Model Based Program Synthesis and Runtime Error Detection for Dependable Embedded Systems

PhD Thesis

Gergely Pintér

Advisor: Dr. István Majzik
Budapest University of Technology and Economics
Department of Measurement and Information Systems
Budapest, 2007
Nyilatkozat (Declaration of Authorship)

Alulírott Pintér Gergely kijelentem, hogy ezt a doktori értekezést magam készítettem és abban csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból vettem át, egyértelműen, a forrás megadásával jelöltem.

A dolgozat bírálatai és a védésről készült jegyzőkönyv elérhető a Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Karának Dékáni Hivatalában.


Pintér Gergely
Abstract

This thesis presents achievements of over three years of research focusing on various aspects of software dependability. The first part of the thesis presents a novel formal operational semantics for statecharts of the UML 2.0 modeling language. The solid foundations of formalization efforts are established by defining the metamodel of precise statecharts with a set of well-formedness rules, formal definition of refinement concepts, compound transition structures and compound activity structures. The approach puts emphasis on modeling parallel execution by defining compound activity structures based on PERT graphs enabling this way the straightforward exploitation of parallel computing resources of modern platforms. The operational semantics is defined by a Kripke transition system and finally mapped to imperative algorithms, enabling this way the straightforward implementation in popular programming languages.

The second part presents a method for automatic source-code level implementation of behavior specified by UML 2.0 statecharts. According to my knowledge this solution is the most complete published approach for automatic source-code level implementation of UML 2.0 statecharts even supporting resource constrained embedded systems. The remarkable features of the method are as follows: (i) the solution corresponds to the steps of OMG’s Model Driven Architecture i.e., conceptually different tasks (identification of target platform’s features, abstract mapping to the available resources and the implementation) are clearly separated; (ii) a significant effort was laid on proving the correctness of subsequent transformation steps; (iii) being based on the semantics presented in the first part of the thesis, the solution supports the most complete toolkit of the most recent version of the UML standard, (iv) the mapping to resource constrained platforms does not restrict the set of usable modeling concepts and (v) for sufficiently complex models the performance and size of the resulting applications is even more beneficial as compared to the ones built according to less sophisticated approaches supporting much less modeling features.

The third part presents two techniques for runtime detection of behavioral errors in UML statechart implementations: (i) model refinement faults are addressed by introducing a propositional linear temporal logic language (PSC-PLTL) for defining dependability criteria and evaluating these criteria during the execution of the application; (ii) faults in the implementation of statechart models are addressed by a statechart-level watchdog solution (PSC-WD) synthesized from the formal semantics of statecharts. The viability of both approaches is demonstrated by experimental evaluation of error detection capabilities.

Finally the last part presents a novel approach for automatic identification of key factors determining the behavior of systems in the presence of faults by intelligent data processing methods. The corresponding chapter discusses how to exploit the benefits of automated tools enabling this way the application of most advanced data processing intelligence for analyzing experiment results by the dependability community.
Kivonat (Abstract in Hungarian)

Ez a disszertáció több mint háromévenyi, a szolgáltatásbiztonság különböző aspektusaira koncentrált kutatás eredményeit foglalja össze. A munka első része egy új formális működési szemantikát mutat az UML 2.0 nyelv állapotérkökeihez. A tárgyalás stabil alapjait a preciz állapotérköpek metamodeljének definíciója, a kapcsolódó jólformáltsági szabályok megadása, az állapotfinomítás fogalmainak formális definíciója, illetve az összetett átmenet és aktivitás struktúrák definíciója teremti meg. A megoldás jelentős hangsúlyt helyez a párhuzamosan végzhető tevékenységek modellelésére azáltal, hogy az összetett aktivitás struktúrákat a PERT gráfok eszközkezelésére alapozva vezeti be, lehetővé téve ezáltal a modern platformok párhuzamos számítási erőforrásainak hatékony kihasználását. A szemantikát egy Kripke tranziciós rendszer definíálja, ami végül a népszerű programozási nyelveken közvetlenül megvalósítható imperatív algoritmusokra képződik.

A disszertáció második része bemutat egy módszert, amely UML 2.0 állapotérkökkekkel specifikált viselkedés automatikus forrásnyelvi megvalósítást teszi lehetővé. Ismereteim szerint jelenleg ez a legteljesebb publikus megoldás állapotérköpek megvalósítására, amely erőforrásokban szegény beágyazott rendszereket is támogat. A módszer említésre méltó tulajdonságai: (i) a megoldás belső szervezése illeszkedik az OMG Model Driven Architecture javaslatához vagyis a koncepcionalisan különböző feladatok (celplatform tulajdonságainak meghatározása, absztrakt leképzés a rendelkezésre álló erőforrásokhoz és a megvalósítás) tisztán elkülönülnek; (ii) jelentős hangsúlyt kapott az egymást követő transzformációs lépések helyességének bizonyítása; (iii) mivel megoldásunk a disszertáció első részében bemutatott szemantikán alapul, a módszer támogatja a legfrissebb UML szabvány gazdag eszközkezelését; (iv) az erőforrásokban szegény platformokra való leképzés nem szorítja meg a használható modelllezési fogalmak körét és (v) kölén összetett modellök esetén a létrejövő alkalmazások teljesítménye és mérete kedvezőbb a korábban publikált megoldásoknál, amelyek lényegesen kevesebb modelllezési elemet támogatnak.

A harmadik rész két futási idejű hibadetektároló megoldást mutat UML állapotérköpek megvalósításokhoz: (i) a modellfinomítási hibák kezelésére bevezet egy lineáris temporális kijelentéstélogikai nyelvet (PSC-PLTL), amely lehetővé teszi szolgáltatásbiztonsággal kapcsolatos követelések megfogalmazását, illetve ezen kritériumok ellenőrzését az alkalmazás futása közben; (ii) állapotérköpekkel leírt rendszerek megvalósítási hibáinak kezelésére egy állapotérkép szintű watchdogot (PSC-WD) mutat be, ami a formális szemantika specifikációiból szintetizálható. Mindkét megoldás élettépességét a hibadetektároló képesség kísérleti kiértékelése demonstrálja.

Végül az utolsó rész mutat egy új, intelligens adatfeldolgozó technikákra alapozott automatikus megoldást amely lehetővé teszi azon kulcsfontosságú jellemzők azonosítását, amelyek elsősorban meghatározóak egy rendszer viselkedését hibák jelenlétében. A kapcsolódó fejezet megmutatja, hogyan használhatók ezen a téren az ismert automatikus eszközök, lehetővé téve ezáltal, hogy a legfejlettebb adatfeldolgozási intelligenciamintak bekerüljön a szolgáltatásbiztonsági szakértők eszköztárába.
Contents

Introduction xix
A Formal Operational Semantics for UML 2.0 Statecharts xx
Automatic Implementation of UML 2.0 Statecharts xxii
Error Detection in UML 2.0 Statechart Implementations xxiii
Supporting Experiment Evaluation by Data Mining xxvi

1 A Formal Operational Semantics for UML 2.0 Statecharts 1
1.1 Introduction 1
1.2 Notation 5
1.3 Metamodel and Well-Formedness 6
  1.3.1 Core Constructs and Shorthand Notations 7
  1.3.2 Metamodel of Precise Statecharts 7
  1.3.3 Well-Formedness of Precise Statecharts 8
1.4 Refinement Formalism 9
  1.4.1 Refinement Concepts 9
  1.4.2 Well-Formedness of Refinement 13
  1.4.3 Relations of Refinement Concepts 13
  1.4.4 State Hierarchies 14
1.5 Compound Transition Structures 14
  1.5.1 Possible Roles of Transitions 15
  1.5.2 Transition Conglomerate Classes 17
  1.5.3 Triggers and Guards 19
  1.5.4 Sources, Targets and Containers 19
  1.5.5 Priorities and Conflicts 23
1.6 Compound Activity Structures 24
  1.6.1 PERT Graphs 24
  1.6.2 Entering and Leaving State Hierarchies 27
  1.6.3 Effects of Transition Conglomerates 28
1.7 Introduction of New Concepts Into the Metamodel 28
1.8 Operational Semantics 29
  1.8.1 Modeling Extended Variables 29
  1.8.2 Definition of the Kripke Transition System 29
  1.8.3 High-Level Algorithms 35
  1.8.4 Formal Algorithm Definitions and Algorithmic Complexity 37
  1.8.5 Correspondence to the Informal Semantics of the Standard 37
1.9 Conclusions and Future Work 39
## 2 Automatic Implementation of UML 2.0 Statecharts

2.1 Introduction .......................................................... 41
2.2 Overview on Resource Constrained Embedded Systems .......... 47
  2.2.1 Application Area, Hardware and Implied Architecture ....... 48
  2.2.2 Design Decisions ................................................. 49
2.3 Platform-Specific Behavior Modeling .................................. 49
  2.3.1 Modeling Concepts Affected by Design Decisions ............ 50
  2.3.2 A Smart Data Structure for Storing Sets and Hierarchies .. 52
  2.3.3 Representation of PERT Graphs by Activity Sequences ....... 56
  2.3.4 Platform-Specific Metamodel ................................... 57
  2.3.5 Platform-Specific Mapping of Algorithms ..................... 58
2.4 Implementation ..................................................... 59
  2.4.1 Implementation of the Static Structure ....................... 60
  2.4.2 Implementation of the Operation Algorithms ................. 61
2.5 Code Generation .................................................... 64
2.6 Experimental Evaluation ............................................ 65
  2.6.1 Memory Layout and Experiment Setup ........................ 65
  2.6.2 Experiments ................................................... 67
  2.6.3 Summary ....................................................... 71
2.7 Conclusions and Future Work ........................................ 72

