th>
<th>Rule Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>SDFDomainModel</td>
<td>Model_to_SDFDomainModel</td>
</tr>
<tr>
<td>Activity</td>
<td>SDF</td>
<td>Activity_to_SDF</td>
</tr>
<tr>
<td>ActivityNode</td>
<td>Node</td>
<td>ActivityNode_to_Node</td>
</tr>
<tr>
<td>ControlFlow</td>
<td>Arc</td>
<td>ControlFlow_to_Arc</td>
</tr>
<tr>
<td>ObjectFlow</td>
<td>Arc</td>
<td>PinToPin_ObjectFlow_to_Arc</td>
</tr>
<tr>
<td>ObjectFlow</td>
<td>Arc</td>
<td>APNToPin_ObjectFlow_to_Arc</td>
</tr>
<tr>
<td>ObjectFlow</td>
<td>Arc</td>
<td>PinToAPN_ObjectFlow_to_Arc</td>
</tr>
</tbody>
</table>

- MMSDF represents the SDF metamodel previously defined with Ecore in Section 7.2.1.2.
- MMSYSML represents the SysML metamodel provided by Topcased.
- MMUML represents the UML metamodel provided by Topcased.

The module activity2sdf is composed of 8 rules as shown in Table 7.2.

**Rule 1:** The rule Model_to_SDFDomainModel in Listing 7.8 translates SysML model’s root element Model to SDFDomainModel root element. The rule source pattern specifies one variable \( m \) of type Model conforming the metamodel MMUML and the target pattern contains one element \( s \) of type SDFDomainModel conforming the metamodel MMSDF. As the Model element within a SysML model is unique, only one SDFDomainModel model will be created. The SDFDomainModel will include all the SDFs of the system being designed.

This rule presents a binding that initializes the attribute sdf of the SDFDomainModel. The value is assigned by obtaining all the Activity elements found in the model. The resolution algorithm of the ATL engine resolves the binding assigning the output of the rule Activity_to_SDF to the variable sdf.

**Rule 2:** The rule Activity_to_SDF in Listing 7.9 implements the semantic mapping \( M_{SDF} \) described in Section 6.4.3:

\[
M_{SDF} : A_{Activity} \rightarrow S_{SDF}
\]
7.2 Implementing Behavioural Semantic Mapping

Listing 7.8: Model_to_SDFDomainModel ATL rule.

```
rule Model_to_SDFDomainModel {
  from
    m : MMUML!Model
to
    s : MMSDF!SDFDomainModel {
      sdf <- m.allOwnedElements() ->
        select(a | a.oclIsTypeOf(MMUML!Activity))
    }
}
```

Listing 7.9: Activity_to_SDF ATL rule.

```
rule Activity_to_SDF {
  from
    a : MMUML!Activity
to
    s : MMSDF!SDF {
      name <- a.name,
      nodes <- a.node ->
        select(n | n.oclIsKindOf(MMSYSML!ActivityFinalNode)
            or n.oclIsKindOf(MMSYSML!InitialNode)
            or n.oclIsKindOf(MMSYSML!Action)
            or n.oclIsKindOf(MMSYSML!ActivityParameterNode)),
      arcs <- a.edge
    }
}
```

The rule Activity_to_SDF transforms each Activity into a SDF element. The rule source pattern specifies one variable a : MMUML!Activity and the target pattern contains one variable s : MMSDF!SDF.

The name of an Activity is assigned to the name attribute of a SDF. ActivityFinalNode elements, InitialNode elements, Action elements and ActivityParameterNode elements are transformed into nodes of the SDF s. The node attribute of a SDF will be populated by the Rule 3.

Furthermore, the edge elements of the Activity will be assigned to the attribute arcs by the ATL Rules 4, 5, 6 and 7. Each of these rules
Listing 7.10: ActivityNode_to_Node ATL rule.

```plaintext
rule ActivityNode_to_Node {
  from
    an : MMUML!ActivityNode(
        an.oclIsKindOf(MMUML!ActivityFinalNode)
    or an.oclIsKindOf(MMUML!InitialNode)
    or an.oclIsKindOf(MMUML!ActivityParameterNode)
    or an.oclIsKindOf(MMUML!Action)
    )
  to
    n : MMSDF!Node {
      name <- an.name
    }
}
```

implements a part of the semantic mappings

\[ A = A_{CF} \cup A_{OF1} \cup A_{OF2} \cup A_{OF3} \]
\[ P = P_{CF} \cup P_{OF1} \cup P_{OF2} \cup P_{OF3} \]
\[ C = C_{CF} \cup C_{OF1} \cup C_{OF2} \cup C_{OF3} \]

**Rule 3:** InitialNode elements \((IN)\), ActivityFinalNode elements \((AFN)\), ActivityParameterNode elements \((APN)\) and Actions \((AC)\) are transformed into SDF Nodes \((N)\) by the rule ActivityNode_to_Node. The rule ActivityNode_to_Node in Listing 7.10 implements the following semantic mapping transformation defined in Eq. (6.54a):

\[ N = AFN \cup IN \cup APN \cup AC \]

The rule source pattern specifies a variable \(an : MMUML!ActivityNode\) and a guard condition which specifies that only ActivityFinalNode elements, InitialNode elements, ActivityParameterNode elements, or Action elements are matched by this rule. The target pattern contains one variable \(n : MMSDF!Node\).

**Rule 4:** The rule ControlFlow_to_Arc transforms ControlFlow elements into a Arc element. The rule ControlFlow_to_Arc in Listing 7.11 implements the following semantic mapping transformations

\[ A_{CF} = \{(source(cf), target(cf)) \mid cf \in CF\} \]
\[ P_{CF} = \{(source(cf), \langle source(cf), target(cf) \rangle), 1 \mid cf \in CF\} \]
\[ C_{CF} = \{(source(cf), \langle source(cf), target(cf) \rangle), 1 \mid cf \in CF\} \]
7.2 Implementing Behavioural Semantic Mapping

Listing 7.11: ControlFlow_to_Arc ATL rule.

```plaintext
rule ControlFlow_to_Arc {
  from
    cf : MMUML!ControlFlow
  to
    n : MMSDF!Arc {
      name <- cf.name,
      source <- cf.source,
      target <- cf.target,
      incoming_tokens <- 1,
      outgoing_tokens <- 1,
      initial_tokens <- 0
    }
}
```

The rule source pattern defines a variable a : MMUML!ControlFlow and the target pattern includes a variable s : MMSDF!Arc.

This rule sets the attributes incoming_tokens and outgoing_tokens to 1.

All the rules that create Arc elements assign the source and the target nodes to source and target attributes, respectively. Furthermore, the initial_marking attribute of the arcs is set to 0.

Rule 5: The rule PinToPinObjectFlow_to_Arc maps ObjectFlow elements between Pins owned by Actions into Arc elements. This rule is described in Listing 7.12 and it implements the following semantic mapping transformations:

\[ A_{OF1} = \{ \langle s, t \rangle \mid \{ s, t \} \subset AC, of \in OF, source(of) \in output(s), target(of) \in input(t) \} \]

\[ P_{OF1} = \{ \langle s, (s, t), upperBound(op) \rangle \mid \{ s, t \} \subset AC, of \in OF, \]
\[ op \in PIN, source(of) = op, op \in output(s), target(of) \in input(t) \} \]

\[ C_{OF1} = \{ \langle t, (s, t), upperBound(ip) \rangle \mid \{ s, t \} \subset AC, of \in OF, \]
\[ ip \in PIN, source(of) \in output(s), target(of) = ip, \]
\[ ip \in input(t) \} \]
Listing 7.12: PinToPinObjectFlow_to_Arc ATL rule.

```java
rule PinToPinObjectFlow_to_Arc {
    from
    of : MMUML!ObjectFlow {
        of.source.oclIsKindOf(MMUML!Pin)
        and of.target.oclIsKindOf(MMUML!Pin)
    }
    to
    n : MMSDF!Arc {
        name <- of.name,
        source <- of.source.owner,
        target <- of.target.owner,
        incoming_tokens <- of.source.upperBound,
        outgoing_tokens <- of.target.upperBound,
        initial_tokens <- 0,
        token_type <- of.source.type.name.toString()
    }
}
```

The rule source pattern specifies a variable \( an : \text{MMUML!ObjectFlow} \) and a guard condition which specifies that only ObjectFlows with Pins in both ends are matched by this rule. The target pattern contains one variable \( n : \text{MMSDF!Arc} \).

This rule sets the attribute \( \text{incoming\_tokens} \) with the \( \text{upper\_Bound} \) value of the source Pin and sets the attribute \( \text{outgoing\_tokens} \) with the \( \text{upper\_Bound} \) value of the target Pin.

**Rule 6:** APNToPinObjectFlow_to_Arc rule translates ObjectFlow elements between a ActivityParameterNode and a Pin to Arc elements. The rule APNToPinObjectFlow_to_Arc in Listing 7.13 implements the following semantic mapping transformations:

\[
A_{OF2} = \{ (p,t) \mid p \in \text{APN}, o \in \text{OF}, t \in \text{AC}, \\
\text{source}(of) = p, \text{target}(of) \in \text{input}(t) \}
\]

\[
P_{OF2} = \{ (p, (p,t), 1) \mid p \in \text{APN}, o \in \text{OF}, t \in \text{AC}, \\
\text{source}(of) = p, \text{target}(of) \in \text{input}(t) \}
\]

\[
C_{OF2} = \{ (t, (p,t), \text{upperBound}(ip)) \mid p \in \text{APN}, o \in \text{OF}, \\
ip \in \text{PIN}, t \in \text{AC}, \text{source}(of) = p, \text{target}(of) = ip, \\
ip \in \text{input}(t) \}
\]
7.2 Implementing Behavioural Semantic Mapping

Listing 7.13: APNToPinObjectFlow_to_Arc ATL rule.

```plaintext
rule APNToPinObjectFlow_to_Arc {
  from
    of : MMUML!ObjectFlow {
      of.source.oclIsKindOf(MMUML!ActivityParameterNode)
      and of.target.oclIsKindOf(MMUML!Pin)
    }
  to
    n : MMSDF!Arc {
      name <- of.name,
      source <- of.source,
      target <- of.target.owner,
      incoming_tokens <- 1,
      outgoing_tokens <- of.target.upperBound,
      initial_tokens <- 0,
      token_type <- of.source.type.name.toString()
    }
}
```

The rule source pattern specifies a variable an : MMUML!ObjectFlow and a guard condition which specifies that only ObjectFlows with an ActivityParameterNode as source and a Pin as target are matched by this rule. The target pattern contains a variable n : MMSDF!Arc.

This rule sets the attribute incoming_tokens to 1 and sets the attribute outgoing_tokens with the upperBound value of the target Pin.

**Rule 7:** PinToAPNObjectFlow_to_Arc rule translates ObjectFlow elements between a Pin and an ActivityParameterNode to Arc elements. The rule PinToAPNObjectFlow_to_Arc in Listing 7.14 implements the following semantic mapping transformations:

\[
A_{OF3} = \{(s, p) \mid s \in AC, \ of \in OF, \ p \in APN, \ source(of) \in output(s), \ target(of) = p\}
\]

\[
P_{OF3} = \{(s, (s, p), \text{upperBound}(op)) \mid s \in AC, \ op \in PIN, \ of \in OF, \ p \in APN, \ source(of) = op, \ target(of) = p, \ op \in output(s)\}
\]

\[
C_{OF3} = \{(p, (s, p), 1) \mid s \in AC, \ o \in OF, \ p \in APN, \ source(of) \in output(s), \ target(of) = p\}
\]
Listing 7.14: PinToAPNObjectFlow_to_Arc ATL rule.

```plaintext
rule PinToAPNObjectFlow_to_Arc {
  from
    of : MMUML!ObjectFlow {
      of.source.oclIsKindOf(MMUML!Pin)
      and of.target.oclIsKindOf(MMUML!ActivityParameterNode)
    }
  to
    n : MMSDF!Arc {
      name <- of.name,
      source <- of.source.owner,
      target <- of.target,
      incoming_tokens <- of.source.upperBound,
      outgoing_tokens <- 1,
      initial_tokens <- 0,
      token_type <- of.source.type.name.toString()
    }
}
```

The rule source pattern specifies a variable an : MMUML!ObjectFlow and a guard condition which specifies that only ObjectFlows with a Pin as source and an ActivityParameterNode as target are matched by this rule. The target pattern contains a variable n : MMSDF!Arc.

This rule sets the attribute incoming_tokens with the upperBound value of the source Pin and sets the attribute outgoing_tokens to 1.

All the rules that create Arc elements assign the source and the target nodes to source and target attributes, respectively. Furthermore, the initial_marking attribute of the arcs is set to 0.

7.3 IMPLEMENTING SYSTEMC-TLM CODE GENERATION

Acceleo is an Eclipse based product created and developed by Obeo [143]. Acceleo is a code generator implementing the MOF Model to Text Transformation Language (MOFM2T) standard [144] specified by Object Management Group (OMG). Acceleo is a template-based approach to transform a source model $M_s$ conforming to a metamodel $MM_s$ into a text file.

The Acceleo platform is proposed for the code generation process depicted in Figure 7.6. The objective of this M2T workflow is to produce the source files of the SystemC-TLM executable model from the FSM, SDF and
SysML models describing the system under design. In a previous step, the FSM and the SDF models are created from the State Machine Diagrams and the Activity Diagrams using the ATL transformations described in Section 7.2.2. The FSM and SDF models specify the behavioural aspects of the system while the structural features are extracted from the BDD and IBD diagrams of the SysML model.

Additionally, the M2T framework described in Figure 7.6 is supported by the proposed ASSYST C++ library. This C++ library aims to simplify the M2T template definition providing reusable components to describe heterogeneous executable models.
7.3.1 ASSYST C++ LIBRARY

The proposed C++ library is built on top of SystemC v2.2.0 and TLM v2.0.1 library. This library defines C++ base classes for the description of a heterogeneous executable model of the system being designed. The key benefit is that the Acceleo templates can produce an executable system model creating software artifacts that are derived from base classes defined in the C++ library. Thus, the code generation process and the complexity of the Acceleo templates are simplified.

The base classes defined in the C++ library are presented in Figure 7.7. These classes describe the base features needed to describe a system.

7.3.1.1 Structural Class

Structural features are modelled with the class Block declared in Figure 7.8. Block elements are derived from the SystemC class sc_module and they allow to describe the hierarchy of the system being designed.
7.3 Implementing SystemC-TLM Code Generation

### Figure 7.8: Block class.

```plaintext
scmodule

Block

+ Block(block\_name : sc\_module\_name)
+ ~Block()
```

### Figure 7.9: Transaction class.

```plaintext
Transaction

- Id : int
- InitTime : double
- EndTime : double
+ Transaction()
+ ~Transaction()
+ Get\_Transaction\_Name() : string
+ Get\_Transaction\_Report() : void
+ Set\_Id(id : int) : void
+ Set\_StrId(id : const char*) : void
+ Get\_InitTime() : int
+ Set\_InitTime(init\_time : int) : void
+ Set\_StrInitTime(init\_time : const char*) : void
+ Get\_EndTime() : double
+ Set\_EndTime(end\_time : int) : void
+ Set\_StrEndTime(end\_time : const char*) : void
+ Get\_EndTime() : double
```

### 7.3.1.2 Communication Classes

The classes `Transaction` and `TransactionsSocket` describe the communication features of the system. TLM–2.0 [9] resources have been used to support the communication aspects.

The class `Transaction` models the events of the system. The class `Transaction` represents the unit of information exchanged between the external stimuli and the system, and between the several parts composing the system. `Transaction` defines the following three fundamental attributes for system transactions:

- **Id** is the transaction identifier.
- **InitTime** indicates when the transaction shall be sent.
TransactionsSocket class.

- *EndTime* indicates when the transaction has been received.

*Transaction* provides setter and getter functions to handle its attributes. Additionally, the function *Get_Transaction_Name()* returns the name of the transaction and the function *Get_Transaction_Report()* prints a report with the values of the attributes in the execution terminal. The latter functions are virtual, so derived classes can redefine them.

The class *Transaction* class defines a base class; specific system implementations create new transactions including additional attributes extending the class *Transaction*.

The class *TransactionsSocket* describes the *Initiator* and *Target* sockets for *Transaction* elements, as presented in Figure 7.10. TLM–2.0 sockets are transport interfaces. The TLM–2.0 core interfaces pass transactions between initiator sockets and target sockets.

On one hand, the class *multipassthrough_initiator_socket* from the TLM v2.0.1 library is used to define the *Initiator* sockets. On the other hand, the class *multipassthrough_target_socket* from the TLM v2.0.1 library is used to define the *Target* sockets. A new TLM protocol has been defined within the class *assyst_protocol*. The *assyst_protocol* uses the previously described *Transaction* as the payload and the phase type *tlm_phase* defined in TLM–2.0 [9]. The *Initiator* and the *Target* sockets are imple-
7.3 Implementing SystemC-TLM Code Generation

```
<<interface>>
<TransactionFactory_if

+ CreateTransaction(transaction : TiXmlElement*) : boost::shared_ptr<Transaction>
+ CreateXmlElement(transaction : Transaction*) : TiXmlElement
```

Figure 7.11: TransactionFactory_if class.

mented specialising the classes `multipassthrough_initiator_socket` and `multipassthrough_target_socket`, respectively, by means of the class `assyst_protocol`.

The SystemC TLM–2.0 defines several types of sockets. The sockets `multipassthrough_initiator_socket` and `multipassthrough_target_socket` have been used due to their flexibility. On one hand, a single socket can be bound to multiple sockets on other components. On the other hand, unlike other TLM–2.0 sockets, these also support hierarchical child-to-parent socket binding on both the initiator and the target side.

### 7.3.1.3 XML Management Classes

The external event management features are described using the classes `XMLTransactionReader` and `XMLTransactionWriter`. eXtensible Markup Language (XML) files are used for the description of both the transactions to be injected in the system and the transactions received from the system.

The ASSYST C++ library uses TinyXML v2.6.1 [145] to handle XML files. TinyXML is a simple and small C++ XML parser. TinyXML parses an XML document, and builds from that a Document Object Model (DOM)\(^1\) [146] that can be read, modified, and saved.

The class `TransactionFactory_if` is a C++ interface class\(^2\), as declared in Figure 7.11. This class describes an interface to transform XML elements (class `TiXmlElement` from TinyXML) into `Transaction` elements and vice versa, through the functions `CreateTransaction` and `CreateXmlElement` respectively. Particularly, the function `CreateTransaction` produces smart pointers (class `shared_ptr` from Boost C++ libraries [122])

\(^1\)The Document Object Model (DOM) is a platform- and language-neutral interface for representing and interacting with objects in HTML, XHTML and XML documents.

\(^2\)In C++, an interface class is an abstract base class containing only pure virtual function declarations. Sometimes, an exception is made adding a virtual destructor to the interface class.
XMLTransactionReader

- _xmlDoc : TiXmlDocument*
- _transaction_factory : TransactionFactory*
- _current_element : TiXmlElement*
- _xmlDoc_loaded : bool
- _transaction_factory_loaded : bool

+ XMLTransactionReader(pFilename : const char*, transaction_factory : TransactionFactory*)
+ ~XMLTransactionReader()
+ GetNewTransaction() : boost::shared_ptr
- GetFirstXmlElement() : TiXmlElement*
- GetNextXmlElement() : TiXmlElement*

Figure 7.12: XMLTransactionReader class.

which store pointers to dynamically allocated Transaction objects. Smart pointers provide automatic and correct object lifetime management. These Transaction objects are guaranteed to be deleted when the last shared_ptr pointing to it is destroyed or reset.

The TransactionFactory_if is used for both the transactions injection from the XML files into the system, and the recording of the transactions received from the system into the XML files. Specific class implementations shall be derived from the class TransactionFactory_if within the system being designed. These specific classes shall implement the pure virtual functions CreateTransaction and CreateXmlElement.

XMLTransactionReader reads an input XML file and injects the transactions into the system. The class XMLTransactionReader is declared in Figure 7.12. The constructor of the XMLTransactionReader has two parameters:

- pFilename represents the name of the input XML file containing the transactions.
- transaction_factory receives an specific transaction factory conforming the TransactionFactory_if interface.

The class XMLTransactionReader is responsible for handling the XML file, creating the DOM structure and retrieving the input transactions using the function GetNewTransaction. This function returns transactions obtained through transaction_factory.
XMLTransactionWriter

- _pFilename : const char*
- _xml_doc : TiXmlDocument*
- _transaction_factory : TransactionFactory*
- _transaction_factory_loaded : bool

XMLTransactionWriter(pFilename : const char*, transaction_factory : TransactionFactory*)
~XMLTransactionWriter()
SetNewTransaction(transaction : Transaction*) : void

Figure 7.13: XMLTransactionWriter class.

Listing 7.15: ASSYST input stimuli XML file.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
2
3 <!-- This is a comment -->
4 <Transaction1 Id="0" InitTime="1.5" EndTime="0.0"
5     IntegerAttribute="10" DoubleAttribute="8.2" />
6 <Transaction2 Id="1" InitTime="3.0" EndTime="0.0"
7     StringAttribute="string" />
8 <Transaction3 Id="2" InitTime="7.2" EndTime="0.0" />
9 <Transaction4 Id="3" InitTime="13.8" EndTime="0.0"
10     BoolAttribute="false" />
11 ...
```

XMLTransactionWriter receives the transactions from the system and writes them into the output XML file. The class XMLTransactionWriter is declared in Figure 7.13. The constructor of the class XMLTransactionWriter is identical to that defined for the class XMLTransactionReader. The objective of the class XMLTransactionWriter is to handle the output XML file, and store in it the output transactions using the function SetNewTransaction. This function saves output transactions be means of the transaction_factory object.

ASSYST employs a simple XML format for transaction injection and reception, using simple XML nodes containing XML attributes, as described in the example shown in Listing 7.15. In this example, a sequence of XML nodes is specified. XML nodes represent input transactions, such as Transaction1, Transaction2, etc. These transactions shall have their corresponding classes derived from the class Transaction. XML attributes represent the attributes of the transactions.
7.3.1.4 Behavioural Classes

The class *UseCase* represents the interface between the structural aspects and the behavioural aspects. The class *UseCase* is declared in Figure 7.14. *UseCases* describe the functionality of a system in relation to how the users use that system. This functionality is expressed in terms of behaviours, such as FSMs and SDFs. Moreover, the services represented by *UseCases* have to be offered by the *Block* elements of the system.

The class *Behavior*, shown in Figure 7.15, defines the common behavioural features, such as the event management and the event processing. *Behavior* is a C++ abstract class. The class *Behavior* declares a pure virtual function named *process*; as a result, the classes derived from *Behavior* shall implement it. The aim of the function *process* is to define an interface in order to receive input transactions and process them.

FSM and SDF MoCs are implemented taking *Behavior* as the base class. Each of them implements a particular event processing algorithm conforming the runtime semantics defined in Section 6.3.2 (FSM runtime semantics) and Section 6.4.2 (SDF runtime semantics). More MoCs can be described deriving from the class *Behavior*, and thus, the ASSYST C++ library can be easily extended.

---

**Figure 7.14:** *UseCase* class.

<table>
<thead>
<tr>
<th>UseCase</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ UseCase()</td>
</tr>
<tr>
<td>+ ~UseCase()</td>
</tr>
</tbody>
</table>

**Figure 7.15:** *Behavior* class.

<table>
<thead>
<tr>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Behavior()</td>
</tr>
<tr>
<td>+ ~Behavior()</td>
</tr>
<tr>
<td>+ process(tag : int, trans : Transaction&amp;, delay : sc_time&amp;) : void</td>
</tr>
</tbody>
</table>
7.3 Implementing SystemC-TLM Code Generation

### 7.3.1.5 FSM Behavioural Classes

The behavioural aspects of the FSM MoC are described through three C++ classes: the class `State`, the class `Transition` and the class `FiniteStateMachine`.

The class `State` is declared in Figure 7.16. `State` is a C++ abstract class representing the state elements of the FSM MoC. The class `State` contains three pure virtual functions: `entry_behavior`, `exit_behavior` and `do_behavior`. These functions shall be implemented in derived classes.

The abstract class `Transition` shown in Figure 7.17 represents the transition elements of the FSM MoC. The pure virtual function `check` of the class `Transition` shall be implemented by derived classes.

Figure 7.18 presents the class `FiniteStateMachine`. `FiniteStateMachine` is the main class representing the FSM MoC. The constructor of the `FiniteStateMachine` receives the initial state (class member `init_state`) of the state machine.

The pure virtual function `process` declared in the class `Behavior` is implemented within the class `FiniteStateMachine`. This function processes incoming transactions and performs a runtime step of the FSM MoC as described in Section 6.3.2. The function `process` takes the incoming transaction and checks whether an enabled transition is available in the current
active state represented by the class member \(_\text{current\_state}\). If a transition is enabled, the exit function of the current state is performed and the current active state is set to the target state of the transition. Finally, the entry function of the newly activated state is executed.

### 7.3.1.6 SDF Behavioural Classes

The SDF nodes are implemented with the abstract class \(\text{Node}\) declared in Figure 7.19. The function \(\text{do\_behavior}\) is a pure virtual function that the derived classes shall implement. This function represents the data processing carried out by the \(\text{Node}\).

The abstract class \(\text{Arc}\), shown in Figure 7.20, represents the arc elements of a SDF. The functions \(\text{get\_production\_rate}\), \(\text{get\_consumption\_rate}\) and \(\text{get\_initial\_tokens}\) are pure virtual functions. Each class derived from \(\text{Arc}\) shall implement them, indicating how many tokens are produced, consumed, or initially set in the specific arc class.
7.3 Implementing SystemC-TLM Code Generation

Figure 7.20: Arc class.

*SynchronousDataFlow* is the main class describing the SDF MoC. The declaration of *SynchronousDataFlow* is shown in Figure 7.21. Similarly to *FiniteStateMachine*, *SynchronousDataFlow* is derived from the base class *Behavior*.

The functions `sdf_register_node` and `sdf_register_arc` register the *Node* and *Arc* objects into the *SynchronousDataFlow*. Only registered nodes and arcs are taken into account in the calculations of the topology matrix and the static scheduling.

The function `sdf_init` initializes the *SynchronousDataFlow* creating the topology matrix and the static scheduling. The class *SynchronousDataFlow* uses the GNU Scientific Library (GSL)\(^3\) [147] for vector and matrix calculations performed during the creation of both the topology matrix and the static scheduling.

First, the *SynchronousDataFlow* builds the topology matrix, \(\Gamma\) (as defined in Section 6.4.2), by means of the function `create_topology_matrix`. The topology matrix \(\Gamma\) is constructed with the token consumption and production rates of the *SynchronousDataFlow* elements. Once the topology matrix has been correctly constructed, the rank of the topology matrix (class member `topology_matrix_rank`), \(\text{rank}(\Gamma)\), is calculated.

If \(\text{rank}(\Gamma) = |N| - 1\), being \(|N|\) the number of registered nodes, the strictly positive integer vector \(\mathbf{q}\) (class member `nullspace_vector`) is calcu-

---

\(^3\)The GNU Scientific Library (GSL) provides a collection of numerical routines for scientific computing. GSL contains several mathematical routines such as vectors and matrices computations, sorting routines, random number generators, root-finding routines and functions for linear algebra.
Chapter 7: Automatic SysML to SystemC-TLM code generation

Behavior

+ Behavior()
+ ~Behavior()
+ process(tag : int, trans : Transaction&, delay : sc_time&) : void

SynchronousDataFlow

- _node_vector : Node* [*]
- _node_name_vector : string [*]
- _arc_vector : Arc* [*]
- _arc_name_vector : string [*]
- _topology_matrix : gsl_matrix*
- _topology_matrix_rank : unsigned int
- _nullspace_vector : gsl_vector*
- _initial_tokens_vector : gsl_vector*
- _scheduled_node_vector : Node* [*]
- _scheduled_node_name_vector : string [*]

+ SynchronousDataFlow()
+ ~SynchronousDataFlow()
+ process(tag : int, trans : Transaction&, delay : sc_time&) : void

Figure 7.21: SynchronousDataFlow class.
lated in the right nullspace of $\Gamma$ as: $\Gamma q = 0$. This functionality is implemented in the function `get_matrix_rank_and_nullspace`.

On the other hand, a static scheduling is calculated within the function `create_static_scheduling` following the scheduling algorithm described in Section 6.4.2. As a result, the scheduled firing order of the SDF `Node` elements is stored within the member vector `scheduled_node_vector`.

Finally, the pure virtual function `process` declared in the class `Behavior` is implemented in `SynchronousDataFlow`. This function processes the input transactions of the `SynchronousDataFlow` and executes the static scheduling. During the static scheduling execution the function `do_behavior` of the `Node` elements stored in the vector `scheduled_node_vector` are executed one by one.

### 7.3.2 ASSYST ACCELEO CODE GENERATION

Acceleo is a template-based code generation platform to transform models into text files. An Acceleo module specifies an input template, a source metamodel $MM_s$ and a source metaclass $MC_s$ within $MM_s$ in order to produce a target text file. The Acceleo M2T generation process is depicted in Figure 7.22. When an Acceleo module is applied to a source model $M_s$ conforming the metamodel $MM_s$, the Acceleo generation engine extracts the source model elements that conform to the metaclass $MC_s$ and employs the specified template to produce the output text file.

Acceleo modules are implemented in `.mtl` files and comply with the language syntax defined in the Meta-Object Facility (MOF) Model to Text Transformation Language specification [144]. The Acceleo module file is composed of static text parts and dynamic text parts. The static text parts are common to all model elements conforming to the metaclass $MC_s$ specified in the module. On the other hand, the dynamic text parts depend on the particular values of the source models. An Acceleo template specifies both static text and model queries. Placeholders are included within the generation template to implement queries and extract data from models. The placeholders are delimited by `['` and `']`, that is, placeholders follow the format `['query/']`. The queries are expressions specified over the source $MM_s$ metamodel for extracting the values from the $M_s$ model. Having defined a source metamodel, a source metaclass, a source model, and a generator template, the Acceleo generation engine executes the model
queries on the source model $M_s$ and merges the results of the queries with the static text to produce the target text file.

An Acceleo generation process can be launched several times over the same input model. The text files produced in a previous generation are overwritten every time an Acceleo generation module is relaunched. However, some degree of specialization of the files that have been generated is often desired. Acceleo provides protected user code sections to support an incremental generation process. User code is the piece of text inside an output file that will be preserved from one generation to another. This text is identified by a pair of delimiting tags. The template code written between the tags \{protected\} and \{/protected\} is protected, and is not overwritten during the next generation. This allows a design engineer to include text manually inside the protected user code sections contained in the files already generated by Acceleo. If the generation module is relaunched, the new output file will implement any modifications performed in the input model but the protected text will not be overwritten. If the protection tags are deleted because of a modification of the template, or due to the deletion of an element of the model, the design engineer is informed of the loss of the code. A “lost” file containing the suppressed user text is created alongside the file that has been generated.
The Acceleo modules of the ASSYST framework are described in the following sections. The code generation templates have been divided into three groups:

- Templates translating metaclasses of the SysML metamodel.
- Templates translating metaclasses of the FSM metamodel.
- Templates translating metaclasses of the SDF metamodel.

### 7.3.3 ASSYST SYSML CODE GENERATION

The SysML code generation process takes the SysML model as the source model. The SysML model conforms to the SysML metamodel provided by the Topcased toolkit.

The proposed code generators provide Acceleo M2T templates to generate source code files from Model, Actor, Block, UseCase, Signal, Datatype, and Enumeration elements of the SysML metamodel.

#### 7.3.3.1 Actor code generation

The Actor elements of the SysML model are translated into C++ classes. The code generation template of the Actor elements is specified in the Acceleo module “generate_actor.mtl” shown in Listing 7.16.

The module generate_actor includes the UML metamodel provided by Eclipse and the SysML metamodel provided by Topcased in order to traverse the elements of the SysML model. As explained in Section 7.2.2, Topcased describes SysML as a metamodel inherited from the UML metamodel, and not as a profile. The UML elements directly reused in SysML are not redefined but imported from the UML metamodel into the SysML metamodel, and thus, some elements conform to the UML metamodel. As a result, some elements of a Topcased SysML model conform to the UML metamodel.

The module generate_actor defines the template generate_actor for the model elements a of type Actor (expressed as a : Actor). The template generate_actor contains two file blocks in order to create two code files implementing a C++ class from each Actor element, as
Listing 7.16: Acceleo template file “generate_actor.mtl”.

```acceleo
[module generate_actor('http://www.eclipse.org/uml2/3.0.0/ UML', 'http://www.topcased.org/2.0/sysML')]
[template public generate_actor(a : Actor)]
[file ('include/Actor/'.concat(a.name).concat('_Actor.h'), false, 'UTF-8')]
...
[/file]
[file ('src/Actor/'.concat(a.name).concat('_Actor.cpp'), false, 'UTF-8')]
...
[/file]
[/template]
```

described in Listing 7.16. C++ classes are implemented in two source code files: the header file “[a.name/]

Actor.h” created in the relative directory path include/Actor/ and the source file “[a.name/]

Actor.cpp” created in the relative directory path src/Actor/4. The query specified inside the file blocks is ‘[a.name/]’. For example, assuming the name of the SysML Actor being processes by Acceleo is User, the files generated are “User_Actor.h” and “User_Actor.cpp”. The first file block creates the file “[a.name/]

Actor.h”. The contents of this file block are detailed as an example in Listing 7.17.

The module generate_actor produces classes called “[a.name/]

Actor”, which are derived from the class sc_module of the SystemC library, as shown in Figure 7.23. The actor class includes the transaction factory specified for the model in order to inject/receive transactions to/from the system. Actor classes contain an initiator TransactionsSocket, a target TransactionsSocket, a XMLTransactionReader, and a XMLTransactionWriter. Actor objects read transactions from an input XML file using the XMLTransactionReader object, and inject those transactions into the system through the initiator TransactionsSocket object. On the other hand, actor objects receive transactions from the system through the target TransactionsSocket object, and write those transactions to an output XML file

4The base directory path is specified to the Acceleo generator engine.
Listing 7.17: Header file template details of “generate_actor.mtl”.

```cpp
[file ('include/Actor/'.concat(a.name).concat('_Actor.h'), false, 'UTF-8'))]
#elseif [a.name.toUpperCase()].ACTOR_H_
#define [a.name.toUpperCase()].ACTOR_H_
#include "assyst/assyst.h"
using namespace assyst;
#include [getModel().name/].TransactionFactory.h"