## 3 Error Detection in UML 2.0 Statechart Implementations

3.1 Introduction .......................................................... 75
3.2 Background on Temporal Logic Languages ......................... 80
  3.2.1 Basic Concepts ............................................... 80
  3.2.2 Classification of Temporal Logic Languages ................. 81
  3.2.3 A Propositional Linear Temporal Logic Language .......... 81
3.3 A Temporal Logic Language for Statecharts ......................... 82
3.4 Previous Approaches for PLTL Evaluation ......................... 84
  3.4.1 Naive Methods ............................................... 84
  3.4.2 Method Proposed in the Literature .......................... 85
3.5 A Novel High-Performance PLTL Evaluation Method ................ 87
  3.5.1 Overview of the Idea ....................................... 87
  3.5.2 Iterative Decomposition of Formulae ........................ 89
  3.5.3 Evaluation Node Types and Instances ....................... 91
  3.5.4 Basic Operations on Evaluation Nodes ...................... 93
  3.5.5 Algorithm of Evaluation ................................... 95
  3.5.6 Correctness and Algorithmic Complexity .................... 96
  3.5.7 Implementation Considerations .............................. 96
  3.5.8 Memory Consumption ....................................... 96
  3.5.9 Experimental Evaluation of Performance .................... 97
  3.5.10 Summary ................................................... 100
3.6 Background on Watchdog Processors ................................. 100
3.7 A Watchdog Architecture for UML 2.0 Statecharts ................ 103
3.8 Experimental Evaluation ............................................ 105
  3.8.1 Background on Fault Injection .............................. 106
  3.8.2 Experiment Setup .......................................... 108
  3.8.3 Experiment Results ......................................... 111
3.9 Conclusions and Future Work ....................................... 113
4 Supporting Experiment Evaluation by Data Mining

4.1 Introduction

4.2 Background on Data Mining

4.3 Analysis of Data from DBench-OLTP Experiments

4.3.1 Experiment Setup

4.3.2 Applying Data Mining to Identify Dominant Factors

4.3.3 Discussion of Results and Analysis

4.4 Analysis of Data from Evaluation of Error Detection Techniques

4.4.1 Failures Experienced

4.4.2 Effectiveness of the Watchdog

4.4.3 Effectiveness of Low-Level Mechanisms

4.5 Conclusions and Future Work

5 Conclusions

Appendices

A Details on Formal Semantics

A.1 Citations from the UML 2.0 Standard

A.2 Well-Formedness of Precise Statecharts

A.2.1 Final States

A.2.2 Initial Pseudostates

A.2.3 Join Pseudostates

A.2.4 Fork Pseudostates

A.2.5 Termination States

A.2.6 Regions

A.2.7 States

A.2.8 State Machines

A.2.9 Transitions

A.3 Detailed Definitions

A.3.1 Entering and Leaving State Hierarchies

A.3.2 Effects of Transition Conglomerates

A.3.3 Derived Transition Conglomerate Classes in the Metamodel

A.4 AsmL Specification of the Platform-Independent Semantics

A.4.1 Base Data Types

A.4.2 Algorithms

A.5 Detailed Proofs

A.5.1 Refinement Formalism

A.5.2 Compound Transition Structures

A.5.3 Correctness of Operation

A.6 Implementation of Shorthand Constructs

A.7 Detailed Examples

A.8 Example for Simulation

B Details on Automatic Implementation

B.1 Detailed Examples for Implementation Patterns

B.2 AsmL Specification of the Platform-Specific Semantics

B.2.1 Base Data Types

B.2.2 Operation Algorithms

B.3 An Example for Code Generation
C Details on Error Detection

C.1 Proof of Correctness .............................................................. 207
C.2 Algorithmic Complexity ....................................................... 208
C.3 An Example for Code Generation ............................................ 209
  C.3.1 Evaluation Node Classes and Top Interfaces ......................... 209
  C.3.2 Bottom Interfaces and Evaluation Node Base Class .................. 210
  C.3.3 Head and Next Nodes ..................................................... 211
  C.3.4 Evaluation of Expressions in Ternary Logic ......................... 211
  C.3.5 External Propagation .................................................. 212
  C.3.6 Top-Level Algorithms .................................................. 213
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Overview of the Thesis</td>
<td>xx</td>
</tr>
<tr>
<td>1.1</td>
<td>Identified Issues, Research Goals and Achievements</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Overview of the Formal Semantics</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>UML State Machine Metamodel</td>
<td>5</td>
</tr>
<tr>
<td>1.4</td>
<td>Precise Statechart Metamodel</td>
<td>7</td>
</tr>
<tr>
<td>1.5</td>
<td>Direct Refinement, Containment, Substate and Subregion Relations</td>
<td>9</td>
</tr>
<tr>
<td>1.6</td>
<td>State Refinement Example</td>
<td>11</td>
</tr>
<tr>
<td>1.7</td>
<td>State Hierarchies</td>
<td>14</td>
</tr>
<tr>
<td>1.8</td>
<td>Transitions Between Vertices</td>
<td>15</td>
</tr>
<tr>
<td>1.9</td>
<td>Transition Conglomerate Classes</td>
<td>17</td>
</tr>
<tr>
<td>1.10</td>
<td>Examples for Transition Conglomerates (i) – Sources, Targets and Containers</td>
<td>20</td>
</tr>
<tr>
<td>1.11</td>
<td>Examples for Transition Conglomerates (ii) – State Hierarchies Possibly Left</td>
<td>21</td>
</tr>
<tr>
<td>1.12</td>
<td>Examples for Transition Conglomerates (iii) – State Hierarchies Entered</td>
<td>23</td>
</tr>
<tr>
<td>1.13</td>
<td>Examples for Transition Conglomerates (iv) – Priorities and Conflicts</td>
<td>24</td>
</tr>
<tr>
<td>1.14</td>
<td>PERT Graphs</td>
<td>25</td>
</tr>
<tr>
<td>1.15</td>
<td>Basic PERT Graph Operations</td>
<td>25</td>
</tr>
<tr>
<td>1.16</td>
<td>Implied Dependencies and Restrictiveness</td>
<td>26</td>
</tr>
<tr>
<td>1.17</td>
<td>Examples for Application of PERT Graphs</td>
<td>27</td>
</tr>
<tr>
<td>1.18</td>
<td>New Concepts in the Metamodel</td>
<td>28</td>
</tr>
<tr>
<td>1.19</td>
<td>Illustration of Operation – Initialization</td>
<td>34</td>
</tr>
<tr>
<td>1.20</td>
<td>Illustration of Operation – Processing a Trigger</td>
<td>35</td>
</tr>
<tr>
<td>1.21</td>
<td>High-Level View of the KTS Operation</td>
<td>36</td>
</tr>
<tr>
<td>2.1</td>
<td>The Quantum Hierarchical State Machine Pattern and an Example for its Application</td>
<td>44</td>
</tr>
<tr>
<td>2.2</td>
<td>Implementation of Statecharts in the MDA Framework</td>
<td>46</td>
</tr>
<tr>
<td>2.3</td>
<td>Overview of Automatic Implementation of UML 2.0 Statecharts</td>
<td>47</td>
</tr>
<tr>
<td>2.4</td>
<td>Economical Forces, Engineering Aspects and Implied Architecture</td>
<td>48</td>
</tr>
<tr>
<td>2.5</td>
<td>Architectural Forces and Design Decisions</td>
<td>50</td>
</tr>
<tr>
<td>2.6</td>
<td>Modeling Concepts Affected by Mapping to the Target Platform</td>
<td>51</td>
</tr>
<tr>
<td>2.7</td>
<td>Design Decisions and Affected Modeling Concepts</td>
<td>51</td>
</tr>
<tr>
<td>2.8</td>
<td>Examples for Identifier Assignment</td>
<td>54</td>
</tr>
<tr>
<td>2.9</td>
<td>Conversion of PERT Graphs to Activity Sequences</td>
<td>57</td>
</tr>
<tr>
<td>2.10</td>
<td>Platform-Specific Metamodel</td>
<td>57</td>
</tr>
<tr>
<td>2.11</td>
<td>Platform-Specific Metamodel (Base Metaclasses)</td>
<td>58</td>
</tr>
<tr>
<td>2.12</td>
<td>Metamodel of the ANSI-C Implementation</td>
<td>62</td>
</tr>
<tr>
<td>2.13</td>
<td>Synthetic Benchmark Models</td>
<td>67</td>
</tr>
<tr>
<td>2.14</td>
<td>Read-Only Memory Consumption (“Loop” and “Mesh” Models)</td>
<td>68</td>
</tr>
<tr>
<td>2.15</td>
<td>RAM Consumption (“Loop” and “Mesh” Models)</td>
<td>69</td>
</tr>
<tr>
<td>2.16</td>
<td>Execution Times (“Loop” and “Mesh” Models)</td>
<td>70</td>
</tr>
<tr>
<td>2.17</td>
<td>Read-Only Memory Consumption (“Hierarchy” and “Interlevel” Models)</td>
<td>71</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table Number</th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Memory Consumption (&quot;Loop&quot; and &quot;Mesh&quot; Models)</td>
<td>68</td>
</tr>
<tr>
<td>2.2</td>
<td>Execution Times (&quot;Loop&quot; and &quot;Mesh&quot; Models)</td>
<td>69</td>
</tr>
<tr>
<td>2.3</td>
<td>Memory Consumption (&quot;Hierarchy&quot; and &quot;Interlevel&quot; Models)</td>
<td>70</td>
</tr>
<tr>
<td>2.4</td>
<td>Execution Times (&quot;Hierarchy&quot; and &quot;Interlevel&quot; Models)</td>
<td>71</td>
</tr>
<tr>
<td>3.1</td>
<td>Rewriting Kripke Transition Systems to Kripke-Structures</td>
<td>82</td>
</tr>
<tr>
<td>3.2</td>
<td>Evaluation Node types in the Example</td>
<td>91</td>
</tr>
<tr>
<td>3.3</td>
<td>Truth Table of Ternary Logic Operators</td>
<td>94</td>
</tr>
<tr>
<td>3.4</td>
<td>Comparison of Execution Times with Havelund and Rosu’s Approach ($\phi_1$ and $\phi_2$)</td>
<td>98</td>
</tr>
<tr>
<td>3.5</td>
<td>Comparison of Execution Times with Havelund and Rosu’s Approach ($\phi_3$)</td>
<td>98</td>
</tr>
<tr>
<td>3.6</td>
<td>Comparison of Execution Times with Finkbeiner and Sipma’s Approach</td>
<td>99</td>
</tr>
<tr>
<td>3.7</td>
<td>Overview of Models, Sources and Experiments</td>
<td>111</td>
</tr>
<tr>
<td>3.8</td>
<td>Results of the Fault Injection Campaign</td>
<td>112</td>
</tr>
</tbody>
</table>
Listings

1.1 Pseudocode of the Fireable Method ............................................. 33
1.2 Pseudocode of the FireSingle Method ......................................... 33
1.3 Pseudocode of the Fire Method ..................................................... 34
1.4 Pseudocode of the InitializationStep Method ............................... 37
1.5 Pseudocode of the TriggerProcessingStep Method ........................ 38
2.1 Implementation of the Time State ................................................ 45
2.2 Implementation of the Display State ............................................. 45
2.3 AsmL Specification of the Traverse Hierarchy Function .................. 54
2.4 Pseudocode for Conversion of PERT Graphs to Activity Sequences .... 56
2.5 AsmL Implementation of the Enabled Function (PSM) .................... 60
2.6 AsmL Implementation of the LeavingHierarchy Function (PSM) ....... 61
2.7 ANSI-C Implementation of Transition Conglomerates .................... 62
2.8 ANSI-C Implementation of the Enabled Function ......................... 62
2.9 ANSI-C Implementation of the LeavingHierarchy Function ............. 64
3.1 Pseudocode of the RecProp Method ............................................. 95
3.2 Pseudocode of the CheckTrace Method ........................................ 96
A.1 Pseudocode of the EnteringHierarchy Method .............................. 150
A.2 Pseudocode of the EnteringHierarchySet Method ......................... 151
A.3 Pseudocode of the LeavingHierarchy Method .............................. 151
A.4 Pseudocode of the LeavingHierarchySet Method ......................... 152
A.5 Effects of Transition Conglomerates (Class D) ............................ 152
A.6 Effects of Transition Conglomerates (Class E) ............................ 153
A.7 Effects of Transition Conglomerates (Class F) ............................ 153
A.8 Phase of Operation (PIM) ......................................................... 154
A.9 State of the Kripke Transition System (PIM) .............................. 154
A.10 Evaluation of Variables (PIM) ................................................... 155
A.11 Key Data Members of the KTS class (PIM) ................................... 155
A.12 AsmL Specification of the Enabled Function (PIM) ...................... 157
A.13 AsmL Specification of the NonOverpowered Function (PIM) ......... 157
A.14 AsmL Specification of the ConflictFree Function (PIM) ............... 158
A.15 AsmL Specification of the Fireable Function (PIM) ...................... 158
A.16 AsmL Specification of the LeavingHierarchy Function (PIM) ....... 159
A.17 AsmL Specification of the LeavingHierarchySet Function (PIM) .... 160
A.18 AsmL Specification of the EnteringHierarchy Function (PIM) ....... 161
A.19 AsmL Specification of the EnteringHierarchySet Function (PIM) .... 161
A.20 AsmL Specification of the LeavingSource Function (PIM) ............ 162
A.21 AsmL Specification of the PerformingEffect Function (PIM) ......... 162
A.22 AsmL Specification of the EnteringTarget Function (PIM) .......... 163
A.23 AsmL Specification of the FireSingle Function (PIM) ................. 163
A.24 AsmL Specification of the Fire Function (PIM) ........................... 164
A.25 AsmL Specification of the Init Function (PIM) ........................... 164
A.26 AsmL Specification of the InitializationStep Function (PIM) ........................................... 165
A.27 AsmL Specification of the TriggerProcessingStep Function (PIM) ................................. 166
A.28 AsmL Specification of the Operation Function (PIM) ...................................................... 168
A.29 Call Syntax of the Simulator .............................................................................................. 177
A.30 Initialization in the Simulator ............................................................................................ 178
A.31 Processing a Trigger in the Simulator ................................................................................ 179
B.1 Enumeration of States and Triggers ..................................................................................... 182
B.2 Implementation by the Nested Switch Pattern ..................................................................... 183
B.3 Implementation by the State Table Pattern ......................................................................... 184
B.4 Phase of Operation (PSM) .................................................................................................. 185
B.5 State of the Kripke Transition System (PSM) .................................................................... 185
B.6 Evaluation of Variables (PSM) ............................................................................................ 186
B.7 Key Data Members of the KTS Class (PSM) ....................................................................... 186
B.8 AsmL Specification of the Enabled Function (PSM) .......................................................... 187
B.9 AsmL Specification of the NonOverpowered Function (PSM) ........................................... 188
B.10 AsmL Specification of the ConflictFree Function (PSM) .................................................. 189
B.11 AsmL Specification of the Fireable Function (PSM) .......................................................... 189
B.12 AsmL Specification of the LeavingHierarchy Function (PSM) ......................................... 190
B.13 AsmL Specification of the LeavingHierarchySet Function (PSM) ..................................... 191
B.14 AsmL Specification of the EnteringHierarchy Function (PSM) ......................................... 191
B.15 AsmL Specification of the EnteringHierarchySet Function (PSM) ..................................... 192
B.16 AsmL Specification of the LeavingSource Function (PSM) .............................................. 192
B.17 AsmL Specification of the PerformingEffect Function (PSM) ........................................... 193
B.18 AsmL Specification of the EnteringTarget Function (PSM) ............................................... 193
B.19 AsmL Specification of the FireSingle Function (PSM) ....................................................... 194
B.20 AsmL Specification of the Fire Function (PSM) ................................................................. 196
B.21 AsmL Specification of the Init Function (PSM) .................................................................. 196
B.22 AsmL Specification of the InitializationStep Function (PSM) .......................................... 198
B.23 AsmL Specification of the TriggerProcessingStep Function (PSM) ................................. 199
B.24 AsmL Specification of the Operation Function (PSM) ...................................................... 200
B.25 Simplified AsmL Specification of the Operation Function (PSM) ................................. 200
B.26 Entry Activity of State Red-Yellow ..................................................................................... 204
B.27 Call Syntax of the Code Generator ..................................................................................... 204
B.28 Generation of Source Code for the Traffic Light ............................................................... 204
C.1 Enumeration of Top Interface Expression Identifiers ......................................................... 211
C.2 Evaluation Node Classes (Headers) ..................................................................................... 212
C.3 Enumeration of Bottom Interface Expression Identifiers .................................................. 212
C.4 The Evaluation Node Base Class (Fragment) ...................................................................... 213
C.5 Implementation of the Head Method ................................................................................... 213
C.6 Implementation of the Next Method in Evaluation Node Classes EN47613 and EN12544 214
C.7 Evaluation of Top Interface Expressions in Ternary Logic .............................................. 214
C.8 Implementation of the IntProp Method in Evaluation Node Class EN47613 .................... 214
C.9 Implementation of the ExtProp Method in Evaluation Node Class EN47613 .................... 215
C.10 Implementation of the TermTrace Method in Evaluation Node Class EN47613 .............. 215
C.12 Implementation of the RecProp Method ............................................................................ 216
C.13 Implementation of the CheckTrace Method ...................................................................... 217
Introduction

As computer-based systems pervasively extend to virtually all facets of life ranging from power plant control through everyday PCs to nearly invisible computers working as embedded intelligence in vehicles, health-care, or mobile devices, the dependence of the society on their correct behavior leads to growing impact of these systems’ dependability.

This PhD thesis presents research achievements of over three years focusing on various aspects of software dependability. Most of the results are related to state-based event-driven systems specified by statecharts of the UML 2.0 modeling language: Chp. 2 presents a novel method for automatic source-code level implementation of statecharts with low CPU and memory consumption enabling its application in resource constrained embedded environments; Chp. 3 presents two runtime error detection techniques for indicating abnormal behavior exposed by statechart-based implementations; our first solution introduces a temporal logic language for statecharts enabling the definition of temporal correctness criteria and a corresponding high-performance solution for evaluating these formulae during runtime; the second approach is a watchdog scheme for runtime monitoring the event-driven behavior and checking whether the operation corresponds to the statechart of the system. As both the implementation of statecharts and runtime error detection necessitates the unambiguous definition of statechart semantics, Chp. 1 presents a detailed, formal operational semantics for statecharts providing a well-established foundation for the further discussions. The last subject addressed in this document is related to automatic processing of data obtained from experimental evaluation of dependability: Chp. 4 presents a method for automatic identification of key infrastructural factors that determine the dependability properties of complex HW/SW systems built up of COTS components by the application of data mining. This method was used in the evaluation of the runtime error detection techniques.