class [a.name/].Actor: public sc_module {
  public:
    TransactionsSocket<[a.name/].Actor>::Initiator
    [a.name/].Actor_Output;
    TransactionsSocket<[a.name/].Actor>::Target [a.name/].Actor_Input;
    XMLTransactionReader xml_transaction_reader;
    XMLTransactionWriter xml_transaction_writer;
    SC_HAS_PROCESS([a.name/].Actor);
    [a.name/].Actor( sc_module_name module_name );
    ~[a.name/].Actor();
    void thread_process();
    virtual void b_transport(int tag, Transaction& trans, sc_time& delay);
};
#endif /* [a.name.toUpperCase()].ACTOR_H_ */
[/file]
```

using the XMLTransactionWriter object. Actor objects read input XML files named “/[a.name/].Actor_In.xml”. Actor objects write output XML files named “/[a.name/].Actor_Out.xml”. The actor architecture is shown in Figure 7.24.

The function thread_process of the actor classes is responsible for injecting the input transactions into the system. This function takes into account the attribute InitTime of the transactions to inject them at the correct time-stamp during the system execution. Additionally, the actor classes contain the function b_transport to store the transactions received from the system in the output XML file. This function sets the attribute EndTime of the transactions with the reception time-stamp.
7.3.3.2 DataType code generation

The Acceleo module `generate_datatype` produces a C++ class for each `DataType` element extracted from a SysML model, as depicted in Figure 7.25. The data type classes are called “[d.name/]”. Data type classes are described in two files: “[d.name/].h” and “[d.name/].cpp”.

Compositions of data types are supported, that is, previously implemented data type classes may be used to type the member variables or the member functions of a new data type class. In this case, data type classes shall include other data type classes needed to declare member variables and member functions. A public member variable is created in the data type class for each of the `Properties` of the SysML `DataType` elements. ASSYST supports the `Property` types defined in Table 7.3.
By default, a simple variable is generated. However, ASSYST supports the following member variable constructions:

- A C++ pointer is created if the aggregation attribute of the SysML Property element is equal to shared.

- A C++ array is created if the upper attribute of the SysML Property element is greater than 1. The upper attribute specifies how many elements are contained in the array.

A public member function is created in the data type class for each of the Operations of the SysML DataType elements. These member functions
contain a protected user code section in order to let the design engineers define manually the contents of the function.

### 7.3.3.3 Enumeration code generation

The SysML Enumeration elements are translated into C++ `enum` data types, as shown in Figure 7.26. This Acceleo module is straightforward as the enumeration literals are directly mapped into the C++ `enum` type.

### 7.3.3.4 Transaction code generation

The Signal elements from a SysML model are translated into C++ classes, as depicted in Figure 7.27. The generated transaction classes are called “[s.name/]” and they are derived from the base class `Transaction` of the ASSYST C++ library. Transaction classes are described in two files: the header file “[s.name/].h” and the source file “[s.name/].cpp”.

The specific transaction classes define additional attributes to the fundamental attributes (Id, InitTime and EndTime) defined for the base class `Transaction`. Moreover, getter and setter functions are implemented to handle the attributes defined for a particular transaction class “[s.name/]”.

### 7.3.3.5 UseCase code generation

A C++ class called “[u.name/]_UseCase” is produced by the module `generate_usecase` from each UseCase element (u : UseCase) of a SysML model, as shown in Figure 7.28. The use case class is derived from
7.3 Implementing SystemC-TLM Code Generation

Figure 7.27: Transaction Acceleo generation.

Figure 7.28: UseCase Acceleo generation.
the class UseCase of the ASSYST C++ library.

The class “[u.name]/_UseCase” creates a member variable for each FSM or SDF implemented. Furthermore, the C++ use case class contains two variables of type TransactionsSockets: a initiator socket internal_socket and a target socket process_socket. The use case class receives the input transaction through the target socket process_socket. On the other hand the initiator socket internal_socket routes the outgoing transactions outside the use case.

The function process of the use case class distributes the transactions received by the target socket process_socket to the FSM and SDF behaviours contained in the use case.

### 7.3.3.6 Block code generation

The SysML Block elements are translated into C++ classes, as shown in Figure 7.29. The Acceleo module generate_block produces a block class named “[b.name]/_Block”. This class produced by Acceleo is implemented in two source files, as specified in Figure 7.29.

Structural features are modelled with the classes “[b.name]/_Block”. These block classes are derived from the class Block of the ASSYST C++ library. The block classes contain member variables of type initiator TransactionsSocket in order to implement the input and output FlowPort elements of the SysML Block element being processed by Acceleo. The UseCase elements implemented by a Block are instantiated as member variables.
A SysML Block can define relationships with other Block parts using IBDs in order to describe a system hierarchy. The parent block C++ class instantiates child block classes as member variables.

The block classes are responsible for performing the following interface bindings. An example of these bindings is shown in Figure 7.30:

- **Port-to-UseCase** binding: the block classes bind the input initiator TransactionsSockets to the process_socket attributes of the owned UseCase objects. The transactions received in the input sockets of the block classes are then routed to all the UseCase elements.

- **UseCase-to-UseCase** binding: the UseCase elements owned by block classes are interconnected. The internal_socket of each UseCase is bound to the process_sockets of the other UseCases. These bindings allows the UseCases to send transactions to other UseCases.

- **Port-to-Port** binding: if a block hierarchy is modelled, the input and the output sockets of the parent Block may be bound to the output and the input sockets of its child Blocks, respectively. As a result, transactions can flow through the block hierarchy.

Finally, block classes contain member variables representing data. The generation pattern for the creation of the member variables follows the same pattern defined for DataType elements and their attributes in Table 7.3.
Furthermore, member functions are defined to handle the data variables of block classes. These member functions contain a protected user code section in order to let the design engineers define manually their contents.

The classes “[b.name/]_Block_Interface” provide an access interface to the corresponding classes “[b.name/]_Block”. The block interface classes implement the access interface through member functions.

### 7.3.3.7 Model code generation

The module `generate_model` translates SysML `Model` elements into several software artifacts, as described in Figure 7.31. `Models` are the root elements of the SysML models. A `Model` contain all the elements describing a SysML model under design.

The module `generate_model` creates the header file “`[m.name/]_h`” and the source file “`[m.name/]_cpp`” for each `Model` within a SysML model. These files contain the function `sc_main`, which is the top level function of a SystemC-based executable system description. As the `Model` element is unique within a SysML model, the function `sc_main` is guaran-
7.3 Implementing SystemC-TLM Code Generation

The actor classes and the block classes of the system are instantiated inside the `sc_main`.

The Acceleo module `generate_model` also creates a C++ class named `[m.name/]_TransactionFactory` for each model element `m : Model`. This C++ class derives from the interface class `TransactionFactory_if` provided by the ASSYST C++ library.

On one hand, the class `[m.name/]_TransactionFactory` implements the function `CreateTransaction`. The objective of this function is to define an interface between the input XML stimuli files and the C++ transaction objects. This function receives XML nodes (of type `TiXmlElement`) representing transactions from an input XML stimuli file and converts them into C++ transaction objects (of type `Transaction`) in order to be injected into the system. The attributes of the input XML nodes are mapped to the attributes of the C++ transaction objects. On the other hand, the class `[m.name/]_TransactionFactory` implements the function `CreateXmlElement`. This function aims to define an interface between the C++ transaction objects leaving the system and the XML nodes of the output XML files. This function receives C++ transaction objects and converts them into XML nodes representing outgoing transactions coming from the system. The classes `XMLTransactionReader` and `XMLTransactionWriter` instantiated in an actor receive as a construction parameter the transaction factory of the SysML model.

As shown in Figure 7.31, the module `generate_model` also produces the header files `[m.name/]Transactions.h` and `[m.name/]DataTypes.h` in order to include all the transactions and data types of the system, respectively.

7.3.4 ASSYST FSM BEHAVIOURAL CODE GENERATION

The source model for the FSM code generation process is the FSM model produced by ATL from SysML State Machine diagrams during the previous M2M transformation. The FSM model conforms to the FSM metamodel described in Section 7.2.1.1.
7.3.4.1 State code generation

The Acceleo module `generate_state` creates a C++ class for each State element contained within the FSM model, as depicted in Figure 7.32. The produced C++ class is called “[s.name/]State”. The state class derives from the class State class of the ASSYST C++ library.

The Acceleo module `generate_state` implements the entry, exit and do behaviour functions. These functions contain protected user code sections where the design engineer can implement the atomic functionality of the state after the code generation process.

A member variable is created in the C++ class for each of the output Transition elements of the FSM State elements. Additionally, the module `generate_state` implements the checking process of the output transitions.

7.3.4.2 Transition code generation

The code generation of the FSM Transition elements is performed by the Acceleo module `generate_transition`. As shown in Figure 7.33, each Transition (t : Transition) is translated into a C++ class called “[t.name/]Transition”. The produced class derives from the class Transition of the ASSYST C++ library.
The function `check` determines whether the transition is enabled during the runtime step of the FSM. This function examines the trigger of the transition whenever a Transaction is given as a parameter.

### 7.3.4.3 FSM code generation

The module `generate_fsm` translates FSMs into C++ classes. A FSM produces a C++ class called “[f.name/] FiniteStateMachine”, as depicted in Figure 7.34. The C++ class derives from the class `FiniteStateMachine` of the ASSYST C++ library.
The source model for the SDF code generation process is the SDF model produced by ATL from SysML Activity diagrams during the previous M2M transformation. The SDF model conforms to the SDF metamodel described in Section 7.2.1.2.

### Arc code generation

The module `generate_arc` produces a C++ class from an input SDF `Arc (a : Arc)`, as shown in Figure 7.35. The C++ class “Arc” contains a member variable `buffer` implemented as a queue container in order to store the tokens flowing through the arc. The variable `buffer` is specified as a First In First Out (FIFO) container.

The module `generate_arc` implements three member functions to handle the tokens within the arc. The function `get_production_rate` specifies how many tokens are produced in the source node and then sent to the arc. The function `get_consumption_rate` specifies how many tokens do the target node consume from the arc. The function `get_initial_tokens` indicates how many tokens are initially in the arc. Additionally, this Acceleo module creates the functions `write_token` and `read_token` in order to write tokens to `buffer` and read tokens from `buffer`, respectively.
7.3 Implementing SystemC-TLM Code Generation

7.3.5.2 Node code generation

The module `generate_node` translates SDF `Node` elements into C++ classes. A `Node` object `n` produces a C++ class called “`[n.name/]_Node`”, as depicted in Figure 7.36. The C++ class is derived from the class `Node`. The module `generate_node` implements member variables specifying both the incoming and the outgoing arcs of the SDF node.

The class “`[n.name/]_Node`” contains a member function `do_behavior`. This function implements both the reading of tokens from the incoming arcs and the writing of tokens into the outgoing arcs. Additionally, a protected user code section is specified within the function `do_behavior` to let the design engineers define the atomic functionality of the node. The protected user code section shall define how the input tokens are converted into the output tokens.

7.3.5.3 SDF code generation

The code generation of the SDFs is performed by the Acceleo module `generate_sdf`. As shown in Figure 7.37, each SDF (`s : SDF`) is translated into a C++ class called “`[s.name/]_SynchronousDataFlow`”. The specific SDF C++ classes are derived from the class `SynchronousDataFlow`. Each SDF C++ class is described in two files:

- The header file “`[s.name/]_SynchronousDataFlow.h`”.
- The source file “`[s.name/]_SynchronousDataFlow.cpp`”.
The Node and Arc elements contained in the SDF are instantiated as class member variables and then registered into the SDF scheduler.

7.4 CONCLUDING REMARKS

In recent years, MDD has been proposed as a methodology for electronic systems design. Specifically, the SysML visual modelling language offers a promising perspective for the efficient specification of complex electronic systems using MDD. On the other hand, SystemC–TLM has been proposed as a promising library to create executable code for early electronic system definition.

In this chapter, a MDD framework called ASSYST has been proposed to translate electronic systems described using SysML to SystemC–TLM heterogeneous executable models. ASSYST is a Eclipse plug-in based on M2M and M2T techniques using ATL and Acceleo. ASSYST implements both the formal semantic mapping described in Chapter 6 and the SystemC–TLM executable code generation processes. The clearly defined MoCs and the formal behavioural semantic mappings described in Chapter 6 effectively enables the application of MDD methodologies and the implementation of frameworks for electronic systems design.

ASSYST offers the capability to generate SystemC–TLM code describing both control-based and data-flow based behaviours. This heterogeneity improves the flexibility and the practical use of ASSYST within an electronic system design process.
7.4 Concluding Remarks

The automatic code generation is a key characteristic of ASSYST; the SystemC–TLM executable model can be created immediately. This characteristic enables to easily modify the architecture of the system model, re-organizing the system structure (SysML Blocks) or the behavioural aspects (SysML Use Cases, State Machines or Activities), and then to generate again the SystemC–TLM code in a very short time period. As a result, ASSYST improves the initial iterations carried out to tune the architecture of the high abstraction functional model and facilitates what-if analysis.

The global benefits of the proposed approach are clear: the error-prone manual tasks are greatly reduced, thus improving the design productivity.

This work focused on electronic systems design using SysML. However, the formalization of the models of computation and the automation of the code generation open new opportunities in other key areas such as, functional verification and system validation.

This chapter has been the precursor for the paper [148].
Chapter 8

Case study

Contents

8.1 Assessment methodology ........................................ 188
8.2 CONFIDENCE Project ............................................ 190
8.3 CONFIDENCE system development with SAVY .......... 192
8.4 CONFIDENCE system verification .............................. 202
8.5 CONFIDENCE system algorithmic design .................... 218
8.6 CONFIDENCE system development with ASSYST .......... 226
8.7 Concluding Remarks ............................................. 255

This chapter presents the assessment of the verification framework, the MATLAB framework, and the Automatic SysML to SystemC Translator (ASSYST) framework that supports a Systems Modeling Language (SysML) to SystemC–Transaction Level Modeling (TLM) development flow.

The CONFIDENCE project has been employed to validate the design and verification frameworks described in this thesis dissertation. The CONFIDENCE project, developed within the Seventh Framework Programme (FP7), is introduced in Section 8.2 and an extended description is provided in Appendix C. The main objective of the CONFIDENCE project, led and coordinated by Centro de Estudios e Investigaciones Técnicas de
Gipuzkoa (CEIT), is the development and integration of innovative technologies to develop a care system for the detection of abnormal events (such as falls) or unexpected behaviours that may be related to a health problem in elderly people.

8.1 ASSESSMENT METHODOLOGY

The assessment methodology takes as reference the development flow proposed by SAVY [75]. A description of the SAVY approach is provided in Appendix B. As depicted in Figure 8.1(a), the design team creates an Unified Modeling Language (UML) model describing the CONFIDENCE system. The UML model is then manually translated into a SystemC–TLM executable model using SAVY. Section 8.3 describes how the system–design team uses the SAVY library to develop TLM executable models from the UML description of the CONFIDENCE system.

In the next step, the verification framework proposed in Chapter 4 has been coupled to the reference executable models created previously using SAVY, as shown in Figure 8.1(b). Assertions have been specified for several abstraction level executable models and the performance of the verification framework has been measured. Section 8.4 presents the validation of the verification framework.

The validation of the MATLAB framework is presented in Figure 8.1(c). The MATLAB framework has been coupled to the SystemC–TLM executable models created previously during the assessment of the verification framework in Section 8.4. Section 8.5 presents the assessment of the MATLAB algorithmic framework proposed in Chapter 5. An algorithmic model of the CONFIDENCE system has been constructed to measure the performance of the MATLAB framework.

On the other hand, the CONFIDENCE system has been implemented using the ASSYST framework described in Chapter 7, as depicted in Figure 8.1(d). A SystemC–TLM executable model has been automatically generated using ASSYST from a SysML model describing the CONFIDENCE system. The ASSYST framework has been compared with the SAVY methodology analysing both the executable models and the development flows. Both the UML and the SysML models have been developed taking as input the same CONFIDENCE system requirements. The results of the comparison are presented in Section 8.6.
8.1 Assessment methodology

(a) SAVY Framework Assessment.

(b) Verification Framework Assessment.

(c) MATLAB Framework Assessment.

(d) ASSYST Framework Assessment.

Figure 8.1: Assessment methodology.
8.2 CONFIDENCE PROJECT

The main objective of the CONFIDENCE Project [149, 150] is to create a care system that will work both outdoors and indoors. This care system will be able to reconstruct the user’s posture and detect abnormal situations, such as falls or loss of consciousness. The propose system will raise an alarm if an abnormal situation is detected.

This care system will also be able to detect changes in the user’s behaviour and issue a warning. For instance, if the system notices changes in the user’s gait that may involve a lack of stability, the CONFIDENCE system will warn the user about an increased risk of falling, and prevent an accident. Detection of anomalous behaviour will utilize prior expert knowledge as well as learnt movement patterns of particular users.

The CONFIDENCE system consists of a central device, which plays the role of a Base Station (BS), a small Portable Device (PD), which looks similar to a mobile phone, and several tags, as it can be seen in Figure 8.2.

The BS will be placed inside the house and could be designed to look like a decorative item. It will be able to determine the position of each tag in the three dimensional (3D) space.

The PD will automatically enter in a setup mode and will guide the user (end-user or technician) during the installation process indicating where to
The Confidence system is intuitive, easy to setup, and user-friendly.

The user will wear small size and low cost tags. Additionally, some tags can be placed in specific positions such as the corners of the bed, chairs or some other pieces of furniture. Some tags on the furniture will make it possible to distinguish situations such as the user lying in bed or on the sofa. Other tags placed on the walls or furniture are also necessary for the localization subsystem.

The procedure followed during the installation of the tags is guided by a wizard that is setup automatically when the power of the system is turned on. The wizard indicates to the user, who will probably be a technician, where to place the tag. These tags are programmed by the system with a numeric identifier when the User has placed it on the body or on the furniture.

Indoors, the BS will be able to determine the position of each tag. Based on this information, the system will reconstruct the posture of the body and decide if the user has either suffered a fall or is acting abnormally. When a fall or an atypical situation is detected, the system raises an alarm.

When leaving the house, the user takes the PD with him/her, as shown in Figure 8.2(a). The BS will be able to locate the PD, thus, it will be able to notice that the user is leaving the house without PD. In this case, the system will detect this event and will warn him/her.

Both the BS and the PD can raise an alarm or a warning. The BS will have priority in indoor environments. Typically, the Confidence system will follow this alarm protocol: First, it makes a phone call to the user through the PD. If the user picks up the PD, he/she is requested to indicate that he/she feels well by pressing a button or telling a certain word. This stops the alarm. This way, the user keeps control of the system, which is an important feature and considerably reduces the false alarm rate. If the user does not pick up the phone, the system makes a phone call to a series of contacts, such as relatives, friends, caretakers or the emergency services. The system will explain to the call recipient the reasons of the alarm or warning.

Additionally, if the user stops the alarm, he or she shall be able to indicate the system that the alarmwarning was false pushing the “Normal
Acknowledge” button which means that it was not an alarm situation, but could have been. In this case, the system will send a warning to the user, just to make sure that the user does not panic and unwillingly cancel an actual alarm situation by mistake.

The user will be able to customize the alarm protocol and will select the time that the system waits for user to answer before triggering the alarm; a reasonable default value will be provided. Additionally, the user will be able to allow the system to transmit all the available information about the situation: localization, posture and how long the user has been in that posture.

At least two levels of alarm will be considered, i.e. alarm, warning. What to do within each level will be configurable by the user. The alarm receivers will be selected by the user.

The user, who in this case can be the technician following the orders of the user, inserts a list of call receivers in case of alarm or warning. He/she can customize the protocol, assigning priorities of each receiver. The receiver’s priorities are taken into account by the system which will call first the call receiver with highest priority. These priorities can be different for alarms and/or warnings. Moreover, the information related to the different call receivers is saved in the memory of the PD, allowing the user to activate or deactivate the different receivers.

8.3 CONFIDENCE SYSTEM DEVELOPMENT WITH SAVY

The methodology proposed by SAVY has being employed for the CONFIDENCE system development. This development is considered the reference for the assessment of the verification framework, the MATLAB framework and the proposed design framework.

During the first stage of a system design flow the system–design team creates the initial UML description taking as reference the functional requirements of the CONFIDENCE system as depicted in Figure 8.3. SAVY is used to easily create a SystemC–TLM executable model from the system behaviour capture carried out using UML Use Case diagrams and UML Statechart diagrams. SAVY enables to reuse models and to fully support a model driven design methodology where several system abstraction levels are described.
The employed methodology is an iterative process, where new design details are added in each iteration. The system–design team refines the UML model, creating new models as show in Figure 8.3. SystemC–based executable models are derived from UML models by means of the SAVY framework. The CONFIDENCE system models have been classified according to their abstraction level into Level 0, Level 1, and Level 2 models.

8.3.1 LEVEL 0 UML MODEL

The focus of the Level 0 model is to define the interfaces of the system with the external actors, user and Alarm Receiver (AR); define the transactions between the system and the environment; and define what the main functions of the system are. The inputs for the system–design team are the functional requirements of the CONFIDENCE system.
Figure 8.4: CONFIDENCE Level 0 Use Case Diagram.

The use case diagram of the Level 0 of CONFIDENCE system is shown in Figure 8.4. The system box represents the whole CONFIDENCE system and contains use cases, one for each top-level service that the system provides to its actors. **System Setup** use case describes how the system setup will be implemented. The **Analyze Situation** and **Call Alarm Receiver** use cases describes how the system will analyse the situation and will determine if an alarm or warning has occurred.

### 8.3.2 LEVEL 1 UML MODEL

In a second step of the design process, the system–design team refines the Level 0 model by dividing the system into functional devices. New requirements appear: the functions defined for each device; and the interfaces (transactions) between the devices. Both the actors and the transactions defined in the Level 0 model are reused. This design activity yields the Level 1 model. The Level 1 model of the CONFIDENCE system consists of the following functional devices: a central device, which plays the role of a BS; a small PD, which looks similar to a mobile phone; and several tags.

Figure 8.5 shows the architecture of the Level 1 system model which is composed by the three components BS, PD and Tag. The Tag component represents all the necessary human and environment tags of the system which are used to detect the posture of the User. The PD interacts with the User and the AR. This way, the User makes the set up of the system through the PD and the PD programs the Tag and sends the set up infor-
The design team treats the Tag, the PD, and the BS as subsystems and thus, a Use Case Diagram is specified for each of these three system components. Additionally, the behavioural aspects of the use cases defined in these Use Case Diagrams are described using Statechart Diagrams.

As an example, the Figure 8.6 shows the BS Use Case diagram. This device has four Use Cases: Analyse Situation, Call Alarm Receiver, Base Station Setup and Test Base Station, and it interacts with four actors: the Tag, the AR, the PD and the Developer.

The PD is responsible for the set up of the BS. This process is made in the Base Station Setup Use Case. When the PD indicates that the set up process has finished, the Analyse Situation Use Case starts the analysis of the situation. To do this operation, the Tag actor sends the corresponding signal. This way, the BS can calculate the 3D coordinates of the Tag in
order to determine the posture of the User. Moreover, the Analyse Situation Use Case decides if it is necessary to generate an alarm or not. This alarm is sent to the PD when the system works indoors. This is the task of the Call Alarm Receiver Use Case. Moreover, if the call operation of the PD does not work correctly and the system is indoors, the PD will inform to the BS in order to carry out the call operation.

### 8.3.3 LEVEL 2 UML MODEL

In a third step, each device is divided into its main functional modules. The resulting requirements are translated to a new executable model, the Level 2 model. In the Level 2 model, both the PD and BS are divided into four submodules: the localization (L) submodule, the reconstruction (R) submodule, the interpretation (I) submodule and system interface (SI) submodule. At this point, the system–design team has an architecture
The localization subsystem performs two tasks: identification and localization of the tags. It provides to the reconstruction subsystem the location of each tag. An indoor localization subsystem will be placed in the BS and an outdoor localization subsystem will be located in the PD.

The reconstruction subsystem receives the estimates of the locations of the tags and generates a model of the User and the environment. This model will be sent to the interpretation subsystem. An indoor reconstruction subsystem will be placed in the BS and an outdoor reconstruction subsystem will be located in the PD.

The interpretation subsystem interprets this data to make a decision about the situation. An indoor interpretation subsystem will be placed in the BS and an outdoor interpretation subsystem will be located in the PD.

The system interface subsystem is responsible for the user interface, system setup and alarm handling. The system interface subsystem of the PD will be able to work indoors and outdoors. However, the system interface subsystem of the BS will only work indoors.

8.3.4 UML TO SAVY TRANSLATION PROCEDURE

During the first stage of a system design flow the system–design team creates a UML description of the system to be developed. SAVY can be used to easily create a SystemC–TLM executable model from the system behavior capture carried out using UML Use Case diagrams and UML Statechart diagrams. SAVY enables to reuse models and to fully support a model driven design methodology where several system abstraction levels are described. The CONFIDENCE care system development allowed to assess the design methodologies proposed by SAVY within a complex electronic system design process.

Figure 8.7 illustrates the transformation of the UML diagrams into an executable model using the actor and entity architecture patterns of SAVY. In [151], a step–by–step translation procedure has been defined to easily go from a UML based system description to a TLM–SystemC executable model using design patterns and SAVY in a structured way. Let us assume the design team has described the system under design by means of use case
Figure 8.7: UML-to-SAVY transformation.
and statechart UML diagrams. The UML–to–SAVY translation procedure follows the next steps:

1. Identify UML actors and interactions in the UML use case diagrams.
   (a) For each UML use case, define the possible information exchanged between actors and entities as transactions. A new derived class is defined for each transaction.
   (b) Define the input/output interfaces of the entity: the Transactional Input Interfaces (TIIs) and the Transactional Output Interfaces (TOIs). For each TII, implement the two sc_threads. For each TOI, implement its sc_thread.

2. Identify the data that the system must store.
   (a) In SAVY, the data is stored in a container class (context) that must be defined.

3. Identify the control flow and actions in the UML statechart diagram.
   (a) For each state of each UML state diagram, extract the possible state transitions and actions. This is achieved by defining a new class derived from State and implementing its method ProcessEventState.
   (b) Declare the actions that the states can perform through the actuator_if.
   (c) For each UML use case, define a new SAVY Use Case module and specify the initial state. For each Use Case, define the events the Use Case is sensitive to. This is achieved by creating a sc_thread sensitive to each event.

4. Build the model of the entity.
   (a) Define the model of the entity which will be a sc_module. This sc_module will contain objects of the elements created in the former steps: TIIs, TOIs, Context and Use Cases. It will also contain an Event Communication Channel (ECC) object. Additionally, this sc_module will implement the actions of actuator_if.

5. Build the SystemC executable model.
   (a) Define transactors if needed.
(b) Create the model of the actors using the appropriate objects of the SAVY classes XML Transaction Generator (XTG), Traffic Injector (TI), XML Transaction Recorder (XTR) and Traffic Receiver (TR). Create an object of the entity. Bind the former objects.

6. Define the stimuli to be applied.

(a) For each actor that applies stimuli, write the eXtensible Markup Language (XML) file that describes the transaction injection sequence.

Once these steps are fulfilled, the model can be executed and analysed.

This procedure guides the designer to separate the definition of the actors, the entity to be designed and the information exchanged between actors and entity. Furthermore, the model of the entity is clearly divided into its input/output interface, control flow, data to be stored and actions to be performed. This design patterns result in models easy to understand and share between the different teams involved in a project using SAVY.

Level 0, Level 1 and Level 2 executable models based on SystemC have been manually created following the transformation steps described previously. The iterative refinement of the executable models is depicted in Figure 8.8.

8.3.5 EVALUATION OF THE SAVY METHODOLOGY

In order to evaluate the system design methodology using design patterns and SAVY, the cost of creating a TLM system model from an UML system description and the simulation performance of the executable model have been measured for each level of abstraction. The cost of creating a TLM system model from an UML description is estimated by means of the Lines of Code (LoC) written by the design team. The cost can be split in several parts according to the steps described for UML-to-SAVY translation procedure. Table 8.1 shows the LoC written to achieve the system Level 0 (L0), Level 1 (L1) and Level 2 (L2) models. As it can be observed, the LoC increase as the system model gets more detailed.

The LoC measurement greatly depends on the programming style of the design engineers and the formatting rules (if any) of the Integrated Devel-
Figure 8.8: CONFIDENCE system design flow.
Table 8.1: The LoC necessaries to create TL system models from the UML system descriptions.

<table>
<thead>
<tr>
<th>UML-to-SAVY translation steps</th>
<th>No Items</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L0</td>
<td>L1</td>
</tr>
<tr>
<td>Definition of transactions</td>
<td>31</td>
<td>75</td>
</tr>
<tr>
<td>Definition of entities I/O</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Definition of the context</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Definition of states</td>
<td>21</td>
<td>51</td>
</tr>
<tr>
<td>Declaration of actions</td>
<td>149</td>
<td>511</td>
</tr>
<tr>
<td>Definition of UCs</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Creation of the system model</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Creation of Transactors</td>
<td>4</td>
<td>13</td>
</tr>
<tr>
<td>Creation of the models of the actors, entity(ies) for system and binding</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TOTAL</td>
<td>212</td>
<td>677</td>
</tr>
</tbody>
</table>

opment Environment (IDE) used to write the source code. In order to solve this issue and give a standardized measurement, Eclipse IDE’s built-in automatic C++ code formatter has been used. The BSD/Allman indentation style has been applied to format all the source files with the same coding style and avoid any programmer-dependent LoC measurements. Additionally, blank lines have been ignored to provide a consistent LoC count in Table 8.1. In the following sections, this formatting technique is used for LoC determination.

8.4 CONFIDENCE SYSTEM VERIFICATION

The methodology explained in Section 8.3 has been employed for the development of the CONFIDENCE care system. The design team has used the SAVY library [75] to create the executable models that describe the system at the highest levels of abstraction. The executable models created have been classified according to the abstraction level of the functionality into Level 0, Level 1 and Level 2. The verification team has used the verification framework proposed in Chapter 4 to specify the assertions for each model.
8.4 CONFIDENCE system verification

The verification framework has been coupled to the CONFIDENCE system model, enabling early functional verification. The same input stimuli are reused for each level of abstraction. However, new monitors and assertions appear as the model increases its details. The proposed Verification Framework enables the verification designers to easily define the new system assertions and relate them to the new monitors.

8.4.1 LEVEL 0 VERIFICATION

In the first step of the employed methodology, the group responsible for the system design created the Level 0 model, while the verification group defined the first assertion set of the system. System designers and verification designers used the first system specifications as a reference. The focus of the Level 0 model was to define the interfaces of the system with the external actors: the user and the AR.

The verification framework is coupled to the CONFIDENCE Level 0 system model as shown in Figure 8.9. The coupling process follows the steps described in Section 4.2.2.

The first step is to replace the transactional ports supplied by SAVY with the Monitorizable Transactional Ports (MTPs) as shown in Listing 8.1. These modifications are carried out within the “ConfidenceLevel0.cpp” file which describes the CONFIDENCE Level 0 system model. Both the transactional input/output ports of the user and the alarm receiver actors are replaced. The Monitored C++ namespace defined by the verification library is added to regular transactional ports to transform them into monitored ones, for the transactional event monitorization purposes.

Figure 8.9: CONFIDENCE Level 0 verification.
The next step to integrate the verification framework and the Level 0 system model is to create an `AssertionsManager` object inside the system testbench code in order to enable all the verification framework capabilities. A `MonitorsCollection` object is set up registering all the MTPs to be monitored by the verification engine. The `MonitorsCollection` object is passed to the `AssertionsManager`'s constructor along with the XML file containing the assertions to be verified. The monitor creation and `AssertionManager` creation commands are presented in Listing 8.2. The `AssertionManager` is then prepared to record transactional events and verify the assertions during the system simulation. Once the SystemC simulation stops, the statistics of the verification process are obtained and saved into a XML report file.

### 8.4.1.1 Level 0 assertion specification

The Level 0 assertion list checks that the design team has correctly interpreted the specification of the user and AR interaction. An example of the type of properties checked is the following:

> *If the CONFIDENCE system makes a phone call to the user due to a possible abnormal situation and the user does not answer in 30 seconds, the system will inform to the alarm receiver.*

When the CONFIDENCE system detects an abnormal situation it sends the `InterpretationAlarm` transaction to the user. This transaction has two attributes: `AlarmType` describes whether the abnormal situation is an alarm or a warning, and `StatusAlarm` indicates whether an abnormal situation has been detected or not. The user sends the `UserStopProtocol`
Listing 8.2: Monitors and AssertionManager creation within `sc_main`.

```cpp
//Create monitors
MonitorsCollection monitors;
monitors.add((Monitor*)(&(confidenceLevel0->user_in)));
monitors.add((Monitor*)(&(confidenceLevel0->user_out)));
monitors.add((Monitor*)(&(confidenceLevel0->AR_in)));
monitors.add((Monitor*)(&(confidenceLevel0->AR_out)));

//Create AssertionsManager object
AssertionManager assertionsManager = new AssertionManager(monitors,
    "Assertions_L0.xml");
assertionsManager->bindAssertionsToMonitors();

//sc_start() SystemC simulation launch

//Recount Statistics
assertionsManager->recountStatistics();

//Save Statistics
assertionsManager->SaveStatistics("AssertionsReport",
    "AssertionsReport.xml");

//Release
delete assertionsManager;
```

transaction to the system to answer and stop the current system alarm or warning.

The CONFIDENCE system sends the SystemCall transaction to the alarm receiver to report him/her the detection of an abnormal situation. Similarly to the InterpretationAlarm transaction, the SystemCall transaction contains the AlarmType attribute to describe whether the abnormal situation is an alarm or a warning. Finally, the property is defined in the assertion description XML file as shown in Listing 8.3.

### 8.4.2 LEVEL 1 VERIFICATION

In a second step, the system was divided into functional devices. New requirements appeared for each device and the interfaces between the devices. The golden model was refined by the design team to take into account these new requirements. Additionally, the verification team added new assertions to the ones defined in the first step in order to cover the new model details. This way, the Level 1 model was verified against the
Listing 8.3: Confidence level 0 assertion example.

```xml
<?xml version="1.0"?>
<AssertionsManager xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="Assertions.xsd" Id="Confidence_Assertions">
    <Assertions DefaultMaxOverlappingInstances="20">
        <Assertion Id="Alarm_NoUserStop_AlarmRXCall" MaxOverlappingInstances="100">
            <Property>
                <Antecedent>
                    <Proposition Mode="POSITIVE" Id="1" Entity="confidenceLevel0_User_out"
                        Expression="((@1{InterpretationAlarm.AlarmType}=='ALARM') &&
                        (@1{InterpretationAlarm.StatusAlarm}==true))"/>
                    <Proposition Mode="NEGATIVE" Id="2" Entity="confidenceLevel0_User_in"
                        Operation="READ" Message="UserStopProtocol" StartTime="0" EndTime="30000"/>
                </Antecedent>
                <Consequence>
                    <Proposition Mode="POSITIVE" Id="3" Entity="confidenceLevel0_AR_out" Operation="WRITE"
                        Message="SystemCall" Expression="(@1{InterpretationAlarm.AlarmType} ==
                        @3{SystemCall.AlarmType})"/>
                </Consequence>
            </Property>
        </Assertion>
    </Assertions>
</AssertionsManager>
```
initial requirements and the new ones that take into account details of the implementation.

The Level 1 model of the CONFIDENCE system consists of the following functional devices: a central device, which plays the role of a BS; a small PD, which looks similar to a mobile phone; and several tags. The BS is placed at home and will only be working when the user is indoors. In this case, the three modules communicate among themselves sending and receiving transactions. The user only interacts with the PD; whereas the alarm receiver is connected both to the PD and BS.

The verification framework is coupled to the CONFIDENCE Level 1 system model as shown in Figure 8.10. The coupling process follows the steps described in Section 4.2.2. In this executable model, the Tag (class TagLevel1), the PD (class PDLevel1) and the BS (class BSLevel1) functional devices composing the system are created in the ‘sc_main.cpp’ source file as shown in Listing 8.4.

Both the transactional input and output ports of the functional devices are replaced with the monitored ones. The transactional ports of the BS functional device are presented in Listing 8.5 as an example. The BS func-
Listing 8.4: Functional devices creation within sc_main.

```cpp
// Tag functional device
TagLevel1* tagLevel1 = new TagLevel1("tagLevel1");

// Portable device functional device
PDLevel1* pdLevel1 = new PDLevel1("pdLevel1");

// Base station functional device
BSLevel1* bsLevel1 = new BSLevel1("bsLevel1");
```

Listing 8.5: Monitors creation within BSLevel1.

```cpp
//Tag
Monitored::TransactionalInterfaceIn<Transaction*> tag_in;
Monitored::TransactionalInterfaceOut<Transaction*> tag_out;

//AlarmRX
Monitored::TransactionalInterfaceIn<Transaction*> AR_in;
Monitored::TransactionalInterfaceOut<Transaction*> AR_out;

//Portable Device
Monitored::TransactionalInterfaceIn<Transaction*> pd_in;
Monitored::TransactionalInterfaceOut<Transaction*> pd_out;
```

Functional device is connected to the Tag functional device, the PD functional device and the alarm receiver actor.

Listing 8.6 shows how verification monitors are generated from the monitored transactional ports. Analogously to Level 0 executable model, a MonitorsCollection object is set up registering all the MTPs to be monitored by the verification engine.

The commands related to AssertionsManager creation, MTP registration in the verification engine, and verification statistics described in the Level 0 model verification in Section 8.4.1 remain unaltered.

### 8.4.2.1 Level 1 assertion specification

The Level 0 assertions are also used in the Level 1 model, to verify that the new more detailed Level 1 model is functionally equivalent to the Level 0 model. However, the new details require more assertions to verify the correct behaviour of the system.
8.4 CONFIDENCE system verification

Listing 8.6: Monitors and AssertionManager creation within sc_main.

```java
MonitorsCollection monitors;
monitors.add((Monitor)*(&(tagLevel1->tag_in)));
monitors.add((Monitor)*(&(tagLevel1->tag_out)));
monitors.add((Monitor)*(&(bsLevel1->tag_in)));
monitors.add((Monitor)*(&(bsLevel1->tag_out)));
monitors.add((Monitor)*(&(bsLevel1->AR_in)));
monitors.add((Monitor)*(&(bsLevel1->AR_out)));
monitors.add((Monitor)*(&(bsLevel1->pd_in)));
monitors.add((Monitor)*(&(bsLevel1->pd_out)));
monitors.add((Monitor)*(&(pdLevel1->user_in)));
monitors.add((Monitor)*(&(pdLevel1->user_out)));
monitors.add((Monitor)*(&(pdLevel1->tag_in)));
monitors.add((Monitor)*(&(pdLevel1->tag_out)));
monitors.add((Monitor)*(&(pdLevel1->AR_in)));
monitors.add((Monitor)*(&(pdLevel1->AR_out)));
monitors.add((Monitor)*(&(pdLevel1->bs_in)));
monitors.add((Monitor)*(&(pdLevel1->bs_out)));
```

Listing 8.7: CONFIDENCE level 1 IsIndoor global variable definition.

```xml
<GlobalVariables>
  <Variable Name="PDIndoor" Type="BOOLEAN" Value="false"/>
  ...
</GlobalVariables>
```

In this model, the BS sends a beacon every 5 seconds, so that the PD knows that both the user and the PD are indoors. There will be new properties to verify that the system detects correctly whether the user and PD are indoors or outdoors. These properties will set up the global variable IsIndoor to true if the system is indoors and to false if it is outdoors.

Listing 8.7 shows the IsIndoor definition in the preamble of the XML file containing the assertions.

Two assertions managing the value of IsIndoor are presented in Listing 8.8. The PDIndoor assertion states that whenever a BsPdIndoor transaction is received by the PD, the PDIndoor global variable is set to true. The PD_Outdoor assertion specifies that if a BsPdIndoor transaction is received by the PD and no other BsPdIndoor transaction is received in
5 seconds, then the PDIndoor global variable is set to false. There is a possibility where no BsPdIndoor transaction is received by the PD during the system simulation. In this case, BsPdIndoor remains in false state, that is, the default value set in Listing 8.7.

Additionally, when the user is indoors, some redundancy is added to the call to the alarm receiver. First, the PD will call to the alarm receiver using a wireless network. However, if the wireless communication system is not working, the BS will try to call to the alarm receiver using a fixed line. The system property is quoted below:

*If the PD makes a phone call to the alarm receiver due to a possible abnormal situation and the alarm receiver does not answer in 30 seconds while the user is indoors, then the BS will inform to the alarm receiver.*

The property is defined in the assertion description XML file as shown in Listing 8.9.

### 8.4.3 LEVEL 2 VERIFICATION

In a third step, each device was divided into several modules. The new requirements were translated both to a new executable model, the Level 2 model, and to new assertions. The Level 2 model was verified against the Level 0, Level 1 and Level 2 assertions.

Figure 8.11 shows the Level 2 block diagram of the CONFIDENCE system. Both the PD and BS are divided into four submodules: the localization (L) submodule, the reconstruction (R) submodule, the interpretation (I) submodule and system interface (SI) submodule. At this level, the number of transactions increases in order to propagate the information among the different modules of the system.

The functional devices composing the Level 2 system are created in the ‘sc_main.cpp’ source file as shown in Listing 8.10. In this executable model, the localization, reconstruction, interpretation and system interface submodules for both the PD and the BS are new. However, the Tag (class TagLevel1) module from Level 1 has been completely reused. All the functional devices describe MTPs that comply with the interfaces depicted in Figure 8.11.
Listing 8.8: CONFIDENCE level 1 assertions managing IsIndoor.

```
<Assertion Id="PD_Indoor" MaxOverlappingInstances="100">
  <Property>
    <Antecedent>
      <Proposition Imp="NON_OVERLAPPING" Mode="POSITIVE" Id="1"
        Entity="portabledeviceLevel1_Basestation_in" Operation="READ" Message="BsPdIndoor"/>
    </Antecedent>
    <Consequence>
      <Proposition Mode="POSITIVE" Id="2" Expression="(G{PDIndoor} = true)"/>
    </Consequence>
  </Property>
</Assertion>

<Assertion Id="PD_Outdoor" MaxOverlappingInstances="100">
  <Property>
    <Antecedent>
      <Proposition Imp="NON_OVERLAPPING" Mode="POSITIVE" Id="1"
        Entity="portabledeviceLevel1_Basestation_in" Operation="READ" Message="BsPdIndoor"/>
      <Proposition Imp="NON_OVERLAPPING" Mode="NEGATIVE" Id="2"
        Entity="portabledeviceLevel1_Basestation_in" Operation="READ" Message="BsPdIndoor" StartTime="0" EndTime="5"/>
    </Antecedent>
    <Consequence>
      <Proposition Mode="POSITIVE" Id="3" Expression="(G{PDIndoor} = false)"/>
    </Consequence>
  </Property>
</Assertion>
```
Listing 8.9: CONFIDENCE level 1 assertion example.

```xml
<Assertion Id="PD_AlarmCall_NoAlarmRXStop BS_AlarmCall" MaxOverlappingInstances="100">
  <Property>
    <Antecedent>
      <Proposition Imp="OVERLAPING" Mode="POSITIVE" Id="1" Entity="pdLevel1_AR_out" Operation="WRITE"
        Message="SystemCall" Expression="(G(PDIndoor) == true)="/>
      <Proposition Mode="NEGATIVE" Id="2" Entity="pdLevel1_AR_in" Operation="READ"
        Message="UserStopProtocol" Expression="(G(PDIndoor) == true)" StartTime="0" EndTime="30000"/>
    </Antecedent>
    <Consequence>
      <Proposition Mode="POSITIVE" Id="3" Entity="bsLevel1_AR_out" Operation="WRITE"
        Message="SystemCallTransaction" Expression="[@1,SystemCall.AlarmType] == @3(SystemCall.AlarmType)">/
    </Consequence>
  </Property>
</Assertion>
```
Figure 8.11: CONFIDENCE Level 2 verification.
Listing 8.10: Monitors and AssertionManager creation within `sc_main`.

```cpp
TagLevel1* tagLevel1 = new TagLevel1("tagLevel1");
BSLocalisationLevel2* bslLevel2 = new
    BSLocalisationLevel2("bslLevel2");
BSReconstructionLevel2* bsrLevel2 = new
    BSReconstructionLevel2("bsrLevel2");
BSInterpretationLevel2* bsiLevel2 = new
    BSInterpretationLevel2("bsiLevel2");
BSSystemInterfaceLevel2* bssiLevel2 = new
    BSSystemInterfaceLevel2("bssiLevel2");
PDLocalisationLevel2* pdlLevel2 = new
    PDLocalisationLevel2("pdlLevel2");
PDRecostructionLevel2* pdrLevel2 = new
    PDRecostructionLevel2("pdrLevel2");
PDIInterpretationLevel2* pdiLevel2 = new
    PDIInterpretationLevel2("pdiLevel2");
PDSystemInterfaceLevel2* pdsiLevel2 = new
    PDSystemInterfaceLevel2("pdsiLevel2");
```

As well as in Level 1 executable model, the commands related to the assertions manager creation, MTP registration, and verification statistics described in the Level 0 model verification are reused.

8.4.3.1 Level 2 assertion specification

In the Level 2 model, the communications between PD and AR and between BS and AR are handled by the PD System Interface (PD–SI) submodule and the BS System Interface (BS–SI) submodule respectively. The PD Interpretation (PD–I) module and the BS Interpretation (BS–I) should detect abnormal situations, so they need to know when an alarm has been unattended by the alarm receivers. Thus, new assertions appear.

If the PD–SI makes a phone call to the alarm receiver due to a possible abnormal situation and the alarm receiver does not answer in 30 seconds while the user is indoors, then the PD–SI will inform to the PD–I.

If the BS–SI makes a phone call to the alarm receiver due to a possible abnormal situation and the alarm receiver does not answer in 30 seconds while the user is indoors, then the BS–SI will inform to the BS–I.
8.4 CONFIDENCE system verification

Table 8.2: Verification Framework Integration Cost.

<table>
<thead>
<tr>
<th>Abstraction Level</th>
<th>Level 0</th>
<th>Level 1</th>
<th>Level 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SystemC Model</td>
<td>5,691 lines</td>
<td>14,807 lines</td>
<td>16,442 lines</td>
</tr>
<tr>
<td>Verification Integration</td>
<td>20 lines</td>
<td>43 lines</td>
<td>76 lines</td>
</tr>
</tbody>
</table>

Both properties are defined respectively in Listing 8.11 and Listing 8.12.

Note that both assertions make use of the global variable \texttt{PDIndoor}, which is controlled by the Level 1 assertions.

The presented example shows how assertions can be reused among different levels of abstraction models. Additionally, we have shown how the verification designer adds new assertions to take into account the new details of each level. These steps can be further repeated taking into account more implementation details. The propositions presented are an example of how simple is to translate the natural language specification to assertions for the proposed Verification Framework.

8.4.4 EVALUATION OF THE VERIFICATION FRAMEWORK

In order to assess the verification framework, two different executable models have been created at each level of abstraction: one with verification capabilities; and another one without verification capabilities (the verification framework is not coupled to the design model).

A key point of the proposed verification framework is its simplicity to be coupled to a SystemC–TLM system model. The lines of code for the system models at the three abstraction levels are shown in Table 8.2. It can be observed that few lines of code are necessary to integrate the proposed verification framework to the system model.

On the other hand, the Central Processing Unit (CPU) execution time is measured to compare the performance of the system models with verification capabilities and the models without verification capabilities. For each level of abstraction, both models have been simulated in scenarios that represent more than two hours of work of the system. The three levels of abstraction use the same input stimuli. The input stimuli has been designed to create more than 9,000 different situations where more than
Listing 8.11: CONFIDENCE level 2 assertion example.

```
<Assertion Id="PD_SI_Alarm_NoAlarmRXStop" MaxOverlappingInstances="100">
  <Property>
    <Antecedent>
      <Proposition Imp="OVERLAPING" Mode="POSITIVE" Id="1">
        Entity="portabledeviceLevel1_interpretationInChannel" Operation="READ"
        Message="InterpretationAlarmTransaction"
        Expression="(@1{InterpretationAlarmTransaction.AlarmType} == 'ALARM') &&
        (@1{InterpretationAlarmTransaction.StatusAlarm} == true) && (G{PDIndoor} == true)"
      </Proposition>
      <Proposition Mode="NEGATIVE" Id="2" Entity="portabledeviceLevel1_User_in" Operation="READ"
        Message="UserStopProtocolTransaction" Expression="(G{PDIndoor} == true)" StartTime="0"
        EndTime="30000"/>
    </Antecedent>
    <Consequence>
      <Proposition Mode="POSITIVE" Id="3" Entity="portabledeviceLevel1_AR_out" Operation="WRITE"
        Message="SystemCallTransaction" Expression="(@1{InterpretationAlarmTransaction.AlarmType} ==
        @3{SystemCallTransaction.AlarmType})"/>
    </Consequence>
  </Property>
</Assertion>
```
Listing 8.12: **CONFIDENCE level 2 assertion example.**

```xml
<Assertion Id="BS_SI_Alarm_NoAlarmRXStop" MaxOverlappingInstances="100">
  <Property>
    <Antecedent>
      <Proposition Imp="OVERLAPING" Mode="POSITIVE" Id="1">
        Entity="portabledeviceLevel1_interpretationInChannel" Operation="READ"
        Message="InterpretationAlarmTransaction"
        Expression="(@1{InterpretationAlarmTransaction.AlarmType} == 'ALARM') &&
        (@1{InterpretationAlarmTransaction.StatusAlarm} == true) &&
        (G{PDIndoor} == true)"/>
      </Proposition>
      <Proposition Mode="NEGATIVE" Id="2" Entity="portabledeviceLevel1_User_in" Operation="READ">
        Message="UserStopProtocolTransaction" Expression="(G{PDIndoor} == true)" StartTime="0"
        EndTime="30000"/>
    </Antecedent>
    <Consequence>
      <Proposition Mode="POSITIVE" Id="3" Entity="portabledeviceLevel1_AR_out" Operation="WRITE">
        Message="SystemCallTransaction" Expression="(@1{InterpretationAlarmTransaction.AlarmType} ==
        @3{SystemCallTransaction.AlarmType})"/>
    </Consequence>
  </Property>
</Assertion>
```
Table 8.3: Verification Framework Performance.

<table>
<thead>
<tr>
<th>Abstraction Level</th>
<th>Level 0</th>
<th>Level 1</th>
<th>Level 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verification Disabled</td>
<td>1.56 s</td>
<td>10.01 s</td>
<td>11.7 s</td>
</tr>
<tr>
<td>Verification Enabled</td>
<td>2.7 s</td>
<td>21.9 s</td>
<td>27.84 s</td>
</tr>
<tr>
<td>Assertion Evaluations</td>
<td>1,395</td>
<td>846,735</td>
<td>1,032,010</td>
</tr>
<tr>
<td>Verification Overhead</td>
<td>42.2%</td>
<td>54.3%</td>
<td>57.9%</td>
</tr>
</tbody>
</table>

100 alarms need to be handled. This simulation scenario involves almost 19,000 input transactions.

Table 8.3 shows the CPU simulation time for the executable models at the three levels of abstraction. The executable models have been compiled using GNU/g++ v3.4.6 in a PC with a CPU Intel Xeon E5405 @ 2.00 GHz, running Linux Red Hat Enterprise. We can observe that the simulation time increases with the detail of the models. Additionally, the next-to-last row of Table 8.3 presents the number of assertion evaluations within the system model with verification capabilities. The number of assertion evaluations represents the activity of the verification framework.

The performance loss due to the verification framework depends on the number of assertion evaluations. In turn, the number of evaluations depends on the number of assertions defined for the system model. The performance loss increases slightly as the system model gets more detailed. This execution time increment of the verification process is due to the new assertions added in order to verify the new system details. Despite more than one million assertion evaluations are performed in the Level 2 system model, the verification process represents only 58% of the execution time.

8.5 CONFIDENCE SYSTEM ALGORITHMIC DESIGN

As described in Section 8.4, SAVY has been employed for the Confidence system development. SAVY library enables to develop TLM executable models with a high level of abstraction. The executable models have been classified according to their abstraction level into Level 0, Level 1, and Level 2 models. The verification team has used the verification framework described in Chapter 4 to apply Assertion-based verification (ABV) techniques to the executable models. The verification team has refined the
assertions iteratively, taking into account the new design details of successive executable models.

At this point, the system–design team has an architecture proposal, with the functionalities and transactions between the modules specified: *what* the system modules shall do has been defined. As a result, the algorithm–design team can develop the algorithms (the *how*) for the different functionalities. The result of the activity of the algorithm–design team are executable specifications commonly developed in MATLAB as M–files. The system–design team can now insert the algorithm execution into selected Level 2 functional modules for the verification of their correct cooperation to achieve the high–level functionalities. The *MatlabEngine++* library proposed in Chapter 5 can help to achieve this SystemC–MATLAB interaction in an easy and efficient way. As an example, in this case study, the tag module and the BS Localization (BS–L) submodule are replaced by their algorithmic models to yield a Level 3 model. The resulting model is illustrated in Figure 8.12.

The MATLAB model of the tag module simulates an Impulse–Radio (IR) Ultra-Wideband (UWB) system, where sequences of UWB Gaussian pulses are transmitted among the tags over an Additive White Gaussian Noise (AWGN) channel. A tag performs cross–correlations on the received signal in order to detect the sequences transmitted by other tags. Then, the tag transmits a response to the other tags. In each tag, the delay between the transmitted and the received signal is used to estimate the distances between the tags. This Round–Trip–Time (RTT) ranging algorithm is described in [152]. The IR–UWB system is simulated with a sampling frequency of 8GHz. The set of estimated distances is the transaction between the tag module and BS–L. The BS–L shall estimate the tags coordinates. The MATLAB model implements a Recursive Least Square (RLS) algorithm [153].

### 8.5.1 LEVEL 3 ALGORITHMIC DESIGN

The system–design team insert the MATLAB algorithm execution into the Tag and BS–L Level 2 functional modules following the operation steps presented in Section 5.2.2. The MATLAB simulation framework is set up in three different modules: the SystemC *sc_main* routine, the Tag module and the BS–L module.
Figure 8.12: CONFIDENCE Level 3 algorithmic executable model.
The SystemC sc_main routine manages the MATLAB session and its configuration as presented in Listing 8.13. In the level 3 model, a unique MATLAB session has been used to run both the Tag and the BS–L algorithms. A MATLAB session called “ConfidenceSession” has been created using the MatlabManager. The configuration of the “ConfidenceSession” MATLAB session has been customized to meet the requirements of the host machine where the algorithmic simulations are executed. In this case, a host named rita has been configured to use MATLAB version 7.3 with the proper license path. The configured MATLAB engine is opened before starting the SystemC simulation calling the sc_start() function, and it is closed after the simulation is finished. Additionally, the SystemC sc_main routine configures the global parameters for the algorithmic simulation as shown in Listing 8.14. The parameters of the simulation are set up after the MATLAB engine is opened but before the SystemC simulation is launched.

First, the parameters are created within the configured MATLAB session (session variable). To get a flexible simulation platform and avoid a re-
Listing 8.14: Set parameters in MATLAB for Tag and BS operation.

```matlab
//Set parameters in Matlab for Tag and BS operation
MatlabVar c_snr_db("snr_db", session);
MatlabVar c_t_pulse("t_pulse", session);
MatlabVar c_t_symbol("t_symbol", session);
MatlabVar c_pulse_type("pulse_type", session);
MatlabVar c_ppm_tag("ppm_tag", session);
MatlabVar c_ppm_sensor("ppm_sensor", session);
MatlabVar c_human_speed("human_speed", session);
MatlabVar c_room_x_dim("room_x_dim", session);
MatlabVar c_room_y_dim("room_y_dim", session);
MatlabVar c_room_z_dim("room_z_dim", session);
MatlabVar c_sep_sensors_xy("sep_sensors_xy", session);
MatlabVar c_sep_sensors_z("sep_sensors_z", session);

//Assign values to parameters
session->EvalString("configure_pos_est_AWGN;");
```

compilation of the executable model, the values of the MATLAB parameters are not hard coded into the `sc_main` routine. A XML file has been prepared to contain all the configuration related values. Every time a simulation is launched, this XML file is read and its values are assigned to the global parameters defined in Listing 8.14.

The `configure_pos_est_AWGN` is a MATLAB M–script that is executed after the global parameters are properly configured. This script processes all the parameters configured previously and sets up the MATLAB algorithmic simulation platform, such us, the sampling frequency, UWB pulse width, Signal-to-noise ratio (SNR), signal preamble type, etc.

During the system simulation the Tag and the BS–L modules simulate ranging and localization algorithms, respectively. Both modules execute their algorithms within the “ConfidenceSession” MATLAB session.

The Tag manages the MATLAB session and the MATLAB parameters within its process execution loop as described in Listing 8.15. First of all, the Tag module needs to obtain a pointer of the MATLAB session previously configured in the `sc_main` routine. Then, all the necessary parameters are created in the “ConfidenceSession” session. The Tag defines the input parameters `xtag`, `ytag` and `ztag` representing its real location coordinates.
8.5 CONFIDENCE system algorithmic design

Listing 8.15: Ranging parameters in Tag.

MatlabManager* m = MatlabManager::GetHandler();
MatlabSession* session = m->GetSession("ConfidenceSession");

//Input parameters for Matlab
MatlabVar c_xtag("xtag", session);
MatlabVar c_ytag("ytag", session);
MatlabVar c_ztag("ztag", session);

//Output parameters for Matlab
MatlabVar c_distance1_est("distance1_est", session);
MatlabVar c_distance2_est("distance2_est", session);
MatlabVar c_distance3_est("distance3_est", session);
MatlabVar c_distance4_est("distance4_est", session);

Listing 8.16: Ranging algorithm in Tag.

session->EvalString("[distance1_est, distance2_est, distance3_est, distance4_est] = dist_cal_AWGN(xtag, ytag, ztag, Fs, SNR_db, multipath, Tp, Tsym, Tsym_mono, pulse_type, ppm_t, ppm_s, Cap_reac_hum, pos1, pos2, pos3, pos4, room_x_dim, room_y_dim, room_z_dim, sensors_position, symbols_before_code, std_preamble, preamble, code_to_sync, tag_ID, tproces, study_interval, sep_sensors_xy, sep_sensors_z);");

The four output parameters (distance1_est to distance4_est) represent the estimated distances from each of the four anchors to the Tag.

During the system simulation the Tag module performs EvalString calls to the dist_cal_AWGN algorithm script from their SystemC threads, as presented in Listing 8.16. In this case, the Tag implements a ranging algorithm that processes its (x, y, z) coordinates to obtain the distances to each of the four anchors. The calculated distances are then sent within a transaction to the BS–L or PD Localization (PD–L) modules for further processing, depending on whether the user is indoor or outdoor.

Similarly to Tag, the BS–L handles the “ConfidenceSession” session and defines all its necessary input and output parameters, as presented in Listing 8.17.

The BS–L implements a localization algorithm that estimates the tag location based on its estimated distances to each of the four anchors. During
the system simulation, the BS–L module performs EvalString calls to the pos_cal_RLS_3D algorithm script, as presented in Listing 8.18. This script calculates the estimated location of the Tag. The BS–L module sends the calculated Tag location within a transaction to the BS Reconstruction (BS–R) reconstruction module.

### 8.5.2 EVALUATION OF THE MATLAB FRAMEWORK

In order to measure the benefits of the proposed framework to integrate MATLAB algorithms and SystemC–TLM models, we pay attention to two issues: the cost of including the selected algorithm execution in the Level 2 system model to yield the Level 3 system model; and the simulation performance.

The cost of building the Level 3 system model can be split in two parts: cost of modifying the MATLAB executable specification; and cost of modifying the SystemC Level 2 system model. The algorithm–design team developed their algorithms with a clear specification of the required functionality and input/output arguments. This produced algorithm executable specifi-
Table 8.4: Cost of Modifying SystemC Level 2 Model.

<table>
<thead>
<tr>
<th>Level 2 System component</th>
<th>MatlabEngine++ LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>BS–L</td>
<td>12</td>
</tr>
<tr>
<td>Tag</td>
<td>11</td>
</tr>
<tr>
<td>sc_main</td>
<td>25</td>
</tr>
<tr>
<td>Total</td>
<td>48</td>
</tr>
</tbody>
</table>

Table 8.5: Level 3 System Model Performance Analysis.

<table>
<thead>
<tr>
<th>Level 3 System component</th>
<th>Execution Time</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>SystemC Model (SAVY platform)</td>
<td>2.89 s</td>
<td>0.28%</td>
</tr>
<tr>
<td>Matlab algorithms</td>
<td>1,047.5 s</td>
<td>97.76%</td>
</tr>
<tr>
<td>MatlabEngine++ interaction</td>
<td>21.1 s</td>
<td>1.96%</td>
</tr>
<tr>
<td>Total</td>
<td>1,071.49 s</td>
<td>100%</td>
</tr>
</tbody>
</table>

cations that needed no change for their use in the Level 3 system model. Therefore, the cost of modifying the MATLAB executable specification was zero.

Table 8.4 illustrates the cost of modifying the SystemC Level 2 model. The table shows the number of code lines added by the system-design team in the SystemC Level 2 model to yield the SystemC Level 3 model. As it can be observed, the cost of modifying the SystemC model was very small, despite the complexity of the algorithms included in this case study.

The proposed approach achieves system models that enable verification of correct algorithm cooperation with a very small cost. The reduced cost of building these models enables early high-level design exploration, where different architectures and algorithms can be analysed to optimize the system.

Table 8.5 analyses the simulation performance. The data in the table corresponds to a simulation where the BS–L performs more than 600 coordinate estimations. For each coordinate estimation, the tags simulate the IR–UWB system. The first row is the time consumed by the SystemC–TLM model. The second row is the time consumed by the MATLAB engine executing the algorithms. The third row is the time consumed in the SystemC–MATLAB interaction. It can be seen that the overhead due
226 Chapter 8: Case study

to the SystemC–MATLAB interaction is very small, despite the amount of data/command transfers between the SystemC and the MATLAB domains.

8.6 CONFIDENCE SYSTEM DEVELOPMENT WITH ASSYST

The system that is being developed within the CONFIDENCE Project [149] is presented as a case study for the ASSYST framework.

The CONFIDENCE system development process is divided into two steps.

In the first step the design team creates a SysML model describing the CONFIDENCE system. The behavioural modelling has been carried out using Use Case diagrams, State Machine diagrams and Activity diagrams. On the other hand, the structural modelling has been carried out using Block Definition Diagrams (BDDs).

The second step involves the proposed ASSYST framework. ASSYST has been used to automatically translate the CONFIDENCE SysML model into a SystemC–TLM heterogeneous executable code using the ATL Transformation Language (ATL) Model-to-Model (M2M) tools and the Acceleo Model-to-Text (M2T) generation workflow described in Chapter 7.

8.6.1 CONFIDENCE SYSML MODEL

The design team uses the Eclipse IDE and the Topcased toolkit to create the SysML model that describes the CONFIDENCE Level 0 model. This SysML model describes both the structural and behavioural features of the care system at the highest level of abstraction. The SysML model is created following the ASSYST modelling methodology described in Section 6.5.

In the first step, the design team creates a general Topcased project called CONFIDENCE Level 0 and a folder called Models. This folder will store the SysML model describing the system.

A new model is created using the SysML wizard from the Topcased toolkit, as shown in Figure 8.13. An empty SysML model called CONFIDENCE Level 0 is created with a default Use Case diagram.

The SysML model creation wizard produces two new files under the Models folder, as shown in the outline view in Figure 8.14. The SysML
8.6 CONFIDENCE system development with ASSYST

(a) New SysML model.

(b) New SysML model options.

Figure 8.13: SysML model creation.
model data is stored in the file “Confidence_Level_0.sysml”. The file “Confidence_Level_0.sysml” stores the properties of the diagrams and the graphical elements of the SysML model. “Confidence_Level_0.sysml” has references to “Confidence_Level_0.sysml”. Opening “Confidence_Level_0.sysml” brings up a graphical model–based editor where the design engineer can develop the CONFIDENCE Level 0 model.

For better readability and flexibility for future model extensions, the Confidence_Level_0 SysML model has been organized by means of Package elements, as presented in Figure 8.15.
8.6 CONFIDENCE system development with ASSYST

8.6.2 MODELLING FUNCTIONALITY WITH USE CASES

Use Case Diagrams describe the functionality of a system in relation to how the users use that system. The use case diagram of the Level 0 of CONFIDENCE system is shown in Figure 8.16. The system box represents the whole CONFIDENCE system. Four use cases and two actors are identified in the CONFIDENCE system.

8.6.2.1 User Actor

The User actor can be the end-user or the technician. He/She interacts with the system through an interface with the following use cases:

- *SystemSetup* use case: the User is the responsible for the set up of the System.
- *CallAlarmReceiver* use case: the User can stop the protocol if a false alarm raise.

- *ProcessReconstruction* use case reconstructs the posture of the user.

- *ProcessSituation* use case analyses the posture of the user.

### 8.6.2.2 AlarmReceiver Actor

He/She is going to interact with the System through an interface with the following use case:

- *CallAlarmReceiver* use case: the System will call the corresponding AR if an alarm/warning occurs. These ARs will stop the call protocol, in case he/she will be called.

### 8.6.2.3 SystemSetup Use Case

*SystemSetup* use case describes how the system set up will be implemented. The *User* will install the different tags for the correct operation of the system. Also, he/she will insert the receiver’s information which is required for the call procedure. In addition, the *User* can configure the call procedure. In this case, the *User* can be a technician who is responsible for the correct configuration of the system and for educating the end-user in order to learn the CONFIDENCE system working.

### 8.6.2.4 ProcessReconstruction Use Case

The *ProcessReconstruction* use case reconstructs the posture of the user.

### 8.6.2.5 ProcessSituation Use Case

The *ProcessSituation* use case analyses the reconstructed posture and determines whether there is an alarm situation, a warning situation or a normal situation. In case of alarm or warning, the system enters in the alarm or warning mode, the detected situation is saved and the system begins with the call procedure.
8.6.2.6 CallAlarmReceiver Use Case

The CallAlarmReceiver use case describes the call protocol. In case of an emergency, the system will determine if an alarm or warning has occurred. If it is so, the System will call the corresponding AR. It will follow the call protocol configured by the User in the set up of the System.

8.6.2.7 Elaborating Use Cases with Behaviours

The use case models presented previously, describe the functionality of a system. The detailed functionality of each use case has been described using either a State Machine diagram or an Activity diagram, depending on the modelling requirements of the use case. Use cases describing control-based behaviours have been modelled by means of State Machine diagrams, while data flow oriented behaviours have been described using Activity diagrams. SystemSetup and CallAlarmReceiver use cases have been modelled as State Machines while ProcessReconstruction and ProcessSituation use case have been modelled as Activities.

8.6.3 MODELLING EVENT–BASED BEHAVIOUR WITH STATE MACHINES

State Machine Diagrams define a set of concepts that can be used to model discrete behaviour through finite state transition systems. The state machine represents a event-driven behaviour. State Machines typically describe the behaviour of blocks and external events received by blocks control the State Machine execution. A state machine models discrete event-based behaviour by means of states and transitions connecting states.

8.6.3.1 SystemSetup State Machine Diagram

The SystemSetup state machine is created as the classifier behaviour of the SystemSetup use case as shown in Figure 8.17. The SystemSetup state machine describes how the system set up will be implemented.

At the beginning, the System is in the Initial state and the User can start with the installation of the tags, configure the call protocol or configure the CONFIDENCE alarm scenarios. When the User sends:
The *UserRequestInstallMenu* transaction, the tags installation procedure is started.

The *UserRequestCPMenu* transaction, the call protocol configuration procedure is started.

The *UserSetLyingScenarios* transaction, the *UserInhibitAlarm* transaction or the *UserReactivateAlarm* transaction, the alarm scenarios are configured.

The first set up of the system runs a wizard in tag installation and call protocol configuration cases. This wizard helps the User, in this case a technician, during the installation in order to avoid errors. That is, the User is guided during the set up process. The User selects the wizard that he/she would like to start with, and when this wizard finishes, the other one will begin automatically.

As the *SystemSetup* state machine is complex, it has been divided into three views for better understanding. That is, the *SystemSetup* state machine is unique but its information has been split into three diagrams: *SystemSetup TagInstallation* diagram, *SystemSetup CallProtocolConfiguration* diagram and *SystemSetup ConfigureAlarms* diagram. Note that the Initial state that appears in those diagrams is the same.
SystemSetup TagInstallation diagram

The TagInstallation diagram of the SystemSetup state machine is shown in Figure 8.18.

In the Initial state the User has the option to select the installation of the tags or the configuration of the call protocol. If he/she selects Tag Installation (UserRequestInstallMenu), the System goes to the Tag Installation State. The internal states of Tag Installation State are shown in Figure 8.18. If he/she selects the install wizard (UserRequestInstallMenuWizard), the state machine goes to InstallHumanWizard state.

InstallHumanWizard state: In the InstallHumanWizard state, a wizard starts automatically requesting to the User to install the tags one by one. Once the User inserts a tag, the System receives the UserSendParam transaction and the System goes to the CheckHumanTag state.

CheckHumanTag state: In the CheckHumanTag state, it is checked that the status of the tag is active. Otherwise, it is re-requested a number of times fixed by the System configuration. If these conditions are satisfied, the tag is saved in the table of ActiveHumanTags and the System assigns a numeric identifier to it. The User has to confirm that he/she would like to continue with the wizard by means of UserFinish transaction to allow the System to continue.

InstallEnvironmentWizard state: The group of tags placed on the furniture and the sensors on the walls are called environment tags. The purpose of these tags is to improve the accuracy of the System in the reconstruction and localization of the User. The System will request to install these tags in the wizard of the InstallEnvironmentWizard state.

The InstallEnvironmentWizard state has the same functionality of the InstallHumanWizard state and the CheckEnvironmentTag state the same as the CheckHumanTag state.

ConfigTag state: The configuration of one tag without the wizard is possible after the first set up of the System. The User requests the install menu using UserRequestInstallMenu transaction, and then the System goes to the ConfigTag state.
Figure 8.18: CONFIDENCE SystemSetup TagInstallation state machine diagram.
In the ConfigTag state the User has some options: The addition of one new tag using the UserRequestAddTag transaction; the deletion of one tag using the UserRequestDeleteTag transaction and the replacement of one tag using the UserRequestReplaceTag transaction.

**AddTag state:** The System requests the identifier and the group of the tag using the SystemUserRequestTag transaction transaction. The User sends this information by means of the UserSendParam transaction and the System checks if the identifier has been used and if the tag with this identifier is active. When everything is OK, the System allocates a numeric identifier for the tag and saves it in the corresponding table of active human or environment tags. Otherwise, a failure message is sent to the User using the SystemUserSendMessage transaction transaction.

**DeleteTag state:** The list of active tags is sent to the User and he/she has to send the tag identifier that he/she would like to delete using the UserSendParam transaction. The System checks if this tag identifier belongs to the default tags. If this is the case, the tag cannot be deleted and it must be replaced by another one. Otherwise, the tag is removed from the corresponding active tag table.

**ReplaceTag state:** In this state, the System sends the list of active tags to the User. Then, the User sends the tag that he/she would like to replace using the UserSendParam transaction. The System checks the status of the tag and, if it is inactive, the tag is replaced and the new one is given the same identifier.

**SystemSetup CallProtocolConfiguration diagram**

The call protocol configured by the User is going to be followed by the System when an alarm or warning rises. The configuration of the ARs is done by the User during the system setup following the state machine diagram depicted in Figure 8.19.

The parameters required by the call protocol configuration are:

- The name of the AR.
- The maximum time interval the system has to wait for an answer from the receiver (in seconds).
Figure 8.19: CONFIDENCE SystemSetup CallProtocolConfiguration state machine diagram.
• The information needed to call the AR.

• The alarm priority assigned by the User (0–10, 0=no for alarms, 1=highest priority)

• The warning priority assigned by the User (0–10, 0=no for warnings, 1=highest priority)

• The possibility of sending more information about the alarm/warning (yes/no)

• The status of the AR (Active/Inactive). If the AR is inactive means that the data of this person are stored in the CONFIDENCE system, but the system can not call him/her.

The System must have at least one AR with active status. Only the active ones will be called in case of alarms or warnings. If an alarm occurs, the System will call the active AR that has the biggest alarm priority. In case of no answer after waiting the maximum time interval, the System calls the AR activated with the following biggest priority and so on. If a warning rises, the same call protocol as in alarms will be followed. However, in this case the warning priority will be assessed. The User decides if the System will send more information about the alarm/warning to the AR in case of alarm or warning. In the Initial state the User has the option to select the configuration of the call protocol using the UserRequestCPMenu transaction. If it is the first time that the system is set up and the option of call protocol configuration selected, the state machine goes to the CPWizard state; otherwise it goes to the ConfigCP state.

CPWizard state: In the CPWizard state, a wizard that requires the parameters of the receiver needed by the CONFIDENCE System is run automatically. The User inserts all these parameters using the UserSendAlarmRx transaction and the state machine goes to the CheckAlarmRx state.

CheckAlarmRx state: In the CheckAlarmRx state, it is checked that no phone number and no name is repeated. Moreover, the priority for alarms and warnings is reorganized to avoid duplicated priorities. However, several ARs can have priority equal to zero which means that the AR will not be called by the system because of an alarm or warning. This way, an ARs can be configured only to receive alarms, only to receive warnings or to receive both alarms and warnings. Finally, the presence of at least one
active receiver is checked. Otherwise, the System requests other receiver. If these conditions are satisfied, the System stores the parameters of the receiver and assigns a numeric identifier to the new call receiver. Finally, the User has to confirm that everything is OK using the UserFinish transaction to allow the System to continue with the wizard.

**ConfigCP state:** If the first set up has been done and the User requests the call protocol configuration by means of the UserRequestCPMenu transaction, the System goes directly to the ConfigCP state.

In the ConfigCP state, the User can add a new receiver using the UserRequestAddAlarmRx transaction; delete a receiver using the UserRequestDeleteAlarmRx transaction, and edit parameters of the receiver using the UserRequestEditAlarmRx transaction, including the activation or deactivation of the receiver.

**AddAlarmRX state:** The User must send the AR information using the UserSendAlarmRx transaction and the System checks that there is no repetition of the name. The User must finish the action by means of the UserFinish transaction and the state machine returns to the Initial state.

**DeleteAlarmRX state:** The System sends the list of alarm receivers to the User and he/she replies with the name of the receiver that wants to delete by means of the UserSendAlarmRx transaction. The System checks if there is at least one AR active. If the User wants to delete the only AR that is active, the system will give an error. After receiving a message of acknowledgement or failure by means of the SystemUserMessage transaction, the User must finish the action using the UserFinish transaction and the state machine returns to the Initial state.

**EditAlarmRX state:** The User has to send the AR, the parameter that he/she would like to edit and the value of the new parameter by the means of the UserSendAlarmRx transaction. The following parameters of the receiver can be changed: maximum time interval for an answer, information needed to call the AR, alarm priority, warning priority, possibility of sending more information about the alarm/warning and the status of the receiver. In each case, the system checks the new parameter and sends an information message to the User by means of the SystemUserMessage transaction. Then the User finishes the action by means of the UserFinish transaction and the System returns to the Initial state.
SystemSetup ConfigureAlarms diagram

The ConfigureAlarms diagram of the SystemSetup state machine is shown in Figure 8.20. The User can inhibit the alarms in certain situations using the transaction called UserInhibitAlarm. Additionally, the User can indicate to the system the normal situations to be lied down by means of the transaction called UserSetLyingScenarios as described in Figure 8.20. If the User wants to reactivate an inhibited situation or a lying scenario, he/she will send the transaction called UserReactivateAlarm. These three transactions will trigger different events that will produce a transition in the SystemSetup state machine to the following states:

**InhibitAlarm state:** If a new scenario shall be ignored, the system receives from the User the UserSendSituation transaction with the information about the scenario to be ignored that the system needs to store. The User must indicate how long he/she wants that situation to be ignored. Additionally, the system will decide if that situation can be ignored or not.

**ReactivateAlarm state:** If a scenario shall be reactivated, the User sends the UserSendSituation transaction with the information about the scenario to reactivate.
The User sends the transaction UserSendSituation with the information about the scenario to be considered as a normal lying scenario. The lying scenario can have the following values: “LYING\_IN\_BED”, “LYING\_ON\_COUCH”, “LYING\_IN\_BATH” and “THIS SITUATION”. When the value “LYING\_IN\_BED”, “LYING\_ON\_COUCH” or “LYING\_IN\_BATH” is received, all the possible situations where the User is lying in bed, couch or bath, respectively, are considered as normal situation by the system. When the value “THIS SITUATION” is received, the situation that the system is interpreting at this moment is considered as a normal lying scenario. Then, the state machine goes to the Initial state.

**8.6.3.2 CallAlarmReceiver State Machine Diagram**

The CallAlarmReceiver state machine is created as the classifier behaviour of the CallAlarmReceiver use case. The CallAlarmReceiver state machine implements the call protocol configured by the User in the set up of the System. The diagram of the CallAlarmReceiver state machine is shown in Figure 8.21.

At the beginning, the System is in the Idle state and the state machine goes to the Analyse state when the transaction called StartAnalyseSituation is received.

The possibility of raising the alarm by the User is considered by means of UserRaiseAlarm transaction, the state machine goes from the Idle state to the CallReceiver state and the System calls the configured receiver. In case the state machine would be in Analyse state and the User raises the alarm (UserRaiseAlarm transaction), it goes to the CallReceiver state and the system calls the corresponding receiver.

During the whole process of the CallAlarmReceiver state machine, the possibility of resetting the system is available using the UserResetSystem transaction.

**Analyse state:** In the Analyse state, the system determines the situation of the User, such as alarm, warning or normal situation.

In case of alarm or warning, the system enters in the alarm or warning mode. In that case, the state machine goes to the CallReceiver state and
the System begins with the call procedure for alarm or warning in each case.

**CallReceiver state:** When a warning or alarm is raised, the first call is always made to the User actor. A timer is enabled to count the time the system waits for the User to answer. The User has the possibility of stopping the protocol by means of *UserStopProtocol*. It means that the User responds to the alarm to indicate that it is a false alarm. In this case, the alarm mode is disabled. If the timer expires without a User answer, the System continues with the call procedure.

Then, the call protocol configured by the User in the set up of the System is followed. The System will call the corresponding receiver and enable a timer to wait for the receiver to answer.

The alarm receiver has the possibility of responding to the alarm by means of the *AlarmReceiverStopProtocol* transaction. Then, the alarm mode is disabled and the state machine goes to the *Analyse* state. If the timer expires, the System continues with the call procedure. The call pro-
procedure finishes when an alarm receiver stops the protocol using the *Alarm-ReceiverStopProtocol* transaction, allowing the system to disable the alarm or warning mode.

When a deficient tag is detected, the system will call the User using the *SystemCall* transaction, informing that a tag is not working properly. Then, the state machine will go to the *Analyse* state.

### 8.6.4 MODELLING FLOW–BASED BEHAVIOUR WITH ACTIVITIES

As mentioned previously, a use case scenario can also be represented by an Activity Diagram. Activity Diagrams represent flow-based behaviour through the execution of a sequence of actions. Activities define the transformation of input data into output data. SysML extended the data-flow concepts of UML adding new features, such as, continuous flow behaviors and probability in activity edges [49]. *Activity* diagrams are similar to the well-known functional flow diagrams.

Activities can describe the behaviour of blocks or parts. An activity may be specified as the main behaviour of a block that describes how input data of the block are transformed into outputs. The activity can also be specified as the method for an operation of the block. Alternatively, when a behaviour is described using state machines, activities can specify the effects of transitions or the entry/exit behaviours of a state.

#### 8.6.4.1 ProcessReconstruction activity diagram

The *ProcessReconstruction* activity diagram reconstructs the posture of the user. The *CONFIDENCE ProcessReconstruction* activity diagram is shown in Figure 8.22

*ProcessReconstructionActivityInput input parameter:* The *ProcessReconstruction* activity diagram receives through the *ProcessReconstructionActivityInput* input parameter the information about the localisation of the user tags. The information about the localisation of the user tags is then sent to the *ProcessReconstruction* node.
ProcessReconstruction node: This node reconstructs the posture of the user using the localisation information received through the ProcessReconstructionActivityInput input parameter.

The information about the processed tag type is sent to the SetTagType node and the status of the processed tag is sent to the SetTagStatus node.

SetTagType node: The SetTagType node saves the processed tag type in the ConfidenceTagTable attribute defined for the system block Confidence_System.

SetTagStatus node: The SetTagStatus node saves the status of the processed tag also in the ConfidenceTagTable attribute of the system block Confidence_System.

CheckSituationDateReconstruction node: This node is executed after the SetTagType node and the SetTagStatus node.
8.6.4.2 ProcessSituation activity diagram

The ProcessSituation activity diagram analyses the posture of the user. The CONFIDENCE ProcessSituation activity diagram is shown in Figure 8.23.

ProcessSituationActivityInput input parameter: The ProcessSituation activity diagram receives the user posture information through the ProcessSituation.ActivityInput input parameter. The information about the posture of the user is then sent to the ProcessSituation node.

ProcessSituation node: The ProcessSituation node analyses the reconstructed posture and determines whether there is an alarm situation, a warning situation or a normal situation. In case of alarm or warning, the system enters in the alarm or warning mode. The detected scenario information is sent to the SetScenarioType node and the processed alarm type is sent to the SetAlarmType node.

SetScenarioType node: In case of alarm or warning, the SetScenarioType node saves the detected scenario information in the HistoricScenariosTable attribute of Confidence_System_Block.
8.6 CONFIDENCE system development with ASSYST

SetAlarmType node: In case of alarm or warning, the SetAlarmType node stores the processed alarm type in the alarmType attribute of Confidence_System.Block.

CheckSituationDateInterpretation node: The CheckSituationDateInterpretation node is executed after the SetScenarioType node and the SetAlarmType node.

8.6.5 MODELLING STRUCTURE WITH BLOCKS

In SysML, the structural constructs are modelled using structure diagrams, such as, BDDs and Internal Block Diagrams (IBDs). The structural diagrams define the relationship between the elements that compose the system.

The BDD defines features of blocks and relationships between blocks such as associations, generalizations, and dependencies. The block definition diagram is the most widely used diagram in SysML. It captures the definition of blocks in terms of properties and operations, and relationships such as a system hierarchy.

The IBD captures the internal structure of a block in terms of sub-blocks (parts) and connections between them. A block includes properties to specify its values, parts, and references to other blocks. Ports are a special class of property used to specify the interactions between blocks.

8.6.5.1 Block Definition Diagram of Level 0 system model

The block definition diagram is the most widely used diagram to describe the structural elements that compose the system. The design team has defined the Confidence_System block as the top level element of the system being designed. This block contains the properties and operations of the system as presented in Figure 8.24. The Confidence_System block can define relationships with other block parts to describe system hierarchy. However, as this is the highest abstraction level model the system has been modelled as a single block. Future designs can refine the system model described in this section to include structural hierarchy information.

Designers described the interfaces of the block Confidence_System by means of input and output Flow ports as show in the diagram of the Fig-
(a) Confidence_System block outline.

(b) Confidence_System SysML block.

Figure 8.24: Confidence_System block.
8.6 CONFIDENCE system development with ASSYST

Figure 8.25: CONFIDENCE Level 0 datatypes.

ure 8.24(b). A pair of input/output ports are added for each actor to enable a bidirectional communication with the Confidence_System block.

UML/SysML offers basic data types, such as: Boolean, Integer, String and Real data types. All the custom data types have been defined by means of Datatype and Enumeration SysML elements. The Datatype elements of SysML can represent complex data types by means of attributes and operations. As an example, two CONFIDENCE Level 0 data types are shown in Figure 8.25.

8.6.6 AUTOMATIC SYSML TO SYSTEMC–TLM CODE GENERATION

The CONFIDENCE Level 0 executable code is generated using the proposed ASSYST framework. ASSYST automatically produces the SystemC–TLM source code from the SysML model describing the system. The code generation process is launched from the context menu of the CONFIDENCE_Level0.sysml model as shown in Figure 8.26.

The first step of the generation process involves the creation of a new C++ Eclipse project that will store the SystemC–TLM source code generated by ASSYST. The proposed framework enables the designer to input the name C++ project as presented in Figure 8.27(a). Then, ASSYST
Figure 8.26: ASSYST launch.
creates the C++ project and configures its include paths and required libraries to be able to compile the generated code. Additionally, two C++ compiler configurations are set up: Debug and Release.

During the automatic generation process, ASSYST reports the progress of the tasks as shown in Figure 8.27(b). All the activities of the ASSYST plug-in are visualized in the progress-bar dialog.

The second step involves the formalization of the State Machine and Activity diagrams of the SysML system model. In this step formal Finite State Machine (FSM) and Synchronous Data Flow (SDF) behavioural models are automatically created from State Machine and Activity diagrams using the ASSYST ATL transformations proposed in Section 7.2.2. At the end of this step, the behavioural aspects of the CONFIDENCE system are described in the FSM and SDF models, as depicted in Figure 8.28.

In the third step the SysML, FSM and SDF models are automatically translated into a SystemC–TLM heterogeneous executable code using the
Figure 8.28: ASSYST FSM and SDF models generation.
8.6 CONFIDENCE system development with ASSYST

ASSYST M2T generation process described in Section 7.3. As depicted in Figure 8.29, ASSYST creates the required folder structure in order to store the source files produced during the code generation process.

8.6.7 CONFIDENCE SYSTEMC–TLM SIMULATION

A compilable SystemC–TLM source code is the result of the generation process. Then, the designers complete the executable model incorporating the atomic behaviours of the FSM states and the SDF nodes.

The CONFIDENCE Level 0 model simulation involves the creation of the system stimuli and the execution of the produced SystemC–TLM code. The system stimuli are specified using XML files where the input transactions for the system model are defined. A XML stimuli file must be prepared for each system actor. In the case of the CONFIDENCE Level 0 model, one
stimuli file is specified for the *User* actor and another stimuli file is defined for the *AR* actor.

In Listing 8.19 an example is presented showing the output of a CONFIDENCE Level 0 execution run. In this example, the *Release* configuration of the executable model has been executed in a Linux terminal.

The information printed on the terminal during the simulation of the executable model is:

- Progress information of the stimuli files: loading, reading and saving status.
- Simulation start and stop.
- Initialization and simulation timers (user, system and real timings). The initialization timer includes the time needed to initialize the sys-
tem objects and load in memory the input XML files. Additionally, a high precision timer is provided for the real simulation timing.

8.6.8 ASSESSMENT OF THE PROPOSED SYSTEM DEVELOPMENT FRAMEWORK

In order to assess the ASSYST framework, the CONFIDENCE system has been developed following two different design flows, as depicted in Figure 8.30.

On one hand, the design methodology proposed in Chapter 7 has been used for the development of the CONFIDENCE system. The design team has specified the system using a SysML model from which the SystemC heterogeneous executable model is created using ASSYST.

On the other hand, the SAVY library [75], created by our research group, has been employed for the development of the CONFIDENCE care system [151]. In this case, UML modelling language was used for the development of the CONFIDENCE system.

In order to evaluate the proposed system design methodology, the cost of creating an executable system model from a UML/SysML system description and the simulation performance of the executable model have been measured. The results are presented in Table 8.6. ASSYST utilizes SysML as the modelling language and supports the use of State Machine Diagrams (SMD) for control–based behavioural parts and Activity Diagrams (AD) for data flow behavioural descriptions. This provides a higher expressive power over SAVY, as SAVY only supports State Machine Diagrams.

One of the key benefits of ASSYST is the automation of the design process. The automated Model-driven Design (MDD) process reduces the error–prone manual design tasks and greatly improves productivity and reliability.
Table 8.6: Assessment of the ASSYST framework.

<table>
<thead>
<tr>
<th>Design Framework</th>
<th>SAVY</th>
<th>ASSYST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modelling Language</td>
<td>UML</td>
<td>SysML</td>
</tr>
<tr>
<td>Behavioural diagrams</td>
<td>SMD</td>
<td>SMD &amp; AD</td>
</tr>
<tr>
<td>Executable code generation</td>
<td>Manual</td>
<td>Automatic</td>
</tr>
<tr>
<td>C++ library LoC</td>
<td>5,977</td>
<td>1,726</td>
</tr>
<tr>
<td>Total executable LoC</td>
<td>16,282 LoC</td>
<td>22,598 LoC</td>
</tr>
<tr>
<td>Manually generated LoC</td>
<td>100% (16,282 LoC)</td>
<td>14.2% (3,209 LoC)</td>
</tr>
<tr>
<td>Automatically generated LoC</td>
<td>–</td>
<td>85.8% (19,389 LoC)</td>
</tr>
<tr>
<td>Simulation Time</td>
<td>1.56 s</td>
<td>0.65 s</td>
</tr>
<tr>
<td>Development Time</td>
<td>6 months</td>
<td>1 month</td>
</tr>
</tbody>
</table>

Both SAVY and ASSYST are supported by C++ libraries. The idea behind SAVY is to provide a C++ library to enable UML–to–SystemC design patterns and facilitate the work of design engineers as much as possible. SAVY enables the creation of executable system models offering the designers a friendly Application Programming Interface (API). However, code fragments added exclusively to ease the library usage by designers lead to an extra complexity of the SAVY C++ library. In the ASSYST framework, as the code generation process is carried out by an automatic tool, the proposed C++ library is not primarily focused on facilitating the design engineers use and many designer–centric constructs are avoided. Thus, the proposed C++ library is much more lightweight.

The cost of creating an executable system model from an UML/SysML description is estimated by means of the LoC written by the design team and the overall development time.

Using the SAVY library, the executable model of the system has 16,282 LoC. In the case of SAVY, although design patterns are provided to ease the translation process from UML to SystemC, all the code of the executable model is manually created. The executable system model developed following the ASSYST design methodology has 22,598 LoC. The increment in the LoC generates no cost for the designers as 85.8% of the code is created automatically from the system model described using SysML. Only 14.2% (3,209 LoC) of the code is manually created. The hand-coded part of the
executable code corresponds to the description of the atomic functionality of the States and the Actions of State Machine Diagrams and Activity Diagrams, respectively.

On the other hand, the CPU execution time is measured to compare the performance of the two executable system models. Both executable models use the same input stimuli that represent more than two hours of work of the system. The input stimuli has been designed to create more than 9,000 different situations where more than 100 alarms need to be handled. This simulation scenario involves almost 19,000 input transactions.

The executable models have been compiled using GNU/g++ v3.4.6 in a PC with a CPU Intel Xeon E5405 @ 2.00 GHz, running Linux Red Hat Enterprise. As shown in the last row of Table 8.6, the executable code created with ASSYST is 2.4 times faster than the executable model designed with SAVY.

As a global result, the design methodology that has been proposed by ASSYST has greatly reduced the development time of the CONFIDENCE system from 6 months to 1 month.

8.7 CONCLUDING REMARKS

This chapter presented a case study in order to assess and analyse the applicability of the proposed methodology and framework. The CONFIDENCE project has been employed to validate the Verification Framework, MATLAB Framework and the ASSYST Framework proposed in previous chapters.

Initially, SAVY has been used to manually create three SystemC–TLM executable functional models (Level 0, Level 1 and Level 2) from UML models describing the system. Besides, UML-to-SystemC translation steps were defined.

In the next step, the proposed Verification Framework has been coupled to the reference executable models created previously using SAVY. A key point of the proposed verification framework is its simplicity to be coupled to a SystemC–TLM system model. Few lines of code are necessary to integrate the proposed verification framework to the system model. Assertions
have been specified for several abstraction level executable models and the
performance of the verification framework has been measured.

Then, the MATLAB framework has been coupled to the SystemC–TLM
executable models created previously. This framework simplifies the task of
integrating MATLAB algorithms into a SystemC model; the cost of building
these models is very small and the resulting model code is clear.

Finally, the ASSYST Framework has been used to develop the Confidence
project. The system-design team described in SysML a functional
model of the CONFIDENCE system. Then, a SystemC–TLM model has been
automatically generated from the SysML model using ASSYST. The AS-
SYST framework and the resulting SystemC–TLM model have been com-
pared with the SAVY methodology. The SysML model has been described
at the highest level of abstraction, equivalent to the UML Level 0 model
developed using the SAVY methodology. Both the UML Level 0 model
and the SysML model have been developed taking as input the same Con-
fidence system requirements. Additionally, both the executable models
created by means of SAVY and ASSYST have been simulated using the
same input stimuli. One of the key benefits of ASSYST is the automation
of the design process; the ASSYST framework greatly reduces the devel-
opment time and the error-prone manual tasks. As a result, the design
productivity is improved.

The results of this case study have been included in the papers [125,
127, 141, 148, 151].
Contents

9.1 Conclusions ................................................. 258
9.2 Areas for further research .............................. 263

A methodology to assist in the high abstraction level design and functional verification has been proposed in this research work. The contributions of this dissertation have been divided into three workflows: a functional verification methodology supported by a framework implementing Assertion-based verification (ABV) techniques, an approach integrating MATLAB algorithms for the creation of functional models at Transaction Level Modeling (TLM), and a Model-driven Design (MDD) approach for the generation of SystemC–TLM models from a system described using Systems Modeling Language (SysML).

This chapter is divided into two sections. Section 9.1 presents the main conclusions of this research work and Section 9.2 introduces the areas for further research.
9.1 CONCLUSIONS

First, a design flow integrating ABV into SystemC–TLM simulations has been presented in this dissertation. This design flow enables the verification of complex systems using assertions during the system model simulation. The proposed ABV approach is based on the Assertion Specification Language (ASL) and a novel ABV verification framework supported by a C++ library built on top of SystemC. An integrated, powerful and flexible framework has been created in order to simulate and verify complex systems starting from a very high abstraction level. The following can be remarked about the proposed ABV approach:

- The ASL language has been proposed for assertion based transaction level system specification. ASL enables the verification team to write assertions simply and fast, with increased detail within a SystemC–TLM modelling environment. Thus, unwanted behaviours are detected in the early stages of the modern electronic systems design flow, resulting in greater margin of reaction and reduced development cost.

- ASL enhances antecedent–consequence style properties with temporal constructs, and thus, assertions can be created for both timed and untimed SystemC–TLM coding styles. As a result, ASL is not only valid in the initial stages of the design; it is prepared to support a SystemC based design flow from TLM to Register Transfer Level (RTL) models.

- Assertions are described in user friendly eXtensible Markup Language (XML) files following the proposed ASL syntax. XML files are easily readable by human beings and machines. Thus, verification engineers can specify system assertions easily.

- The application of the proposed verification framework to a SystemC–TLM model is very simple. Clear steps have been presented to apply the proposed verification framework to a SystemC–TLM model.

- The proposed ABV framework enables independent design and verification teams. As the assertion specification is not embedded in the design model code, the design and verification tasks are decoupled. The redundancy introduced by working with these two independent teams mitigates the probability of a specification misinterpretation.
9.1 Conclusions

- A modification in the assertion descriptions does not require any system model recompiling. The proposed verification framework loads dynamically the XML assertions file during the start-up phase of the system model execution. Modifications to the input assertion set are applied to the ABV process the next time the system is executed.

The proposed verification framework has been applied to a complex electronic system development: the CONFIDENCE project. The verification framework has been coupled to the reference executable models created using SAVY. From this case study, it has been concluded that:

- A key point of the proposed verification framework is its simplicity to be coupled to a SystemC–TLM system model. The case study demonstrated that few lines of code are necessary to integrate the proposed verification framework to the system model.

- The proposed ABV framework is flexible and enables to reuse assertions between abstraction levels. Assertions have been successfully specified for several abstraction level executable models.

- The performance loss due to the verification framework depends on the number of assertion evaluations. In turn, the number of evaluations depends on the number of assertions defined for the system model. The performance loss increases slightly as the system model gets more detailed. This execution time increment of the verification process is due to the new assertions added in order to verify the new system details.

Next, the MatlabEngine++ library has been proposed. This library abstracts efficiently the MATLAB Application Programming Interface (API). The inclusion of a MATLAB Engine in a SystemC model enables building TLM models, where architecture (modules and transactions) is described in SystemC and algorithms are described in MATLAB. About MatlabEngine++ approach, the following can be said:

- MatlabEngine++ enables the creation of SystemC–MATLAB models at TLM level to verify the correct algorithm cooperation within the system. Thus, architecture and algorithm tuning can be performed efficiently early in the design, with the system–design and the algorithm–design teams working in their preferred environments.
• The proposed abstraction of MATLAB Engine simplifies the task of integrating a MATLAB Engine in a SystemC model, with negligible additional costs in development. Thanks to the proposed approach, the cost of building these models is very small and the resulting model code is clear.

• The MatlabEngine++ framework is responsible for automatically keeping the variables of the MATLAB session synchronized with the SystemC application. This feature avoids the error-prone manual coding process to exchange data between the SystemC model and MATLAB.

• As MatlabEngine++ is based on the MATLAB Engine API, both Simulink models and traditional MATLAB M-file scripts are supported for simulation.

• The proposed methodology defines the SystemC model as the main controller of the simulation during the integration of the MATLAB algorithms. The benefit of this approach is twofold: it avoids the creation of S-Function wrappers in MATLAB and it allows to maintain the same simulation environment used in the SystemC–TLM design and verification flow.

• Although the MatlabEngine++ library is used in a SystemC environment, the proposed approach is not exclusively built for SystemC models; it can be integrated into any C++ based application.

The MATLAB framework has been coupled to the SystemC–TLM executable models of the CONFIDENCE system created previously during the assessment of the verification framework. An algorithmic model of the CONFIDENCE system has been constructed to measure the performance of the MATLAB framework. From this case study, it has been concluded that:

• The cost of building a SystemC–TLM system model with integrated MatlabEngine++ can be split in two parts: cost of modifying the MATLAB executable specification (M-files or Simulink model); and cost of modifying the SystemC–TLM system model. On one hand, the cost of modifying the MATLAB executable specification is zero. On the other hand, the case study demonstrated that the cost of modifying the SystemC model was very small, despite the complexity of the algorithms included in this case study.
The overhead due to the SystemC–MATLAB interaction is very small, despite the amount of data/command transfers between the SystemC and the MATLAB domains.

The proposed approach allows to use ABV techniques to verify a SystemC–TLM with integrated MATLAB algorithms.

Finally, a MDD approach based on the Automatic SysML to SystemC Translator (ASSYST) framework has been presented in order to translate electronic systems described using SysML to SystemC-TLM heterogeneous executable models. ASSYST is supported both by a SysML based methodology and a modelling tool. The ASSYST modelling tool is a Eclipse plug-in based on Model-to-Model (M2M) and Model-to-Text (M2T) techniques using ATL Transformation Language (ATL) and Acceleo. About the ASSYST approach, the following can be said:

- Formally defined behavioural semantics of SysML State Machine diagrams and Activity diagrams have been presented. Initially, the abstract syntax supported for SysML State Machine diagrams and Activity diagrams have been defined. Then, the semantic domains for SysML State Machine diagrams and Activity diagrams have been defined using Finite State Machine (FSM) and Synchronous Data Flow (SDF) Model of Computations (MoCs), respectively. Finally, the semantic mappings from the abstract syntax of SysML State Machine diagrams and Activity diagrams to FSM and SDF MoCs have been formalized. As a result, clear semantic transformation rules have been defined for State Machine diagrams and Activity diagrams.

- The proposed semantic formalization established the theoretical background to effectively implement a MDD based framework to create SystemC–TLM executable code with heterogeneous behavioural MoCs for early electronic system definition. Rigorously defined semantic mappings facilitated the definition of M2M and M2T specifications.

- ASSYST framework promotes the use of a MDD methodology for visual electronic systems modelling and automatic SystemC–TLM executable code generation. The development of visual models is more intuitive than the development of source code.
• ASSYST offers the capability to generate SystemC–TLM code describing both control-based and data-flow based behaviours. This heterogeneity improves the flexibility and the practical use of ASSYST within an electronic system design process.

• The automatic code generation is a key characteristic of ASSYST; the SystemC–TLM executable model can be created immediately. This characteristic enables to easily modify the architecture of the system model, reorganizing the system structure (SysML Blocks) or the behavioural aspects (SysML Use Cases, State Machines or Activities), and then to generate again the SystemC–TLM code in a very short time period. As a result, ASSYST improves the initial iterations carried out to tune the architecture of the high abstraction functional model and facilitates what-if analysis.

• The global benefits of the proposed approach are clear: the error-prone manual tasks are greatly reduced, thus improving the design productivity.

The ASSYST methodology and framework has been applied to the CONFIDENCE system development process in order to illustrate the benefits of the proposed approach. The CONFIDENCE system has been developed following two different design flows. On one hand, ASSYST has been used for the development of the CONFIDENCE system. The design team has specified the system using a SysML model from which the SystemC executable model is created using ASSYST. On the other hand, the SAVY library has been employed for the development of the CONFIDENCE care system [151]. In this case, Unified Modeling Language (UML) modelling language was used for the development of the CONFIDENCE system. ASSYST and SAVY design flows have been compared in the case study. The following can be remarked about the proposed ASSYST approach:

• One of the key benefits of ASSYST is the automation of the design process. Thus, error-prone manual tasks are minimized. Within the CONFIDENCE development using ASSYST, 85.8% of the code is created automatically from the system model described using SysML. The manually created part of the executable code corresponds to the description of the atomic functionality of the States and the Actions of State Machine Diagrams and Activity Diagrams, respectively.
The design methodology proposed by ASSYST has greatly reduced the development time of the first executable model of the CONFIDENCE system from 6 months to 1 month.

9.2 AREAS FOR FURTHER RESEARCH

The objective of this research work is to enhance the electronic system design and verification methodologies in order to improve the productivity of the modern electronic device development process. During this work, several topics have been found that deserve further research:

- **Coverage-driven ABV**
  The effectiveness of the proposed ABV framework depends on the stimuli prepared by the verification team. On one hand, random stimuli can be injected into the system under verification in order to check the verification assertions. This approach requires huge system simulation times in order to get an acceptable verification coverage. On the other hand, the verification team could prepare stimuli data based on some desired use cases or scenarios. In this case, reaching corner-cases during the verification process is a difficult task. A coverage-driven ABV may be a topic for further research. This approach enables the verification framework to autonomously guide the stimuli injection based on the obtained coverage information, thus improving the verification process. Artificial Intelligence (AI) techniques may be promising for the automatic stimuli generation process as they have been successfully applied to adaptive search tasks in other engineering areas.

- **Extension of the supported SysML modelling elements**
  ASSYST methodology relies on SysML State Machine diagrams and Activity diagrams. An analysis could be done to extend the supported SysML behavioural elements and include features such as hierarchical state machines, parallel activities, etc.

- **Model refinement techniques**
  Using the proposed MDD approach, the system-design team is able to create several system models for the architecture design process. An interesting research topic would be to define new models based on some refinement rules over the previous system model. These refinement rules would completely capture the design intent of the designer.
for the new model. This approach enables to define a mechanism by which a new model is automatically produced based on M2T techniques implementing the refinement rules. As a result, the design process would be a sequence of design intentions represented by refinement rules that would improve the tracking of the development evolution and the architecture design exploration and analysis. This approach would likely provide benefits to the requirements management and also to the refinement of verification assertions.

- **System on Chip (SoC) modelling support**
  This research work focused on the generation of executable functional SystemC–TLM models at high level of abstraction. The executable model generated by means of this approach enables the analysis of the architecture design and algorithm tuning prior to the Software (SW)/Hardware (HW) partitioning. Another topic for further research is the extension of the proposed MDD framework for SoC modelling where the system is composed of SW parts and HW parts and where more detailed timing information plays an important role. In this scenario, the management of SysML modelling elements such as allocations is fundamental. An analysis would be necessary to determine whether a pure SysML based modelling approach provides an adequate framework, or an extension of SysML using specific real time modelling profiles such as Modeling and Analysis of Real–time Embedded Systems (MARTE) is required.
References


“A UML Profile for SysML-Based Comodeling for Embedded Systems Simulation and Synthesis”. In: Dresden, Germany 2010. See pp. 27, 30, 32.


COMPLEX. URL: https://complex.offis.de. See p. 31.


APPENDIX A

Foundations of Model-driven Initiatives

Contents

A.1 Model, Metamodel and Meta–metamodel . . . . . . . . 286
A.2 MBE, MDE, MDD and MDA . . . . . . . . . . . . . . . . 288
A.3 Model Transformations . . . . . . . . . . . . . . . . . . . 291
A.4 Visual Modelling Languages . . . . . . . . . . . . . . . . 293
A.5 Syntax and Semantics . . . . . . . . . . . . . . . . . . . . 298
A.6 Modelling Tools . . . . . . . . . . . . . . . . . . . . . . . 298

Model-driven initiatives promoted an evolution from code-centric software development strategies towards visual model-centric approaches as such code-centric development strategies did not scale well with large systems. Additionally, model-centric approaches provided many benefits such as automatic code generation. This evolution provided a high abstraction level development process that enabled the models to direct and guide the understanding, analysis, design, implementation, test, integration, deployment and maintenance of the systems under development.

Some of the most prominent benefits of using model-driven initiatives are presented in the following list:
Model-driven processes improve the productivity of the development team by means of code-generation techniques.

Models facilitate early evaluation of the system, thus reducing the number of defects in the final code.

Model-driven methods increases the decomposition and modularization of the system under development. As a result, they promote early exploration of functional and/or architectural alternatives.

Model-driven initiatives facilitate the system evolution and the reuse of parts.

The communication between project members and development teams is improved. Additionally, the communication with the customers and stakeholders is also improved.

**A.1 MODEL, METAMODEL AND META–METAMODEL**

The objective of this section is to briefly introduce the key model-driven concepts that are used throughout this PhD dissertation, namely, *Model, Metamodel* and *Meta–metamodel*.

The common element among model-driven initiatives is the *model*. Although more abstractly, the term *model* is widely used in all the engineering disciplines and can be defined by the following sentence: “A model is an abstract and simplified representation of a system” [154]. Thus it is clear that the terms *model* and *system* are tightly coupled.

A *system* is defined by Institute of Electrical and Electronics Engineers (IEEE) as “a collection of components organized to accomplish a specific function or set of functions” [155]. Thus, a *system* is composed of three concepts: a predefined objective or behaviour, a set of components or elements, and how these components are interconnected.

As defined previously, a *model* represents a view of a *system*. However, a *system* may be represented by several different *models*, which may differ in the abstraction level. The abstraction level defines a particular view of a *model* that focuses on the relevant aspects for a particular purpose, leaving other aspects out of the view. As a result, a *model* may not focus on all the aspects of a *system*, i.e., it needs not be complete. Moreover, being a
model complete or incomplete does not mean it is a good or bad model, respectively. A model is good if its abstraction level satisfies the particular application or purpose it was created for.

In a system development process, the abstraction level of a model may be qualified as high abstraction level model or a low abstraction level model depending on the details of the system representation. On one hand, a high abstraction level model represents few details of the system being modelled. The purpose of high abstraction level models is to express the requirements, describe the system context and analyse the system. On the other hand, a low abstraction level model is very detailed. The purpose of low abstraction level models is to describe the design intents and the implementation details.

A metamodel is defined as a model of models. A model can be defined by means of the elements, attributes and connections described by a metamodel. When a model is specified in accordance with a metamodel, it is said that the model conforms to the metamodel.

A meta–metamodel introduces the elements that are required to specify metamodels. As a model with its metamodel, a metamodel conforms to the meta–metamodel. Note that a meta–metamodel is usually able to describe its own structure, i.e. it can be specified by means of its own elements. In such a case, a meta–metamodel conforms to itself.

### A.1.1 FOUR-LAYER METAMODEL ARCHITECTURE

Figure A.1 presents a four-layered infrastructure as explained in [45]. This layered infrastructure describes the relationship between models, metamodels and meta-metamodels. In this infrastructure, a layer is used to specify the layer below. Furthermore, a layer can be viewed as an ‘instance’ of the layer above, as described in Figure A.1.

M0 specifies the user data layer where the data objects are defined.

M1 specifies the model layer. This layer contains the models of the user data.

M2 specifies the metamodel layer. In this layer, metamodels are defined by means of well-defined modelling languages.
M3 specifies the *meta-metamodel* layer. This layer is *meta-circular* as the *meta-metamodel* are self-defined.

### A.2 MBE, MDE, MDD AND MDA

Several model-driven related acronyms such as, Model-based Engineering (MBE), Model-driven Engineering (MDE), Model-driven Design (MDD) and Model-driven Architecture (MDA) (often defined as MD*) are found in the literature focused on system modelling. MDE, MDD and MDA are often defined as MD*. Although these acronyms are sometimes used interchangeably and their differences may seem often vague, guidelines are found in [156] in order to cope with them. The relationship between these acronyms is summarized in Figure A.2.