The logical organization and interdependence of subjects are illustrated in Fig. 1. The four top-level (orange) rectangles correspond to the four main research areas (thus the four main chapters of the thesis). The internal structure of these top-level regions reflects the main areas covered in the corresponding chapter similarly to the usual illustration of complex hardware-software systems or protocols in which the services are decomposed to multiple layers where a higher layer uses the lower ones – this way the sub-figures are not to be read in a top-to-bottom, left-to-right order but from bottom to the top. The black arrows between the top-level rectangles indicate the application of achievements of some research area in another field.

The four chapters are composed according to the same structure: the introduction outlines the actual focus of discussion and presents an overview on related work (with the exception of Chp. 3 where the discussion of approaches published previously in the literature is divided into two parts according to the two solutions), then the main part discusses the corresponding solution, finally the last section summarizes the discussion, highlights the novelties and contribution, enumerates our related publications, outlines the state of implementation, emphasizes the application possibilities of achievements and indicates the possible directions of future research. The introductions of all chapters contain a figure similar to Fig. 1 zooming the corresponding part of the overview illustration and annotating it with indication of section numbers. The remaining four sections of this introduction discuss the motivations for having chosen the research areas discussed in the thesis.
A Formal Operational Semantics for UML 2.0 Statecharts

The first subject addressed in the thesis (Chp. 1) is related to the operational semantics of statecharts in the UML 2.0 visual modeling language. With respect to the dominating trends in modern software industry, the importance of visual modeling and the actuality of subject does not need much discussion: the straightforward graphical notation enables the construction of models of high documentation value supporting the communication with the users, helping the organization of work within the development team and providing powerful facilities for ruling the complexity. Apart from the documentation role the direct application of visual models in the development receives ever growing attention: a semantically well established model can be the basis of automatic implementation (code generation), various verifications can be performed on models to prove their correctness with respect to some dependability requirements (model checking), moreover, formal models play even the role of executable specification in runtime verification approaches.

UML is the most popular modeling language in the software industry, its most recent version, 2.0 offers sophisticated modeling facilities throughout the entire development chain (requirement analysis, design, implementation and deployment). Many professional UML modeling tools are available and most of them have already been integrated into full featured development environments supporting requirement management, development, debugging, version control, etc. The rigorously defined syntax of the language (metamodel) and the XML metadata interchange built upon it (XMI) [109] enables interchanging models between various tools.

At present, UML is primarily applied for modeling static aspects of the software under development (use cases, data structures, package hierarchies, deployment etc.). Although the language provides various facilities for modeling dynamic aspects (activity diagrams, statecharts, communication diagrams etc.) the visual modeling of the behavior is usually less emphasized (with the possible exception of interface and test design where interaction and activity diagrams are used). As we will see, the probable reason for this is the fact that the meaning of behavior models is less intuitive as the semantics of e.g.,
class diagrams. This unbalanced situation at the expense of behavior models should be undoubtedly corrected since the application of model-based software development approaches like automatic program synthesis, testing and runtime verification are expected to considerably reduce development costs, increase productivity and enhance the quality of the software delivered. In order to achieve this, we are obviously in the need of formally well established visual behavior models. The first three parts of the thesis focus on various aspects of modeling behavior, synthesizing programs and runtime verification; the visual notation used is the statechart formalism of the UML 2.0 modeling language.

In order to enable the application of visual models in engineering practice, naturally emerges the need for assigning unambiguous meaning to graphical models i.e., we have to define the formal semantics of these models. Similarly to all non-trivial languages, also in case of UML holds, that the pure definition of the grammar does not prevent the construction of semantically meaningless models thus various well-formedness criteria are to be defined in order to prevent the uncontrolled usage of modeling artifacts. In contrast to the detailed definition of syntax and the sophisticated graphical notation, the formal specification of semantics is unfortunately a weak point of UML resulting in the emergence of many non elaborated details and open questions:

- The standard uses OMG’s Object Constraint Language (OCL) [110] for ensuring the well-formedness of models. Unfortunately with respect to some important artifacts, the corresponding well-formedness rules are simply missing from the standard and some requirements are discussed only informally.

- Focusing on statecharts it is to be highlighted that some concepts related to state refinement, hierarchical organization, compound transition structures and compound activities are defined by the standard only informally, their unambiguous formal definition is missing.

- Finally the most important deficiency of the standard is that it does not define a formal operational semantics for statecharts at all. The operation is meant to be introduced by textual discussion relying on human intuition but unfortunately in case of non-trivial models we can easily encounter such situations where the expected behavior is not clear i.e., which transitions are to be fired, what is the resulting state configuration etc.

There were many formalization efforts published in the literature aiming at the definition of a formal semantics for statechart models. Unfortunately (as discussed in-depth in Chp. I) the well-known approaches are usually not fully elaborated or the mathematical formalisms applied in them are far from the engineering practice. According to the discussion above, the goal of my research was to define a formal operational semantics for UML 2.0 statecharts supporting the entire practically relevant set of modeling elements. The solution should be a pragmatic approach close to engineers’ way of thinking. In order to achieve this, I identified the following sub-tasks:

(i) We need a formalism for the unambiguous definition of concepts related to state refinement and we have to formally define the corresponding well-formedness rules since the standard mostly provides textual discussion only, the OCL constraints defined by it are sometimes contradictory or incomplete.

(ii) Although the standard informally uses the concept of compound transitions built up of multiple basic transitions and pseudostates, the formal definition is missing and the related well-formedness constraints are incomplete. We need to explicitly define the notion of compound transition structures, clarify the concepts related to enabled transitions, discuss the tasks to be carried out when firing a transition and we have to unambiguously define the corresponding well-formedness rules.

(iii) We need a formalism for compound activity structures that is capable of unambiguously expressing the subsequence relations amongst basic activities to be carried out when firing transitions (state exit activities, effects, entering states) and indicate the possibilities for parallel execution.
(iv) Finally, based on the formalisms introduced above, we have to define the formal operational semantics of statecharts. The approach has to be both mathematically well-established and easily applicable in the engineering practice. Mathematical preciseness should be aimed at by building the formalism on a finite state-transition system (e.g., a Kripke transition system) and this specification should be finally mapped to algorithms that are easy to implement.

**Automatic Implementation of UML 2.0 Statecharts**

The widespread application of visual modeling languages naturally implies the need for automatic implementation of these models. The substitution of the time-consuming and labor intensive manual programming with automatic tools promises considerable enhancement of productivity and software quality. Modern modeling environments usually provide some support for automatic implementation of static aspects of software models (e.g., generation of data types, function declarations, class headers, interfaces etc. according to the syntax of a programming language) i.e., these solutions are restricted to the interpretation of data models (package, class and deployment diagrams). Automatic implementation of dynamic models (activity and collaboration diagrams, statecharts) is supported only by a very few number of high-end tools. The probable reason for this is the fact that while the implementation of a class model in a programming language can be easily achieved by some template-based approaches (since there is a direct association between data modeling concepts and object-oriented programming language artifacts), the implementation of behavior means actually writing programs. This is such an intellectual challenge for which the solution requires both the in-depth understanding of the complex formal semantics of behavior models, solid knowledge on the characteristics of the target platform and obviously a considerable expertise in the programming language used. These difficulties render the automatic implementation of statecharts a much more complicated task than generating data type declarations from a class model according to some simple templates. Even with respect to the issues mentioned above, the automatic implementation of behavior specified by statecharts is an important and up-to-date research area since because of the rich modeling toolkit and the complexity of the operational semantics the manual implementation of behavior specified by statecharts is a highly labor intensive and error-prone task and a minor modification of the model may require rewriting the complete implementation.

The reactive, state-based, event-driven behavior is typical for embedded systems that are in direct connection with their environment. In case of this application area not only the theoretical issues mentioned above are to be considered but we also have to take into account that the computing resources of the underlying platform (operative memory, processing power) are seriously constrained (small memory, weak CPU or microcontroller) thus the popular modern object-oriented languages and the corresponding virtual machines (Java, .Net, etc.) are not available in most cases.

None of the approaches published previously in the literature for the implementation of event-driven state-based systems are sophisticated enough for the implementation of UML 2.0 statecharts. The best known corresponding approach is the State design pattern [47], but unfortunately it is not much more than a set of suggestions for the representation of states by classes in a plain (non-hierarchical) state machine and does not propose any solutions for handling state hierarchies, concurrent operation etc. Some popular modeling environments also advertise providing support for the implementation of statecharts but taking a closer look we usually find that these solutions do not provide more than the automatic declaration of some function signatures or the approach is restricted to seriously simplified non-hierarchical state diagrams [16]. The most elaborated implementation method, the Quantum Hierarchical State Machine (QHsm) pattern published in the literature was proposed by Miro Samek [144, 145]. The QHsm pattern provides support for the implementation of those UML statecharts that do not contain concurrent state decomposition. According to my knowledge the most complete solution is provided by the Rhapsody and VisualSTATE software families from the I-Logix and IAR companies respectively; however since these are high-end commercial integrated modeling and development
environments, the implementation algorithm of statecharts was not published.

With respect to the discussion above, the goal of my research was to propose a method for automatic programming language level implementation of UML 2.0 statecharts according to the operational semantics presented in Chp. 1. I decided to put explicit emphasis on the applicability of my approach in resource constrained embedded systems. In order to achieve this I identified the following sub-tasks:

(i) We have to identify the typical resources available on the target platform (resource constrained embedded systems) and based on this analysis the platform-specific syntax and semantics of statecharts is to be specified according to OMG’s Model Driven Architecture initiative. The platform-specific representation maps the general UML artifacts and behavioral concepts to the resources available on the target platform (e.g., the CPU available on an embedded platform is not capable of the parallel execution of multiple threads thus concurrent operation should be substituted by a valid sequential scheduling of activities, with respect to the low amount of available memory we have to find compact solutions for storing data, furthermore in case of real-time system we may have to substitute recursive algorithms by iterative implementations whose execution time analysis is easier, etc.). Having defined the platform-specific representation we have to prove that the resulting semantics corresponds to the platform-independent original one with respect to the characteristics of the target platform.

(ii) The algorithms and data structures of the platform-specific semantics are to be implemented in a programming language that seamlessly fits to the target platform. Similarly to the previous step we have to prove that the resulting source code correctly implements the platform-specific semantics.

(iii) The resource consumption of the applications built according to our approach is to be analyzed (memory consumption and execution time) and we also have to experimentally evaluate these properties for some benchmark models. With respect to the practical applicability of the approach I also find important to actually compare my solution with the most elaborated method published previously in the literature (QHsm) instead of barely presenting measurement results.

Error Detection in UML 2.0 Statechart Implementations

The issues related to the semantics and automatic implementation of visual behavior models are addressed in the first two main parts of the thesis; Chp. 3 focuses on another emerging research area by discussing the application of behavior models as reference information for runtime detection of errors caused by design, implementation or physical faults.

Aiming at the enhancement of software dependability is another goal that does not require much justification: the use of both general-purpose and specialized computers pervasively extends to virtually all facets of life, the dependence of the society on their correct behavior leads to growing impact of these systems’ dependability. In the past, the term “safety critical” was primarily used for referring to such evident cases as flight control systems or nuclear power plant operation etc. At present, the wide variety of applications ranging from on-board embedded microcontrollers in cars to the use of high performance servers driving the Internet infrastructure all constitute “critical” systems whether for their clear impact on human health or financial conditions [30]. The typical origins of failures have also changed over the years: while in early computer-based systems the primary causes of service unavailability was related to hardware faults, modern systems are usually built on highly dependable hardware thus the main cause of failures is related to design or implementation faults in the software [27, 51, 167]. Due to to its importance, runtime error detection is explicitly required even by standards of software development for safety critical applications (e.g., IEC 61508, EN 50128).

Using the approaches presented here complements the exploitation of behavior models’ expressive power: the same statechart can play the role of design documentation, provides the input for automatic code generation and can be used as reference information for runtime error detection.
Having decided to aim at runtime checking of dynamic behavior, the first obvious task is to select those classes of faults that our approaches should address and identify those phases of the software development when these faults may be introduced. As a beneficial property of model-based development, we are not restricted to checking informal and hard to capture textual requirements during the execution of the implementation but we can provide integrated and formally well-established solutions for addressing faults causing behavioral errors introduced at any key phases of the development.

In correspondence to the V-model of safety-critical software development (see Chp. 3) I considered the following development phases during my research (still focusing on the specification of control know, i.e., evolution and implementation of statecharts): (i) during requirement analysis those fundamental temporal dependability requirements are collected that the implementation should meet in any cases (e.g., avoidance of dangerous operation modes etc.); (ii) the early versions of statecharts that capture only the key aspects of behavior (states, notion of safe and dangerous situations, operation modes, etc.) are constructed in the specification phase; although these early models require further elaboration, they enable the formal specification of the previously collected temporal dependability requirements in the context of statecharts; (iii) the fully elaborated statecharts that unambiguously specify the behavior of the system are prepared in architecture and module design phases; these statecharts correspond to the draft ones prepared in the specification phase (i.e., they can be considered as refinements of early models) and they are ready for being implemented; (iv) the source code level representation of behavioral models is prepared in the implementation phase by manual programming or automatic code generation; (v) finally after verification, integration and validation phases the system is certified and deployed. According to the observation above the following points can be identified where faults can be introduced into the system (Fig. 3.2):

- If the set of requirements collected in the requirement analysis phase is inconsistent or faulty or their initial formalization carried out in the specification phase is incorrect, the entire development is likely to fail due to fundamental misunderstanding between the developers and future users of the system. Ensuring that the user requirements (functional and non-functional ones) are fully understood by the developers and these requirements are correctly formalized in the specification is usually aimed by in-depth communication with users involving the construction of visual models, screenshot concepts and fast prototypes and discussion of typical application scenarios. As the correctness of the initial specification primarily depends on informal human interaction, the earliest entry points of our solutions are the temporal requirements defined in the specification phase (thus faults introduced during requirement analysis and specification phases are out of the scope of our work). Note that most of the approaches published previously in the literature expect much more elaborated inputs (e.g., design models or the implementation) thus they can not address faults introduced during the development (i.e., during which these more elaborated models are constructed, e.g., architecture and module design, implementation phases etc.).

- During the architecture and module design phases such model refinement faults may be introduced that result in incorrect models violating some temporal requirements of the specification.

- The application built in the implementation phase may not correspond to the behavior model elaborated in design phases; the possible reasons for this are programming bugs or misunderstanding the semantics; the possibility of these faults can be reduced by automatic code generation but can not be totally eliminated while the entire source code is not automatically synthesized by proven correct techniques.

- Finally, transient and permanent faults may occur during operation in the underlying hardware-software infrastructure that affect the behavior.

The corresponding research presented in the thesis primarily focuses on two of these fault classes i.e., the ones introduced in design and implementation phases. Obviously for addressing a fault introduced at some phase we have to use models prepared at an earlier phase i.e., a reference that was not
affected by the fault. Thus we will address model refinement (design) faults by the runtime verification of temporal requirements defined in specification and analysis phases while we will aim at the detection of implementation faults by comparing the actual behavior of the application to the fully elaborated reference statechart model. It is important to highlight, however, that although our solutions are primarily aimed at addressing faults introduced during the development they may be capable of detecting behavioral errors that are manifestations of low-level physical faults; this is especially important in case of such resource constrained embedded systems where the sophisticated error detection mechanisms of modern microprocessors (e.g., memory segmentation, privilege levels) and the solutions for ensuring memory integrity (e.g., error correcting codes) are not available.

Having decided to primarily aim at addressing model refinement and implementation faults, we have to discuss how to use models prepared in specification and design phases as reference information and how to actually check whether the software behaves according to these models – below we will outline these considerations.

For addressing design faults (i.e., faults in the model refinement) it seems to be beneficial to define a temporal logic language for the definition of temporal requirements and checking that these requirements hold during runtime. According to our expectations by defining requirements in the context of specification models we will be able to address faults introduced even during the development (model elaboration) in contrast to previous approaches that use (fully elaborated and possibly faulty) design models as reference.Having defined the language, the evaluation of its formulae over execution traces of observed applications is another issue. The thesis presents a detailed overview about previously published methods and tools; this survey can be summarized as follows: although there have been several approaches proposed in the literature from the field of model checking, these solutions are not usable for handling finite execution traces for semantic reasons or can not be directly applied in resource constrained embedded systems due to their high resource consumption. (Model checking typically aims at proving various properties of protocols and automata expecting them to be running infinitely long. Since runtime error detection has to observe real applications that may happen to be terminated, the language and the method of verification has to be able to semantically handle the notion of finite execution traces.)

The goal of the implementation phase is to map the fully elaborated models to a programming language thus it seems to be obvious to specify a runtime monitor that compares the application’s actually exposed behavior to the design models (i.e., fully elaborated statecharts) in order to detect errors caused by implementation and operational (physical) faults. The monitor discussed here closely resembles some watchdog solutions proposed previously in the literature. Watchdog processors (WDP) are relatively simple coprocessors that detect if the application running on the main CPU deviates from the behavior specified by the control flow graph. A control flow graph (CFG) is a directed graph whose nodes represent those blocks of machine instructions that do not contain branches while edges of the graph correspond to syntactically enabled control transfer (jump instructions, subroutine calls etc.). The correct (expected) behavior described be the control flow graph is called the reference information. The watchdog processor checks the behavior of the application by directly observing its instruction fetch operations on the system buses, or the watchdog can be implemented as an ordinary peripheral device to which the application explicitly transmits information about its current activities. The fragments of observed behavior (instructions fetched or information explicitly transmitted) are called signatures of operation. The control flow graph can be stored in the watchdog (by an adjacency matrix or list, etc.) or it can be embedded in the signatures sent by the application. The idea of our approach is to construct a watchdog-like structure whose reference “control flow graph” is actually a fully elaborated statechart. Although methods based on control flow graphs were successfully applied for monitoring relatively low-level constructs (functions, interrupt routines, etc.) our task is more complicated than the comparison of a control flow graph and an execution trace since states of a statechart can be organized into a hierarchy even involving concurrent operation. The need for runtime error detection based on a statechart specification has already occurred in the literature but the solution has still remained
an open issue. From an implementation point of view we have to take into consideration that in our
case the reference information is of much higher abstraction level than in case of previous approaches:
while in case of a traditional watchdog processor an edge in the control flow graph corresponds to a
single or a few number of machine instructions, in our case an edge corresponds to the entire response
of the application to an event – this means that the steps to be checked are delivered to the monitor
at a significantly lower bandwidth but actually checking these steps is a much more complicated task
than searching for an edge in a graph. According to this observation, in contrast to the inherently
hardware-based approach of traditional watchdog processors, we will develop a software solution (but
obviously not excluding the hardware implementation).