A.2 MBE, MDE, MDD and MDA

A.2.1 MODEL-BASED ENGINEERING (MBE)

In a MBE approach, the models are important artifacts within the systems development process, but they may not lead the development.

A.2.2 MODEL-DRIVEN ENGINEERING (MDE)

MDE is considered a subset of MBE. In MDE, models are key elements of the development process, as they drive the processes that ultimately produce a real system. The MDE initiative goes beyond of the pure implementation activities and proposes its usage for the complete engineering process, participating in tasks such as analysis, design, implementation, verification and test [47, 157].

A.2.3 MODEL-DRIVEN DESIGN (MDD)

MDD is proposed as a subset of MDE. The main proposition of MDD is to capture in visual models all the information of the system being designed in order to concentrate on generating implementation artifacts from models. In MDD, the design products are models instead of source code. The first experiences with MDD date back to the late 90s [43, 44] It has been in the last decade, when MDD has emerged as a development methodology for software engineering [45, 46].

Figure A.2: Relationship between model-driven acronyms.
A key point of MDD is that the transformation and generation processes can be applied to graphical models in order to create diverse products, such as, text documentation, source code, other models, etc. However, a MDD approach reports limited success in the development time if models are specified exclusively for documentation purposes [158]. The automation of the source code generation from system models simplifies the work of engineers and avoids the error-prone and time-consuming manual translation process. As a result, the development time of the device being designed is reduced significantly.

In a MDD approach the model plays the primary role. Thus, despite all the benefits of automation, MDD approaches are only as good as the models they help the engineers construct [46]. Hence, it is vital to construct a good model in order to take advantage of MDD methods. A good model shall:

- Abstract away not-currently-relevant details
- Accurately reflect the relevant aspects of the system
- Help the engineers analyse the system
- Be cheaper to build than code
- Improve the communication between people involved in the system development

A.2.4 MODEL-DRIVEN ARCHITECTURE (MDA)

In 2001, the Object Management Group (OMG) defined a particular realization of MDD using the term MDA [159]. Therefore, MDA is depicted as a subset of MDD in Figure A.2.

MDA is an approach to using models for the development of software systems [160]. It provides a set of guidelines for the structuring of specifications, which are expressed as models. One of the main aims of the MDA is to separate the specification of what the system under development shall do from how that system uses the capabilities of its platform [161]. OMG defined a modelling and design methodology for MDA supported by OMG standards and tools such as Unified Modeling Language (UML) and
Meta-Object Facility (MOF). The methodology is supported by three abstraction level and their corresponding models: Computation Independent Model (CIM), Platform Independent Model (PIM) and Platform Specific Model (PSM). The MDA modelling methodology is based on transformations between CIM, PIM and PSM constructs [162], as described in Figure A.3.

**CIM** System requirements are structured in a CIM. This model describes the system under development and its context. It is useful for presenting and understanding the system under development. Typically such a model is independent of how the system is implemented. The CIM is sometimes called a domain model or a business model. In a system specification, CIM requirements should be traceable to the PIM and PSM constructs that implement them, and vice versa.

**PIM** The PIM describes the system and its functionality, but does not show details of its use of its platform. The PIM does not specify any implementation technology.

**PSM** The PSM produced by the transformation is a model of the same system specified by the PIM. The PSM contains details that have meaning only within a specific platform. The PSM has a relation with implementation technologies as it specifies how that system makes use of the chosen platform.

### A.3 MODEL TRANSFORMATIONS

In MD*, the model transformations are fundamental methods within the system development process. The model transformations can be used for tasks such as, modifying, creating or merging models as well as for code generation [163]. A model transformation is defined by an input model (source model), an output model (target model) and the transformation rules. Model transformation languages have been classified in [163] and [164].

![Figure A.3: MDA modelling methodology.](image-url)
**Change of Abstraction:** Model transformations can change the level of abstraction between a source model and a target model. The abstraction level represents the amount of details in a model. A model transformation either increases the abstraction level, reduces the abstraction level or the abstraction level is unmodified.

- A *refinement transformation* produces the target model by adding details to the source model.
- An *abstraction transformation* produces the target model by reducing the details of the source model.
- A model transformation may modify the source model but it may keep the level of abstraction of the model unchanged.

This transformation types are independent of the change in the metamodel; source and target metamodel can be the same or different.

**Change of Metamodels:** The source and target metamodels may be the same or different [164, 165]. As a result, two types of transformations are defined:

- In an *endogenous transformation* the source metamodel and target metamodel are the same. This type of transformation is also called *rephrasing transformation*.
- An *exogenous transformation* map concepts between different metamodels. This type of transformation is also called *translation transformation*.

**Supported Target Type:** We can distinguish model transformations with respect to the type of the target. The target can be model or text.

- In a Model-to-Model (M2M) transformation both the source and the target are models. Elements in the source model are mapped to elements in the target model.
- In a Model-to-Text (M2T) transformation the source is a model but the target is a text. Elements in the source model are mapped to fragments of text. If the text produced by the transformation is source code, the transformation is also called *code generation*. 
A.4 VISUAL MODELLING LANGUAGES

Graphical modelling languages, such as UML [48] and Systems Modeling Language (SysML) [49], have been proposed to be applied within the MDD methodology. The UML standard was conceived by the OMG as an object-oriented software modelling language, whereas SysML is defined as a general-purpose modelling language for systems engineering. The International Council on Systems Engineering (INCOSE) defined SysML, in close collaboration with OMG, as an extension of a subset of UML.

These modelling languages define a graphical notation based on diagrams that allows to create visual models of systems under development. The purpose is to promote intuitive visual design patterns that raise the abstraction to a higher level and improve the communication between engineer teams developing the system. Moreover, the use of standard modelling languages guarantees the compatibility between different modelling frameworks.

A.4.1 UNIFIED MODELING LANGUAGE (UML)

UML [166] is a general-purpose modelling language for software-intensive systems that is designed to support mostly object oriented programming. UML is used for a wide variety of purposes across a broad range of domains. Although UML was initially announced as a software domain modelling language, the OMG changed the UML notation into a general-purpose modelling language which can be used for various application domains within engineering modelling [99, 100].

UML 2 describes 13 official modelling diagram types [167], as shown in Figure A.4. The modelling diagrams are grouped in Structure diagrams, Behaviour diagrams and Interaction diagrams:

- **Structure diagrams:**
  - *Class diagrams* define the basic building blocks of a model: the types, classes and general materials used to construct a full model.
  - *Component diagrams* are used to model higher level or more complex structures, usually built up from one or more classes, and providing a well defined interface.
Figure A.4: UML diagrams.

- **Composite structure diagrams** diagrams provide a means of layering an element’s structure and focusing on inner detail, construction and relationships.

- **Deployment diagrams** show the physical disposition of significant artifacts within a real-world setting.

- **Object diagrams** show how instances of structural elements are related and used at run-time.

- **Package diagrams** are used to divide the model into logical containers, or 'packages', and describe the interactions between them at a high level.

- **Profile diagrams** provide a visual way of defining extensions to the UML specification.

• **Behaviour diagrams:**
A.4 Visual Modelling Languages

- **Activity diagram** emphasizes the inputs, outputs, sequences, and conditions for coordinating other behaviours. It provides a flexible link to blocks owning those behaviours.

- **State machine diagram** defines a set of concepts that can be used for modelling discrete behaviour through finite state transition systems. The state machine represents behaviour as the state history of an object in terms of its transitions and states.

- **Use case diagram** describes the usage of a system (subject) by its actors (environment) to achieve a goal, that is realized by the subject providing a set of services to selected actors.

- **Interaction diagrams:**
  - **Communication diagrams** show the network, and sequence, of messages or communications between objects at run-time, during a collaboration instance.
  - **Interaction overview diagrams** fuse activity and sequence diagrams to allow interaction fragments to be easily combined with decision points and flows.
  - **Sequence diagrams** are closely related to communication diagrams and show the sequence of messages passed between objects using a vertical timeline.
  - **Timing diagrams** fuse sequence and state diagrams to provide a view of an object’s state over time, and messages which modify that state.

UML includes a profile mechanism that allows it to be constrained and customized for specific domains and platforms. UML profiles use stereotypes, stereotype attributes and constraints to restrict and extend the scope of UML to a particular domain. The best known example of customizing UML for a specific domain is SysML, a domain specific language for systems engineering.

A.4.2 SYSTEMS MODELLING LANGUAGE (SYSML)

OMG and INCOSE developed the SysML graphical modelling language as a response to the UML for Systems Engineering Request For Proposal
(RFP). SysML is a UML profile that represents a subset of UML 2 with extensions. SysML [168, 169] supports the specification, analysis, design, verification, and validation of systems that include hardware, software, data, personnel, procedures, and facilities.

Figure A.5 depicts an overview of the SysML/UML interrelationship. SysML is based on UML 2. SysML reuses a subset of UML, discard a subset of UML, and provides additional extensions needed to address system engineering requirements not present in the UML. The intersection of the two circles indicates the UML modelling constructs that SysML reuses.

The modelling diagrams provided by SysML are shown in Figure A.6. SysML reuses the Activity diagram, Sequence diagram, State Machine diagram, Use Case diagram and Package diagram from UML 2. However, the Block definition diagram and Internal Block diagram are modified from UML 2, and the Requirement diagram and Parametric diagram are new in SysML.

- **Block definition diagram** is based on the Class diagram from UML 2. The Block Definition Diagram (BDD) in SysML defines features of blocks and relationships between blocks such as associations, generalizations, and dependencies. It captures the definition of blocks in terms of properties and operations, and relationships such as a system hierarchy or a system classification tree.

- **Internal block diagram** is based on the Composite structure diagram from UML 2. The Internal Block Diagram (IBD) in SysML captures the internal structure of a block in terms of properties and connectors
A block can include properties to specify its values, parts, and references to other blocks.

- **Parametric diagram** is a new diagram in SysML. Parametric diagrams include usages of constraint blocks to constrain the properties of another block. The usage of a constraint binds the parameters of the constraint, such as $F$, $m$, and $a$, to specific properties of a block, such as a mass, that provide values for the parameters. The constrained properties, such as mass or response time, typically have simple value types that may also carry units, quantity kinds, or probability distributions.

- **Requirement diagrams** are new diagrams in SysML. The requirement stereotype represents a text-based requirement. A requirement specifies a capability or condition that shall be satisfied. A requirement may specify a function that a system must perform or a performance condition a system must achieve. SysML provides modelling constructs to represent text-based requirements and relate them to other modelling elements. A requirement can also appear on other diagrams to show its relationship to other modelling elements. The requirements modelling constructs are intended to provide a bridge between traditional requirements management tools and the other SysML models.
A.5 SYNTAX AND SEMANTICS

Model-driven initiatives rely on modelling languages to describe models. Visual modelling languages such as UML, SysML and Modeling and Analysis of Real–time Embedded Systems (MARTE) are defined on the basis of metamodels. Metamodels have a syntax as well as semantics [129].

The semantics of a metamodel defines how information which is represented in its models is to be applied. It defines the meaning of a model’s contents. The syntax defines a notation in which the model is represented. The abstract syntax covers all the information that has a semantic meaning, stripped from layout and formatting information. The concrete syntax refers to the data related to human-readable representations of a model, that is the layout and formatting information. In the case of a visual representation, the concrete syntax describe positions, sizes, colours and shapes of graphical elements. Since a model can have multiple representations, there may be multiple concrete syntaxes for the same model.

A.6 MODELLING TOOLS

This section introduces the most distinguished modelling tools and frameworks.

A.6.1 ECLIPSE

Eclipse [140] is free and open source Integrated Development Environment (IDE) mostly written in Java and released under the terms of the Eclipse Public License. The Eclipse Project was created by IBM in 2001. In 2004, the Eclipse Foundation was created in order to establish a vendor neutral and open community around Eclipse.

Eclipse provides a multi-language development environment supported by an extensible plug-in framework. Eclipse can be used natively for the development of Java applications. Other programming languages such as C, C++, Python, Perl and PHP are supported using plug-ins.

The development of Eclipse and its plug-ins is organized into projects. The Eclipse Modeling Project (EMP) is one of the most active projects within the Eclipse community. It comprises several modelling frameworks
Eclipse Modeling Framework Project (EMF)

The Eclipse Modeling Framework (EMF) project [170] is a modelling framework and code generation facility for building modelling tools and applications. The core EMF framework includes the metamodel Ecore for describing models and runtime support for the models. The metamodel Ecore is based on OMG’s MOF [171]. EMF provides tools to produce a set of Java classes from a model specification. Moreover, EMF is able to create
a set of adapter classes and a basic editor that enable viewing and editing of the model. EMF provides the foundation for interoperability with other EMF-based tools and applications.

**Eclipse Modeling Framework Technology (EMFT)**

The Eclipse Modeling Framework Technology (EMFT) project [172] aims to support new technologies to extend or complement EMF. Two projects stand out within EMFT: *EMF Compare* and *Ecore Tools*.

**EMF Compare:** EMF Compare provides a model comparison framework to EMF. EMF Compare this tool provides generic support for any kind of metamodel in order to compare and merge models.

**Ecore Tools:** The Ecore Tools component provides a complete environment to create, edit and maintain Ecore models. This component eases handling of Ecore models with a Graphical Ecore Editor and bridges to other existing Ecore tools.

**Model Development Tools (MDT)**

The objective of the Eclipse Model Development Tools (MDT) [173] is to provide an implementation of industry standard metamodels. Additionally, MDT aims to support the creation of models conforming those industry standard metamodels by means of modelling tools. *The Object Constraint Language (OCL) project*, *Papyrus* and *UML2* are the main projects within MDT.

**OCL project:** The OCL project is an implementation of the OMG’s OCL [174] standard for EMF-based models. OCL is a language for the formal description of expressions in models. The expressions can represent invariants, preconditions, postconditions, initializations, guards, derivation rules and object queries. It is a language without side effects, so the verification of a condition, which assumes instantaneous operation, never alter the model objects. Its main role is to complete the various artifacts of the models to formally expressed requirements.

**Papyrus:** Papyrus is a component of the MDT project that aims to provide an IDE to edit any kind of EMF model. Papyrus is particularly
suitable for UML and its related modelling languages such as SysML and MARTE.

**UML2:** UML2 is an EMF-based implementation of the OMG’s UML version 2.x metamodel for the Eclipse platform.

**Model To Model (M2M)**

Model-to-Model (M2M) is a subproject of the top-level EMP. The M2M provides a modelling framework for model-to-model transformation languages and tools such as, ATL Transformation Language (ATL) and Query/View/Transformation (QVT).

**ATL:** The ATL Transformation Language (ATL) [142, 175] is a Model-to-Text Language (MTL) and toolkit developed by OBEO and INRIA. It is a model transformation language specified both as a metamodel and as a textual concrete syntax. Recently, ATL became an M2M Eclipse component, inside of the EMP. In the field of a model-driven approach, ATL provides developers with a means to specify the way to produce a number of target models from a set of source models. An ATL transformation program is composed of rules that define how source model elements are matched and navigated to create and initialize the elements of the target models.

**QVT:** The Query/View/Transformation (QVT) language implementation is based on the OMG document MOF 2.0 Query/View/Transformation Final Adopted Specification [176]. The QVT Declarative (QVTd) component aims to provide a complete Eclipse based IDE for the Core (QVTc) and Relations (QVTr) Languages defined by the OMG QVT Relations (QVTR) language. This goal includes all development components necessary for development of QVTc and QVTr programs and Application Programming Interfaces (APIs) to facilitate extension and reuse.

**Model To Text (M2T)**

The Model-to-Text (M2T) project [177] focuses on infrastructures to support model-to-text engines based on industry standards. The model-to-text engines are also known as code generators. Code generators are tools aimed at the generation of code from input models. Acceleo and Xpand are the main contributions under the Eclipse M2T project.
**Acceleo:** Acceleo [143] is an actively developed open source project, licensed under the Eclipse Public License (EPL). Acceleo is an implementation of the OMG’s MOF Model to Text Transformation Language (MOFM2T) [144] standard. Acceleo is a framework to specify code generators to generate textual artifacts (code) from models. Additionally, Acceleo generators can be deployed as Eclipse plug-ins.

**Xpand:** Xpand is a statically-typed template languages for code generation. Xpand is known to provide a very open architecture in order to integrate profilers and pretty printers in its interpreter. Xpand was originally developed as part of openArchitectureWare (oAW) project before it became a component under Eclipse.

**Textual Modeling Framework (TMF)**

The Eclipse Textual Modeling Framework (TMF) [178] project provides a means to create editors for custom textual notations. **Xtext** is the main contribution of TMF.

**Xtext:** Xtext [179] is a framework for development of external textual Domain-specific Languages (DSLs). Xtext proposes to describe a DSL using Xtext’s Extended Backus-Naur Form (EBNF) [180, 181] grammar language. Then, the Xtext generator creates a parser, an Abstract Syntax Tree (AST) metamodel (implemented in EMF) as well as a full-featured Eclipse Text Editor The Framework integrates with other Eclipse modelling technologies such as EMF, Graphical Modeling Framework (GMF), M2T and parts of EMFT.

**A.6.2 TOPCASED**

TOPCASED [139] aims to provide a development toolkit dedicated to critical and embedded systems, software and hardware. **TOPCASED** is an open-source and free software based on Eclipse Modeling Framework (EMF) and Eclipse Rich Client Platform since 2004. It takes advantage of the Eclipse UML 2 API and M2M/M2T technologies in order to offer new modelling services such as, UML editor, SysML editor, SAM editor, model rules checker, document generators and requirements traceability engine. TOPCASED model data are stored in XML Metadata Interchange (XMI) format for interoperability. TOPCASED can import models from any UML 2
XMI compliant tool, such as RSM/RSA, Rhapsody, Enterprise Architect and Magic Draw.

The Topcased project is supported by partners from various organisations such as, Airbus France, AdaCore, AtoS, EADS Astrium, ESEO, Institut National de Recherche en Informatique et en Automatique (INRIA), OBEO and Thales. The Topcased project also collaborates with Eclipse and OMG.

A.6.3 IBM RATIONAL RHAPSODY

Rhapsody is a visual development environment for systems engineers and software developers creating real-time or embedded systems and software. Rational Rhapsody uses graphical models to generate software applications in various languages including C, C++, Ada and Java.

Rational Rhapsody helps diverse teams collaborate to understand and elaborate requirements, abstract complexity visually using industry standard languages (UML, SysML, AUTOSAR, DoDAF, MODAF, UPDM), validate functionality early in development, and automate delivery of high quality products.

A.6.4 YAKINDU

Yakindu [182] is an open source toolkit based on the EMP. Yakindu provides an integrated modelling environment for the specification and development of reactive systems using the Statechart Tools (SCT) [183]. SCT support the specification and the development of reactive, event driven systems based on the concept of statecharts (state machines). Yakindu also provides a set of modelling modules such as, graphical and textual editors, simulators and code generators. Recently, Yakindu has been updated with the Dynamical Systems Modeler (Damos). Damos focuses on the development of data flow-oriented systems.
The **SAVY** [75] is built on top of the C++ standard library, SystemC library, Xerces library [123] and the Xalan library [124]. **SAVY** library extends SystemC capabilities providing an actor architecture pattern and an entity architecture pattern to easily translate Unified Modeling Language (UML) based system descriptions into Transaction Level Modeling (TLM)—SystemC executable models.

During the first stage of a system design flow the system-design team creates a UML description of the system to be developed. **SAVY** can be used to easily create a SystemC TLM executable model from the system behavior capture carried out using UML Use Case diagrams and UML Statechart diagrams.
For the conception and definition of a system, it is common to start with UML use case diagrams. These diagrams are used as a tool to identify both the actors and the use cases. At this stage, it is also possible to define the transactions between the actors and the system. The next step is to try to determine the behaviour of the system in each use case. A common approach to capture the behaviour corresponding to a use case is a UML state machine diagram. The state machine is a complete description of the behaviour of the system for a use case. This approach is summarized in Figure B.1.

At this point designers would like to validate their concept of the system against the specification. This validation is carried out by defining stimuli for different scenarios and analysing the state machines to obtain the response of the system. An executable model approach can be very helpful. SAVY proposes structured procedure to go from the former UML diagrams to the SystemC executable model used during the design and verification of the system.

In order to study the system behaviour, stimuli are applied to the design and record the responses produced by the design. The external actors of the UML use case diagram apply stimuli and receive the system response. The actors inject and receive transactions. SAVY provides a basic class for the transaction, which is based on the SystemC TLM 1.0 standard. This class implements the features needed to handle the transactions within SAVY: a
transaction identifier, the time instant in which the transaction is injected and the time instant in which the transaction is recorded. When a type of transaction is needed for the study of a specific system, a new class of transaction is derived from the base transaction class. This derived class adds the application specific information in the form of new attributes.

SAVY can be used to easily create a SystemC executable golden model from the system behaviour capture carried out using UML diagrams. The following design patterns are proposed to model the environment (actors) and the system:

- Entity modelling: entity architecture pattern and state machine pattern.
- Actor modelling: actor architecture pattern and file format pattern for transactions.

These design patterns are described in detail in subsequent section.

B.2 ENTITY MODELLING USING SAVY

B.2.1 ENTITY ARCHITECTURE PATTERN

Figure B.2 shows a block diagram of an entity using the SAVY entity architecture pattern. An entity can be used to represent the system or a module within the system. The entities are reactive to external stimuli. For each actor, input port is defined if the actor stimulates the entity and/or an output port if the entity produces a response to that actor. These ports use the methods of the interface transactor_if. This interface is compliant with TLM 1.0. The SAVY entity architecture pattern comprises the following elements:

**Transactional Input Interface (TII):** The TII collects the stimuli received at its input port and broadcasts an event through the Event Communication Channel (ECC) to the different Use Cases. This element is built using a First In First Out (FIFO) and two sc_threads (SystemC threads). One of the sc_threads monitors the arrival of new transactions through the input port. When a new transaction arrives, it performs a blocking read, inserts the transaction in the FIFO and sends an event to the second sc_thread. The second sc_thread, reacts to this event broadcasting a
new event through the ECC. The second sc_thread remains blocked until an Acknowledgement (ACK) is received from the use cases. The acknowledgement mechanism is used to ensure that all the UCs have consumed the event. If new transactions arrive while the second sc_thread is blocked, they are put in the FIFO. When the second sc_thread receives the first ACK, it checks if there are more transactions in the FIFO and processes them until the FIFO is emptied. This way, no transaction is lost.

**Event Communication Channel (ECC):** This element works as a hub. Any event put in this channel is replicated so that all the UCs are notified. An interface called EvtCommsChannel_if has been defined. This interface is implemented in the ECC.

**Use Cases (UC):** There is a correspondence between this element and the use cases of the UML diagram. The UC is a sc_module (SystemC module) whose ports are of the class EvtCommsPort that derives from sc_port (SystemC port) and uses the interface EvtCommsChannel_if. These UCs are implemented by means of state machines. The reaction of each state machine to an input stimulus will depend on the Context and on the current states of the state machines. There is one state machine per use case. The design pattern and the tools provided by SAVY for the implementation of the UML state diagrams are described in subsection B.2.2. The reac-
tion of the UC will consist on a state transition and a set of actions to be performed by the entity.

**Actuator:** The actuator is responsible for carrying out the actions requested by the UCs. These actions can be the execution of operations that modify the Context, the raise of events or the production of output transactions. The execution of these actions can be untimed or timed. The entity architecture pattern defines an interface `actuator_if`. This interface declares the actions (methods) that the entity can perform. These actions are implemented in the entity. Thus, the entity is a class derived from both `actuator_if` and `sc_module`.

**Context:** The Context is implemented as a container class, which stores data for the entity. This context class also defines the methods that Actuator uses to access the data.

**Transactional Output Interface (TOI):** The TOI produces transactions to the environment of the entity. This element comprises a FIFO and a `sc_thread`. When a UC is to produce an output transaction, the UC uses the corresponding Actuator method to write to the FIFO and to generate an event for the `sc_thread`. When the `sc_thread` receives the event, it performs a blocking write in the output port.

### B.2.2 STATE MACHINE PATTERN

The behaviour associated to each UC is described as a state machine. In the proposed pattern, each UC has to keep track of two things: the current state and the received event. The class UC contains a pointer to the current state and declares a method called `ProcessEvent`.

Each particular UC employs the following technique to identify the received event. For each event the UC is sensitive to, there is a `sc_thread` sensitive to that event. When an event arrives, the `sc_thread` sensitive to that event is activated. Then, the `sc_thread` of that particular UC calls the method `ProcessEvent`, sending as parameter an event identifier. This method calls the method of the current state responsible for processing the event. This idea is illustrated in FigureB.3.

The state base class defines a virtual method `ProcessEventState`. Each particular state implements this method so that it can request the corresponding actions and return the name of the next state. When the method
ProcessEvent of UC receives the name of the next state, it deletes the current state and creates a new object of the class of the next state.

The class state receives in the constructor a pointer to the Actuator. Thus, state can access all the methods of Actuator to perform actions.

B.3 ACTOR MODELLING USING SAVY

SAVY proposes patterns and provides tools to create elements in the executable model that play the role of the actors. These elements inject stimuli to the design and they receive (and record) the response of the system.

B.3.1 ACTOR ARCHITECTURE PATTERN

SAVY proposes to use the pattern shown in Figure B.4 to implement the actors of the UML use case diagram in the SystemC executable model. As the figure shows, it is proposed to use eXtensible Markup Language (XML) files for the description of both the transactions to be injected and the transaction to be recorded. The XML Transaction Generator (XTG) and the Traffic Injector (TI) blocks are used for the injection of the transactions. The XML Transaction Recorder (XTR) and the Traffic Receiver (TR) module are used in the recording of the transactions. The elements of the proposed actor architecture pattern are described below:
B.3 Actor Modelling Using SAVY

Event Communication Channel (ECC): This element works as a hub. Any event put in this channel is replicated so that all the UCs are notified. An interface called `EvtCommsChannel_if` has been defined. This interface is implemented in the ECC.

XML Transaction Generator (XTG): The XTG is a class that implements the method `GetNextTransaction` to produce transactions from the XML file. The implementation exploits the data introspection utilities of the SystemC Verification (SCV) library [121] to build the transactions in runtime. The XML file access is based on the Xerces library [123].

Traffic Injector (TI): The TI is a `sc_module` with two ports. The port connected to XTG uses the method `GetNextTransaction` provided by the interface `TG_if`. The port connected to the transactor uses the interface `transactor_if`. The TI has a single `sc_thread` that request transactions using `TG_if` and performs a blocking write on the other port using the `transactor_if`.

Transactor: The transactor unit is depicted with dashed lines in Figure B.4, because it can be present or not depending on the abstraction level of the model of the entity, the abstraction level of the transactions and the particularities of the system under study. This transactor unit implements abstraction level translations between its input and its output. The transactor can also be used to introduce some degree of actor reactivity.
Traffic Receiver (TR): The TR is a `sc_module` with two ports. The port connected to the transactor uses the interface `transactor_if`. The port connected to the XTR uses the method `RecordTransaction` provided by the interface `TR_if`. The TR has a single thread that performs a blocking read on the transactor port. When a transaction arrives, the transaction is recorded using the method `RecordTransaction`.

XML Transaction Recorder (XTR): The XTR class implements the method `RecordTransaction`. This implementation writes the transactions to an XML file. Although SAVY provides classes to handle XML files, the structure of the transaction injection and recording patterns is independent of the format of the transaction repository file. Therefore, it would be very easy to extend SAVY to work with other file formats.

B.3.2 FILE FORMAT FOR TRANSACTIONS

For directed tests, the engineer must specify the stimuli to be applied. It is proposed to use XML files for this purpose. Each transaction is written as an XML node, which has as XML attributes the base transaction class attributes plus the specific attributes of that type of transaction. The engineer writes the transactions to be injected by one actor in one XML file.

Note that the XML node corresponding to a transaction contains an attribute that is the desired injection time. The patterns proposed to model the entity support both timed and untimed modelling. The injection of transactions also supports both concepts: an increasing value of the injection time produces a timed behaviour and an injection time equal to zero for all the transactions will produce an untimed behaviour.
APPENDIX C

CONFIDENCE Project

In the following a brief summary of the CONFIDENCE project is presented as described in the Description of Work (DoW), which was the input for the design and implementation engineers.

The main objective of this proposal is the development and integration of different Information and Communications Technology (ICT) technologies for the detection of abnormal events, such as falls, losses of consciousness or unexpected behaviours that may be related to a health problem in elderly people. This care system will be able to interpret situations both indoors and outdoors and raise an alarm or issue a warning when necessary. As a consequence, the elderly people will gain confidence and security and they will have a better quality of life and a longer active participation in the social life. Not only will the elderly people benefit from the system, but also the people around them, namely family and caregivers, since the burden on them will be substantially reduced.

The idea is to create a care system that will work both outdoors and indoors. This care system will be able to reconstruct the user’s posture.
and detect abnormal situations, such as falls or loss of consciousness. The propose system will raise an alarm if an abnormal situation is detected. This care system will also be able to detect changes in the user’s behaviour and issue a warning. For instance, if the system notices changes in the user’s gait that may involve a lack of stability, the CONFIDENCE system will warn the user about an increased risk of falling, and prevent an accident. Detection of anomalous behaviour will utilise prior expert knowledge as well as learnt movement patterns of particular users.

The target group of CONFIDENCE have the following characteristics:

- Elderly over 65 years
- Cared or not by some kind of home assistance provided by Public Administration (Municipality or National Health Service)
- Mobility independent and with no particular difficulty with ADL (activities of daily living)
- With fear of falls
- At risk of social exclusion

CONFIDENCE aims to detect abnormal situations that may affect the daily activities of the elderly and raise an alarm, if necessary.

An important characteristic of CONFIDENCE is that the system performance will improve as it is used. Any alarm or false alarm will be taken into account for similar situations in the future, improving the system “intelligence” and making it easier to interpret the user’s behaviour. The system will adapt itself to the user and it will become more accurate and its efficiency will increase as time wears on.

One of the goals of the proposed care system is to introduce only small and low cost changes in the user’s home. The system must be as simple as possible from the user’s point of view. A complex system would cause reluctance among the users, who would regard it as a problem rather than as a solution. Therefore, the CONFIDENCE system will be easy to setup and to use, so that it does not limit the user’s daily activity in any way. As it can be seen in Figure C.1, the system consists of a central device, which plays the role of a base-station (BS), a small portable device (PD), which looks similar to a mobile phone, and several tags.
The base-station will be placed inside the house and could be designed to look like a decorative item. It will be able to determine the position of each tag in the three dimensional (3D) space.

The portable device, which will automatically enter in a setup mode, will guide the user during the installation process and will indicate him/her where to place the tags. This is one of the innovative aspects of this solution, since the user can easily install the system by himself/ herself without any external help. The CONFIDENCE system is intuitive, easy to setup, and user-friendly, and, thus, the user feels independent since the beginning.

The user will have to wear small size and low cost tags, either in the form of bracelets to be worn in wrists and ankles or in the form of necklaces. If the user prefers it, the tags may be sewed into the clothes (socks, underwear, etc.). These tags will be easy to wear and, due to their small size, easy to hide. Additionally, some tags can be placed in specific positions such as the corners of the bed, chairs or some other pieces of furniture. These tags on the furniture will make it possible to distinguish situations such as the user lying in bed or on the sofa. The system will be technically tested both with and without furniture tags. The latter version is expected to be more reliable, but body tags should be enough for basic operation.

Indoors, the base-station will be able to determine the position of each tag. Based on this information, the system will reconstruct the posture of the body and decide if the user has either suffered a fall or is acting...
abnormally. When a fall or an atypical situation is detected, the system raises an alarm.

When leaving the house, the user takes the portable device with him/her, as shown in Figure C.1(b). In case the user leaves the house and forgets the device, the system will detect this event and will warn him/her. The portable device will use electronic compasses based on magnetic or inertial sensors to determine its orientation and will determine the 3D relative positions of the tags the user will be wearing. Combining this information, the portable device will be able to reconstruct the posture of the body. Additionally, this portable device comprises a Global Positioning System (GPS) receiver and a mobile phone. Hence, the portable device will always know where the user is.

Both the base-station and the portable device can raise an alarm or a warning. The base-station will have priority in indoor environments. Typically, the CONFIDENCE system follows the following alarm protocol: First, it makes a phone call to the user. If the user picks up the phone, he/she is requested to indicate that he/she feels well by pressing a sequence of buttons or telling a certain word. This stops the alarm. This way, the user keeps control of the system, which is an important feature and considerably reduces the false alarm rate. If the user does not pick up the phone, the system makes a phone call to a series of relatives or friends. If nobody answers the phone, the system calls the emergency services. In both cases, the system will explain to the call recipient the reasons of the alarm or warning.

The user will be able to customise the alarm protocol and will select the time that the system waits for him to answer, before triggering the alarm. Additionally, the user will be able to allow the system to transmit all the available information about the situation: localisation, posture and how long the user has been in that posture. The recipient of the emergency call will receive this information as a multimedia message (MMS). This information will help to react to the emergency call.

Figure C.2 shows a detailed description of the CONFIDENCE system. In order to simplify the development of the system, the base-station and portable device have been divided into four subsystems:

Localisation subsystem: This subsystem relies on radio technology and performs two tasks: identification and localisation of the tags. It provides
Figure C.2: Detailed CONFIDENCE system description
the position of each tag with an accuracy of a few centimetres. Two local-
isation subsystems will be built; the first one will work indoors and will be
placed in the base-station. The second one will work outdoors and will be
located in the portable device.

Reconstruction subsystem: This second subsystem receives the esti-
mates of the positions of the tags and generates a model of the user and
the environment. Two reconstruction subsystems will be developed; the
first one will be placed in the base-station and will work indoors. The
second will be located in the portable device and will function outdoors.

Interpretation subsystem: The third subsystem interprets this data to
make a decision about the situation. This subsystem will be provided with
"intelligence", so that it can learn from the user’s habits and help to detect
eyeal symptoms of illness, such as Parkinson, spine cancer, etc. An indoor
interpretation subsystem will be placed in the base-station and an outdoor
interpretation subsystem will be located in the portable device.

System interface subsystem: The fourth subsystem is the system in-
terface that is responsible for the user interface, system setup and alarm
handling. The system interface of both the base-station and the portable
device will be developed. The system interface subsystem of the portable
device will be able to work indoors and outdoors. However, the system
interface subsystem of the base-station will only work indoors.
APPENDIX D

Research Results

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.1 International Journal Papers</td>
<td>321</td>
</tr>
<tr>
<td>D.2 International Conference Papers</td>
<td>353</td>
</tr>
<tr>
<td>D.3 National Conference Papers</td>
<td>377</td>
</tr>
</tbody>
</table>
D.1 INTERNATIONAL JOURNAL PAPERS


A MDD approach to generate heterogeneous SystemC–TLM executable models from SysML – Part I: Behavioral Mapping Formalization
(Submitted).
A MDD approach to generate heterogeneous SystemC–TLM executable models from SysML – Part I: Behavioral Mapping Formalization

Koldo Tomasaera, Juan F. Sevillano, Member, IEEE, and Ignacio Vélez, Member, IEEE

Abstract—Recently, the Model-driven Design (MDD) methodology has emerged as a methodology for complex electronic systems. However, existing development tools cannot be applied within an MDD environment to automatically produce SystemC–TLM executable models from systems described using visual modeling languages such as SysML. Although this approach is promising, visual modeling languages with clear semantics are required in order to implement effective code generation frameworks. Besides, executable models with clear models of computation (MCs) are also required to uniformly and effectively integrate existing software and hardware components. This paper focuses on defining formal semantics for Statechart diagrams and Activity diagrams of SysML. Semantic mappings are provided in order to translate Statechart diagrams and Activity diagrams into Finite State Machine (FSM) and Synchronous Data Flow (SDF). MCs. This paper establishes the necessary background for a successful framework implementation. An associated paper proposes an MDD framework that implements the semantic mappings defined in this paper and describes the generation of SystemC–TLM source code from a system model described using SysML.

Index Terms—Electronic system level (ESL), Model-driven design (MDD), High abstraction level, SysML, structural modeling, behavioral modeling, formal semantics, models of computation (MCs), SystemC, Transaction-level modeling (TLM).

1 INTRODUCTION

Modern electronic systems have arrived to a degree of complexity that makes traditional design methodologies inefficient and error-prone. In order to address the complexity, electronic system level (ESL) design tools and languages are demanded to enable the creation of high level executable functional descriptions of the system being designed [1]. In this context, the SystemC [2] language has become the de facto standard to create executable models of electronic systems. SystemC is an ANSI standard C++ class library for systems design. SystemC provides resources to model electronic systems at a wide range of abstraction levels, from the register-transfer level (RTL) up to the high abstraction level Transaction-level modeling (TLM). TLM [3] is being proposed for early stage development, architecture exploration, software development and platform design.

On the other hand, the last decade, Model-driven Design (MDD) has emerged as a development methodology for software engineering [4, 5]. The main proposition of MDD is to capture in visual models all the information of the system being designed. In MDD, the design products are visual models instead of source code. The purpose is to promote intuitive visual design patterns that raise the abstraction to a higher level and improve the communication between engineer teams developing the system. Moreover, the use of standard modeling languages guarantees the compatibility between different modeling frameworks. Thus, the Unified Modeling Language (UML) [6] standard was conceived by the Object Management Group (OMG) as an object-oriented software modeling language applicable within the MDD methodology. The MDD approach is not exclusively limited to pure software development. In order to extend the benefits of this approach to other domains, the International Council on Systems Engineering (INCOSE) in close collaboration with OMG, defined a general-purpose modeling language for systems engineering: Systems Modeling Language (SysML) [7]. SysML is defined as an extension of a subset of UML and is UML. It defines a graphical notation supported by modeling diagrams that allows to create visual models of systems under development.

In recent years, MDD has been proposed as a promising methodology for electronic systems modeling and design [8, 9]. Model-driven (MD) and Model-based (M2T) techniques can be used to create SystemC–TLM code from visual modeling languages such as UML or SysML. However, some requirements shall be fulfilled in order to effectively use these techniques. On one hand, visual modeling languages with clear semantics are required. The lack of formal semantics makes difficult the acceptance of visual modeling languages by developers and complicates the exchange of complex models among different tools [10]. On the other hand, the model of computation (MC) of the generated executable model shall be clearly de-
fixed. MoCs, together with design languages, provide the foundation for defining system behavior [11]. The well-defined communication semantics of MoCs allow the unambiguously capture the required functionality and enable the application of formal design and verification techniques [12]. The automation tools for electronic system development can only be applied if the behavioral semantics and the corresponding MoCs are well-defined.

The objective of this paper and its associate [13] is to propose a MoC-based method to translate electronic systems described using SysML into SystemC-TLM executable code. The SystemC-TLM executable code supports heterogeneous behaviors both state-based behaviors and data flow behaviors can be implemented. The main goal of this paper, Part I, is to formalize the semantics of SysML State Machine diagrams and Activity diagrams using MoCs as semantic domains. Rigorous semantic mappings are provided in order to translate the abstract syntax of State Machine diagrams and Activity diagrams to Finite State Machine (FSM) and Synchronous Data Flow (SDF) MoCs, respectively. The Part II addresses the generation of SystemC-TLM source code from a system model described using SysML. The proposed approach is supported by Eclipse-based tool called ASSIST (Automatic SystemC to SystemC Translator) that automates the SystemC-TLM source code generation using M2M and M2T techniques. ASSIST implements the semantic mappings defined in Part I to support the creation of heterogeneous executable models of electronic systems. Additionally, Part II presents a case study where the applicability of the tools is demonstrated and the proposed design flow is assessed.

The rest of this paper is organized as follows. Section 2 discusses some related work in the literature and points out the objectives of the proposed approach. Section 3 presents the preliminaries of our approach and introduces the proposed workflow. Then, Section 4 and Section 5 describe the formal semantics of SysML State Machines and Activities by means of FSM and SDF models of computation, respectively. Finally, Section 6 summarizes the conclusion of this paper.

2 RELATED WORK

Many contributions in the literature focus on electronic systems design methods involving graphical modeling languages and SystemC language. The objective of these works is to generate SystemC executable code from a graphical system model described using UML or SysML. ASSIST, the tool defined in YAVML [14], which only uses UML, to capture the structural aspects of the system under design. No behavior is captured and thus only a structural snapshot of the design is obtained. Executable code is generated from certain structural aspects.

Several works in the literature proposed similar approaches to produce SystemC code from UML models. Three works that represent three different approaches of creating UML profiles to model specifically SystemC elements and modeling system behaviors with state machine diagrams in [15]: the authors define an UML profile to model SystemC elements. The behavioral aspects of UML models are translated to SystemC in this work. Kusubone et al. presented several contributions focusing on a MDD methodology based on UML, UML profiles, and SystemC, to model embedded systems and generate executable code [16], [17], [18]. The authors proposed UML profiles to support SystemC constructs for HW/SW co-designs. The behavioral aspects of the system under design are only captured using state machine diagrams. There are also several works that model behavioral aspects by means of state machine diagrams [19], [20], [21]. These contributions are also focused on UML-to-SystemC methodologies proposing specific UML profiles defining SystemC elements.

Previously described works do not address the generation of SystemC-TLM executable code. MDD methodologies generating TLM models are required to fill the gap between the system specifications and the TLM abstraction level. Moreover, these research works do not define the MoC of the generated SystemC code. The UML-to-SystemC approaches presented in recent years are focused on generating only control-based applications using state-oriented modeling techniques by means of UML. State Machine diagrams. Modern electronics systems are often composed of control-based parts and signal processing oriented parts, but the modeling requirements for control-based applications and signal processing applications are very different. Control-based parts are efficiently modeled by finite state machines while signal processing algorithms are better described using data flow style representations. This heterogeneity is supported within UML using State Machine diagrams for state-oriented control modeling and Activity diagrams for data flow behavior models. The presented research works lack support for modeling heterogeneous behaviors, and thus, the practical use of these tools within an electronic system development process is very restricted.

While UML has been widely criticized for being software-centric, SysML offers a promising perspective for a high abstraction level electronic systems design methodology [22]. SysML simplifies UML by discarding some diagrams and providing new ones for general systems engineering. As a result, SysML is better suited for high abstraction level functional electronic system modeling and contributes to the applicability of MDD techniques for electronic design. In recent years, several works have been proposed that focus on creating SystemC code from SysML.

Rashid et al. proposed one of the earliest approaches addressing SystemC code generation from SysML [23]. SysML models specified in Rhapsody are translated into SystemC, however, mapping rules are presented informally and only a few SystemC code snippets are presented. Although the proposed method supports the generation of SystemC code from SysML, State Machine
diagrams and Activity diagram, no MoC is specified and thus, the behavioral semantics of the generated code remains unknown. Moreover, the approaches proposed by the authors do not create UML executable models.

A work involving SysML is found in [25]. This work provides a study of the SystemC code generation capabilities of SysML and MARTE, but it does not present any formal semantics. An informal, example-based mapping, is provided only for structural features of the system being designed using SysML or MARTE. The code generation process for behavioral features is not provided, the presented methodology only generates a SystemC code skeleton.

In [26–28] the SATURN UML profile for SW/HW co-modelling is presented. The SATURN profile is composed of three subprofiles: a synthesizable SystemC profile to model HW components, a synthesis profile to translate SystemC models into VHDL, and a C profile to model the embedded SW. These SysML-based proposals model behavioral aspects by means of UML Activity diagrams, but they do not define the MoC of the generated code and they do not address clear behavioural mappings. Muller et al. [28] proposed an additional UML profile for SATURN including stereotypes representing different MoCs such as Kahn Process Network (KPN), Bounded Kahn Process Network (BKPN) and Synchronous Data Flow (SDF). In another recent work [29], systematic studies of the potential MoCs for electronic systems and their formalisms are described. Previously introduced works mainly selected UML as the graphical modelling language for electronic systems design. However, the UML standards [30, 31] concentrate on syntax and do not offer a rigorous definition of the semantics. This issue is specially significant in behavioral aspects where the UML standards provide explanations about the semantics in informal natural language and lack many details. Although the UML profile mechanism proposed by the works in the literature is flexible and powerful, it does not provide a method for precisely defining semantics associated with extensions [32]. Recently, some works in the literature focused on using SysML as the electronic systems modelling language. SysML focuses on the efficient specification of complex systems, but it also lacks formal semantics, as its standard [7] raises the concern of constructs of UML. The lack of formal semantics makes difficult the acceptance of visual modeling languages for developers and complicates the exchange of complex models among different tools [10].

3 PROPOSED APPROACH

In this research work we propose a MOO approach to translate electronic systems described using SysML to SystemC UML heterogeneous executable models. In order to effectively implement a design automation tool that supports this MOO approach, well-defined behavioral semantics and MoCs are required.

Using the modeling language concept introduced by [33] and [34] we will assume the SysML modeling language $\mathcal{M}$ as a sample

$$(\mathcal{A}, \mathcal{C}, \mathcal{S}, \mathcal{M})$$

SysML language consists of an abstract syntax ($\mathcal{A}$), a concrete syntax ($\mathcal{C}$), a semantics domain ($\mathcal{S}$), a syntactic mapping ($\mathcal{M}_A$), and a semantic mapping ($\mathcal{M}_S$). The abstract syntax $A$ defines the concepts of the SysML language and their relationships. The concrete syntax $C$ defines the physical appearance of the language, that is, its notation. The syntactic mapping

$M_A: \mathcal{C} \rightarrow \mathcal{A}$

maps language notation elements to abstract syntax elements. The semantics describe the meaning of a model specified in SysML language. The semantics of the modeling language is given in terms of semantic domain $S$. The semantic mapping

$M_S: \mathcal{A} \rightarrow \mathcal{S}$

assigns abstract syntax elements to elements of the semantic domain so that each syntactic construct is mapped to its meaning.

The OMGs SysML standard specification [7] is defined as an extension of the OMGs UML v2 superstructure specification [6]. Both OMGs standards and works in the literature deeply discussed the syntax of the SysML modeling language. The concrete syntax $C$ and the abstract syntax $A$ are defined using graphical notations of the language concepts and metamodeling techniques, respectively. The semantic mapping $M_S$ relating the graphical notation and the elements of the abstract syntax is clearly described. However, those documents only provide an informal description of the semantics in natural language.

In this paper, we focus on the definition of both formal semantic domain and semantic mapping of SysML behavioral constructs specifically State Machine diagrams and Activity diagrams. Section 4 deals with the formalization of the SysML State Machine diagrams using Finite State Machine (FSM) MoC as the semantic domain. First, we define an abstract syntax $\mathcal{A}_{StateMachine}$ for State Machine diagrams. Then, we propose a semantic domain $\mathcal{S}_{FSM}$ and a semantic mapping

$\mathcal{M}_{FSM}: \mathcal{A}_{StateMachine} \rightarrow \mathcal{S}_{FSM}$

On the other hand, Section 5 presents the formalization of the SysML Activity diagrams using Synchronous Data-Flow (SDF) MoC as the semantic domain. We define an abstract syntax $\mathcal{A}_{Activity}$ for Activity diagrams and we propose a semantic domain $\mathcal{S}_{SDF}$ and a semantic mapping

$\mathcal{M}_{SDF}: \mathcal{A}_{Activity} \rightarrow \mathcal{S}_{SDF}$

The associated paper part II [35] describes the AS-SYST framework. ASSYST implements the semantic mappings $\mathcal{M}_{FSM}$ and $\mathcal{M}_{SDF}$ using model-to-model
APPENDIX D: Research Results

4 FORMAL SEMANTICS FOR SYMFL STATE MACHINE DIAGRAMS

State Machine Diagrams define a set of concepts that can be used to model discrete behavior through finite state transition systems. The state machine represents a event-driven behavior. SysML restricts the StateMachine concept defined in UML, while UML supports both BehaviorStateMachines and ProtocolStateMachines types. SysML restricts the UML’s state machine abstraction syntax to BehaviorStateMachines. In this section, we will discuss the semantics of the State Machine diagrams in SysML.

4.1 Abstract syntax of State Machine Diagrams

The SysML standard [7] reuse the abstract syntax defined for the BehaviorStateMachine package in the UML standard [6]. A simplified version of the abstract syntax of the BehaviorStateMachine package is shown in Figure 1.

A StateMachine element is composed of one or more Region elements. Regions are orthogonal behavioral parts that contain Vertex elements and Transitions connecting various vertexes.

A Vertex is a generalization of State and Pseudostate elements. States are also allowed to contain Regions. This feature leads to several kinds of states: simple states, composite states, and submachine states. A simple state does not have any regions and it does not contain any references to a substate machine. On the contrary, a composite state contains one or more regions. A submachine state includes a reference to a substate machine. A FinalState is a special state that represents the termination point of the enclosing Region. Pseudostate elements model transient various such as the initial vertex of a StateMachine.

The Vertex elements of a StateMachine are connected using Transitions. A StateMachine traverses its vertexes based on the input events detected. Transitions are fired by events. The Trigger attribute of a transition specifies which event can fire the transition between two Vertex elements.

Each region of the state machine can only have one initial pseudostate vertex. The initial vertex is connected to the default state of the StateMachine through a single transition. This transition is not allowed to have a trigger.

During a StateMachine execution, a State can be active or inactive. States have incoming transitions and outgoing transitions. When an outgoing transition is fired, the state is exited and it becomes inactive. When an incoming transition is fired, the state is entered and it becomes active. As Transitions are connections between two states, whenever a transition is fired the source state becomes inactive and the target state becomes active.

There can be at most one active State in a Region at any instant during the execution of the StateMachine. When the StateMachine is executed for the first time the default state becomes the active state.

We assume a mathematical representation of a State Machine Diagram using a graph theory notation. We will focus on basic state machines with a single region and no composite states.

Definition 1: The abstract syntax of a State Machine

A StateMachine is an ordered pair of sets

\( \text{StateMachine} = (V, T) \)  

where \( V \) is a finite set of state machine Vertex elements, and \( T \) is a set of Transitions. The use of Regions is deviated as we are focusing on state machines with a single region. Thus, State Machines consist of Vertex and Transition elements.

Definition 2: Vertex are partitioned into States and Pseudostates

\( V = S \cup P \)  

where \( S \) is a finite set of States, and \( P \) is a finite set of Pseudostates.

Definition 3: A vertex \( v \in V \) has a set of incoming transitions and a set of outgoing transitions

\( \forall v \in V, \text{incomings}(v) \subseteq T \)  

\( \forall v \in V, \text{outcomings}(v) \subseteq T \)

Definition 4: The function \( \text{label}(v) \) determines the precise type (PseudostateKind) of the Pseudostate \( p \in P \),

Definition 5: A Transition \( t \in T \) has a trigger \( \text{by} \in TEG \)

\( \text{trigger}: T \rightarrow TEG \)
4.2 Semantic Domain of State Machine Diagrams

We propose to use the Finite State Machine (FSM) model of computation as the semantic domain $S_{FSM}$ for the State Machine diagrams of SysML.

A Finite State Machine (FSM) is an event-based MoC suitable to describe state-oriented models such as sequential control logic.

An FSM consists of an input event alphabet, a set of states and a set of transitions between these states. During its execution, an FSM receives input events and processes them while traversing its states.

The relation between states and transitions is specified by a transition function. The transition function describes the allowed transitions between states. During the execution of an FSM, it processes input events in order to fire transitions and traverse states.

**Definition 7** A FSM is a 4-tuple

$$S_{FSM} = (Q, q_0, \Psi, \Delta)$$

where

- $Q$ is a finite set of symbols denoting states
- $q_0$ is an initial state of the FSM, $q_0 \in Q$
- $\Psi$ is a set of symbols denoting the possible input events (input alphabet)
- $\Delta$ is a transition function $\Delta : Q \times \Psi \rightarrow Q$

The execution of a FSM starts at the state $q_0$. A FSM processes the incoming events $\psi \in \Psi$ one at a time in order to determine in which order the states will be traversed. Each event is processed in a runtime step. A runtime step can be launched only if the previous runtime step has been successfully finished. First, a runtime step selects an enabled transition. A transition $(q_0, \psi)$ can be enabled in the current state only if $\psi$ is the event processed in the current runtime step. In order to guarantee the determinism of the FSM execution, there may be at most one enabled transition. If there is no enabled transition, the runtime step terminates. If an enabled transition is found, the transition’s target state $q_f$ is activated, and the runtime step is terminated.

**Definition 8** The execution of a FSM is a sequence of states

$$q_0 \xrightarrow{\Delta} q_1 \xrightarrow{\Delta} q_2 \xrightarrow{\Delta} \cdots$$

where $q_0$ is the initial state of the FSM, $q_i \in Q$ and $\Delta_i$ is a transition $(q_i, \psi_i) \xrightarrow{\Delta} q_{i+1}$ that goes from $q_i$ to $q_{i+1}$.

**Definition 9** A transition $q_i \xrightarrow{\Delta} q_{i+1}$ is enabled if the following condition is satisfied

$$(q_i, \psi_i, q_{i+1}) \in \Delta$$

**4.3 Semantic Mapping of State Machine Diagrams**

The semantic mapping from a State Machine diagram to an FSM is straightforward.

State (S) elements from SysML. State Machines are translated to $Q$ elements of the FSM model of computation. A StateMachine can only have one initial pseudostate vertex which is translated to the initial state $q_0$ of a FSM graph. The set of input events $\Psi$ is populated with the events that trigger the Transitions (T) of a StateMachine. The transitions (T) of a StateMachine are translated into the FSM transition function $\Delta$. The triggers and guards of Transitions are directly related to the sets $\Psi$ and $\Sigma$ of FSMs.

**Definition 10** The transformation from a State Machine diagram to an FSM is a function $M_{FSM}$ that

$$M_{FSM} : AStateMachine \rightarrow S_{FSM}$$

where

$$Q = S$$

$$q_0 = \{p | kind(p) = InitialState, p \in PS\}$$

$$\Psi = \{\text{trigger}(t) \mid t \in T\}$$

$$\Delta = \{(source(s), trigger(\psi), target(t)) \mid t \in T\}$$

5 FORMAL SEMANTICS FOR SYSML ACTIVITY DIAGRAMS

Activity Diagrams represent flow-based behavior through the execution of a sequence of actions. Activities define the transformation of input data into output data. Activities can describe the behavior of blocks or parts. An activity may be specified as the main behavior of a block that describes how input data of the block are transformed into outputs. In this section, we discuss the semantics of the Activity Diagrams in SysML.

5.1 Abstract Syntax of Activity Diagrams

The SysML standard [8] reuses the abstract syntax defined for the BasicActivities package in the UML standard [6]. Activity diagrams consist of two primary elements ActivityNodes and ActivityEdges as shown in the abstract syntax of the BasicActivities package depicted in Figure 2.

An Activity represents a data flow behavior where ActivityNodes are connected using ActivityEdges. An Activity specifies the data transformation functions, input data is transformed into output data. This data transformation function is performed by the individual execution of ActivityNodes which are basic data transformation units. Data communication in an Activity diagram is token-based. Tokens are elemental information units controlled by ActivityNodes and exchanged between ActivityNodes.

The ActivityNode element is categorized into three types: Actions, ObjectNodes and Constraints. There are
two types of ControlNodes: InitialNodes and ActivityFinalNodes. ObjectNodes has two subtypes: Pins and ActivityParameterNodes. In turn, ActivityEdges consists of ControlFlows and ObjectFlows as depicted in Figure 2. Actions are the main executable nodes and they can be connected through ControlFlows or ObjectFlows.

ControlFlow is used to establish the execution order of Action nodes and ControlNodes through the communication of control tokens. The InitialNode sets the starting point of the execution while the ActivityFinalNode is the last node of the control flow. Tokens emitted by the source node are all offered to the target node.

On the other hand, ObjectFlows describe the flow of data between ObjectNodes. Pins are attached to Actions in order to receive or transmit data tokens. Pins are interfaces to allow Actions to interoperate data tokens with other Actions or ActivityParameterNodes. There are input and output Pins depending on whether they are connected to incoming or outgoing ObjectFlows. An Action consumes the data tokens placed on its input Pins, processes them, and places the outgoing data tokens on its output Pins.

An Action is executed when all its ObjectFlow and ControlFlow requirements have been satisfied. A ControlFlow requirement is satisfied when each of the flows is offered one token. An ObjectFlow requirement is satisfied when all of the input Pins are offered all necessary tokens, as specified by their minimum multiplicity, and accept any tokens as specified by their maximum multiplicity. When an Action is completed, it emits any object tokens that have been placed on its output Pins and control tokens on all its outgoing ControlFlows.

ActivityParameterNodes are used to specify input and output Activity parameters. Using ActivityParameterNodes, the designer can model a scenario where an external agent transmits data into or gets data from an Activity.

Similarly to State Machines, we assume that the abstract syntax of an Activity is described using a graph theory notation as an ordered pair of sets. We propose modeling process based on the BasicActivities level of Activity Diagrams. The BasicActivities package supports control sequencing and data flow between actions, but forks, joins, decisions and messages are not supported.

Definition 1: The abstract syntax of an Activity $A_{activity}$ is an ordered pair of sets

$$A_{activity} = (AN, AE)$$

where $AN$ is a finite set of ActivityNodes and $AE$ is a finite set of ActivityEdges.

Definition 2: ActivityNodes are partitioned into Actions, ObjectNodes and ControlNodes

$$AN = AC \cup ON \cup CN$$

where $AC$ is a finite set of Actions, $ON$ is a finite set of ObjectNodes, and $CN$ is a finite set of ControlNodes.

Definition 3: The set of ActivityEdges is partitioned into ControlFlow and ObjectFlow sets

$$AE = CF \cup OF$$

where $CF$ is a finite set of ControlFlows, and $OF$ is a finite set of ObjectFlows.

Definition 4: The set of ActivityNodes is partitioned into ActivityParameterNodes and Pins sets

$$ON = APN \cup PIN$$

where $APN$ is a finite set of ActivityParameterNodes, and $PIN$ is a finite set of Pins.

Definition 5: Pin elements have a upperBound function. This function is interpreted as the token capacity of the respective Pin. The upperBound function is defined as

$$\text{upperBound}: PIN \rightarrow \mathbb{N}$$

Definition 6: The set of ControlNodes is partitioned into InitialNode and ActivityFinalNode sets

$$CN = IN \cup AFN$$

where $IN$ is a finite set of InitialNodes, and $AFN$ is a finite set of ActivityFinalNodes.

Definition 7: An ActivityNode element $a \in AN$ has a set of incoming ActivityEdges and a set of outgoing ActivityEdges.

$$\forall a \in AN, \text{incoming}(a) \subseteq AE$$

$$\forall a \in AN, \text{outgoing}(a) \subseteq AE$$

The incoming function returns the ActivityEdges elements reaching an ActivityNode. On the other hand, the outgoing function returns the ActivityEdges elements leaving an ActivityNode. An ActivityNode may have several incoming and/or outgoing edges.

In the following, the incoming and outgoing functions of ObjectNodes, Actions and ControlNodes are specified.
Definition 18: ObjectNodes only accept ObjectFlows as incoming or outgoing elements.

\[ \forall n \in \text{ON}, \text{incoming}(n) \subseteq OF \]  
\[ \forall n \in \text{ON}, \text{outgoing}(n) \subseteq OF \]  

Definition 19: ActionNodes and ControlNodes only accept ControlFlows as incoming or outgoing elements.

\[ \forall n \in (\text{AC} \cup \text{CN}), \text{incoming}(n) \subseteq CF \]  
\[ \forall n \in (\text{AC} \cup \text{CN}), \text{outgoing}(n) \subseteq CF \]  

Definition 20: Both ActionNodes and ControlNodes has a set of input Plus and a set of output Plus.

\[ \forall n \in (\text{AC} \cup \text{CN}), \text{input}(n) \subseteq P\text{IN} \]  
\[ \forall n \in (\text{AC} \cup \text{CN}), \text{output}(n) \subseteq P\text{IN} \]  

Definition 21: An ActivityEdge element \( e \in AE \) has a target ActivityNode and a source ActivityNode.

\[ \text{target}(e) \in \text{AN} \]  
\[ \text{source}(e) \in \text{AN} \]  

The source function returns the ActivityNode from which the ActivityEdge originates. The target function returns the ActivityNode to which the ActivityEdge points to. The ActivityEdge produces tokens. An ActivityEdge has only one source node and one target node.

Particularly, ControlFlow elements produce tokens to and consumes tokens from Actions and ControlNodes elements.

\[ \forall e \in CF, \text{target}(e) \in (\text{AC} \cup \text{CN}) \]  
\[ \forall e \in CF, \text{source}(e) \in (\text{AC} \cup \text{CN}) \]  

However, ObjectFlow elements connects together ObjectNodes only.

\[ \forall e \in OF, \text{target}(e) \in \text{ON} \]  
\[ \forall e \in OF, \text{source}(e) \in \text{ON} \]  

5.2 Semantic Domain of Activity Diagrams

We propose to use Synchronous Data Flow (SDF) model of computation as the semantic domain SDFy for the Activity diagrams of SysML. The SDF model of computation is a special case of data flow in which the number of data samples produced or consumed by each node on each firing is fixed in the design process [34].

A SDF graph consists of a set of vertex (nodes) connected through a set of directed edges (arcs). The SDF graph is a digraph as defined in graph theory [35]. The nodes are connected defining arcs between them, as are communication channels. Each node has a set of input arcs and a set of output arcs. Each arc has a source node and a target node.

A SDF node defines a token (data) rate in each of its input/output arc. A token rate is specified using a positive integer number. A token rate defined for an output arc determines how many data tokens the node produces on that arc each time the node is fired. On the other hand, a token rate defined for an input arc determines how many data tokens the node consumes from that arc each time the node is fired.

The token rates are defined for all the arcs connected to a node. When a node fires, it consumes tokens from its input arcs and produces tokens in its output arcs. In order to a node to be fired, its input arcs must have, at least, as many tokens as specified by the input token rates. If that condition is satisfied the node is ready to be fired.

The SDF model of computation is widely used in signal processing applications, however, few works are found formally describing its semantics.

Definition 22: An SDF graph \( \langle S_{\text{DF}} \rangle \) is a five-tuple

\[ S_{\text{DF}} = \langle N, A, P, C, \mu \rangle \]  

where

\[ N \] is the set of nodes,
\[ A \subseteq N \times N \] is the set of arcs,
\[ P : \ N \times A \rightarrow N_0 \] is the set of data tokens produced at node \( n \in N \) and to be carried by arc \( a \in A \),
\[ C : \ N \times A \rightarrow N_0 \] is the set of data tokens carried by arc \( a \in A \) and consumed in node \( n \in N \),
\[ \mu : A \rightarrow N_0 \] is the initial marking, \( \mu(a) \) initial data tokens in arc \( a \in A \).

\( N_0 \) is the set of all the natural numbers (0) including zero, \( N_0 = \{0, 1, 2, \ldots\} \).

The set of nodes \( N \) and the set of arcs \( A \) describe the structural aspects of a SDF graph. On the other hand, the production rate \( P \), the consumption rate \( C \), and the initial marking \( \mu(a) \) describe the behavioral aspects of the SDF graph.

The execution semantics of a SDF graph are described in terms of its marking.

Definition 25: The marking \( \mu \) of a SDF graph describes how many tokens each arc has in a certain instant of the execution. Similarly to Petri Nets, a marking function can be defined for SDF as

\[ \mu : A \rightarrow N_0 \]  

The marking in the initial state of the SDF graph is represented by \( \mu_0 \).

Definition 26: A node \( n \in N \) can fire if

\[ \forall a \in A, \mu(a) \geq C(a) \]  

When a node fires the marking of a SDF graph is modified.

Definition 27: The execution of a SDF is a sequence of marking steps

\[ \mu_0 \overset{\mu_1}{\rightarrow} \overset{\mu_2}{\rightarrow} \overset{\mu_3}{\rightarrow} \cdots \]
where \( \mu_k \) is the initial marking of the SDL, \( \mu_k \) is the SDL marking at step \( k \) and \( n_k \) is a firing node \( n_k \in N \) that goes from \( \mu_k \) to \( \mu_{k+1} \).

**Definition 26:** The node execution function \( \xi_n \) is defined for any firing node \( n_k \in N \) as
\[
\mu_k \stackrel{\xi_n}{\rightarrow} \mu_{k+1} \tag{26}
\]
where \( \mu_{k+1} \) is computed as
\[
\forall a \in A, n_k, a(a) = \mu_k(a) - C(n_k, a) + P(n_k, a) \tag{29}
\]

The proposed SDF formal semantics can be directly related to the matrix representation proposed in [34]. In [34], Lee and Messerschmitt introduced a topology matrix (denoted as \( D \)) similar to the incidence matrix associated with directed graphs in graph theory in order to characterize a SDL graph. The \( (i,j) \) entry in the topology matrix is the amount of data produced by node \( j \) on arc \( i \) each time it is invoked. If node \( j \) consumes data from arc \( i \), the number is negative, and if it is not connected to arc \( i \) then the number is zero.

The relation between the proposed SDF formal semantics and the topology matrix proposed in [34] is neither straightforward. The topology matrix \( D \) has size \([N] \times [A]\), and its value is calculated as
\[
\Gamma = P - C \tag{30}
\]
where \( P \) and \( C \) are the production and consumption functions expressed as \([N] \times [A]\) sized matrices.

### 5.3 Semantic Mapping of Activity Diagrams

In this section, the semantic mapping \( M_{SDF} \) is described. \( M_{SDF} \) maps elements from the abstract syntax of SysML Activities and the elements of the SDL semantic domain; that is, the formal semantics mapping function takes an \( A_{abstract} = (A, V, E) \) as input and generates a \( A_{SDF} = (N, A, E, \mu_0) \).

**Definition 27:** The semantic mapping function \( M_{SDF} \) is defined as
\[
M_{SDF} : A_{abstract} \rightarrow A_{SDF} \tag{31}
\]

First, an intuitive outline of the semantic mapping process is provided. Then, a detailed mapping specification is provided.

An intuitive mapping relation between the elements from the abstract syntax of SysML Activities and the elements of the SDL semantic domain is depicted in Figure 3.

**ActivityFlow** elements in SysML Activities and **ActionFlow** elements in Activity diagrams are translated into SDF nodes.

**ControlFlow** elements are directly mapped into SDL **Act nodes**. **ControlFlow** elements communicate control tokens that establish the execution order of Action nodes, **InitialNode** and **ActivityInitialNode**. When the source element finishes the execution, it puts one token of the **ControlFlow** element. Then, the **ControlFlow** sends that token to the target element. Both the token production rate and the token consumption rate of the Act mapped from ControlFlow elements are equal to 1, as shown in Figure 3.

**ObjectFlow** elements carry data tokens between **ObjectNodes** (**ActivityParameterNodes** or **Pin**). In the last three rows of Figure 3 all the possible connections of ObjectFlow are represented. ObjectFlow elements are mapped into SDL Act. The token production rate of these Act is related to the number of tokens that the target ObjectNode gets. Analogously, the token consumption rate depends on the number of tokens that the source ObjectNode puts on the ObjectFlow.

The detailed mapping specification is based on the intuitive mapping provided previously. **ActivityFlow** (**AFN**), **InitNode** (**IN**), **ActivityParameterNodes** (**APN**), and **Actions** (**AC**) of Activity diagrams become SDF nodes (N) as described in the following equation
\[
N = \text{AFN} \cup \text{IN} \cup \text{APN} \cup \text{AC} \tag{32}
\]

Both **ControlFlow** (**CF**) elements and **ObjectFlow** (**OF**) elements are mapped into SDF Act (N). The semantic mapping of **ObjectFlow** (**OF**) elements is split in 3 cases, for each supported connection configuration previously shown in Figure 3:
- **Case OF1:** A **ObjectFlow** element connects two **Action** nodes through their **Pins**.
- **Case OF2:** A **ObjectFlow** element connects an **ActivityParameterNode** to an **Action**.
- **Case OF3:** A **ObjectFlow** element connects an **Action** to an **ActivityParameterNode**.
Thus, the semantic mapping to translate ControlFlow (CF) elements and ObjectFlow (OF) elements into SDFA can be given by

\[ A = A_{CF} \cup A_{OF} \cup A_{CF2} \cup A_{OF2} \]

where \(A_{CF1}, A_{CF2}, A_{OF1}, \) and \(A_{OF2}\) denote the mapping of ControlFlow (CF) elements and ObjectFlow (OF) elements (Case OF1, Case OF2 and Case OF3) into SDFA, respectively.

Similarly, the mapping of the production (P) and consumption (C) rates can be decomposed as:

\[ P = P_{CF} \cup P_{OF1} \cup P_{CF2} \cup P_{OF2}\]

\[ C = C_{CF} \cup C_{OF1} \cup C_{CF2} \cup C_{OF2}\]

On one hand, all the CF elements are translated into SDFA arcs

\[ A_{CF} = \{\text{source}(c), \text{target}(c); | c \in CF\} \]

On the other hand, for a ControlFlow of \( c \in CF\), the number of tokens produced at \( \text{source}(c)\) and consumed at \( \text{target}(c)\) are also 1. Hence

\[ P_{CF} = \{\text{source}(c); \text{source}(c), \text{target}(c); 1\}; | c \in CF\}

\[ C_{CF} = \{\text{source}(c); \text{source}(c), \text{target}(c); 1\}; | c \in CF\}

The token production and consumption rates related to SDFA arcs created from ObjectFlow elements depend on the upperBound attribute of the ObjectNode connected to the ObjectFlow. The upperBound attribute from SysML is interpreted as the number of tokens that an ObjectNode can produce or get from an ObjectFlow element. Specifically, the token production rate related to an SDFA arc created from an ObjectFlow of \( OF\) is equal to the upperBound value of the source ObjectNode \( \text{source}(c)\). Similarly, the token consumption rate related to an SDFA arc created from an ObjectFlow of \( OF\) is equal to the upperBound value of the target ObjectNode \( \text{target}(c)\).

ObjectFlow OF elements connecting an ActivityNode \( APN\) to an Action \( AC\) (Case OF3) are mapped as

\[ A_{OF2} = \{\text{input}(i); | p \in APN, o \in OF, t \in AC; \text{source}(c) = p, \text{target}(c) = p\}

\[ P_{OF2} = \{\text{input}(i); | p \in APN, o \in OF, t \in AC; \text{source}(c) = p, \text{target}(c) = p\}

\[ C_{OF2} = \{\text{input}(i); | p \in APN, o \in OF, t \in AC; \text{source}(c) = p, \text{target}(c) = p\}

Finally, ObjectFlow OF elements connecting an Action \( AC\) to an ActivityNode \( APN\) (Case OF3) are mapped as

\[ A_{OF3} = \{\text{input}(i); | s \in AC, p \in APN, o \in OF, t \in AC; \text{source}(c) = \text{output}(s), \text{target}(c) = p\}

\[ P_{OF3} = \{\text{input}(i); | s \in AC, p \in APN, o \in OF, t \in AC; \text{source}(c) = \text{output}(s), \text{target}(c) = p\}

\[ C_{OF3} = \{\text{input}(i); | s \in AC, p \in APN, o \in OF, t \in AC; \text{source}(c) = \text{output}(s), \text{target}(c) = p\}

6 CONCLUSIONS
Many research works address the generation of SystemC-TLM executable models from systems described using UML or SysML. However, they do not solve a well-known issue: the lack of formally defined behavioral semantics of State Machine diagrams and Activity diagrams. Moreover, very few works in the literature define the MOFs to be implemented within the generated executable system model.

Initially, the paper defines the abstract syntax supported by SysML State Machine diagrams and Activity diagrams. Then, the semantic domains for SysML State Machine diagrams and Activity diagrams have been defined using FSM and SDFA models of computation (MOC) respectively. Finally, the semantic mapping from the abstract syntax of SysML, State Machine diagrams and Activity diagrams to FSM and SDFA models of computation has been formalized. As a result, clear semantic transformation rules have been defined for State Machine diagrams and Activity diagrams.

The proposal of this work enables to effectively implement a MDD-based framework to create SystemC-TLM executable code with heterogeneous behavioral MOCs for early electronic system design. The implementation of a framework for the formal semantic mappings presented in this paper is addressed in Part II [15].
APPENDIX D: Research Results

Part II also describes a code generation process to produce SystemC-IML models from a SysML model. Besides, Part II presents a practical case study where the proposed approach has been applied to a complex electronic system development process in order to illustrate its benefits.

REFERENCES

L. Vázquez (M’08) received the M.Sc. degree in electrical, electronic, and control engineering and the Ph.D. degree, both from the University of Navarra, San Sebastián, Spain, in 2000 and 2005, respectively.

Her work at CER, San Sebastián, Spain, involves the digital design of the physical layer of wireless communication systems, notably WLAN, RFID, and UWB. Her research interests include digital signal processing and hardware development. She is a Lecturer at TECNUN, University of Navarra, in the Electronic CAD Laboratory.
A MDD approach to generate heterogeneous SystemC–TLM executable models from SysML – Part II: ASSYST Framework and Case Study
A MDD approach to generate heterogeneous SystemC–TLM executable models from SysML – Part II: ASSYST Framework and Case Study

Koldo Tomasea, Juan F. Sevilla, Member, IEEE, and Ignace Ville, Member, IEEE

Abstract—In recent years, Model-Driven Design (MDD) has been proposed as a promising methodology for electronic systems modeling and design. In this paper, we propose a MDD framework called ASSYST to translate electronic systems described using SysML into SystemC–TLM heterogeneous executable models. An associated paper previously defined semantic mappings to translate SysML State Machine diagrams and Activity diagrams to formal models of computation (MoCs), Assyst has generated several semantic mappings and provides an automatic approach to generate SystemC–TLM code containing controlled behaviors and class-based objects. A case study demonstrates both the application of the ASSYST framework within a complex electronic system and the benefits of the proposed approach: design productivity and system development time are greatly improved.

Index Terms—Electronic system level (ESL), Model-driven design (MDD), high abstraction level (HAL), formal modeling, behavioral modeling, formal semantics, modeled computation (MoC), SystemC, Transaction-level modeling (TLM), model-based (MB), model-based (MBE).

1 INTRODUCTION

The modern electronic systems are increasing their complexity to meet market requirements. However, present Electronic Design Automation (EDA) tools can not handle this complexity growth efficiently. This productivity loss has led the so-called design productivity gap [1], as documented by several editions of the Design Report [2] published by the International Technology Roadmap for Semiconductors (ITRS). The ITRS suggested three foundations in order to address the design productivity challenge: increasing the level of abstraction, automation processes and tools, and producing executable specifications. Therefore, in order to reduce the development time and improve the product quality, new languages and design paradigms are needed to handle present electronic systems.

SystemC [3] has become the de facto standard to create executable models of electronic systems. Besides, the Transaction Level Modeling (TLM) [4] supported by SystemC, has been proposed for the creation of high abstraction level executable models. These high abstraction level SystemC–TLM models are fast to build and they provide a very high speed of simulation.

Recently, the MDD methodology has emerged as a promising methodology for complex electronic systems modeling and design [5], [6]. MDD enables the modeling and the automatic code generation of functional SystemC–TLM system models at a high abstraction level from visual modeling languages such as Unified Modeling Language (UML) [7] and Systems Modeling Language (SysML) [8]. SysML offers systems engineers several noteworthy improvements for electronic systems modeling compared with UML, which tends to be software-centric [9]. In recent years several works have been proposed that focus on creating SystemC code from SysML.

Raskin et al. proposed a method supporting the generation of SystemC code from SysML State Machine diagrams and Activity diagrams [10], [11]. However, no model of computation (MoC) of the generated SystemC executable model is specified and the authors do not address the generation of TLM models.

Several works in the literature [12], [13], [14] are related to the SATURN European project [15], which aims to bridge the gap between the modeling phase and the design phase of embedded systems using SysML, MARTE and the SATURN profile. The SATURN profile is composed of three subprofiles: a synthetable SystemC profile to model HW components, an analysis profile to translate SystemC models into VHDL, and a C profile to model the embedded SW. These SysML based proposals are focused on modeling synthesizable HW and embedded SW directly using visual modeling languages. However, these works do not address the creation of purely functional executable models prior to the SW/HW partitioning process. Within SATURN, system engineers must make design decisions about allocating SW into HW parts in order to create the first
APPENDIX D: Research Results

The objective of this paper is to automatically translate SysML graphical models into heterogeneous SystemC-TLM executable code. These executable models enable functional architecture analyses and functional simulation early in the development process before the SW/HW partitioning occurs. Besides, the automation of the SystemC-TLM executable model generation improves the error-prone manual coding process and significantly reduces the development time of the device being designed. However, the automation tools for electronic system development can only be applied if the behavioral semantics and the corresponding MoCs are well-defined [16]. The well-defined computation semantics of MoCs allow to unambiguously capture the required functionality and enable the application of formal design and verification techniques [17].

Part I [18] of this work clearly defined the semantic domains of State Machine diagrams and Activity diagrams using Finite State Machine (FSM) and Synchronous Data Flow (SDF) MoCs, respectively. Part II proposed the following two formal semantic mappings:

$$M_{SM} \rightarrow M_{SM}$$

$$M_{SM} \rightarrow M_{SM}$$

where $$M_{SM}$$ maps the abstract syntax of SysML State Machine diagrams (ASM) to a semantic domain described by the Finite State Machine (FSM) MoC ($$M_{SM}$$). $$M_{SM}$$ maps the abstract syntax of SysML Activity diagrams (AAD) to a semantic domain described by the Synchronous Data Flow (SDF) MoC ($$M_{SM}$$).

In this paper, we propose a MDD approach to model and design electronic systems using a framework called ASSYST. ASSYST implements the behavioral semantic mappings, $$M_{SM}$$ and $$M_{SM}$$, defined in Part I. The objective of ASSYST is to automatically translate SysML graphical models into SystemC-TLM executable code with heterogeneous MoCs.

The paper is organized as follows: Section 2 introduces the workflow and the modeling steps proposed by the ASSYST framework. Section 3 presents the implementation of the behavioral semantic mapping described in Part I [18] of this research work using Model-to-Model (M2M) transformation techniques. Section 4 describes the ASSYST code generation process based on Model-to-Text (M2T) principles. Finally, Section 5 presents a practical application of the methodology and tools proposed in the paper. The objective of this paper is to demonstrate the benefits of the proposed MDD-based automatic code generation process by means of a case study. Finally, Section 7 summarizes the results and discusses possible directions of future work.

2 ASSYST FRAMEWORK

The ASSYST framework is implemented as a set of Eclipse [19] plugins that integrate into the Eclipse Integrated Development Environment (IDE). Additionally, ASSYST proposes a C++ library in order to support the code generation process. The ASSYST C++ library is built on top of SystemC and TLM and defines base classes for the description of heterogeneous MoCs.

The SystemC-TLM code generation process is depicted in Figure 1. ASSYST takes as input a SysML model described using the Temko [20] modeling platform and produces a SystemC-TLM executable model. The translation of SysML models into SystemC-TLM source code is automatically and seamlessly performed by the ASSYST framework.

The generation of the executable models follows two steps. In the first step, the SysML behavioral parts are mapped to formal MoCs using Model-to-Model (M2M) transformation techniques. ASSYST supports heterogeneous behavioral constructs. SysML State Machine diagrams are used to model control-based system functionalities. SysML Activity diagrams are used to model system functionalities based on data-flow. SysML State Machines are mapped to FSMs and SysML Activities are mapped to SDFs following the formal semantic mappings, $$M_{SM}$$ and $$M_{SM}$$, proposed in Part I [18] of this research work. As a result, the behavioral aspects of the system are translated into FSM and SDF MoCs while the structural information is kept in the Block Definition Diagrams (BDDs) and Internal Block Diagrams (IBDs) of the SysML model. During the second step, a Model-to-Text (M2T) process produces a SystemC-TLM executable code from the FSM, SDF and SysML models describing the system being designed.

3 BEHAVIORAL SEMANTIC MAPPING

In general, a MSM transformation describes how a source model $$M_s$$ conforming to a metamodel $$MM_s$$ is transformed into a target model $$M_t$$ that conforms.
to a metamodel $M_M$. In the M2M transformations proposed in this paper, the source model is the SysML system model which conforms to the SysML metamodel available in the Topcased Modeling toolset as shown in Figure 2. The targets of the M2M transformations are the SDF and the FSM models conforming to the SDF and FSM metamodels, respectively. The FSM and SDF metamodel definitions are derived from the formal semantics proposed in the Part I of this work [18]. The Eclipse EMF tool has been used for describing FSM and SDF metamodels. ASMF uses the ATL [21] platform to implement the formal semantic mappings defined in Part I [18]. Developed on top of the Eclipse IDE, ATL is a M2M transformation language. ATL provides a platform to produce a set of target models from a set of source models. An ATL transformation program is described by rules that specify how source model elements are translated into target model elements.

In order to achieve the model transformations, the ATL execution engine needs the following elements:

- A source system model conforming the SysML metamodel provided by the Topcased. This model describes the system being developed.
- A target metamodel for FSM.
- A target metamodel for SDF.
- ATL specifications to transform SysML State Machines into FSMs and SysML Activities to SDFs.

3.1 Metamodels definition

The FSM metamodel specification is represented in the Eclipse diagram depicted in Figure 3. The FSMDomainModel element is the root of the FSM metamodel. FSMDomainModel includes the FSMs of the system being designed. A FSM is composed of a set of States $Q$, a set of Transitions $A$, and a reference to an initial State $s_0$. The source and target attributes of a Transition specify which States are connected by the transition. Transitions also contain a trigger attribute of type string representing the possible input events $I$ of the FSM. The FSM, State, and Transition elements have a unique name attribute. This attribute facilitates the code generation process however, it does not modify the semantics of the MoC.

Similarly to FSM metamodel case, the SDF metamodel is represented in Figure 4. The root of the SDF metamodel is the SDFDomainModel element. SDFDomainModel contains all the SDF graphs of the system. A SDF is composed of a set of Nodes $N$ and a set of Arcs $A$. Each Arc connects a source node and a target node. Additionally, the token production rate $P$, the token consumption rate $C$, and the initial marking $(Q_0)$ have been integrated as attributes into the Arc element in order to facilitate the implementation of the model transformation process. Therefore, an arc specifies three integer type attributes concerning the reception and transmission of tokens: incomingTokens, which specifies how many tokens does the arc receive from the source node; outgoingTokens, which specifies how many tokens
Appendix D: Research Results

<table>
<thead>
<tr>
<th>SysML element</th>
<th>FSM element</th>
<th>Rule Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>StateMachine</td>
<td>FSM</td>
<td>StateMachineToFSM</td>
</tr>
<tr>
<td>state</td>
<td>State</td>
<td>StateToState</td>
</tr>
<tr>
<td>initial</td>
<td>Initial</td>
<td>InitialToState</td>
</tr>
<tr>
<td>transition</td>
<td>Transition</td>
<td>TransitionToTransition</td>
</tr>
</tbody>
</table>

Listing 1: StateMachine_to_FSM ATL rule

does the arc send to the target node; and initial, which denotes how many tokens are initially in the arc. The SDF, Node, and Arc elements have a unique name attribute.

3.2 ATL model transformations specification

An ATL specification includes a set of rules that define how SysML models are transformed into FSM and SDF elements. These rules implement the formal semantics mappings described in the Part I of this work [18].

3.2.1 SysML State Machine to FSM transformation

The transformation of State Machines into FSMs is specified in an ATL module composed of 5 transformation rules, as shown in Table 1.

Rule 1: The rule StateMachine_to_FSMDomainModel maps the SysML model’s root element Model to the FSMDomainModel root element. As the Model element within a SysML model is unique, only one FSMDomainModel model will include all the FSMs of the system being designed.

Rule 2: The rule StateMachine_to_FSM shown in Listing 1 describes how a StateMachine is transformed into an FSM. The rule StateMachine_to_FSM implements the formal semantic mapping FSM described in the Part I of this work [18].

\[ \text{FSM} : \text{StateMachine} \rightarrow \text{FSM} \]

Listing 2: StateToState ATL rule

\[ \text{StateToState} (\text{from} : \text{FSM}) \rightarrow \text{StateToState} (\text{to} : \text{FSM}, \text{transition} : \text{Transition}) \]

Listing 3: Pseudostate_to_State ATL rule

In the first attribute binding declaration, the name of a SysML StateMachine is assigned to the name attribute of a FSM.

Elements of type State and Pseudostate are collected from the region of the StateMachine. These elements will be transformed by the ATL Rule 3 (StateToState) and the Rule 4 (PseudostateToState), respectively, and then assigned to the attribute states of the FSM.

Similarly, Transition elements and initial State elements are gathered and assigned to the attributes transitions and initialState of the FSM, respectively. The transformed FSMs and initial_state are created by the ATL Rule 5 (TransitionToTransition) and the Rule 4 (PseudostateToState), respectively.

Rule 3: State elements (S) from SysML models are transformed into FSM State elements (Q) by the rule StateToState. The rule StateToState in Listing 2 implements the semantic mapping transformation

\[ Q = S \]

The attribute name of a SysML State is assigned to the name attribute of a FSM State.

Rule 4: The rule PseudostateToState transforms each initial pseudostate (I) of the SysML model into an initial State element (q0 ∈ Q) conforming to the metamodel FSM as shown in Listing 3. The rule PseudostateToState implements the semantic mapping transformation

\[ q_0 = (p_0 | \text{initial}(p_0) = \text{InitialState}, p_0 \in P) \]

The attributes of the FSM State are created similarly to the ones defined in the rule StateToState.

Rule 5: The rule TransitionToTransition transforms Transition elements (T) of SysML into FSM Transition elements (D), as shown in Listing 4. This rule also creates the triggers...
D.1 International Journal Papers

Listing 4. Transition_to_Transition ATL rule

rule Transition_to_Transition {
from
  S1 : MINTERMTransition
to
  S2 : MINTERMTransition
  n : MINTERMTransition (\n    rule = true
    source = "source"
    target = "target"
    trigger = "trigger.name"
  ) \n}

Listing 5. Activity_to_SDF ATL rule

rule Activity_to_SDF {
from
  a : MINTERMActivity
to
  s : MINTERM (\n    rule = true
    source = "source"
    target = "target"
    trigger = "trigger.name"
  ) \n}

TABLE 2
Activity-to-SDF transformation rules.

<table>
<thead>
<tr>
<th>Source element</th>
<th>Target element</th>
<th>Rule Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>SDF</td>
<td>Model_to_SDF</td>
</tr>
<tr>
<td>Model</td>
<td>SDF</td>
<td>Model_to_SDF</td>
</tr>
<tr>
<td>Activity</td>
<td>SDF</td>
<td>Activity_to_SDF</td>
</tr>
<tr>
<td>Activity</td>
<td>SDF</td>
<td>Activity_to_SDF</td>
</tr>
<tr>
<td>ActivityNode</td>
<td>SDF</td>
<td>ActivityNode_to_SDF</td>
</tr>
<tr>
<td>ActivityNode</td>
<td>SDF</td>
<td>ActivityNode_to_SDF</td>
</tr>
</tbody>
</table>

of the FSM Transitions. These triggers denote the input events (\$) of the FSM. The rule Transition_to_Transition implements the following semantic mapping transformations

\[
\Psi = \{\text{trigger}(t) | t \in T\} \\
\Delta = \{\text{source}(b), \text{trigger}(t), \text{target}(o) | t \in T\}
\]

Source and the target states are assigned to the attributes \text{source} and \text{target}. Moreover, this rule assigns the name of the trigger parameter of the Transition elements of SysML model to the attribute \text{trigger} of the Transition.

3.2.2 SysML Activity to SDF transformation
The transformation of Activities into SDFs is specified in an ATL module composed of 7 rules as shown in Table 2.

Rule 1: The rule Model_to_SDFDomainModel translates SysML model's root element Model to SDFDomainModel root element. As the Model element within a SysML model is unique, only one SDFDomainModel model will be created. The SDFDomainModel will include all the SDFs of the system being designed.

Rule 2: The rule Activity_to_SDF in Listing 5 implements the formal semantic mapping described in Part I of this work. [18]

\[
\text{Mstart} : \text{Activity} \rightarrow \text{SDF}
\]

The rule Activity_to_SDF transforms each Activity into a SDF element.

Action and ActivityParameterNode are transformed into nodes of the SDF. The node attribute of a SDF will be populated by the Rule 3. Furthermore, the edge elements of the Activity will be assigned to the attribute arcs by the ATL Rules 4, 5, 6 and 7. Each of these rules implements a part of the semantic mappings

\[
\begin{align*}
A &= A_{\text{arc}} \cup A_{\text{para}} \cup A_{\text{para}} \\
P &= P_{\text{para}} \cup P_{\text{para}} \cup P_{\text{para}} \\
C &= C_{\text{arc}} \cup C_{\text{para}} \cup C_{\text{para}} \cup C_{\text{para}}
\end{align*}
\]

Rule 3: InitialNodes (IV), ActivityFinalNodes (AFN), ActivityParameterNodes (APN) and Actions (AC) are transformed into SDF Node elements (V) by the rule ActivityNode_to_SDF. The rule ActivityNode_to_SDF in Listing 6 implements the following semantic mapping, defined in Part I of this work: [18]

\[
N = \text{AFN} \cup \text{IV} \cup \text{APN} \cup \text{AC}
\]

The node source pattern specifies a guard condition which specifies that only ActivityFinalNodes elements, InitialNodes elements, or ActivityParameterNode elements are matched by this rule.

Rule 4: The rule ControlFlow_to_Arc in Listing 7 transforms ControlFlow (CF) elements into Arc elements (V). The rule ControlFlow_to_Arc implements
the following semantic mapping transformations:

\[ ACF = \{ (\text{source}(c), \text{target}(c)) : c \in \text{OF} \} \]
\[ BCF = \{ (\text{source}(c), \text{source}(c), \text{target}(c)) : c \in \text{OF} \} \]
\[ CCF = \{ (\text{source}(c), \text{source}(c), \text{target}(c)) : c \in \text{OF} \} \]

This rule sets the attributes \textbf{incoming\_tokens} and \textbf{outgoing\_tokens} to 1.

All the rules that create \textit{Arc} elements assign the source and the target nodes to \textit{source} and \textit{target} attributes, respectively. Furthermore, the initial marking attribute of the arcs is set to 0.

Rule 8: The rule \textit{PinToPinObjectFlow\_to\_Arc} maps ObjectFlow elements (OF) between Pins (PIN) owned by Activity (AC) into Arc (A) elements. The rule \textit{PinToPinObjectFlow\_to\_Arc} in Listing 8 implements the following semantic mapping transformations defined in Part 1:

\[ AOF_1 = \{ (\text{source}(of), \text{target}(of)) : c \in \text{OF}; \text{source}(of) \neq \text{output}(s), \text{target}(of) \neq \text{input}(t) \} \]
\[ BOF_1 = \{ (\text{source}(of), \text{source}(of), \text{target}(of)) : c \in \text{OF}; \text{source}(of) \neq \text{output}(s), \text{target}(of) \neq \text{input}(t) \} \]
\[ COF_1 = \{ (\text{source}(of), \text{source}(of), \text{target}(of)) : c \in \text{OF}; \text{source}(of) \neq \text{output}(s), \text{target}(of) \neq \text{input}(t) \} \]

The rule source pattern specifies a guard condition which specifies that only ObjectFlow with Pins in both ends are matched by this rule. This rule sets the attribute \textit{incoming\_tokens} with the \textit{upper\_Bound} value of the source Pin and sets the attribute \textit{outgoing\_tokens} with the \textit{upper\_Bound} value of the target Pin.

Rule 9: The rule \textit{PinToPinObjectFlow\_to\_Arc} translates ObjectFlow elements (OF) between a ActivityParameterNode (APN) and a Pin (PIN) to Arc (A) elements. The rule \textit{PinToPinObjectFlow\_to\_Arc} implements the following semantic mapping transformations:

\[ AOF_2 = \{ (\text{source}(of), \text{target}(of)) : c \in \text{OF}; \text{source}(of) \neq \text{output}(s), \text{target}(of) \neq \text{input}(t) \} \]
\[ BOF_2 = \{ (\text{source}(of), \text{source}(of), \text{target}(of)) : c \in \text{OF}; \text{source}(of) \neq \text{output}(s), \text{target}(of) \neq \text{input}(t) \} \]
\[ COF_2 = \{ (\text{source}(of), \text{source}(of), \text{target}(of)) : c \in \text{OF}; \text{source}(of) \neq \text{output}(s), \text{target}(of) \neq \text{input}(t) \} \]

The rule source pattern specifies a guard condition which specifies that only ObjectFlow with an ActivityParameterNode as source and a Pin as target are matched by this rule. This rule sets the attribute \textit{incoming\_tokens} with the \textit{upper\_Bound} value of the source ActivityParameterNode and sets the attribute \textit{outgoing\_tokens} with the \textit{upper\_Bound} value of the target Pin.
4 SYSTEMC-TLM CODE GENERATION

Figure 5 depicts the code generation process. The objective of this M2T workflow is to produce the source files of the SystemC-TLM executable model from the FSM, SDF and SystemC models describing the system under design. The FSM and the SDF models were created from the State Machine Diagrams and the Activity Diagrams using the ATL transformations described in Section 3.2. These FSM and SDF models specify the behavioral aspects of the system while the structural features are extracted from the Block Definition Diagram (BDD) and the Internal Block Diagram (IBD) of the SystemC model.

The Accelero platform [22] is used for the model to SystemC-TLM code generation. Accelero is an IDE-free based product created and developed by Ocuin. Accelero is a code generator implementing the M2T2T2 standard [23] specified by OMG. Accelero is a template-based code generation platform to transform models into text files. An Accelero module specifies an input template, a source metamodel MMm and a source metamodel MMc, within MMm, in order to produce a target text file. When an Accelero module is applied to a source model Ml, conforming to the metamodel MMm, the Accelero generation engine extracts the source model elements that conform to the metamodel MMc, and employs the specified template to produce the output text file.

Additionally, the M2T framework described in Figure 5 is supported by the proposed ASSYST C++ library. This C++ library aims to simplify the M2T template definition providing reusable components to describe heterogeneous executable models.

4.1 ASSYST C++ library

The proposed C++ library is built on top of SystemC v2.2.1 and TLM v2.0.1. This library defines C++ base classes for the description of a heterogeneous executable model of the system being designed. The key benefit is that the Accelero templates can produce an executable system model enabling software artifacts that are derived from base classes defined in the C++ library. Thus, the code generation process and the complexity of the Accelero templates are simplified.

The base classes defined in the C++ library are presented in Figure 6. These classes describe the base features needed to describe a system.

Structural features are modeled with the class Block, Block elements are derived from the SystemC class SystemLib and they allow to describe the hierarchy of the system being designed.
The classes Transaction and TransactionState describe the communication features of the system. The M.C. [3] approach is based on the support communication Aspects. The class Transaction models the events of the system. Transaction extends the class Bean @model of the SystemC-TLM library. TransactionState describes the behavior and target node color. Transaction elements of IB and send transactions through TransactionSocket.

The internal event management features are described using XMLReader, XMLWriter, and TransactionFactory classes. XML files are used for the description of both the transactions to be injected into the system and the transactions received from the system. The class TransactionFactory is used for both the transactions injected from the XML files into the system, and the recording of the received transactions from the system using XML files. XMLReader reads the input XML file and injects the transactions to the system, XMLWriter receives the transactions from the system and writes them into the output XML files. Both XMLReader and XMLWriter use TransactionFactory for XML-Transaction transformations.

The class UseCase represents the interface between the structural aspects and the behavioral aspects. UseCases describe the functionality of a system in relation to how the users use the system. This functionality is expressed in terms of behaviors, such as FSMs and SEs. Moreover, the services represented by UseCases to be offered by the blocks of the system.

The class Behavior defines the common behavioral features, such as event management and event processing. FSM and SEs are implemented using behavior as the base class. Each of them defines a particular event processing algorithm conforming to the runtime semantics defined in Part I [18]. More models of computation can be described deriving from the class Behavior and, thus, the ABSYST C++ library can be extended.

The behavioral aspects of the FSM M.C. are described through three C++ classes: the class State, the class Transition, and the class FSM. The class State represents the state elements of the FSM M.C. The class State contains two pure virtual functions, enterBehavior and exitBehavior. These functions shall be implemented in derived classes.

The class Transition represents the transition elements of the FSM M.C. FSM is the main class representing the FSM M.C. This class processes incoming transactions, which are a runtime step of the FSM M.C. It takes the incoming transaction and checks whether an enabled transition is available in the current active state. If a transition is enabled, the enterBehavior function of the current state is performed and the current active state is set to the target state of the transition. Finally, the exitBehavior function of the newly activated state is executed.

The SDF nodes are implemented with the class Node shown in Figure 6. The function doBehavior is a pure virtual function that the derived class shall implement. This function represents the data processing carried out by the Node. The class Arc represents the arc elements of a SDF. Each class derived from Arc shall indicate how many tokens are produced, consumed, or initially set in the specified arc class. The class SDF is the main class describing the SDF M.C. Similarly to FSM, SDF is derived from the Behavior base class. A static scheduling is calculated within SDF. As a result, the scheduled firing order of the SDF arc elements is determined. SDF processes the input transactions and executes the static scheduling. During the static scheduling execution, the function doBehavior of the Node elements is executed one by one.

4.2 Acceleo code generation

The Acceleo M.T models are composed of static text parts and dynamic text parts. The static text parts are common to all model elements conforming to the metaclass M.C. specified in the module. On the other hand, the dynamic text parts depend on the particular values of the source models. An Acceleo template specifies both static text and model queries. Placeholders are included within the generation template to implement queries and extract data from models. The placeholders are defined by `@` and `@?` that is, placeholders follow the format `@variable/`. The queries are expressions specified over the source metamodel `MMx` for extracting the values from the model `MMx`.

Having defined a source metamodel, a source metaclass, a source model, and a generator template, the Acceleo code generation engine executes the model queries on the source model `MMx` and merges the results of the queries with the static text to produce the target text file.

The Acceleo modules of the ABSYST framework are described in the following sections. The code generation templates have been divided into three groups:

- Templates translating SysML models
- Templates translating FSM models
- Templates translating SDF models

4.3 Behavioral code generation

The Acceleo M.T modules proposed for the behavioral code generation are presented in Table 3.

4.3.1 FSM code generation

The source model for the FSM code generation process is the FSM model produced by AXIOM State Machine diagrams during the M.T transformation in Section 4.2.1. The FSM model consists of the FSM metamodel described in Section 4.1.

State code generation: The Acceleo module generateStates creates a C++ class for each state element contained within the FSM model, as shown in Table 3. The produced C++ class is named
D.1 International Journal Papers 347

TABLE 3
Behavioral code generators

<table>
<thead>
<tr>
<th>Actor module</th>
<th>Generated C++ class name</th>
<th>Base C++ class</th>
</tr>
</thead>
<tbody>
<tr>
<td>generate_transition</td>
<td>Transition</td>
<td>FSM</td>
</tr>
<tr>
<td>getTransition</td>
<td>Transition&lt;FSM&gt;</td>
<td>FSM</td>
</tr>
<tr>
<td>action</td>
<td>Transition&lt;FSM&gt;</td>
<td>FSM</td>
</tr>
<tr>
<td>getAction</td>
<td>Transition&lt;FSM&gt;</td>
<td>FSM</td>
</tr>
<tr>
<td>generate_input</td>
<td>Transition&lt;FSM&gt;</td>
<td>FSM</td>
</tr>
<tr>
<td>getInput</td>
<td>Transition&lt;FSM&gt;</td>
<td>FSM</td>
</tr>
<tr>
<td>generate_output</td>
<td>Transition&lt;FSM&gt;</td>
<td>FSM</td>
</tr>
<tr>
<td>getOutput</td>
<td>Transition&lt;FSM&gt;</td>
<td>FSM</td>
</tr>
</tbody>
</table>

TABLE 4
Structural code generators

<table>
<thead>
<tr>
<th>Actor module</th>
<th>Generated C++ class name</th>
<th>Base C++ class</th>
</tr>
</thead>
<tbody>
<tr>
<td>generate_code</td>
<td>LNode</td>
<td>Node</td>
</tr>
<tr>
<td>generate_node</td>
<td>LNode</td>
<td>Node</td>
</tr>
<tr>
<td>generate_actor</td>
<td>LActor</td>
<td>Actor</td>
</tr>
<tr>
<td>generate_input</td>
<td>LInput</td>
<td>Input</td>
</tr>
<tr>
<td>generate_output</td>
<td>LOutput</td>
<td>Output</td>
</tr>
<tr>
<td>generate_event</td>
<td>LEvent</td>
<td>Event</td>
</tr>
</tbody>
</table>

"transition/LState" and derives from the class State of the ASSYST C++ Library. A member attribute is created in the C++ class for each of the output Transition elements of the FSM State elements.

The module `generate_state` implements the `inTransition` and `outTransition` functions of the class "LTransition/LFacade", These functions contain Acceleo protected user code sections where the design engineer can implement the atomic functionality of the state after the code generation process. Acceleo provides protected user code sections to support an incremental generation process. A user code is the piece of text inside an output file that will be preserved from one text generation to another. This allows a design engineer to include text manually inside the protected user code sections contained in the files already generated by Acceleo. If the generation module is re-launched, the new output file will incorporate any modifications performed in the input model but the protected text will not be overwritten.

Transition code generation: The code generation of the FSM Transition elements is performed by the Acceleo module `generate_transition`. As shown in Table 3, each SysML transition is translated into a C++ class called "Transition" that derives from the class `Transition` of the ASSYST C++ Library.

FSM code generation: The module `generate_fsm` translates FSMs into C++ classes. A FSM produces a C++ class called "FSM<name>/LFSM" as described in Table 3. This C++ class derives from the class `FSM` of the ASSYST C++ Library. The class "FSM<name>/LFSM" implements the execution semantics of FSM MOGs described in the Part I of this work.

4.3.2 SDF code generation

The source model for the SDF code generation process is the SDF model produced from SysML. Activity diagrams during the M2M transformation in Section 3.2.2. The SDF model conforms to the SDF metamodel described in Section A.1.

Node code generation: The module `generate_node` translates SDF Node elements into C++ classes called "Node<name>/LNode", as depicted in Table 3. This C++ class derives from the class `Node<name>/LNode`.

The module `generate_code` implements member attributes specifying both the incoming and the outgoing arcs of the SDF nodes. The class "Node<name>/LNode" contains a member function `behavior`. This function implements both the reading of tokens from the incoming arcs and the writing of tokens into the outgoing arcs. Additionally, an Acceleo protected user code section is specified within the function `behavior` to let the design engineers define the atomic functionality of the nodes. The protected user code section shall define how the input tokens are converted into the output tokens.

Arc code generation: The module `generate_arcs` produces a C++ class called "Arc<name>/LArc" from an input SDF Arc. Each generated class implements the specific token consumption and production rates indicated by the attributes `incoming_tokens` and `outgoing_tokens` of SDF Arcs.

SDF code generation: The code generation of the SDFs is performed by the Acceleo module `generate_sdf`. As shown in Table 3, each Node element is translated into a C++ class called "Node<name>/LSDF" which is derived from the class `SDF`. The Node and Arc elements contained in the SDF are instantiated as class member attributes and registered into the SDF scheduler. Then, the SDF scheduler calculates the static scheduling following the algorithm found in [24], [25].

4.4 Structural code generation

The structural code generation process takes the SysML model as the source model. The SysML model conforms to the SysML metamodel provided by the Topcased toolset.

The code generation tool proposed in Table 4 provides Acceleo M2T templates to generate source code files from `Model`, `Actor`, `Block`, `UseCase`, `Signal`, `DataItem`, and `Enumeration` elements of SysML.

Actor code generation: The module `generate_actor` translates SysML Actor elements into C++ classes called "Actor<name>/LActor", which are derived from the class `Component` of the SysML library as shown in Table 4. The Actor classes contain an Iniator `InitiatorSocket`, a `XMLReader`, and a `XMLWriter`.
Actor objects read transactions from an input XML file using the XMLReader object, and inject those transactions into the system through the initiator TransactionSocket object. On the other hand, actor objects receive transactions from the system through the target TransactionSocket object, and write those transactions in an output XML file using the XMLWriter object. The actor architecture is shown in Figure 7.

**Data type code generation:** The Acceleo module `generate_datatype` produces a C++ class called `“DatatypeName”/T` for each `Datatype` element extracted from a SysML model. Compositions of data types are supported, that is, previously implemented data type classes may be used to type the member attributes or the member functions of a new data type class.

A public member attribute is created in the data type class for each of the Properties of the SysML `Datatype` elements. ASSYST supports the Property types defined in Table 5.

**Enumerated code generation:** This Acceleo module is straightforward as the SysML `Enumeration` elements are directly mapped to the C++ enum type.

**Transaction code generation:** The `Signal` elements from a SysML model are translated into C++ classes. The generated transaction classes are called `“SignalName”/T`. They are derived from the base class `TRANSACTION` of the ASSYST C++ library and they define specific transaction data attributes.

**UseCase code generation:** A C++ class called `“UCasename”/LUseCase”` is produced by the module `generate_usecase` from each `UseCase` element of a SysML model. The use case class is derived from the class `LUseCase` of the ASSYST C++ library. The class `“UCasename”/LUseCase”` creates a member attribute for each FSM or SEF implemented. Furthermore, the C++ use case class contains two attributes of type `TRANSACTION` which are the `internal_socket` and the `target_socket` which contain the attributes `process_socket`.

The use case class receives the input transaction through the `process_socket` and sends the output transaction through the `target_socket`. The other hand, the initiator socket `internal_socket` matches the outgoing transaction outside the use case.

**Block code generation:** The Acceleo module `generate_block` translates the SysML `Block` elements into C++ classes named `“Blockname”/LBlock”`, as shown in Table 4. Structural features are modeled with the classes `“Blockname”/LClass”`. These block classes are derived from the class `LBlock` of the ASSYST C++ library.

The block classes contain member attributes of type `TransactionSocket`, `ProcessSocket` and `Block` element being processed by Acceleo. The `UseCase` elements implemented by a `Block` are instantiated as member attributes.

A SysML `Block` can define relationships with other `Block` parts using Internal Block Diagrams (IBD) in order to describe a system hierarchy. The parent block `C++` class instantiates child block classes as member attributes.

The block classes are responsible for performing the following interface bindings. An example of these bindings is shown in Figure 8:

- `ProcessPort/UseCase binding`: the class `ProcessPort/UseCase` bind the input `TransactionSocket` of the parent `UseCase` to the `ProcessPort` of the child `UseCase`.
- `TransactionPort/UseCase binding`: the class `TransactionPort/UseCase` bind the output `TransactionSocket` of the child `UseCase` to the `TransactionPort` of the parent `UseCase`.
- `ProcessPort/TransactionPort binding`: if a block hierarchy is modeled, the input and output sockets of the parent `Block` may be bound to the output and the input sockets of the child `Block`.

---

**TABLE 5
ASSYST C++ data type generation**

<table>
<thead>
<tr>
<th>SysML type</th>
<th>C++ attribute type</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>integer</td>
<td>int</td>
</tr>
<tr>
<td>real</td>
<td>double</td>
</tr>
<tr>
<td>boolean</td>
<td>bool</td>
</tr>
<tr>
<td>string</td>
<td>std::string</td>
</tr>
</tbody>
</table>
sockets of its child blocks, respectively. As a result, transactions can flow through the block hierarchy.

Finally, block classes contain methods attributes representing data. The generation pattern for the creation of the member attributes follows the same pattern defined for UserInterface elements and their attributes in Table 5. Furthermore, member functions are defined to handle the data attributes of block classes. These member functions contain a protected user code section in order to let the design engineers define manually their contents.

Model code generation: The module generate model causes the header file "modelname".h and the source file "modelname".cpp for the Model element within a SysML model. These files contain the function sc_main, which is the top level function of a SystemC-based executable system description. As the Model element is unique within a SysML model, the function sc_main is guaranteed to be unique. Models are the root elements of the SysML models. A Model contains all the elements describing a SysML model under design. The actor classes and the block classes of the system are instantiated inside the sc_main.

5 CASE STUDY

The system that is being developed within the CONFIDENCE Project [28] is presented as a case study. The main objective of this project is the development of a care system for the detection of abnormal events (such as falls) or unexpected behaviors that may be related to a health problem in elderly people. In order to detect problems, the system monitors the posture of the user from some sensors and analyses the situation. This care system works both outdoors and indoors. It raises an alarm to the user if an abnormal situation is detected. The system can also detect changes in the user's behavior and issue a warning. The system can call to an alarm service or a relative (hereafter alarm receiver) when an alarm occurs.

5.1 CONFIDENCE System Design Flow

The CONFIDENCE system development process is divided into three steps. In the first step, the design team creates a SysML model of the system. The SysML model is then used to construct the CONFIDENCE system at the highest level of abstraction. This SysML model describes both the behavioral and structural features of the care system.

The behavioral modeling has been carried out using Use Case diagrams, State Machine diagrams, and Activity diagrams. The Use Case diagram shown in Figure 9 illustrates how different actors use the CONFIDENCE system. The Statechart use case describes how the User actor configures the system for its correct operation. The CallAlarmReceiver case describes the call protocol in case of an emergency. The system will determine if an alarm or warning has occurred and it will call the corresponding AlarmReceiver actor. The system will follow the call protocol configured by the User actor in the set up process. The ProcessReconstruction use case reconstructs the posture of the user. The ProcessSituation use case analyzes the reconstructed posture and determines whether there is an alarm situation, a warning situation or a normal situation. In case of alarm or warning, the system enters in the alarm or warning mode, the detected situation is saved and the system begins with the call procedure.

The functionality of each use case has been described using either a State Machine diagram or an Activity diagram depending on the modeling requirements of the use case. Use cases describing control-based behaviors have been modeled by means of State Machine diagrams, while data flow-oriented behaviors have been described using Activity diagrams. SystemSetup and CallAlarmReceiver use cases have been modeled as State Machines while ProcessReconstruction and ProcessSituation use case have been modeled as Activities.

The CallAlarmReceiver state machine is created as the classifier behavior of the CallAlarmReceiver use case. The CallAlarmReceiver state machine implements the call protocol configured by the User actor in the set up of the System. The SysML diagram of the CallAlarmReceiver state machine is shown in Figure 10.

At the beginning, the System is in the idle state and the state machine goes to the Alert state when the transaction called StartAlarmReceiver is received. If the User raises an alarm by means of UserRaiseAlarm transaction, the state machine goes from the idle state or the Alert state to the CallAlarmReceiver state and the system calls the configured receiver. During the whole process of the CallAlarmReceiver state machine, the possibility of resetting the system is available using the UserResetSystem transaction.

The ProcessSituation activity diagram analyzes the posture of the user. The CONFIDENCE ProcessSituation SysML activity diagram is shown in Figure 11.
The block definition diagram is the most widely used diagram to describe the structural elements that compose the system. The design team has defined the CONFIDENCE System block as the top-level element of the system being designed. This block contains the properties and operations of the system. The CONFIDENCE System block can define relationships with other block parts to describe system behavior. However, as this is the highest abstraction level model, the system has been modeled as a single block. Future designs can refine the system model described in this section to include structural hierarchy information.