To put together the goals of our research, we will present two complementary approaches: faults
possibly introduced during model refinement (architecture and module design) will be addressed by
defining and checking temporal requirements while errors caused by implementation faults are to be
detected by the runtime monitor. I defined the following sub-tasks:

(i) We need a temporal logic language fitted to statecharts that besides the usual Boolean and
temporal constructs provides facilities for referring to statechart artifacts (states, transitions,
activities etc.).

(ii) We have to develop an efficient method for runtime evaluation of temporal logic expressions on
finite execution traces. The low resource consumption of the solution should enable its application
as an error detection module in resource constrained embedded platforms.

(iii) Analogously to the watchdog processors known from the literature, we have to develop a run-
time monitor that checks whether the behavior exposed by the application corresponds to the
specification using fully elaborated statecharts as reference information.

(iv) Obviously emerges the need for assessing, which are those fault classes (design, implementation
or physical faults) that can be most effectively addressed by our solutions and which are the ones
that can be most beneficially addressed by the so called conventional error detection mechanisms
of the execution platform (memory protection, consistency checking of machine instructions,
sanity checking of arithmetic operations etc.); thus I decided to experimentally evaluate the
error detection capabilities of my solutions by a fault injection campaign.

Supporting Experiment Evaluation by Data Mining

It is widely accepted that the evaluation of dependability attributes of computer systems is a complex
task. Traditional techniques based on analysis and simulation have to be supported with experimental
evaluation of prototypes and observation of real systems. These experimental techniques, including
fault injection, robustness testing, and field measurements, have been successfully used to evaluate
specific fault tolerance mechanisms, validate robustness of software components, or to assess the general
impact of faults in systems.

In spite of the big diversity of techniques and tools now available, all the experimental dependability
evaluation approaches share a common problem: they tend to produce a large amount of raw data
that has to be processed to obtain the desired dependability measures. Very often the analysis of
the experimental data is quite complex, as it has to take into consideration many aspects of the
experiment setup e.g., the target system architecture and configuration, the workload, the type of
faults involved, the environmental aspects, etc. Surprisingly, the problem of coping with the large size
of the experimental data sets and the high complexity of the data analysis has received less attention
in the dependability research effort. Researchers have focused on the development of fault injection
and robustness testing tools and on the mitigation of problems such as experiment representativeness,
non-intrusiveness and portability of tools. In correspondence to this, many fault injection or robustness
testing tools have been proposed for the experimental evaluation of dependability attributes but all
these tools either provide rudimentary means to analyze data or, more frequently, just store the raw results in a spreadsheet format. Although this approach can be acceptable for very specific and simple analysis, it is clearly not enough when the analysis required is complex or when the amount of raw data is very large.

Another force rendering automatic measurement evaluation approaches an up-to-date research area is the occurrence of dependability benchmarking proposals. These new approaches represent an attempt to standardize experimental techniques with the goal of comparing dependability features of different systems or components. This research effort has already caught the attention of various companies and lead to many dependability benchmark proposals. Dependability benchmarks represent a new and important source of raw experimental data but the problem of analyzing that data has been even more neglected than in case of traditional fault injection and robustness testing. In fact, dependability benchmarks rely on a typically small set of measures and the data collected during the benchmark runs is just used to calculate the measures defined in the benchmark.

There are a relatively low number of proposals published in the literature focusing on automatic processing of measurement data: Madeira, Costa and Vieira suggested first [88] to store the large amount of data obtained from dependability evaluation experiments in data warehouses and support the analysis and visualization by OLAP methods (On-Line Analytical Processing) enabling this way the effective sharing and comparison of data between research groups. The approach presented by them considered the data analysis as an interactive process where the analyst has some a priori assumptions and specifies queries for the database using the slice-and-dice approach of OLAP to prove or falsify the original assumption. In this method the specification of relevant queries and understanding the results, thus the success of the analysis, heavily depends on the experience and human intuitions of the data analyst.

The application of data mining, the most advanced method of intelligent data processing was first suggested by Pataricza and Tolvaj [114] for analysis of data obtained from fault injection experiments. Data mining is usually defined as an interdisciplinary field bringing together techniques from machine learning, pattern recognition, statistics, databases, and visualization to address the issue of extracting previously unknown, valid and actionable information from large databases to be used for making crucial business decisions [54]. Data mining surpasses the semi-automatic, human interaction-based approach of OLAP analysis by relying on fully automatic machine learning algorithms enabling this way the analysis of data sets from non-intuitive aspects that are not emerging from any a priori human intuitions.

Although the papers mentioned above propose valuable ideas, the automatic processing of data obtained from experimental evaluation of dependability attributes has still remained an open issue. It is important to highlight that we do not argue that the automatic processing of large data sets has not been discussed in the literature since obviously OLAP analysis and data mining are mathematically well-established and successful research areas – we would like to highlight here that despite of the success of data intelligence in the business field these techniques have not been used for processing data obtained from dependability evaluation experiments.

According to the observation above the goal of my corresponding research was to propose a method for automatic analysis of data obtained from experimental evaluation of dependability by the application of data mining techniques; I identified the following sub-tasks:

(i) We have to elaborate a data mining based method that enables the automatic identification of those (infrastructural) factors and faultload properties that determine the behavior of a complex system in the presence of faults by processing data obtained from fault injection experiments.

(ii) The viability of the approach should be demonstrated by the analysis of data obtained from a dependability benchmarking experiment.

(iii) Finally I would like to demonstrate the joint application of achievements from the four parts of the thesis by a comprehensive experiment campaign according to the following scenario: (i)
let us define a *behavior model* by a UML 2.0 statechart (according to the semantics discussed in the first part of the thesis), (ii) *generate* the source code of applications by the approach discussed in the second part of the thesis, (iii) examine the behavior of applications in the presence of faults applying the *error detection methods* presented in the third part of the thesis (the faultload should contain both model refinement, implementation and low-level faults) and finally (iv) *analyze* the resulting large database by the method elaborated in the fourth part of the thesis. The experiments should aim at the identification of those parameters (configuration of the code generator, compiler options, faultload, workload etc.) that primarily determine the dependability of service delivered by the system and the efficiency of error detection mechanisms.
Chapter 1

A Formal Operational Semantics for UML 2.0 Statecharts

1.1 Introduction

The first area addressed in this document is related to the semantics of UML 2.0 visual modeling language. With respect to the current trends in software development the importance of visual modeling would be hard to over-estimate. The straightforward graphical notation enables the construction of documentation quality models for supporting the communication between users and developers and provides efficient means for ruling the complexity. Beyond the documentation role, direct application of visual models is gaining ever higher importance in actual development: a semantically well-established model can be the basis of automatic code generation, various model checking investigations can be performed on visual models in order to prove the correctness of the system under development according to some dependability requirements and in runtime error detection approaches formal models play the role of executable specification.

This chapter presents a formal operational semantics for statecharts of the UML 2.0 language. Statecharts are used for modeling reactive, state-based systems. A system is said to be reactive if its response to external stimuli is determined by both the event actually received and the previous history of the system. A system exhibits state-based behavior if it operates differently during different periods and if these periods can be partitioned into finite, non-overlapping states [144]. Not all systems reveal state-based behavior, e.g., basic mathematical functions, such as sin(x) return the same value for a given x regardless of previous inputs and similarly although continuous operation may depend on previous inputs e.g., in case of digital filters, it can not be divided into a finite number of states [32].

Multiple approaches have been proposed for modeling reactive state-based systems: (i) there are fundamental mathematical formalisms (finite state machines specified by Kripke structures, labeled transition systems and Kripke transition systems) that are ideal for various analyses, model checking and verification; (ii) multiple engineering interpretations of fundamental facilities have been proposed (Mealy and Moore automata), and finally (iii) these technical notations were further developed to sophisticated visual formalisms used in modern software modeling languages.

In fundamental mathematical approaches finite state transition systems are specified by Kripke structures, labeled transition systems and Kripke transition systems. For an AP set of labels a Kripke structure (KS) is a \( M = (S, I, T, L) \) tuple where \( S \) is the set of states, \( I \) is the set of initial states \( (I \subseteq S) \), \( T \subseteq S \times S \) is the state transition relation and \( L : S \rightarrow 2^{AP} \) is the state labeling function; the traversal in the state space \( S \) specified by \( T \) can represent the operation of the modeled system, while the labels assigned to states by \( L \) enable the reasoning about properties of specific states. In labeled transition systems (LTS) labels are called actions and are assigned to transitions instead of states, i.e., for an action set \( A \) a LTS is an \( M = (S, I, \rightarrow) \) tuple where \( S \) is the set of states, \( I \) is the set of initial states \( (I \subseteq S) \) and \( \rightarrow \subseteq S \times 2^A \times S \) is the labeled transition relation. Kripke transition systems (KTS)
CHAPTER 1. A FORMAL OPERATIONAL SEMANTICS FOR UML 2.0 STATECHARTS

are combinations of KS and LTS systems: for an AP set of labels and A set of actions a KTS system is an \( M = (S, I, L, \rightarrow) \) tuple where \( S \) is the set of states, \( I \) is the set of initial states \( (I \subseteq S) \), \( L : S \rightarrow 2^{AP} \) is the state labeling function and \( \rightarrow \subseteq S \times 2^A \times S \) is the labeled transition relation.