Designers described the interfaces of the block CONFIDENCE System by means of input and output Ports. A pair of input/output ports are added for each actor to enable a bidirectional communication with the CONFIDENCE System block.

SysML offers basic data types, such as Boolean, Integer, String and Real data types. All the custom data types have been defined by means of DataTypes and Enumeration SysML elements.

5.2 CONFIDENCE SystemC-TLM code generation
The CONFIDENCE executable code is generated using the proposed ASSYST framework. ASSYST automatically produces the SystemC-TLM source code from the SysML model describing the system. The first step of the generation process involves the creation of a new C++ Eclipse project that will store the SystemC-TLM source code generated by ASSYST. The second step involves the automatic creation of formal FSM and SOF behavioral models from State Machine and Activity diagrams using the ASSYST ATL transformations proposed in Section 3. At the end of this step, the behavioral aspects of the CONFIDENCE system are described in the FSM and SOF models. In the third step, the SysML FSM and SOF models are automatically translated into a SystemC-TLM heterogeneous executable code using the ASSYST M2T generation process described in Section 4.

6 ASSYST FRAMEWORK ASSESSMENT
In order to assess the proposed MDD framework, the CONFIDENCE system has been developed following two different design flows, as depicted in Figure 13. On one hand, the design methodology proposed in this paper has been used for the development of the CONFIDENCE system. The design team has specified the system using a SysML model from which the SystemC heterogeneous executable model is created automatically by means of ASSYST. On the other hand, the SAWY library [27] previously created by our research group has been employed for the development of the CONFIDENCE core system [28]. SAWY library is built on top of the SystemC library. SAWY library extends SystemC capabilities providing design patterns to manually translate UML-based system descriptions into transaction level SystemC executable models. In this case, UML modeling language was used for the development of the CONFIDENCE system.

In order to evaluate the proposed system design methodology, the cost of creating an executable system model from a UML/SysML system description and the simulation performance of the executable model have been measured. The results are presented in Table 6.

The proposed MDD methodology utilizes SysML as the modeling language and supports the use of State Machine Diagrams (SMD) for control-based behavioral...
TABLE 6
Assessment of the proposed development framework

<table>
<thead>
<tr>
<th>Design Framework</th>
<th>SAWY</th>
<th>Proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modeling Language</td>
<td>UML</td>
<td>SysML</td>
</tr>
<tr>
<td>Event-driven Architecture</td>
<td>SHM</td>
<td>SysML &amp; AD</td>
</tr>
<tr>
<td># of libraries</td>
<td>1022 LOC</td>
<td>2298 LOC</td>
</tr>
<tr>
<td>Size in LOC</td>
<td>1022 LOC</td>
<td>2298 LOC</td>
</tr>
<tr>
<td>Development Time</td>
<td>1 month</td>
<td>1 month</td>
</tr>
</tbody>
</table>

D.1 International Journal Papers 351

7 CONCLUSIONS AND FUTURE WORK
In recent years, MDE has been proposed as a methodology for electronic systems design. Specifically, the SysML visual modeling language offers a promising perspective for the efficient specification of complex electronic systems using MDE. On the other hand, SystemC-TLM has been proposed as a promising library to create executable code for early electronic system definition.

In this paper, a MDE framework called ASYST has been proposed to translate electronic system designs described using SysML to SystemC-TLM heterogeneous executable models. ASYST is a Eclipse plug-in based on M2M and M2E techniques using ATL and Acore. ASYST implements both the formal mapping described in Part I [8] and the SystemC-TLM executable code generation processes.

A case study illustrates both the application of the ASYST framework within a complex electronic system and the benefits of the proposed approach. The CONFIDENCE project described in the case study has been also designed using the ASYST framework. A comparison between both design tools has been presented.

The case study demonstrates that the clearly defined MOCs and the formal behavioral mapping described in Part I of this work can be easily used for the translation of an electronic system design into SystemC-TLM executable code describing both control-oriented and data-flow based behaviors. This heterogeneity improves the flexibility and the practical use of ASYST within an electronic system design process.

The automatic code generation is a key characteristic of ASYST. The SystemC-TLM executable model can be created immediately. This characteristic enables to easily modify the architecture of the system model, reorganizing the system structure (SysML Blocks) or the behavioral aspects (SysML Use Cases, State Machines or Activities), and then to generate again the SystemC-TLM code in a very short time period. As a result, ASYST improves the initial iterations carried out to tune the architecture of the high abstraction functional model and facilitates what-if analysis.
D.2 INTERNATIONAL CONFERENCE PAPERS


A Transaction Level Assertion Verification Framework in SystemC: an Application Study
A Transaction Level Assertion Verification Framework in SystemC: an Application Study

K. Tramosa*, J. F. Sevillano*, J. Ríos*, A. Cortés* and I. Vélez*

*CEIT and TECNUN (University of Navarra), Manuel Lardizabal 15, San Sebastián, Spain
{tramosa,sevillano,cortes,rios}@ceit.es
*SISTEPLAN, Parque Tecnológico de Holanda, 607, Desta, Spain
j.rios@sisteplan.com

Abstract—This paper presents a new transaction level assertion verification framework, built on top of SystemC, to support the integration of Assertion Based Verification in a Model Driven Design methodology. A key point of the proposed framework is that it enables decomposing the work of the design and verification teams. This is possible thanks to data introspection capabilities, the fact that the assertions are not embedded in the design model code, and the abstraction in the property specification. Thus, the two teams can work in parallel starting from the natural language specification, reducing the development time.

Index Terms—Transaction Level Modeling, Assertion Based Verification, SystemC.

I. INTRODUCTION

Model Driven Design (MDD) has been proposed to handle the complexity of the design of modern electronic systems [1]. A MDD approach is based on creating a set of executable models describing the system to be built. Executable models are software models of the system that can be analyzed and executed on a workstations. Each model within a MDD flow represents the system at a different abstraction level. SystemC [2] has become the de facto standard to create executable models of electronic systems. SystemC is an ANSI standard C++ class library for system and hardware design. The SystemC core language provides fundamental system modeling components, for example, modules, ports and channels. Moreover, SystemC enables the simulation of concurrent processes. Transaction level modeling (TLM) is being proposed for early system definition and architecture exploration. The transaction level (TL) models are very fast to simulate. Additionally, they enable early software development. The Open SystemC Initiative (OSCI) has developed TLM v0.1 and TLM v2.0 libraries in order to support TLM system models using SystemC.

The MDD process performs model transformations to increase the detail of the system representation through several abstraction levels. These model transformations are commonly performed manually. Therefore, the correctness of the transformation should be verified.

On the other hand, Assertion Based Verification (ABV) has become a successful method for the verification of electronic systems. Using this method, system properties are expressed by means of assertions. An assertion is a directive given to a tool to check a property on a model of the system. These assertions are intended to detect design process bugs. The executable models based on MDD enable functional verification based on assertions. The use of assertions at RTL is a popular technique. Recently, it has been proposed to raise the abstraction level of the assertions to the TL [3] [6]. A reliable design flow is to have separate design and verification teams [7]. The design team creates design models of the system using the specification as an initial input. The verification team creates assertions from the system specifications. The redundancy introduced by working with these teams mitigates the probability of a specification misinterpretation. In order to reduce the development time, it should be possible to build these early design models and verification assertions fast, at TL; it should be possible to have both teams working in parallel, and it should be possible to reuse assertions between different models. A TL modeling design for verification methodology based on assertion reuse is proposed in [3] [4]. In [5], a TL assertion framework is proposed. The assertions are specified using a custom language. These assertions are then provided to a custom compiler that generates a SystemC implementation of the assertions, which is merged with the SystemC model of the design. A modification of assertions or a modification of the design model needs to go through the whole custom compiler flow which slows down the system development process. On the other hand, the approach described in [5] has limitations when a proposition references the data in a common proposition within the same assertion, which in turn can obscure the coding of the assertion and demands a detailed knowledge of the design model by the verification team.

In [6], the TL assertions are embedded in the code of the design model using a set of proposed functions and macros. Different macros and functions are proposed according to the level of abstraction of the model. Embedding the verification code in the design model leads to a very complex code. Furthermore, it makes it difficult for the verification team to work in parallel with the design team.

This paper proposes a system design flow with integrated ABV. The flow is supported by a library built on top of SystemC. The TL framework enables the verification team to write early design assertions simply and fast, at a very high level of abstraction. It enables the verification team to write assertions with increased detail and it is prepared to
support a TL model driven design flow. Furthermore, the assertion specification is decoupled from the design model. The assertion specification is specified by means of an XML file. This enables the verification team to work in parallel with the design team, reducing the time delay. The assertion specification XML file is read during the elaboration of the design model and the appropriate elements for the verification are built automatically. Thus, a modification in the assertion specification does not need access to the design model code or netlisting. The proposed framework is very flexible and can be attached to any SystemC-TL model.

This paper is organized as follows: Section II describes the integration of a ABV methodology within a MDD design flow. Section III defines the structure and the evaluation mechanism of the proposed assertions. Section IV presents the implementation of the verification framework and it describes the system model verification process. A case study illustrating the application of the proposed verification framework is presented in Section V. Finally, Section VI concludes the paper.

II. INTEGRATION OF ABV IN A MDD FLOW

The MDD methodology is based on executable models describing the system to be built. As the transformations between abstraction levels are performed manually, ABV is used to verify the model transformations. Using ABV, system properties are expressed by means of assertions which are further checked against the system model using dynamic verification methods. The proposed design flow considers separate design and verification teams. The employed methodology is an iterative process, where new design details are added in each iteration. The SystemC executable models can be classified according to their abstraction level of the functionality into Level 0, Level 1, etc.

In the first step of the design flow, the group responsible for the system design creates the Level 0 model, while the verification group defines the first assertion set of the system. Level 0 describes the system and its transactions with the environment at the higher level of abstraction. System designers and verification designers use the same system specifications as a reference. The Level 0 assertion list checks that the design team has correctly interpreted the initial system specification.

In a second step, the Level 0 system model is refined by the design team yielding the Level 1 model. The Level 1 model takes into account the new requirements appeared for each module and the interfaces between the devices. Additionally, the verification team adds new assertions to the ones defined in the first step in order to cover the new model details. This way, the Level 1 model is verified against the initial requirements and the new ones that take into account details of the implementation. The Level 1 assertions are also reused in the Level 1 model, to verify that the new more detailed Level 1 model is functionally equivalent to the Level 0 model. However, the new details require more assertions to verify the correct behavior of the system. These refinement steps can be repeated taking into account more implementation details.

In order to apply ABV techniques efficiently to a model driven methodology, two key elements are necessary: a clear definition of how assertions are interpreted and a verification framework to evaluate the assertions during simulation.

III. TL ASSERTIONS

In a TL model, the behavior of the design under verification (DUV) manifests itself as a flow of transactions. In order to verify the correctness of the DUV, we observe the flow of transactions in the DUV. Thus, the fundamental unit of information for TL verification is the transaction. We define a transactional event (TE) as the occurrence of a transaction.

An assertion is a directive given to a tool to check a property. A property is defined as a sequence of propositions about TEs occurring in a defined order. Those propositions are combined to describe a TE's pattern by means of common binary logical connectives defined in propositional calculi such as conjunction (and), disjunction (or), and implication. A proposition is split into four simple parts:

- mode: According to the behavior of the proposition we can distinguish two types of propositions: positive propositions and negative propositions. This attribute supports the negation (not) logical connective.
- trigger: It specifies the identities of the initiator or the target involved in the TE, the type of transaction, and the sequential relationship between the time instants of the TEs involved in the evaluation of the property.
- timeout: It describes a timeout condition.
- expression: It is only evaluated when the trigger is true. It describes a relationship between the information transmitted in the TEs involved in the evaluation of the property.

By means of the proposed TL assertions, it is possible to describe system properties expressed using Linear Temporal Logic (LTL). Additionally, the timeout attribute of the proposed TL assertions considers simulation timings of the executable model which improves the temporal expressibility of LTL.

Let us analyze an example. We have a system with a CPU and a memory. For this system, we have the following property expressed in natural language:

If a write operation is performed on memory, no more write operations are performed at the same memory address, and the memory is requested to read that address, then the memory shall return the value previously written before 3 ms.

This property can be split into four propositions, where each proposition is focused on one TE:

P1: The target of $t_{w1}$ is the memory, and $t_{w1}$ consists of a memory write operation.
P2: The target of $t_{w2}$ is the memory, but there is no memory write operation $t_{w1}$ at the same memory address as in $t_{w1}$ performed between the time instants of $t_{w1}$ and $t_{w2}$.
P3: The target of $t_{w3}$ is the memory, $t_{w3}$ consists of a request to read operation at the same memory address as in $t_{w1}$, and $t_{w3}$ is performed after the time instant of $t_{w1}$.
TABLE I

<table>
<thead>
<tr>
<th>Propositions</th>
<th>Trigger</th>
<th>Time-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: state: positive</td>
<td>t0 is in memory</td>
<td>true</td>
</tr>
<tr>
<td>P1: mode: positive</td>
<td>t0 is in memory, t1 is a memory read operation, t2 happens later than t0</td>
<td>false</td>
</tr>
<tr>
<td>P2: mode: negative</td>
<td>t0 is in memory, t1 is a memory write operation, and t2 happens after t1</td>
<td>true</td>
</tr>
<tr>
<td>P2: mode: positive</td>
<td>t0 is in memory, t1 is a memory read operation, and t2 happens after t1</td>
<td>false</td>
</tr>
</tbody>
</table>

P1: The initiator of t0 is the memory, t0 is a memory read operation at the same memory address as in t1, the data in t0 is the same data as in t1, and t2 is performed no later than 3 ms from the time instant of t2.

P2: The initiator of t0 is the memory, t0 is a memory write operation, and t2 happens after t1.

P3: The address in t1 is the same address as in t0.

P4: The address in t1 is the same address as in t0.

Table I shows the analysis of the above propositions into their modes, triggers, time-out and expression parts.

A. TL Assertions Evaluation

The concept of state of a proposition is introduced for the TL assertions evaluation. At certain instant of simulation, the propositions can be in the three states: pending, fetched, and unachievable. The behavior of the evaluation of a positive and negative proposition is depicted in Figure 1.

In a positive proposition, either a TE that meets the trigger condition or a time-out can produce a state transition. In a negative proposition, a state transition can also be produced by the state transition of a later proposition within the property.

Initially, all the propositions within the property are in the pending state. As the simulation of the DUV produces TEs, the propositions of the property change their state. When a proposition changes its state, a new instance of the property can be built to allow an overlapping evaluation of the property. The new instance of the property is a clone of the former instance before the change of state. According to the state of the propositions within an instance of a property, the instance can be said to be in four states: pending, partially-fetched, fetched or unachievable. The pending state means that all the propositions in that instance are in the pending state. The fetched state means that all the propositions in that instance have been proven to be unachievable. The partiallly-fetched state means that at least one of the propositions in that instance is in the unachievable state.

IV. VERIFICATION FRAMEWORK IMPLEMENTATION

The proposed verification framework runs in parallel to the simulation engine, evaluating the assertions and collecting statistical data of the evaluation process. The verification framework has been developed using C++, SystemC [2], SystemC Verification Library [8], Boost C++ libraries [9], Xerces [10] and Xalan [11] libraries.

The verification code is not embedded within the system model code, providing independence to the design and verification teams. The assertions are specified by means of XML files, which can be easily understood by human beings.

Figure 2 shows a class diagram of the most relevant parts of the framework. The key components of the verification framework are presented below:
Class `AssertionManager`: this class is responsible for the coordination and resource management of the system verification process.

Class `Monitor`: it creates, manages, and deletes the instances of system properties defined in the XML assertions files.

Class `Monitorable Transactional Ports (MTPs)`: the main task of the classes `MTP_in` and `MTP_out` is to detect TEs occurring in the transactional ports and pass the information to the attached monitor. Each monitorable port is input or output transactional port. A monitor is the base class for MTPs. MTPs are programmed using aspect-oriented patterns to extend the TLM v1.0 transactional port functionality and to provide methods to enable the communication between transactional ports and the verification framework through the monitors.

In the following, we describe how the framework integrates within the SystemC simulation and performs the assertion evaluation.

### A. System Model Integration

The central role of the proposed verification framework is to integrate with the SystemC simulation framework. Figure 3 illustrates how the proposed verification framework is used within the SystemC model.

Three changes are necessary in a TLM SystemC model to integrate the proposed verification framework:

1. Create an `AssertionManager` object inside the SystemC code in order to enable all the verification framework capabilities.
2. Replace transactional ports with MTPs provided by the proposed verification framework. MTPs are needed to capture the transactions occurring in the transactional ports. From the SystemC simulation point of view, the MTPs provided by the verification framework act as TLM v1.0 transactional ports. From the point of view of the verification process, MTPs send information about the transactions passing through the monitorizable ports to the verification framework monitors.
3. Register MTPs in the `AssertionManager`. This code is implemented in the system testbench.

The second step could be avoided if the SystemC simulation framework uses MTPs right from the first model of the system under development.

#### B. System Model Verification Process

The verification process performed by the proposed framework within a TLM SystemC model is divided into three phases:

1. **Phase 1**: This first phase takes place before the SystemC simulation starts. Once the system modules are elaborated and the module hierarchy of the system is set up, the `AssertionManager` parses the XML file where the assertions have been specified. With the information extracted from the XML file, the list of assertions is generated.

   Next, one monitor is created for each MTP in the system model. All the monitors created are registered in the `AssertionManager`.

   At this point of the execution, the `AssertionManager` contains two lists representing all the assertions and all the monitors of the system model. In the final step of this phase, the `AssertionManager` binds the assertions with the appropriate monitors.

   Each transactional monitor stores an assertion list specifying which assertions the monitor must report TEs to.

2. **Phase 2**: The system simulation starts, and the evaluation of the properties takes place after the SystemC `sim_start` function is called. During the system simulation, the monitorable interfaces capture the occurrence of transactions (TIs) one-by-one and pass that information to the attached monitors. The `Monitor` distributes the TIs to all the assertions bound to the monitor for evaluation.

   An assertion can contain one or several instances of the property during the system simulation to support the overlapping and nonoverlapping evaluation modes.

   The `Assertion` uses the TE information received from the verification monitors to evaluate its collection of property instances. Each instance keeps track of the property evaluation process using an index pointer inside the proposition `AlmostEqual`. This pointer points to a proposition pending to evaluate inside the property. The property evaluation is a sequential process where the propositions are evaluated one after the other.

   Figure 4 illustrates an example of the evaluation of a property. In this example, the property states that proposition `P2` must be satisfied if propositions `P1` happens after proposition `P3`. Overlapping mode is assumed to be enabled in propositions `P1` and `P3`, but not in proposition `P2`. Initially, the `AssertionManager` creates an assertion instance using the description from the assertions XML file, and the evaluation index is placed in the first proposition. Step 2 shows what happens if a TE is produced in the DUV such that `P1` is satisfied. Then, the `P1` is marked as `Fetched` and the evaluation index of this assertion instance is shifted to `P2`. As the overlapping mode is enabled for `P1`, a new assertion instance is created. The `AssertionManager` then shifts the evaluation pointer of the new assertion instance to the first proposition. The third step assumes that a new TE is produced such that `P2` of the first property instance is satisfied. In this case, the evaluation
The system that is being developed within the CONFIDENCE Project [12] is presented as a case study. The main objective of this project is the development of a care system for the detection of abnormal events (such as falls or unexpected behaviors that may be related to a health problem in elderly people). This care system works both outdoors and indoors. It raises an alarm to the user if an abnormal situation is detected. The system can also detect changes in the user’s behavior and issue a warning. The system can call to an alarm service or a relative if the alarm is received when an alarm occurs.

V. CASE STUDY

The methodology explained in Section II has been employed for the development of the CONFIDENCE care system. The design team has used the SAVY library [13] to create the executable models that describe the system at the abstract level. SAVY provides a system design framework to easily go from a UML system description to a TL SystemC executable model. The executable models created have been classified according to the abstraction level of the functionality into Level 0, Level 1, and Level 2. The verification team has used the proposed verification framework to specify the assertions for each model. The verification framework has been coupled to the CONFIDENCE system model, enabling early functional verification. Figure 5 illustrates the application of SAVY and the Verification Framework.

The same input stimuli are reused for each level of abstraction. However, new monitors and assertions appear as the model increases its details. The proposed Verification Framework enables the verification designers to easily define the new system assertions and relate them to the new monitors.

The focus of the Level 0 model shown in Figure 5 was to define the interfaces of the system with the external actors (the user and the alarm receiver (AR)). The Level 1 model splits the system into several functional devices: a control device, which plays the role of a Base Station (BS), a Small Portable Device (SPD), which looks similar to a mobile phone and several tags. The BS is placed at home and will only be working when the user is indoors.

Figure 5 presents the Level 1 block diagram. In this case, the three modules communicate among themselves sending and receiving transactions. The user only interacts with the PD, whereas the AR is connected to both the PD and BS.

Figure 5 also shows the Level 2 block diagram of the CONFIDENCE system. In this level, both the PD and BS are further divided into four submodules according to their functionality: the localization submodule (L), the interpretation submodule (I), and System Interface submodule (SI).

B. Verification Framework Performance

In order to assess the proposed verification framework, two different executable models have been created at each level of abstraction: one with verification capabilities, and another one without verification capabilities (the verification framework is not coupled to the design model).
APPENDIX D: Research Results

<table>
<thead>
<tr>
<th>Table III</th>
<th>System Model and Verification Framework Integration Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction Level</td>
<td>Level 0</td>
</tr>
<tr>
<td>System Model Verification Integration</td>
<td>2.971 bas</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table III</th>
<th>Verification Framework Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction Level</td>
<td>Level 0</td>
</tr>
<tr>
<td>Verification Delayed</td>
<td>1.590 s</td>
</tr>
<tr>
<td>Verification Enabled</td>
<td>2.575 s</td>
</tr>
<tr>
<td>Verification Overhead</td>
<td>8.237 %</td>
</tr>
<tr>
<td>Assertion Evaluation</td>
<td>1.095 s</td>
</tr>
<tr>
<td>Assertion Overhead</td>
<td>42.5 %</td>
</tr>
</tbody>
</table>

A key point of the proposed verification framework is its simplicity to be coupled to a TL SystemC system model. The lines of code for the system models at the three abstraction levels are shown in Table III. It can be observed that few lines of code are necessary to integrate the proposed verification framework to the system model.

On the other hand, the CPU execution time is measured to compare the performance of the system models with verification capabilities and the models without verification capabilities. For each level of abstraction, the models have been simulated in scenarios that represent more than two hours of work of the system. The three levels of abstraction use the same input stimuli. The input stimuli has been designed to create more than 900 different situations where more than 100 alarms need to be handled. This scenario involves almost 19,000 input transactions.

Table III shows the CPU simulation time for the executable models at the three levels of abstraction. We can observe that the simulation time increases with the detail of the models. Additionally, the last row of Table III presents the number of assertion evaluations within the system model with verification capabilities. The number of assertion evaluations represents the activity of the verification framework.

The performance loss due to the verification framework depends on the number of assertion evaluations. In turn, the number of evaluations depends on the number of assertions defined for the system model. The performance loss increases slightly as the system model gets more detailed. This execution time increment of the verification process is due to the new assertions added in order to verify the new system details. Moreover, that less than one million assertion evaluations are performed in the Level 2 system model, the verification process represents only the 50% of the execution time.

VI. CONCLUSION AND FUTURE WORK

A design flow integrating MD design and ATEV has been presented. This design flow enables the verification of complex systems using assertions during the system model simulation. A novel TL verification framework supported by a library built on top of SystemC has been presented. This TL framework helps to reduce the development time in design flows with team redundancy (independent design and verification teams) from the initial natural language specification. This is achieved thanks to the decoupling of the assertion creation.

This decoupling is provided by several features of the proposed TL verification framework. Firstly, the assertions are not embedded in the design model code. Furthermore, a modification in the assertion descriptions does not need any model remapping. Secondly, the assertions exploit data introspection to free the verification team from the details of the implementation of the DUV. Additionally, clear steps have been presented in order to apply the proposed verification framework to a SystemC-TL model.

The proposed verification framework has been applied to a complex electronic system. This application example illustrates the benefits of the proposed approach.

Future work will be the comparison of the proposed verification framework and design methodology with relevant approaches in the literature. In order to do a fair comparison, the CONFERENCE project's system model needs to be implemented in the verification frameworks proposed in the literature.

ACKNOWLEDGMENT

This work has been funded in part by the Banque Government. The research leading to these results in Section V has received funding from the European Community’s Framework Programme FP7/2007-2013 under grant agreement n.241836.

K. Morais holds a NewAlum de la Universidad de Navarra grant. A. Cortes holds the Torres Quevedo grant No. PTO06-G03-005.

We would also like to thank Santiago Urcelayeta, Isharun Urgoiti and Maria Angeles Santos for their help.

REFERENCES

Embedding Matlab in SystemC Transaction Level Modeling for Verification


Embedding Matlab in SystemC Transaction Level Modeling for Verification

Koldo Tomasicena, Juan F. Sevillianoa, Naiara Aruebe, Ainhoa Cortesb and Igone Velez
aCETI and TECNUN (University of Navarra)
Manuel de Lardizabal 15, 20080, San Sebastian, Spain.
E-mail: [atomasena, jsevilliano, naia, acortes, velez]@tecun.de

Abstract—This paper presents an approach to perform the verification of correct algorithm cooperation to achieve system functionally using executable models. The approach employs a description of the architecture of the system as a SystemC transaction level model and a description of the algorithms as Matlab M-files. It is proposed to perform simulations using the SystemC architectural elements with a Matlab engine to execute their Matlab algorithm specifications. A library is proposed to abstract the SystemC developer from the low level SystemC—Matlab interaction. This way, both the system and algorithm teams can work efficiently in their preferred environments to perform early architecture-algorithm optimization.

Index Terms—Transaction Level Modeling, SystemC, System Design Methodologies.

1. INTRODUCTION

Model driven design has been proposed to handle the complexity of the development of modern electronic systems [1]. This approach creates executable models describing the system to be built. These executable models are software models of the system that can be analyzed and executed on a workstation. At each design step, these models become an executable specification of the system. These models can be used for verification or correctness of each design step. SystemC [2] has become the standard de facto to create executable models of electronic systems. SystemC is an ANSI standard C++ class library for system and hardware design. The SystemC core language provides fundamental system modeling components, for example, modules, ports and channels. Moreover, SystemC enables the simulation of concurrent processes in order to model hardware components.

In the initial models, the designer focuses on the functional description of the system and on the information exchanged between the system and the environment. During the first design steps, the system is divided recursively into smaller parts. The detail of the models increases to describe the functionality of the different parts and the information exchanged between them. Early system definition and architecture exploration models are very high abstract models. For the functionality, the focus is on what functions shall be performed, not on how the functions shall be performed. For the communications, the focus is on what information is exchanged, not on how the information is exchanged. These models of the system are addressed by means of untimed or approximative timed transaction level (TL) modeling techniques [3], [4]. These TL models are fast to build and they enable very high speed of simulation. Additionally, TL models can be verified using assertions [5], [6].

Once what the system modules shall do has been defined, the next design step is to develop the algorithms. The algorithms devised for each module will be included in the test system model. This new TL model is a first description of the how, and it is intended to check that the different algorithms cooperate correctly to achieve the high-level functionalities. At this stage of the design process, design iterations will be performed for architecture and algorithm tuning.

Algorithm development is commonly carried out using Matlab [7]. The algorithm designers produce executable specifications of the algorithms as M-files or Simulink models. ‘Translating’ the algorithms developed in Matlab from Matlab code to SystemC is a possible approach. However, in order to reduce the development cost, it is preferable to enable both the system design and the algorithm design teams to work in their preferred environments: SystemC and Matlab respectively.

Matlab provides an API [7] to enable access to the Matlab Engine from C programs. This API consists of several routines to handle the Matlab Engine sessions and the exchange of data between the C application and the Matlab Engine. However, the direct use of these routines in the SystemC code is cumbersome, i.e., it results in a awkward code, and it distracts the system design team from the system description.

This paper proposes to integrate a Matlab Engine into the SystemC TL model. During the initial iterations to tune the architecture and the algorithms, the TL SystemC model of the system will execute the Matlab description of the algorithms provided by the algorithm design team. This way, the system design team can work in SystemC, the algorithm design team can work in Matlab and the TL SystemC-Matlab model can reuse the verification resources developed for previous SystemC models. Furthermore, the paper proposes an abstraction layer for the Matlab API. This abstraction layer simplifies the interaction with the Matlab Engine from the SystemC model. Thus, the resulting code for the SystemC-Matlab model is clear, and the system design team can focus on the system description. The proposed approach facilitates verifying that the different algorithms cooperate correctly to achieve the high-level functionalities. Therefore, the proposed approach enables a robust design flow with a reduced development cost. A case study is presented to illustrate the integration of the proposed approach in a model driven design flow.
II. MATLAB’S C API

Matlab offers an API [7] so that the Matlab Engine can be accessed from C programs. This API provides resources to handle the Matlab Engine session, the exchange of data between the C application and the Matlab Engine, and the evaluation of commands by the Matlab Engine.

In order to access the Matlab engine, the Matlab’s C API offers routines to open a Matlab session as a separate process. When a Matlab session is opened, two pipes are created between the C application process and the Matlab Engine process. Opening and closing a Matlab session is time consuming, therefore, for optimization of the simulation time, Matlab sessions should be opened and closed only once per simulation whenever it is possible. Additionally, in order to reduce the number of Matlab Engines used and thus the cost of the simulation setup, the same Matlab Engine should be shared between the SystemC modules whenever it is possible. The SystemC model developer is responsible for handling all the session issues and making accessible the appropriate Matlab Engine session pointer to the different SystemC modules.

The exchange of data between the C application and the Matlab Engine is carried out using variables of type mexObject and a set of functions provided by Matlab. The procedure to transfer data stored in a C variable to a variable belonging to the Matlab Engine can be summarized as follows:

1. Create a mexObject object that reaches the C variable type.
2. Fill the newly created mexObject appropriately with the data stored in the C variable. This task can require handling of pointers at low level, which results in heavy code and is error prone.
3. Send the mexObject object to the Matlab Engine.
4. Delete the mexObject.

And the procedure to transfer data stored in a Matlab variable of the Matlab Engine workspace to a C variable is as follows:

1. Get the mexObject from the Matlab workspace corresponding to the desired Matlab variables.
2. Fill the C variable with the data in the mexObject. This task can also require tedious and error prone low level pointer handling.
3. Delete the mexObject.

If we want to process data stored in a SystemC variable using an algorithm specified in Matlab by means of an M-file or a Simulink model, and retrieve the result of this processing, the SystemC model developer needs to implement the above tedious data exchange procedure. Furthermore, the synchronization between the representations of the data as SystemC variables and as Matlab Engine workspace variables has to be implemented by the SystemC model developer.

The direct use of the Matlab’s C API in a SystemC executable model is cumbersome and awkward task. There are many issues that the system-design team is demanded to implement and look after. Moreover, these issues are exclusively related to the SystemC-Matlab interaction; they are error prone, and they are not related to the system design. In order to be able to use the algorithm M-file (or Simulink) specifications in a SystemC model efficiently, an abstraction layer is needed to free the system-design team from the SystemC-Matlab low level issues. This way, the system-design team avoids tedious and error prone tasks they can concentrate on the design of the system and the development cost is reduced. Sections III and IV describe the proposed abstraction layer.

III. MATLABENGINE++ ARCHITECTURE

The proposed Matlab’s C API abstraction layer is supported by a new C++ library called MatlabEngine++, built on top of the Matlab’s C API. This library is easy to use and exploits the object oriented programming features supported by the C++ language to enable an efficient integration of the Matlab Engine within a SystemC ESL model. The proposed library assists the SystemC model developer in handling the Matlab sessions, and it automates the data exchange between the SystemC model and the Matlab Engine. Thus, the proposed library MatlabEngine++ fixes the system-design team from the low level SystemC-Matlab interaction issues. Although the proposed library is intended for SystemC models, it can be used in any C++ application.

The software architecture of the MatlabEngine++ library is depicted in Figure 1 by means of a UML class diagram. In order to have a flexible and scalable solution, the architecture of the proposed approach relies on four main classes: MatlabSessions, which implements the concept of a Matlab session; MatlabSessionConfig, which stores the configuration of a Matlab session; MatlabVar, which handles the data exchange between the C++ variables and the Matlab session variables; and MatlabManager, which implements the resources to handle the different sessions from the C++ application. Class MatlabVar can be specialized to handle specific data types. The features of these four classes are described below.

A. Class MatlabSession

Class MatlabSession represents a Matlab Engine workspace. This class provides functions to easily manage a Matlab session: open a session, close a session, send commands to a session, and retrieve the result from a session. Commands are sent to Matlab sessions by calling the session’s function
Command string $c_k$, where $c$ is the command as if it were written into the Matlab console.

From the functional point of view, $MatlabSession$ is the access point to the Matlab workspace. When a $MatlabSession$ is created, it creates a Matlab process. This process runs in the background as a separate process from the main C++ program.

### D.2 International Conference Papers

#### 367

**Class MatlabSession**

This class encapsulates the configuration of a Matlab session. A Matlab session has attributes such as $Horiz$, which specifies the name of the machine where the Matlab session will be opened; $Dir$, which is the path to the Matlab executable within the machine specified as $Horiz$; and $Options$, which specifies additional Matlab options. The class provides functions to set and get the values of these attributes.

When a $MatlabSession$ object is created, a $MatlabSessionConfig$ object is passed as a constructor argument. The $MatlabSessionConfig$ object is attached to the $MatlabSession$ object to store its configuration. Moreover, the same $MatlabSessionConfig$ object can be attached to several $MatlabSession$ objects within a multi-session application to handle several identical Matlab sessions. A $MatlabSession$ object gets the configuration properly opened a Matlab session from the attached $MatlabSessionConfig$ object's attributes.

**Class MatlabManager**

Class $MatlabManager$ is the root element of the MatlabEngine++ library. This class is responsible for the resource management and provides routines to easily create and access $MatlabSession$ objects and $MatlabSessionConfig$ objects. $MatlabManager$ supports the management of a multi-session and multi-configuration Matlab environment. $MatlabManager$ contains two lists to store $MatlabSession$ and $MatlabSessionConfig$ objects allocated during the C++ application execution. Moreover, $MatlabManager$ automatically manages the memory of the created $MatlabSession$ and $MatlabSessionConfig$ objects, i.e., the C++ developer does not need to free any memory associated to the Matlab interaction at the end of the application.

A key point of $MatlabManager$ is its implementation using a singleton design pattern. Thanks to the singleton design pattern, $MatlabManager$ is unique in the C++ application and we access the same $MatlabManager$ object from anywhere in the C++ application. This feature presents many benefits in a System C model, where several algorithms need to be executed from different SystemC modules. With the proposed approach, the same Matlab session can be requested from different places of the model to execute different algorithms.

**Class MatlabVar**

This is the base class for the variable types supported by MatlabEngine++. Specialized classes for specific data types are derived from class $MatlabVar$. The constructor of class $MatlabVar$ has two arguments of type string $name$, which is the name of the variable, and $session$, which specifies the MatlabSession object the variable is attached to. $MatlabVar$ provides virtual functions to enable variable transfers and updates between the C++ application and Matlab. These routines are implemented in the specific data type classes. Each specialized $MatlabVar$ based class has its own specific C++ data type and the corresponding $matlab$ implemented as member attributes.

Class $MatlabVar$ also provides routines to manage the synchronization between the representations of data as C++ variables and the representation as $matlab$ for Matlab session interaction. The synchronization is handled by means of the state machine depicted in Figure 2. This state machine automates the data exchange between the C++ application and the Matlab session attached to the $MatlabVar$ object. Three states are possible:

- **Synchronized**: Both the C++ variable and the Matlab session variable store the same value.
- **C++CHANGED**: The C++ variable has changed, but the value of the Matlab session variable has not been updated.
- **MATLABCHANGED**: The Matlab variable has changed, but the C++ variable has not been updated.

Class $MatlabVar$ implements the mechanisms to detect if a variable is accessed for reading or for writing in the C++ program. When a $MatlabVar$ based variable is accessed for writing in the C++ program, its synchronization state is set to $C++CHANGED$ regardless of its original state. If a variable is accessed to read its value, its synchronization state is checked. If the state is $Synchronized$ or $C++CHANGED$, the variable returns its value. Nevertheless, if the state is $MATLABCHANGED$, the variable is updated from the Matlab session and its state is set to $Synchronized$ before returning its value. If a command is sent to a Matlab session, the corresponding $MatlabSession$ object updates all the attached variables that are in state $C++CHANGED$, and changes their state to $Synchronized$ before executing the command. The variables attached to the Matlab session are set to state $MATLABCHANGED$ after executing the command. The proposed synchronization scheme abstracts efficiently the steps needed to exchange data between the C++ application and Matlab, and thus, leads to a very intuitive and clean C++ code.

Table 1 summarizes the features of the data types currently supported. New specializations of class $MatlabVar$ can be easily developed to address other data types.
IV. MATLABENGINE++ OPERATION

The system model depicted in Figure 3 is used as an example to illustrate the operation of the MATLABENGINE++ library within a SystemC model. Source and Sink are simple SystemC sc_module connected to the device under test (DUT). Source generates the input data for the DUT, whereas Sink monitors the output data of the DUT. The DUT consists of two SystemC modules: A and B which perform some data processing.

Let us assume that the algorithms performed by A and B are specified in the Matlab M-files A_algorithm.m and B_algorithm.m respectively.

Listing 1 shows the relevant parts of sc_main where the Matlab Engine sessions are setup. First of all, a pointer to the MatlabManager is obtained by means of function GetManager() of class MatlabManager. This function always returns the same pointer, because MatlabManager is unique in the application thanks to the singleton pattern. A MatlabSession Config object is created by calling MatlabManager’S function CreateSessionConfig(). Once the MatlabSessionConfig object is configured, a Matlab session is created by means of MatlabManager’S function CreateSession(). After the Matlab session has been created and configured, it can be opened by calling the session’S function Open(). As a result, a new Matlab process is created as if command 'open_matlab' was executed at host 'PCT'. When the SystemC simulation is finished, the Matlab process is killed and the session is closed by calling the function Close().

Listing 2 shows the SystemC code for module A. Let us assume that the MatlabModel_A algorithm takes as input a scalar variable \( x \) and returns a 2x1 column variable \( \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \). Within the constructor, the MatlabSession is retrieved by means of the MatlabManager. Additionally, MatlabENGINE++ variables \( \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \) are defined as member attributes of the module. These variables are created within the constructor of the algorithmic modules and they are attached to the MatlabSession retrieved in the previous step. During the system simulation, the SC_MODULE executes \$\text{Command}("open_matlab" - "A_algorithm.m")\$; whenever it wants \$\text{Algorithm}\$ to process the data. The user works with variables \( x_1 \) and \( x_2 \) as if they were standard C++ variables. Their values are automatically synchronized with the Matlab session by MatlabENGINE++.

Thanks to the singleton pattern, MatlabManager is unique throughout the SystemC model. It is not necessary to pass the Matlab session as a constructor argument to the SystemC module, and thus, the software interface of the model does not need to be modified. As a result, pure functional modules and modules executing algorithms in a Matlab Engine are fully interchangeable within the SystemC model. Furthermore, in the example, both SystemC modules A and B can request in their constructors the same Matlab session. Then, both Matlab M-files A_algorithm.m and B_algorithm.m are executed in the same Matlab workspace. This has several benefits: less processes running; avoid consuming licenses unnecessarily; and Matlab workspace variables can be shared.

V. CASE STUDY

The system being developed within the CONFIDENCE Project [8] is presented as a case study. The main objective of this project is the development of a care system for the detection of abnormal events (such as falls) or unexpected behaviors that may be related to a health problem in elderly people. The care system works both outdoors and indoors. It raises an alarm when an abnormal situation is detected. The system can also detect changes in the users behavior and issue a warning. The system can call an alarm service or a relative (hereafter alarm receiver) when an alarm or warning occurs.
A. Proposed Design Approaches

A model-driven methodology is being employed for the system design. The executable models have been classified according to their abstraction level into Level 0, Level 1, and Level 2 and Level 3 models. The design flow considers separate system design, algorithm design, and verification teams. The system design team uses the SAVI library [4] to develop TI executable models with a high level of abstraction. The verification team uses the verification framework in [9] to apply assertion based verification to the executable models.

In the first design step, the system design team creates the level 0 system model taking as reference the functional requirements of the CONFERENCE system. The focus of the level 0 model is to define the interfaces of the system with the external actors, user and alarm receiver (AR), define the transactions between the system and the environment, and define what the main functions of the system are. These system requirements serve also as input to the verification team, so that they can write the first assertion that describes the system functionality. XML files are used to inject and record the transactions between the system and the environment in SAVI. The verification framework places monitors on the system interfaces to verify the functionality by means of the analysis of the transactions flow. The assertions are specified by means of an XML file. At the end of the simulation, the verification framework produces an XML report file. The upper part of Figure 4 illustrates the model setup.

The employed methodology is an iterative process, where new design details are added in each iteration. The system design team refines the models creating the level 1 and level 2 models. The verification team refines the assertions, taking into account the new design details. The level 1 divides the system into three functional devices: a central device, which plays the role of a Base Station (BS), a small Portable Device (PD), which looks similar to a mobile phone, and several tags. The user only interacts with the PD whereas the alarm receiver is connected both to the PD and BS. The tags communicate among themselves to determine the distances between them. This information is transmitted to the PD and the BS. For the level 2 models, both the PD and BS are divided into four submodules: the localization submodule (L), the reconstruction submodule (R), the interpretation submodule (I), and System Interface submodule (S). These initial models are functional descriptions of the system.

At this point, the system design team has an architecture proposal, with the functionalities and transactions between the modules specified. At what the system modules shall be defined in detail. Therefore, the algorithm design team can develop the algorithms (the low) for the different functionalities. The result of the activity of the algorithm design team are executable specifications of the algorithms as M-files. The algorithm design team can now insert the algorithms execution into selected level 2 functional modules for the verification of their correct execution to achieve the high level functionalities. The proposed MatlabScript++ library can help to achieve this SystemC-Matlab interaction in an easy and efficient way. As an example, in this case study, the tag module and the BS submodule are replaced by their algorithmic models to yield a level 3 model. The resulting model is illustrated in Figure 4.

The Matlab model of the tag module simulates an Impulse Radio Ultra Wideband (IR-UWB) system, where sequences
of UWB Gaussian pulses are transmitted among the tags over an additive Gaussian noise channel. A tag performs cross-correlations on the received signal in order to detect the sequences transmitted by other tags. Then, the tag transmits a response to the other tags. Each tag, the delay between the transmitted and the received signal is used to estimate the distances between the tags. This Round-Trip-Time ranging algorithm is described in [15]. The ISM-UWB system is simulated with a sampling frequency of 480 MHz. The set of estimated distances is the transaction between the tag module and BS-L. The BS-L shall estimate the tags coordinates. The Matlab model implements a Recursive Least Square algorithm [11].

### 3. Benefits of the Proposed Approach

In order to measure the benefits of the proposed approach for the verification of correct cooperation of algorithms to achieve high-level functionality, we pay attention to two issues: the cost of including the selected algorithm execution in the level 2 system model to yield the level 3 system model; and the simulation performance.

The cost of building the level 3 system model can be split into two parts: the cost of modifying the Matlab executable specification and the cost of modifying the SystemC level 2 system model. The cost of building the level 3 system model, in the design approach described in section 4.4, is reported in Table II. The algorithm design team developed their algorithms with a clear specification of the required functionality and input/output arguments. This produced algorithms executable specifications that needed no change for their use in the level 3 system model. Therefore, the cost of modifying the Matlab executable specification was zero.

Table III illustrates the cost of modifying the SystemC level 2 model. The table shows the number of code lines added by the system-design team in the SystemC level 2 model to yield the SystemC level 3 model. As it can be observed, the cost of modifying the SystemC model was very small, despite the complexity of the algorithms included in this case study.

The proposed approach achieves system models that enable verification of correct algorithm cooperation with a very small cost. The reduced cost of building these models enables early high-level design exploration, where different architectures and algorithms can be analyzed to optimize the system.

Table III analyzes the simulation performance. The data in the table corresponds to a simulation where the BS-L performs more than 600 coordinate estimations. For each coordinate estimation, the tags simulate the BS-UWB system. The first row is the time consumed by the SystemC model; the second row is the time consumed by the Matlab engine executing the algorithms. The third row is the time consumed in the SystemC-Matlab interaction. It can be seen that the overhead due to the SystemC-Matlab interaction is very small, despite the amount of data transferred between the SystemC and the Matlab domains.

### VI. Conclusions

The Matlab API can be abstracted using the proposed library MatlabEngine++. This abstraction simplifies the task of integrating a Matlab Engine in a SystemC model. The inclusion of a Matlab Engine in a SystemC model enables building UWB models where architecture (modules and transactions) is described in SystemC and algorithms are described in Matlab. Thanks to the proposed approach, the cost of building these models is very small, and the resulting model code is clear. The SystemC-Matlab/Engine++ models enable the verification of correct algorithm cooperation to achieve system functionality, with negligible additional costs in development and simulation performance degradation. Thus, architecture and algorithm tuning can be performed efficiently early in the design, with the system-design and the algorithm-design teams working in their preferred environments.

### VIII. Acknowledgment

The research leading to these results has received funding from the European Community’s Framework Programme FP7/2007-2013 under grant agreement n. 216996 Consortium: CETT, HFG, BIE, IST, Berlin, COOS, Matica, University of Jyväskylä, Umeå Municipality, eChips, CUP2003; and ZENON. K. Trumper holds the Patronato de la Universidad de Navarra grant, A. Conde and N. Arme hold the Torres Quevedo grant No. P10606-12-95 and No. P10606-14-987, respectively.

### References

Efficient Monte Carlo Simulation Using SystemC
FDL: Forum on specification and Design Languages, Darmstadt, 2006
Efficient Monte Carlo Simulation Using SystemC

**Abstract** — SystemC has been proposed as a means of using the same description language throughout the stages of the design process. During the design and study of communication systems, performance metrics are commonly estimated using Monte Carlo. An easy-to-implement methodology is proposed to achieve efficient Monte Carlo simulations.

I. **Introduction**

The development of integrated circuits and embedded systems usually requires the use of heterogeneous tools and languages. This mixture of tools and languages is error-prone and time-consuming. The Open SystemC Initiative (OSCI) proposed SystemC [1] to overcome this problem. SystemC is expected to become a single modeling framework for the different abstraction levels of the electronic-system design flow. Recently, SystemC has gained popularity and has been approved as IEEE standard. Many examples have been presented where SystemC is part of their design flow [2], [3].

The first step of the design of an electronic system is the selection of the algorithms to achieve the required performance. In the design of a communication system, this performance is commonly evaluated by means of Monte Carlo (MC) simulations. If SystemC is to be employed throughout the development of communication systems, MC simulations should run efficiently in SystemC. Additionally, the complexity of the implementation of MC should not distract the designer from the study of the communication system.

This paper studies how to perform MC simulations with SystemC. A simple methodology is proposed and its efficiency is demonstrated by means of a case study.

II. **Requirements of MC Simulations**

The simulation of a communication system should be viewed as an statistical experiment. The MC method is used to estimate metrics of performance [4]. Basically, the MC method averages over realizations of a random variable related to the metric of interest.

For example, the bit-error rate (BER) estimation is an important metric of the performance of any communication system. In order to estimate its value, a random variable \( z_i \) is defined such that \( z_i = 0 \) when a bit is correct at the output of the receiver and \( z_i = 1 \) when it is not. The BER is the expectation of \( z_i \) and an estimate of the BER can be obtained through the sample mean of \( z_i \):

\[
P = \frac{1}{N} \sum_{i=0}^{N-1} z_i
\]  

The communication system is simulated to obtain realizations \( z_i \) of the random variable \( z \). MC is popular due to the simplicity of its implementation.

Additionally, MC can provide information on how good our estimate of the performance metric is. This can be done easily if the \( \{z_i \} \) are independent. Nevertheless, communication systems often include elements with memory (filters, feedback loops, etc.), and the independence cannot always be guaranteed within the same simulation. In order to overcome this problem, the values of \( \{z_i \} \) can be obtained from different simulations. A more efficient approach is to perform partial BER estimations, \( \{P_k\}, k = 0, \ldots, M - 1 \), from \( M \) different simulations. Then, these \( \{P_k\} \) can be treated as independent variables for the MC method.

\[
P = \frac{1}{N} \sum_{k=0}^{M-1} P_k
\]  

In order to ensure the independence of the \( \{P_k\} \), proper control of the random generators is needed during the MC estimation process.

Hence, the SystemC simulation environment must provide a framework where independence of the partial simulations is guaranteed. Furthermore, the environment should permit the rapid execution of a great number of these partial simulations, as \( M \) is often very large.

III. **MC Simulation Using SystemC**

There are two possible approaches to carrying out the kind of simulations described above using SystemC:

- An external program performs all the MC control. It calls the SystemC executable, collects the \( \{P_k\} \), and decides when the MC estimation is finished.
- MC control is carried out within SystemC. All the functions associated with the MC method are performed by the SystemC executable.

The first approach is the most straightforward and easy to implement. The external program can be a shell script, a compiled executable or even MatLab. However, the continuous load and unload of the SystemC simulation executable introduces a large penalty in runtime. Furthermore, measures must be taken to ensure that the random generators of the SystemC model maintain their independence through the MC estimation. The second approach has a big potential to improve the runtime and makes it easier to control the random generators.

A. **MC Control Within SystemC**

If we want to carry out the MC method in a single SystemC executable, the simulation context should be reset to ensure the independence of each simulation. The class \texttt{勠nter} from the SystemC library could be used for this purpose. This class has a member function called \texttt{reset}, that is intended to free the memory allocated for the objects of the SystemC simulation. This way \texttt{reuinit} could be called iteratively within a C++ program. However, this methodology has the drawback that the C++ executable is building and destroying the SystemC model continuously. Moreover, during the study of this
methodology, a memory leakage was found. The problem is documented in file scanpoolapp of the SystemC library source code:

```c
sc2_locator::sc2_locator() {
    // Shouldn’t free the blocklist, since global
    // objects that use the memory pool may not have
    // been destroyed yet ...
    // Let it leak, let it leak, let it leak ...
}
```

Due to these problems a different approach is needed for efficient and robust MC estimation within SystemC.

Generally, each module of the communication system is associated with a class derived from sc-module. Its constructor performs commonly two types of actions:

- It builds the object (it reads parameters from files, carries out memory allocation, etc.
- It gives their initial values to all the variables of the object.

The proposed methodology consists in encapsulating the second action inside a member function of each module. This function is called reset. The constructor of each module performs all the building actions and calls this member function to set the variables to their initial values:

```c
module(sc_module::reset)...

        sc2_locator::sc2_locator();
        // Read config files, allocate memory, etc.
        // Initial values of variables.
        reset();
```

Every time this function reset is called, the corresponding object is restored to its initial state.

Now, independent simulations can be done efficiently within SystemC. A single infinite simulation (sc.start(1)) is run. A control module embedded within the SystemC description monitors the simulation. In the case of BER estimation, this module decides when a partial simulation has finished, i.e., a value of \( P_b \) has been obtained. The controller of the simulation calls the reset function of each module in the system. This way the system is restored to its initial state and a new partial simulation can begin. In hierarchical system descriptions the reset function of the top module calls the reset function of its submodules.

In the simulation of communication systems, random signals are usually obtained from the manipulation of independent pseudo-random generators [4]. In order to guarantee the independence of the partial simulations, the reset function of these pseudo-random generators should not clear their internal registers.

The proposed methodology is easy to implement and no specific expertise is needed in simulators. The engineer can concentrate in the design and study of the communication system. The proposed methodology does not need to build and destroy continuously the SystemC objects, which speeds up the simulation. The managing of the random generators is simplified as well.

IV. CASE STUDY

In order to validate the proposed methodology, the QPSK data transmission system shown in figure 1 has been used. The transmitter, synchronization offsets and channel blocks are data flow models. They are connected using FIFO channels. A new class has been derived from sc_signal where a reset function has been added. The receiver is a clock cycle model, where communication is done by means of channels of type sc_signal.

The degree of complexity of this system is representative of a communication system in a middle stage of development. Some algorithms are not fixed yet and their performance has to be explored. MC is to be used to estimate the BER for different values of \( E_b/N_0 \). This is a computing intensive problem. Table I compares the runtime when an external program is used to control the MC simulation and when the proposed methodology is employed. The external program in table I is Matlab.

Experiments have been conducted using a compiled C++-program as external MC controller, but the results are similar. As it can be seen, the proposed methodology produces gains in runtime of more than 50%.

V. CONCLUSIONS

The Monte Carlo performance metric estimations used during the design and study of communication systems are very computing intensive. The Monte Carlo method needs a specific type of simulations. This kind of simulations can be efficiently implemented within SystemC using the proposed methodology.

REFERENCES

EFFICIENT MONTE CARLO SIMULATION USING SYSTEMC

Juan F. Sevillano, Iñigo Vélez, Koldo Tomasa, Andoni Irizar
CEIT and TECNUN (University of Navarra), Spain, {sevillano, vellez, ktomasa, anizar}@ceit.es

Abstract

SystemC has been proposed as a useful means of using the same description language throughout the stages of the design process. During the design and study of communication systems, performance metrics are commonly estimated using Monte Carlo. An easy-to-implement methodology is proposed to achieve efficient Monte Carlo simulations.

I – Introduction

- The development of integrated circuits and embedded systems usually requires the use of heterogeneous tools and languages.
- SystemC is expected to become a single modeling framework.
- Recently, SystemC has been approved as the EEE standard (IEEE-1500).

II – MC Simulation Requirements

- The simulation of a communication system should be viewed as a statistical experiment.
- Basically, the MC method averages over realizations of a random variable related to the metric of interest, $L$.

\[
E[L] = \frac{1}{N} \sum_{i=1}^{N} f(x_i)
\]

- Independent samples

III – SystemC MC Simulations

A1: External Control

- Most common approach
- External Program shell script, Makefile...

B1: Internal Control

- Great potential to improve runtime.
- Homogeneous framework.

B1: First Approach

- Use of SystemC’s sc_environ class.
- sc_main should be called explicitly.