As fundamental approaches do not explicitly provide means for modeling input events and output activities, engineering interpretations were developed that fill the concepts of state and transition labels with more concrete meanings. Mealy automata\,[101] are finite state machines that generate outputs according to state transitions; a Mealy automaton is an \( M = (S, s_0, \Sigma, A, T, O) \) tuple where \( S \) is the set of states, \( s_0 \) is the initial state \( (s_0 \in S) \), \( \Sigma \) is the set of input events, \( A \) is the set of output activities, \( T : (S \times \Sigma) \rightarrow S \) is the state transition function and \( O : (S \times \Sigma) \rightarrow A \) is the output function. In contrary to Mealy automata, in Moore machines\,[107] outputs are assigned to states, i.e., the output depends only on the actual state of the system but not directly on the events received; this interpretation relaxes the “zero time assumption” of Mealy machines where activities are considered to be performed between two states. A Moore automaton is an \( M = (S, s_0, \Sigma, A, T, O) \) tuple where the meanings of \( S, s_0, \Sigma, A \) and \( T \) are the same but the \( O : S \rightarrow A \) assigns activities to states. The typical visual representation of Mealy and Moore automata are state diagrams.

Although the fundamental formalisms and their engineering interpretations were successfully used for modeling and analysis of relatively small electronic circuits and software modules, they do not scale well to systems of high complexity. In order to address large-scale HW/SW systems, modern modeling languages were developed that provide facilities for organization of states into hierarchies, modeling complex transitions, concurrency, communication, etc. The new concepts were introduced primarily into the visual representation toolkit: flat state diagrams were enhanced to statecharts; unfortunately the underlying semantics was not as elaborated as in case of earlier modeling notations.

Statecharts were originally introduced by Harel\,[55] for modeling the behavior of reactive systems\,[57] in the structured analysis approach STATEMATE\,[56]. As the semantics of statecharts was originally defined in an informal way, shortly after the publication of Harel’s paper an extensive research began in order to improve its syntax and semantics. The first formal semantics published by the original author and his colleagues\,[58] was criticized for number of reasons and multiple alternative semantics were suggested. These alternative approaches focus on various issues, e.g., synchrony\,[12], causality\,[63], inter-level transitions, state references, compositability, timing\,[70], parallel execution, transition refinement, determinism, transition priorities etc. Von der Beeck provides a comparison of 20 Harel statechart variants in\,[163]. A well-known approach was proposed by Mikk, Lakhnech and Siegel in\,[105] describing statecharts by hierarchical automata and specifying the semantics by a Kripke-structure.

Due to the success of Harel statecharts, similar notations were introduced into modern software modeling languages, like ROOM proposed by Bran Selic et al. in 1994\,[150] and UML, the current de-facto modeling language for object-oriented systems also provides its statechart variant.

Similarly to the main issue of Harel statecharts, UML does not define the formal operational semantics of statecharts either, thus as happened in case of Harel’s variant, multiple solutions were proposed by the research community. The roots of most UML statechart semantics specifications can be traced back to papers of Pnueli et al.\,[140] and Mikk et al.\,[105] about Harel’s variant. Mikk et al.’s formalism was elaborated for UML statecharts by Latella, Majzik and Massink in\,[78] their extended hierarchical automaton (EHA) notation and the corresponding semantics was the basis of multiple approaches, e.g., Varró presented the same semantics by graph transformations in\,[160]. The drawback of the EHA-based approaches is that they do not handle some important features like the history mechanism, and the transformation does not support such constructs as final states, fork and join pseudostates, junction and choice vertices etc. Von der Beeck and Cleveland presented multiple enhancements over\,[78] in\,[85] and\,[164]; their approach supports history vertices but does not formally specify the semantics of initial, junction and choice pseudostates. As the approaches mentioned above were developed for earlier versions of the UML standard, they do not support new concepts like local transitions and termination.
1.1. INTRODUCTION

There are formalization approaches based on standard specification languages. Arendt’s paper [31] is claimed to specify a formal semantics for UML statecharts using the PVS specification language but it actually does not provide more than some well-formedness rules. Börger, Cavarra and Riccobene presented a method for modeling the dynamics of UML state machines by agents specified using Abstract State Machines [18]; unfortunately also lacking support for fork, join, junction, choice and history vertices. Another approaches based on graph transformation were presented by Kuske [74] and Gogolla [50] still lacking the support for several constructs.

In model-checking approaches, statecharts are transformed to an executable specification language, mostly to PROMELA, the input language of the SPIN model checker [62], this transformation can be considered as a denotational semantics. Transformation of Harel statecharts to PROMELA was first defined by Mikk, Lakhnech, Siegel and Holzmann in [106] and adapted to UML statecharts by Latella, Majzik and Massink in [77] – these approaches use hierarchical automata again as intermediate syntax. There were solutions proposed that directly translate statecharts to PROMELA without relying on intermediate representation, e.g., the vUML tool by Lilius and Paltor [81, 82] or the HUGO model checker by Schafer, Knapp and Merz [147]. The papers mentioned above do not discuss the semantics of junction, choice and history vertices and being primarily targeted for model checking do not put emphasis on the representation of activities, parallel execution etc. Blech, Glesner and Leitner’s paper [16] is claimed to address formal verification of Java code generation from UML statecharts, but their proposal is restricted to primitive finite state machines.

Eshuis and Wieringa presented a STATEMATE-based formal real-time semantics for UML statecharts in [43]; being based on a semantics originally developed for Harel statecharts their approach is significantly different from the informal specification of the UML standard.

To put together, the previously published solutions typically lack some key behavior modeling constructs and were developed for obsoleted versions of the UML standard. Most of approaches rely on custom formalism for representation of the statechart, only the model transformation-based approaches ([50, 74, 77, 160]) define the semantics in the context of the UML metamodel or connect the custom formalism (EHA) to standard metaclasses. However the direct connection of the semantics to the metamodel would be more than beneficial since using a custom intermediate representation for defining the semantics actually introduces the problem of having to formally specify the relation of the metamodel to the intermediate concepts, e.g., in [85] and [164] statecharts are described by textual terms, but the papers do not discuss how to formally derive them from an actual statechart. A semantics defined on the basis of the metamodel would be much easier to integrate to popular modeling environments (for simulation, code generation etc.) since the XML Metadata Interchange Format (XMI) [109] supported by most tools offers a straightforward basis of inter-tool communication.

The issues identified above are summarized in the left column of Fig. 1.1. Obviously the origins

![Figure 1.1: Identified Issues, Research Goals and Achievements](image-url)
of all problems are the complexity of the UML standard and the lack of a formal semantics defined by it. According to our observation, previously published approaches (i) suffer from the atomicity of concepts used by the standard since although most of basic elements (e.g., transitions between vertices in a compound fork-join path) cannot be considered in isolation, there are no higher level notions for compound transition structures and complex activities – this is the reason for not being able to rule the complexity and having to neglect many modeling artifacts in most approaches, (ii) formal definitions of refinement concepts (e.g., substate relation) are missing, (iii) previously published approaches were mainly targeted for model checking while implementation aspects received less attention and (iv) according to our knowledge no solutions put emphasis on modeling parallel execution possibilities in compound activity structures however exploiting parallel processing capabilities of modern platforms should be a primary concern.

Having taken into consideration the issues discussed above we identified the research goals enumerated in the middle column of Fig. 1.1. The first part of our approach will be the reduction of language complexity by distinguishing core modeling concepts and shorthand notations; we will directly define the semantics of core concepts and present a denotational semantics for shorthand notations by presenting their formal transformation to structures built up of core elements. Next we will formally define refinement-related concepts that are introduced only informally by the standard. In order to rule the complexity, a straightforward formalism is to be introduced that considers transitions as compound structures possibly involving multiple transitions and pseudostates. Explicit modeling compound activity structures and parallel execution also requires the introduction of a high-level formalism.

The rest of this chapter presents our efforts for meeting these research goals (Fig. 1.1 right column). The hierarchy of key concepts is illustrated in Fig. 1.2 by zooming the corresponding box of Fig. 1.1 the conceptual hierarchy is represented by vertical alignment, the section dedicated to discussing the corresponding subject is indicated in the top right corner of boxes (e.g., with respect to compound transition structures (in the middle of Fig. 1.2): we will first identify the possible roles of transitions and based on this investigation we will define transition conglomerate classes; the definition of transition...
conglomerate classes is directly used on one hand for defining notions of triggers and guards on the other hand for defining concepts like source and target states and container regions; finally the source states of transition conglomerates are used for distinguishing between transition conglomerates by their priority and defining the notion of conflicting transition conglomerates; the section dedicated for this discussion is Sec. 1.5.

This introduction (Sec. 1.1) has presented on overview on modeling facilities for event-driven state-based systems; we have introduced fundamental mathematical formalisms, discussed their engineering interpretations and the corresponding facilities provided by modern modeling languages; based on our analysis we have identified the issues to be solved and defined our research goals. Sec. 1.2 introduces the notation used in the document. In Sec. 1.3 we will identify core modeling facilities of UML 2.0 statecharts (states, regions, transitions etc.) and those complex artifacts that are used mainly as shorthand notations (e.g., the history mechanism, facilities for embedding entire state machines etc.). Our formalism for state and region refinement is presented in Sec. 1.4: we will present the formal foundations of concepts mentioned informally by the UML standard, discuss the well-formedness issues and introduce a formalism for straightforward handling of state hierarchies. In Sec. 1.5 we introduce a straightforward representation of compound activity structures that explicitly highlights possibilities for parallel execution based on PERT graphs. Having defined formalisms for state refinement, compound transition and activity structures we introduce these concepts into the metamodel in Sec. 1.7. The formal operational semantics is defined in Sec. 1.8 by a Kripke transition system (KTS); based on the KTS definition we will present the same operation by easy to implement imperative algorithms and outline the algorithmic complexity of our solution. Finally Sec. 1.9 concludes the discussion, highlights the new scientific achievements and outlines the direction of further research.