```c
void sc_main(void) {{
  sc_environ env;
  sc_main_init(&env);
  \( \text{Simulate system} \)
}
```

- Dilemma:
- C++ inavescible is building and destroying the SystemC model continuously.
- Memory leakage found in sc_environ.cpp.

B1: Proposed Approach

- Create a new() member function in each module.
- Create a simulation controller module.

```c
sim_controller_new();
//Build model
for(i=0;i<100;i++)
//Simulate system
//Collect \( \epsilon_i \)
//Prepare system
\( \text{Simulation controller thread} \)

//Destory system

sim_controller_new();
//Prepare system
\( \text{Simulation controller thread} \)
//Destory system
```

IV – Case Study and Results

- In order to validate the proposed methodology a QPSK data transmission system has been designed.
- MC is used to estimate the SER for different values of SNR.
- MC simulations are efficiently implemented within SystemC using the proposed methodology.

<table>
<thead>
<tr>
<th>System</th>
<th>Proposed</th>
<th>External Control</th>
<th>Proposed</th>
<th>External Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>0.9 hrs</td>
<td>3.2 hrs</td>
<td>0.9 hrs</td>
<td>3.2 hrs</td>
</tr>
<tr>
<td>Savings</td>
<td>60%</td>
<td>60%</td>
<td>60%</td>
<td>60%</td>
</tr>
</tbody>
</table>

- The proposed methodology produced gains in runtime above 60%.
The Best Poster Award

For the paper entitled

Efficient Monte Carlo Simulation Using SystemC

J. F. Sevillano, I. Veíez, K. Tomasea, A. Inzar
CEIT and TECUN
San Sebastián, Spain

presented, on behalf of the Programme Committees to

FDL'06
Forum on Specification and Design Languages
Darmstadt, Germany

S. A. Huss - General Chair - September 2006
D.3 NATIONAL CONFERENCE PAPERS


High level System Design Using SAVY: an Application Study
High Level System Design Using SAVY: an Application Study

Tomassena K., Vélez I., Cortés A., Pérez J., Sevillano J. F.

1 CEIT and Tecnun (University of Navarra), Manuel Lardizábal 15, San Sebastián, Spain, {tomassena, velez.asores, jsesevillano}@ceites
http://www.ceites
2 SISTEPLAN, Parque Tecnológico de Bizkaia Ed. 607, Derio, Spain, jpcpcra@sisteflante.com
http://www.sisteflante.com

Abstract. This paper presents the translation procedure to easily go from UML system descriptions to Transaction Level SystemC executable models using the SAVY library. This library simplifies the work of the system-design team: it enables the reuse of models, it facilitates model integration and it reduces the development time of complex electronic systems. A case study is presented to demonstrate the application of SAVY within the design flow of a complex system.

1 Introduction

The model driven design methodology has been proposed to handle the complexity of the design of modern electronic systems. This approach is based on executable models describing the system to be built. These executable models are very useful to specify the functionality of a large system, where different engineers work together to develop the system. The use of design patterns simplifies the work of the engineers, helping them to proceed in a simple and regular way. Moreover, design patterns facilitate the model integration and communication between teams.

A system description is made of numerous executable models, each model representing a different level of abstraction [1]. The models must describe both the interactions and the functionality of the system. In order to abstract the interactions, transaction level modelling (TLM) has been proposed. On the other hand, the executable models can be classified according to their abstraction level of the functionality.

A key point of the model driven methodology is the transformation between models. The behaviour of each model can be compared with the former one to check that the new more detailed modules cooperate correctly to achieve the required functionality. SystemC [2] has become the de-facto standard to create executable models of electronic systems. Within SystemC, the OSCI TLM working group has developed the TLM 1.0 standard [3]. Additionally, this group has recently published the TLM 2.0 standard [4] that solves some of the problems of TLM 1.0.

On the other hand, Unified Modelling Language [5] (UML) is widely accepted in the software engineering community as a system modelling tool. UML is used to describe the functionality of a software system starting from a very high level of abstraction. Several
research studies have focused on methodologies to create SystemC models from an UML
description [6]-[11]. An earlier effort is YAML [6], which only uses UML to capture the
structural aspects of the system under design. In [7] and [8], UML model specifications are
translated into AsML and Rebeca modelling languages respectively. Then these
intermediate models are mapped to SystemC TLM executable models. In both cases, the
creation of a executable model requires two translation steps. References [9]-[11] propose
UML-to-SystemC translation procedures based on the UML profile mechanism [12]. These
proposals do not provide clear UML-to-SystemC translation rules. Besides, these research
works are not focused on generating TLM compliant SystemC code. Recently, SAVY [13]
was proposed to provide clear design patterns to model the functionality of a system at high
levels of abstraction. SAVY enables to create high abstraction SystemC executable models
from UML diagrams. However, the procedure to go from UML to SAVY is not clearly
explained in [13].

This paper proposes a translation procedure to go from a UML system description to a
Transaction Level (TL) SystemC executable model using SAVY. A clear step-by-step
design methodology is presented. Another contribution of this paper is the description of a
case study to demonstrate the application of SAVY within a complex system design flow.

2 SAVY Architecture

During the first stage of a system design flow the system design team creates a UML
description of the system to be developed. SAVY [13] can be used to easily create a
SystemC Transaction Level (TL) executable model from the system behaviour capture
carried out using UML diagrams. Reference [13] proposes to use UML Use Case diagrams
and UML Statechart diagrams to describe the functionality and the interfaces of the system.

Two design patterns are proposed in [13] to model the actors and the system actor
architecture pattern and entity architecture pattern. These design patterns take as input the
information obtained from the UML diagrams. Figure 1 illustrates the transformation of the
UML diagrams into an executable model using the actor and entity architecture patterns.

SAVY is built on top of the C++ standard library, SystemC library, Xerces library [14]
and the Xalan library [15] as depicted in Figure 2. Figure 3 shows the UML class diagram
of the SAVY library. SAVY library extends SystemC capabilities providing design patterns
to easily translate UML based system descriptions into TL SystemC executable models.
Created executable models can be used to analyse the system behaviour at high levels of
abstraction. These TL SystemC models are compliant with TLM 1.0 [3].

2.1 Actor Architecture Pattern

SAVY proposes patterns and provides tools to create elements in the executable model
that play the role of the actors. In a TLM environment the fundamental information unit is
the transaction. For directed tests, the engineer must specify the stimuli, i.e., the
transactions, to be applied to the system model.

Figure 1 shows the patterns to implement the actors of the UML Use Case diagram in
the SystemC executable model. These elements inject stimuli to the design and they receive
(and record) the response of the system. As the figure shows, it is proposed to use XML
files for the description of both the transactions to be injected and the transaction to be
recorded. SAVY proposes to write each transaction as an XML node, which has as XML attributes the base transaction class attributes plus the specific attributes of that type of transaction. The XML transaction generator (XTG) and the traffic injector (TI) blocks are used for the injection of the transactions. The XML transaction recorder (XTR) and the traffic receiver (TR) module are used in the recording of the transactions.

The proposed actor architecture pattern is implemented by means of the following classes shown in Figure 3: the XML Transaction Generator (XTG), the Traffic Injector (TI), the Transactor, the Traffic Receiver (TR) and the XML Transaction Recorder (XTR).

2.2 Entity Architecture Pattern

Entities are created based on the information extracted from the use case and statechart UML diagrams. Figure 1 shows a block diagram of an entity using the proposed entity architecture pattern. An entity can be used to represent the system or a module within the system. The entities are reactive to external stimuli. For each actor, we can have an input port if the actor stimulates the entity and/or an output port if the entity produces a response to that actor. Note that the pattern clearly divides the entity into its control -the Use Cases (UCs) and the States-, its data storage -the Context-, the actions it can perform -Actuators-, and its communication parts -the Transactional Input Interface (TII), the Event Communication Channel (ECC) and the Transactional Output Interface (TOI). Figure 3 shows the classes that implement the proposed entity architecture pattern.

3 SAVY Behaviour

SAVY enables the development of complex systems providing the system-design team clear design patterns to achieve a TL SystemC executable model from a UML system description. During the execution of the TL model, the components of the SAVY library interact with each other to perform the system model simulation. The description of the behaviour of SAVY within a TL executable model has been divided into two parts: the actor pattern behaviour and the entity pattern behaviour.

3.1 Actor Pattern Behaviour

The actor pattern is responsible for feeding the entity with input transactions and for recording the entity’s output. The transactions to be injected by an actor are stored in an XML file. Each actor has its own XML stimuli file. The actor behaviour is depicted in Figure 4 by means of a UML sequence diagram. During the system simulation the XTG generates transactions from the XML stimuli file. The XTG is connected to the TI element. Both elements communicate each other using the XTG_if interface. TI requests new transactions to XTG using the method GetNextTransaction and sends transactions to the Transactor performing a blocking write in the output port. The TI element has a sc_thread which is responsible for injecting the transaction at the correct simulation time instant within a timed simulation scenario. If the system model is untimed, the TI injects the transactions one after the other.
Figure 1. UML-to-SAVY transformation

Figure 2. SAVY software architecture

Figure 3. SAVY class diagram
The TP's output is connected to a Transactor if an abstraction level conversion is necessary. If a Transactor is present, its output is connected to an entity; otherwise, the output of the TI is directly connected to an entity. The TI element communicates both with the Transactor and with the entity by means of the Transactor_if interface.

The entity may send transactions to the actor during the simulation. These transactions pass through the Transactor, if it is present, and reach the TR. The TR has a single thread that performs a blocking read on the transaction port. When a transaction arrives, the transaction is sent to XTR using the method Record/Transaction, which is defined in the XTR if interface. XTR writes the transactions received from TR to an XML file.

### 3.2 Entity Pattern Behaviour

The entity behaviour is described by the UML sequence diagram depicted in Figure 5. The transactions sent from the actor to the entity are received in the TI element. This element has a FIFO and two sc_threads. One of the sc_threads monitors the arrival of new transactions through the input port. When a new transaction arrives, it performs a blocking read; it inserts the transaction in the FIFO and it sends an event to the second sc_thread. The second sc_thread, reacts to this event broadcasting a new event through the ECC as shown in Figure 5. This second sc_thread remains blocked until an acknowledgement (ACK) is received from the UCs. The communication operations between the ECC and the UCs are defined in the ECC if interface. The acknowledgement mechanism is used to ensure that all the UCs have processed the event. If new transactions arrive while the second sc_thread is blocked, they are put in the FIFO. When the second sc_thread receives an ACK, it checks if there are more transactions in the FIFO and it processes them until the FIFO is emptied. This way, no transaction is lost.

The ECC broadcasts the events generated on the reception of new transactions to all the UCs of the entity. The behaviour associated to each UC is described as a state machine.

Each particular UC employs the following technique to identify the received event. For each event the UC is sensitive to, there is a sc_thread sensitive to that event. When an event arrives, the sc_thread sensitive to that event is activated. Then, the sc_thread of that particular UC calls the method ProcessEvent, sending as parameter an event identifier. This method calls the method of the current state responsible for processing the event.
When the method `ProcessEvent` of the UC receives the name of the next state, the UC checks if the next state is equal to the current state. If the next and current states are not equal the UC deletes the current state and creates a new object of the class of the next state.

The class `State` receives in the constructor a pointer to the `Actuator`. Thus, any `State` can access all the methods of `Actuator` to perform actions. Moreover, `States` can create new transactions and send them to the entity’s output by means of the `Actuator`. When a `State` is to produce an output transaction, the `State` uses the corresponding `Actuator` method to write to the `TOI`’s FIFO and to generate an event for the `TOI`’s `sc_thread`. When the `TOI`’s `sc_thread` receives the event, it performs a blocking write in the output port.

4 UML Description to SAVY Based Executable Model Translation Procedure

SAVY proposes design patterns to easily go from a UML based system description to a TLM-SystemC executable model in a structured way. Let us assume the design team has described the system under design by means of use case and statechart UML diagrams. The UML-to-SAVY translation procedure follows the next steps:

1. Identify UML actors and interactions in the UML use case diagrams:
   a) For each UML use case, define the possible information exchanged between actors and entities as transactions. A new derived class is defined for each transaction.
   b) Define the input/output interfaces of the entity: TII’s and TOIs. For each TII, implement the two `sc_threads`. For each TOI, implement its `sc_thread`.

Figure 5. Sequence diagram of Entity Pattern behaviour
2. Identify the data that the system must store:
   a) In SAVY, the data is stored in a container class (context) that must be defined.
3. Identify the control flow and actions in the UML statechart diagram:
   a) For each state of each UML state diagram, extract the possible state transitions and
      actions. This is achieved by defining a new class derived from State and
      implementing its method ProcessEventState.
   b) Declare the actions that the states can perform through the actuator_if.
   c) For each UML use case, define a new SAVY UC module and specify the initial
      state. For each UC, define the events the UC is sensitive to. This is achieved by
      creating a sc_thread sensitive to each event.
4. Build the model of the entity:
   a) Define the model of the entity which will be a sc_module. This sc_module will
      contain objects of the elements created in the former steps: T1s, T01s, Context
      and UCs. It will also contain an ECC object. Additionally, this sc_module will
      implement the actions of actuator_if.
5. Build the SystemC executable model:
   a) Define transactors if needed.
   b) Create the model of the actors using the appropriate objects of the SAVY classes
      XTG, TI, XTR and TR. Create an object of the entity. Bind the former objects.
6. Define the stimuli to be applied:
   a) For each actor that applies stimuli, write the XML file that describes the
      transaction injection sequence.

Once these steps are fulfilled, the model can be executed. After the execution, the
behaviour of the system can be studied analysing the output XML files.

This procedure guides the designer to separate the definition of the actors, the entity to
be designed and the information exchanged between actors and entity. Furthermore, the
model of the entity is clearly divided into its input/output interface, control flow, data to be
stored and actions to be performed. The proposed design patterns result in models easy to
understand and share between the different teams involved in a project.

5 Case Study
The system being developed within the CONFIDENCE Project [16] is presented as a case
study of how this library can help in the design of a complex system. The main objective of
the CONFIDENCE project is the development of a care system for the detection of
abnormal events (such as falls) or unexpected behaviours that may be related to a health
problem in elderly people. This care system works both outdoors and indoors. It raises an
alarm when an abnormal situation is detected. The system can also detect changes in the
user’s behaviour and issue a warning. The system can call to an alarm service or a relative
(hereafter alarm receiver) when an alarm or warning occurs.

5.1 Proposed Design Flow
A model driven methodology is being employed for the system development. The
executable models have been classified according to their abstraction level into Level 0,
Level 1 and Level 2 models as shown in Figure 6. At the beginning of the system design flow, the system design team describes the CONFIDENCE system using UML diagrams. Then, the system design team uses the SAVY library [13] to develop a TL executable model from the UML description of the system at each level of abstraction. The UML-to-SAVY based model translation procedure described in Section 4 is applied to create each of the system executable models. A key point of this approach is the reuse of model elements such as actors, transactions, entities, and UCs, between different abstraction levels.

The focus of the Level 0 model is to define the interfaces of the system with the external actors, user and alarm receiver (AR); define the transactions between the system and the environment; and define what the main functions of the system are. The inputs for the system design team are the functional requirements of the CONFIDENCE system.

In a second step of the design process, the system design team refines the Level 0 model by dividing the system into functional devices. New requirements appear: the functions defined for each device; and the interfaces (transactions) between the devices. Both the actors and the transactions defined in the Level 0 model are reused. This design activity yields the Level 1 model. The Level 1 model of the CONFIDENCE system consists of the following functional devices: a central device, which plays the role of a Base Station (BS); a small Portable Device (PD), which looks similar to a mobile phone; and several tags.

In a third step, each device is divided into its main functional modules. The resulting requirements are translated to a new executable model, the Level 2 model. In the Level 2 model of Figure 6, both the PD and BS are divided into four submodules: the localization
<table>
<thead>
<tr>
<th>UML-to-SAVY translation steps</th>
<th>No Items</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L 0</td>
<td>L 1</td>
</tr>
<tr>
<td>1.a) Definition of transactions</td>
<td>31</td>
<td>75</td>
</tr>
<tr>
<td>1.b) Definition of entities I/O</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>2.a) Definition of the context</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3.a) Definition of states</td>
<td>21</td>
<td>51</td>
</tr>
<tr>
<td>3.b) Declaration of actions</td>
<td>149</td>
<td>511</td>
</tr>
<tr>
<td>3.c) Definition of UCs</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>4.a) Creation of the system model</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>5.a) Creation of Transactors</td>
<td>4</td>
<td>13</td>
</tr>
<tr>
<td>5.b) Creation of the models of the actors, entity(ies) for system and binding</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TOTAL</td>
<td>212</td>
<td>677</td>
</tr>
</tbody>
</table>

Table 1. The LoC necessaries to create TL system models from the UML system descriptions

<table>
<thead>
<tr>
<th>CPU simulation time</th>
<th>Level 0</th>
<th>Level 1</th>
<th>Level 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.56 s</td>
<td>12.73 s</td>
<td>31.58 s</td>
</tr>
</tbody>
</table>

Table 2. CPU simulation times of the three CONFIDENCE system abstraction levels

In order to evaluate the proposed system design methodology using SAVY, the cost of creating a TL system model from an UML system description and the simulation performance of the executable model have been measured for each level of abstraction.

The cost of creating a TL system model from an UML description is estimated by means of the lines of code (LoC) written by the design team. The cost can be split in several parts according to the steps described for UML-to-SAVY translation procedure in Section 4.

Table 1 shows the LoC written to achieve the system Level 0 (L0), Level 1 (L1) and Level 2 (L2) models. As it can be observed, the LoC increase as the system model gets more detailed. However, many LoC have been reused between the different abstraction levels.

Table 2 analyzes the simulation performance of the three CONFIDENCE system abstraction levels in terms of the CPU simulation time. These SAVY based executable models have been compiled using GNU/g++ v3.4.6 in a PC with a CPU Intel Xeon E5405 @ 2.00 GHz, running Linux Red Hat Enterprise.

The three abstraction level simulations shown in Table 2 use the same input stimuli. The input stimuli represent more than 2 hours of real system activity. As it can be observed, the simulation time increases as the system model gets more detailed. There is an eightfold increase from Level 0 to Level 1 and less than a threefold increase from Level 1 to Level 2. Nevertheless, the simulation time of the Level 2 executable model is very small, which enables quick system architecture exploration.
6 Conclusions

A UML system description can be easily transformed into a TLM. SystemC executable model applying the system design approach proposed by SAVY. A clear design flow is defined in order to offer a straightforward procedure to go from UML to TLM. SystemC. SAVY enables the models to reuse and facilitates the model integration, thus, reducing the development time and simplifying the work of the engineers. Moreover, SAVY provides fast simulation speeds in order to enable early system architecture exploration. The case study presented shows that this approach can be successfully used in the design flow of a complex electronic system.

7 Acknowledgement

The research leading to these results has received funding from the European Community’s Framework Programme FP/2007-2013 under grant agreement n. 214986. Consortium: CEIT, Fraunhofer IIS, Jožef Stefan Institute, Ikerlan, COOSS Marche, University of Jyväskylä, Umeå University, eDevice, CUP2000, ZENON. A. Cortés holds the Torres Quevedo grant No. PTQ05-02-02455S awarded by the Spanish Ministry of Education and Science, by the European Regional Development Fund and by the European Social Fund.

References

2. Open SystemC Initiative (OSCI), [Online]; Available: www.systemc.org
16. CONFIDENCE Project, [Online]; Available: www.confidence-eu.org
Ubiquitous Care System to Support Independent Living
Ubiquitous Care System to Support Independent Living

Leticia Zamora-Cadenas\textsuperscript{1,a}, Koldo Tomasena\textsuperscript{1b}, Markos Losada\textsuperscript{1c}, Juan F. Sevillano\textsuperscript{1d}, Matjaz Gams\textsuperscript{2a} and Igone Vélez\textsuperscript{1d}
\textsuperscript{1}CEIT and Tecnun, Spain
\textsuperscript{2}Jozef Stefan Institut, Slovenia
\textsuperscript{a}zamora@ceit.es, \textsuperscript{b}tomasena@ceit.es, \textsuperscript{c}losada@ceit.es, \textsuperscript{d}sevillano@ceit.es, \textsuperscript{e}matjaz.gams@jfs.si, \textsuperscript{f}vellez@ceit.es

Key words: ambient assisted living, elderly people, care system, localisation system.

Abstract. This work proposes a care system for elderly people with fear of falls that want to keep living at their own home. The main innovation of the proposed care system is that it not only detects falls as most of the care systems in the market, but it also identifies short and long-term unexpected behaviours that could indicate health problems. Information about the user’s location, together with some environment information, are analysed to decide whether to trigger an alarm. In case of an abnormal situation such as a fall or an accident, the proposed system will permit a rapid actuation of the health services, which will decrease the negative consequences of the accident, e.g.: worsening of injuries, psychological impact of being alone and injured, etc. With the proposed approach, the elderly will gain confidence and independence. Measurements are presented that demonstrate the high accuracy of the proposed care system.

Introduction

Literature suggests that fear of falling or of being left unattended in case of sudden trouble can decrease quality of life and increase the speed of decline in the ability to perform daily activities. Furthermore, this situation might lead elderly people to a self-imposed isolation, refusal of mobility, and admission to institution care [1-11].

Nowadays, there are systems in the market that aim to prolong the personal autonomy of elderly people. However, most of the care systems in the market are limited to detect falls. These systems either use accelerometers or other types of sensors or are based on video systems. Solutions composed of an accelerometer, such as the one described in [12], raise an alarm if the sensor detects a strong acceleration, reducing the risk of elderly people being left without assistance. Similar systems are proposed in [13-17]. However, these systems usually lack accuracy and have considerably high false alarm rates. These false alarms cause distress to some users, which might be concerned about raising an alarm when it is not necessary, and might even stop doing some activities not to raise the alarm unnecessarily. The technical performance of a care system must provide the user with the confidence that it will work in case of real emergency. Nevertheless, this reliability must not be achieved at the detriment of the false alarm rate.

Another approach based on accelerometers is reported in [18]. This system goes beyond fall detection and aims at activity recognition by analysing data from accelerometers. It was found that activities can be recognised with fairly high accuracy. However, this system does not give information about location and it is not very reliable at detecting falls. In line with this approach, the system described in [19] provides user localisation, based on GPS (Global Positioning System), automatic fall detection and activity monitoring. The solution seems very complete but there is still the problem of high false alarm rates, since it relies on accelerometers.

There are also some care systems based on video surveillance, such as the ones described in [20]-[24]. These systems provide lower false alarm rates. However, video based systems, have one major drawback: they are intrusive and do not respect the user’s privacy. Due to this important disadvantage, most users are reluctant to install such systems at their homes. In order to deal with
the intrusion problem, [22] describes a system which filters the images, as soon as they are captured, and translates them into abstract information, such as vectors. This way, visual images are neither stored nor transmitted and it is impossible to reconstruct the received information into an image showing the visual appearance of the subject. However, video systems present other important disadvantages: it is difficult and extremely expensive to cover the whole home with video cameras.

In summary, the majority of the currently available care systems rely either on accelerometers or on video surveillance. There are also some solutions such as [24] that combine both techniques in order to enhance the system efficiency. However, all these care systems have important disadvantages such as high false alarm rates, lack of portability or privacy intrusion.

This work proposes a non-intrusive care system able to detect with high accuracy abnormal situations such as falls or loss of consciousness. Additionally, this system detects long-term behaviour changes that indicate health problems. The proposed system requires a highly accurate indoor positioning system based on UWB (Ultra-Wide Band) technology. This paper presents a novel calibration method of this indoor positioning system that significantly improves its accuracy.

**Care System Description**

The proposed care system reconstructs the user's posture and detects abnormal situations, such as falls or loss of consciousness, and raises an alarm if necessary. It is also able to detect changes in the user's behaviour and issue a warning. For instance, if the system notices changes in the user's gait that may involve a lack of stability, the proposed care system warns the user about an increased risk of falling, and, thus, can prevent an accident. Detection of anomalous behaviour utilise prior expert knowledge as well as learnt movement patterns of particular users.

Figure 1 shows the different components of the proposed system. This system consists of a central device, which plays the role of a base station, a small portable device, such as a smartphone, and some tags and anchors.
• **Tags and anchors.** The user has to wear four small-size tags: one in the chest, one in the waist and one in each ankle. They can be worn either in the form of bracelets or in the form of necklaces or pendants. If the user prefers it, the tags may be sewed into the clothes (socks, underwear, etc.). The tags can be worn comfortably and the user can easily take them on and put them off by himself/herself without any external help. There are also several anchors, in fixed positions in the room, which receive the tags' signals and are able to determine their distance to the tags.

• **Base station.** The base station is able to determine the position of each tag in the 3-dimensional space based on the information received from the anchors. The system reconstructs the posture of the body using the information about the tags position and decides whether the user has suffered a fall or is acting abnormally. When a fall or an atypical situation is detected, the system raises an alarm.

• **Portable device.** The portable device is a smartphone with a proprietary application installed. The portable device alerts the user when the base station detects an abnormal situation.

**Alarm Protocol.** Typically, the proposed care system follows this alarm protocol: First, it makes a phone call to the user that starts up the application in the portable device. This application requests the user to indicate whether he/she feels well or not. If the user feels well, he/she has to press the 'Dismiss' button to stop the alarm. This way, the user keeps control of the system, which is an important feature and considerably reduces the false alarm rate.

If the user presses the 'Alarm' button or does not press the 'Dismiss' button in a certain time after the alarm was raised, the system makes a phone call to a series of relatives or friends. If nobody answers the phone, the system calls the emergency services. In all cases, the system explains to the call recipient the reasons of the alarm or warning. Moreover, if the user is not feeling well, he/she can press the 'Alarm' button to raise the alarm.

The user can customise the alarm protocol and select the time that the system waits for him to answer, before triggering the alarm. Additionally, the user can allow the system to transmit all the available information about the situation: localisation, posture and how long the user has been in that posture. This information will help to react to the emergency call.

**Detailed architecture.** Figure 2 shows the detailed architecture of the proposed care system.

This system is divided into the following four subsystems:

• **Localisation subsystem**, which allows the system to identify and locate the tags in space with enough precision. This subsystem relies on radio technology and must achieve a positioning accuracy in the centimetre range.

• **Reconstruction subsystem**, which is capable of reconstructing the user's body and environment. This second subsystem receives the estimates of the positions of the tags and generates a model of the user and the environment.

• **Interpretation subsystem**, which is in charge of interpreting the body posture within the environment. This subsystem is provided with intelligence, so that it can learn from the user's habits and help to detect early symptoms of illness. This model must have an accuracy of 95% for differentiating abnormal postures from safe postures and must be fast enough to be used in real-time applications.

• **System interface subsystem**, which comprises all the communication between the user and the system: user interface, alarm handling and set-up of the system.
Description of the Localisation Subsystem

In order to achieve a 95% of accuracy in the interpretation of the situation with the proposed care system, the positioning error of the localisation subsystem shall be below 30 cm inside the area of surveillance, i.e.: at a range of 6-7 m or in a space of about 30 m².

The indoor localisation subsystem of the proposed care system is based on the real-time location system of [25]. The anchors are grouped into cells and in each cell, a master anchor is designed to coordinate the communications and activities. The other anchors, called slaved anchors, are networked to the master with an Ethernet timing cable in order to have a tight synchronisation. All the information collected by the anchors is sent to the BS using a power-over-Ethernet cable and there, it is processed to determine the tags position.

When the tags are triggered by the system, they send a UWB pulse in the frequency range of 6-8 GHz that is received by the anchors. Then, the position of each tag is estimated in the BS using a combination of two different techniques: Time Difference of Arrival (TDoA) and Angle of Arrival (AoA). The selected positioning algorithms require an accurate synchronization of the clock and a precise orientation of the antennas of the anchors, to have good positioning accuracy. Thus, this localisation system needs to be calibrated prior to be used when it is installed for the first time.

Improving the accuracy of the first indoor localisation subsystem

Two manners of calibrating the localisation system are proposed in [26]: automatic calibration, called also full calibration or; manual calibration, called dual calibration. In most of the environments, the full calibration does not work, or does not calibrate the system properly, so in general, it is necessary to manually calibrate the anchors and estimate their exact position and orientation in the room using the manual calibration. In the following, this manual calibration is described. Moreover, two new calibration methods are proposed that aim to improve the accuracy of the manual calibration.

Manual calibration method. In this calibration method, the instructions given by [26] are followed. The anchors are positioned near the corners of the room pointing to the centre point of the room. Then, one tag is positioned in the middle of the room or near, in a position with clear Line of Sight (LOS) from the anchors. The tag position (x, y, z) is measured with a laser-meter. For each slave anchor, the dual calibration is performed and the system calculates the cable offset and the pitch and yaw angles for each anchor. When the dual calibration is done with all the slaves, the system is ready to be used.
First calibration method. In order to achieve better accuracy in the positioning of the tags, a new step is proposed to be added to the manual calibration method. After performing the manual calibration method, the power received by each anchor is analysed in order to adjust their physical orientation. The tag is moved around the room and, if the power each anchor receives is not enough, its physical orientation is adjusted. Then, the system is recalibrated. This adjustment of the orientation of the anchors and recalibration is repeated until all the anchors are oriented in their best position receiving the maximum power possible in all the positions of the room.

Second calibration method. The values obtained for the pitch and yaw angles are directly related with the position chosen in the room and the slave anchor involved in the dual calibration. In fact, each time the dual calibration is performed, the cable offset and the pitch and the yaw angles of the master anchor and the slave anchor are calculated. When the next slave anchor is selected to be calibrated, the system recalculates the angles of the master anchor, being these angles similar to the previous calculated with the other slave anchor, but not the same. This behaviour is also reported in [27] where it is proposed to calibrate the system using twelve different points in the room, and average the values of the angles for the master and the slave anchors. Taking into account this behaviour, the second calibration method is designed as follows:

- **Step 1.** One tag is positioned in the middle of the room or near in a position with clear LOS from the anchors. The tag position \((x, y, z)\) is measured with a laser-meter. For each slave anchor, the dual calibration is performed and the system calculates the cable offset and the pitch and yaw for each anchor. This step corresponds to the manual calibration method.
- **Step 2.** One tag is moved around the room and if the power each anchor receives is not enough, the physical orientation of the anchor is adjusted and the system is recalibrated. This adjustment of orientation of the anchors and recalibration is repeated until all the anchors are oriented in their best position, where they receive the maximum possible power in all the positions of the room. This step corresponds to the first calibration method.
- **Step 3.** The system is calibrated with one tag placed in twelve different points in the central part of the room. The values of the angles for the master and the slaves in each of these positions are stored and averaged at the end of the twelve calibrations. Then, the tag is placed in the centre point of the room and the system is recalibrated as in Step 1. Finally, the averaged values of the angles are manually introduced in the properties of the anchors.

Measurement Results of the Localisation System

Figure 3 shows the floor-plan of the room used for the measurements carried out to determine the accuracy of the localisation system. In these measurements, four anchors are set in fixed positions in the room and four tags are placed in different positions around this room.

![Figure 3: Floorplan of the measurement room.](image-url)
The height of each tag is the same for all \((x, y)\) positions in the room; i.e.: tags 1, 2, 3 and 4 are placed at height 0.0 m, 0.5 m, 0.69 m, 1.5 m respectively. The fixed positions of the anchors, near the corners of the room, are depicted in Figure 3 as rectangles and the different \((x, y)\) positions where the tags where placed are shown as ellipses. Between 2400 and 2500 measurements are captured for each tag in each location for statistical analysis.

Figure 4 shows the average distance error in positioning for each of the tags for the three calibration methods studied in this work. The 95% confidence interval of the average distance error is also shown in the figure. The average estimated bias with the manual calibration, first calibration and second calibration is approximately 0.29 m, 0.25 m, and 0.18 m respectively. It can be observed that the second calibration method is the one that produces less bias and outliers for all tags.

![Graphs showing average distance error for different calibration methods](image)

Figure 4: Average distance error of the positioning system for each tag and each method.

Figure 5 presents the Cumulative Distribution Function (CDF) for the manual, first, and second calibration methods. The CDF provides the probability of the error in the estimation of the tag position to be below a certain distance. It can be observed that in 90% of the measurements, the error in the estimation of the position of a tag has been less than 34 cm for the manual calibration, less than 37 cm for first calibration and less than 29 cm for second calibration. By employing the second calibration proposed in this work, the number of outliers generated by positioning system is reduced and, thus, the accuracy of the positioning system is significantly improved.
In the first and second column of Table 1, the average and standard deviation of the distance error of the positioning measurements for the three calibrations is provided. It can be seen that the measurement results for the second calibration method are much better than for the other two methods. Moreover, as shown in the three last columns of Table 1, the second calibration has the highest probability of having an error smaller than 10 cm, 20 cm, and 30 cm. When the second calibration is employed, the positioning error will be below 20 cm for around 70% of the measurements and below 30 cm for nearly 92% of the cases. Thus, for the system installation, the second calibration method proposed in this work will be employed.

<table>
<thead>
<tr>
<th>Calibration method</th>
<th>Mean distance error</th>
<th>Standard deviation of distance error</th>
<th>Probability of having an error &lt; 10 cm</th>
<th>Probability of having an error &lt; 20 cm</th>
<th>Probability of having an error &lt; 30 cm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual Calibration</td>
<td>0.29 m</td>
<td>0.70 m</td>
<td>9.55%</td>
<td>53.31%</td>
<td>88.34%</td>
</tr>
<tr>
<td>First Calibration</td>
<td>0.25 m</td>
<td>0.34 m</td>
<td>6.76%</td>
<td>43.41%</td>
<td>78.61%</td>
</tr>
<tr>
<td>Second Calibration</td>
<td>0.18 m</td>
<td>0.28 m</td>
<td>28.98%</td>
<td>69.88%</td>
<td>91.74%</td>
</tr>
</tbody>
</table>

Measurement Results of the Proposed Care System

The accuracy of the whole proposed care system has also been measured. In the system tests, 10 volunteers performed five times the activities presented in Table 2. As indicated in this table, some of these activities are falls; whereas the others are activities that the fall detector should not classify as a fall. The activities performed by each volunteer were recorded in single recordings interspersed with short periods of walking. Four tags placed on ankles, waist and chest were used. The accuracy was measured using the leave-one-person-out technique. This means that the classifier was trained on nine people and tested on the tenth. This was repeated ten times using a different person for testing each time.
Table 2: Test scenario.

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>Fall</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sitting down normally on the chair</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>Tripping, landing flat on the ground</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>Lying down normally on the bed</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>Falling slowly (trying to hold onto furniture), landing flat on the ground</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>Sitting down quickly on the chair</td>
<td>No</td>
</tr>
<tr>
<td>6</td>
<td>Falling when trying to stand up (trying to hold onto furniture), landing sitting of the ground</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>Lying down quickly on the bed</td>
<td>No</td>
</tr>
<tr>
<td>8</td>
<td>Sitting down quickly on the chair</td>
<td>No</td>
</tr>
<tr>
<td>9</td>
<td>Falling slowly when trying to stand up, landing sitting of the ground</td>
<td>Yes</td>
</tr>
<tr>
<td>10</td>
<td>Searching for something on the ground on all fours any lying</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 3 presents the accuracy of the system in reconstructing the body posture of the user. Sitting on the ground, on all fours and going down are the postures with worse accuracy. Nevertheless, the system showed more than a 90% of accuracy detecting when a person is walking, sitting and lying, which are the postures that are really important in fall detection. Overall, the correct posture was detected in the 89% of the situations.

Table 4 presents the accuracy of the system detecting different kinds of falls. On average, the fall and non-fall events were correctly detected in 96% of the situations, which provides the care system with a really good accuracy in fall detection.

Table 3: Activity reconstruction accuracy.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Walking</td>
<td>94%</td>
</tr>
<tr>
<td>Sitting</td>
<td>89%</td>
</tr>
<tr>
<td>Lying</td>
<td>93%</td>
</tr>
<tr>
<td>Sit on the ground</td>
<td>44%</td>
</tr>
<tr>
<td>On all fours</td>
<td>22%</td>
</tr>
<tr>
<td>Going down</td>
<td>42%</td>
</tr>
<tr>
<td>Standing up</td>
<td>74%</td>
</tr>
<tr>
<td>Overall</td>
<td>89%</td>
</tr>
</tbody>
</table>

Table 4: Fall detection accuracy.

<table>
<thead>
<tr>
<th>Event</th>
<th>Accuracy</th>
<th>Fall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tripping</td>
<td>100%</td>
<td>Yes</td>
</tr>
<tr>
<td>Falling slowly</td>
<td>98%</td>
<td>Yes</td>
</tr>
<tr>
<td>Falling sitting</td>
<td>92%</td>
<td>Yes</td>
</tr>
<tr>
<td>Falling sitting, slowly</td>
<td>90%</td>
<td>Yes</td>
</tr>
<tr>
<td>Lying quickly</td>
<td>100%</td>
<td>No</td>
</tr>
<tr>
<td>Sitting quickly</td>
<td>100%</td>
<td>No</td>
</tr>
<tr>
<td>Searching on the ground</td>
<td>80%</td>
<td>No</td>
</tr>
<tr>
<td>Average</td>
<td>96%</td>
<td></td>
</tr>
</tbody>
</table>

Conclusions

A care system has been presented that aims to prolong the personal autonomy of elderly people at home, giving them confidence to keep living at their preferred environment. Within this care system, the reconstruction and interpretation subsystem receives coordinates from the localisation subsystem, reconstructs the user’s activity from them and interprets it as normal or abnormal. Its purpose is to detect falls and other health problems manifesting in movement, and inform the user raising an alarm. If unusual behaviour is detected, it raises a warning. Measurements have been presented that show that the care system is able to accurately detect falls and other abnormal events.

The care system relays on UWB technology to provide good accuracy in the estimation of the position of the tags that the user is wearing. The accuracy of the positioning system is really
dependant on the place where the system is installed. Therefore, a good calibration method must be employed in order to achieve the positioning accuracy required by this care system. The second calibration method proposed in this work reduces the bias and outliers of the positioning estimator, and thus, makes the positioning system more accurate.

Acknowledgments

The research leading to these results has received funding from the European Community’s Framework Programme FP7/2007-2013 under grant agreement n. 214986. Consortium: CEIT, Fraunhofer IIS, Jožef Stefan Institute, Ikerlan, COOSS Marche, University of Jyväskylä, Umeå Municipality, eDevice, CUP2000, ZENON.

References