For simplicity reasons, the discussion in this chapter focuses on the behavior of a single object, however it is important to highlight our approach is by no means limited to a single object; in case of systems consisting of multiple objects, the behavior of these objects can be specified according to our semantics one by one.

1.2 Notation

This subsection presents the notation for referring to various types, instances and navigation related to UML statechart artifacts used throughout the document. The basis of the formalism is the UML 2.0 state machine metamodel [111, Chp. 15 State Machines] shown in Fig. 1.3.
A teletype font will be used for referring to metaclasses and a sans serif font for referring to sets of metaclass instances e.g., states are modeled by the State metaclass and all states in the models are in the State set etc. Straightforward short names will be used for referring to individual metaclass instances, e.g., State = \{s_0,s_1 \ldots s_{n-1}\}, etc. As the UML standard allows a behaviored classifier to own multiple state machines through the role ownedBehavior and does not identify the “top-level” one (i.e., the one that represents the top-level behavior possibly using the remaining ones as sub-state machines) the discussion will refer to the top-level (root) state machine instance as sm_0.

An OCL-like notation will be used for referring to various attributes of metaclass instances and for navigating through association relations, e.g., ps_i.kind, tj.source, s_k.region, etc. representing the kind attribute of the pseudostate ps_i, the single source vertex of the transition tj (the multiplicity of the role source is 1) and the set of regions owned by state s_k (the multiplicity of the role region is *) respectively. For simplicity reasons when navigating through an association with arbitrary multiplicity, if we know that in the particular case the multiplicity happens to be definitely one, we simply use the object.role notation for navigating to the single target object e.g., a transition can have zero or one trigger (Fig. 1.3) we can use the \( t_i.trigger \) notation for referring to the trigger of transition \( t_i \) if we are sure that \( t_i \) actually has a single trigger (this way the set of names of triggers of all transitions that have a trigger in the model can be expressed as \{\( t_i.trigger \mid i \in Transition \land |t_i.trigger| = 1 \}\).

Furthermore we introduce sets for specific kinds of pseudostates by indicating the abbreviation of the kind in superscript (In, DH, SH, Jo, Fo, Ju, Ch, En, Ex and Te representing initial, deep history, shallow history, join, fork, junction, choice, entry point, exit point and terminate pseudostates respectively), e.g., for initial pseudostates: Pseudostate\(^{in}\) = \{i ∈ Pseudostate \land i.kind = initial\}.

The inheritance relations of the metamodel appear as subset relations amongst sets, e.g., Pseudostate ⊆ Vertex, State ⊆ Vertex, FinalState ⊆ State, Pseudostate\(^{in}\) ⊆ Pseudostate.

Although this document assumes a basic knowledge about statecharts, the UML 2.0 standard and metamodeling, in order to enable fully unambiguous discussion and underpinning our modeling and design decisions, some key fragments of the UML standard are cited in appendix App. A.1 we will refer to these citations by their number e.g., as “see Cit. i about final states”.

1.3 Metamodel and Well-Formedness

The modeling toolkit of UML 2.0 statecharts contains both fundamental concepts like states and transitions and high abstraction level shorthand notations like history vertices, junction and choice pseudostates etc. Although this mixed toolkit provides a considerable degree of flexibility for the modeler but renders the discussion of operational semantics even more complicated. In order to rule the complexity we will follow a straightforward approach by identifying a core subset of statechart features that can be used for building the high-level constructs used as shorthand notations. Statecharts built up of this reduced toolkit will be called precise statecharts (PSC). Our strategy will be as follows: we will specify the metamodel and corresponding well-formedness rules for PSCs and directly define their formal operational semantics. Having defined the syntax and semantics of the core features, the shorthand constructs of UML statecharts that are missing from PSCs can be introduced as shorthand notations built up of multiple basic modeling elements (e.g., representing history vertices by multiple transitions with special guard predicates etc.). Being built on the well-established core toolkit, we can provide a denotational semantics for shorthand notations by defining their transformation to core constructs. Due to space restrictions the fully elaborated denotational semantics is not presented in this document but can be found in [132] and App. A.6 presents an overview on the transformation.

This section focuses on precise statecharts: (i) identifies core modeling constructs, (ii) presents the metamodel of PSCs and (iii) outlines the corresponding well-formedness rules.
1.3. CORE CONSTRUCTS AND SHORTHAND NOTATIONS

Based on the informal discussion of the standard the following modeling facilities can be considered to be “shorthand notations” i.e., ones that can be substituted by one or multiple fundamental elements: junction and choice pseudostates, all kinds of history vertices and facilities for embedding entire state machines (entry and exit point pseudostates and connection point references):

- **Junction and choice pseudostates** are used for connecting multiple transition segments resulting in a compound transition chain (Cit. 8, Cit. 9). These compound transition chains can be substituted by multiple plain transitions.

- **History vertices** represent the most recent shallow or deep configuration of the containing compound state (Cit. 4, Cit. 5). In case of non-concurrent container states the history vertex can be easily substituted by basic constructs: (i) we have to assign a **history variable** to the container state representing its most recent configuration and (ii) substitute the transition targeting the history vertex by multiple transitions targeting substates of the container; these transitions should be guarded by mutually exclusive predicates selecting the corresponding values of the history variable. In case of concurrent container states the substitution is similar, but the newly introduced transitions should target the (multiple) substates through fork vertices.

- **Embedding entire state machines** is a facility for re-using already existing state machine models (Cit. 10, Cit. 11). It is easy to see that the referenced machine can be merged into the containing one by a straightforward transformation.

1.3.2 METAMODEL OF PRECISE STATECHARTS

Based on the discussion above the metamodel of precise statecharts does not contain junction and choice pseudostates, history vertices and facilities for referring to states. This means that the metaclass **ConnectionPointReference** is removed and the **PseudostateKind** enumerated type is restricted to the initial, fork and join values. The weakly defined concepts of “do activities” and “deferred triggers” were also removed. The last difference between the original UML and the PSC metamodel is the representation of termination. The UML standard uses a pseudostate kind for representing termination of the state machine (Cit. 12). Since pseudostates are by definition “transient vertices” (Cit. 2) the representation of termination by a pseudostate seems to be confusing. In order to alleviate this inconsistency we introduce a new metaclass **TerminationState** derived from the metaclass **State** and substitute all “terminate pseudostates” with this “termination state” notation. The resulting metamodel of precise statecharts is shown in Fig. 1.4.
CHAPTER 1. A FORMAL OPERATIONAL SEMANTICS FOR UML 2.0 STATECHARTS

1.3.3 Well-Formedness of Precise Statecharts

Since as in any non-trivial language, a syntactically correct model can be still semantically meaningless, the UML standard aims at reducing the possibilities for constructing meaningless models by imposing various well-formedness constraints. Unfortunately this constraint set as specified by the standard is neither powerful enough nor complete. In order to overcome the weaknesses of the standard we have to introduce a better established constraint set by formally specifying all well-formedness rules that are needed for building a solid syntactic foundation for our operational semantics. Due to space restrictions the detailed formal discussion was moved to App. A.2 below we give an informal overview:

**Final states** A final state (i) cannot have any outgoing transitions, (ii) cannot have regions, (iii) cannot reference a submachine, (iv) has no entry behavior, and (v) has no exit behavior.

**Initial pseudostates** With respect to initial pseudostates we require that (i) all regions should have exactly one initial pseudostate subvertex, (ii) each initial pseudostate should have exactly one outgoing transition that (iii) targets a state (i.e., not a pseudostate) (iv) in the same region and (v) this transition has neither trigger (vi), nor guard condition or (vii) effect and (viii) an initial pseudostate may not be the target of any transitions.

**Join pseudostates** A join vertex (i) must have at least two incoming transitions, (ii) exactly one outgoing transition, (iii) all transitions targeting a join vertex should originate in different regions of an orthogonal state, (iv) the sources of these transitions are states, (v) these transitions may not have guards or (vi) triggers and (vii) finally we require that the single transition originating in the join vertex should target a fork pseudostate or a state.

**Fork pseudostates** A fork vertex (i) must have at least two outgoing transitions, (ii) exactly one incoming transition, (iii) all transitions originating in a fork vertex should target different regions of an orthogonal state, (iv) the targets of these transitions are states, (v) these transitions may not have guards or (vi) triggers and finally we require that (vii) the single transition targeting the fork vertex should originate in a join, junction or history pseudostate or a state.

**Termination states** As termination states are derived from final states the inherited well-formedness requirements apply seamlessly to terminate pseudostates and there is no need for introducing new ones (see the corresponding discussion in App. A.2 for more details).

**Regions** We introduced the well-formedness rules corresponding to regions in the context of other metaclasses, e.g., a region should have exactly one initial pseudostate subvertex (see the corresponding discussion in App. A.2 for more details).

**States** A state is not allowed to have both a submachine and regions.

**State machines** With respect to state machines we require that (i) connection points of a state machine are pseudostates of kind entry point or exit point and (ii) a state machine should have exactly a single region.

**Transitions** Most of well-formedness rules corresponding to transitions were defined in the context of other metaclasses, the only aspect left open corresponds to the kind of transitions. The representation of internal transitions has been modified since the previous version of the standard (1.5) and the notion of local transitions was introduced (Cit. 13). Kind internal implies that the transition, if fired, occurs without exiting or entering the source state i.e., it does not cause a state change. An internal transition can be taken even if the state machine is in one or more regions nested within that state. Kind local implies that the transition, if fired, will not exit the composite (source) state, but it will apply to any state within the composite state, and these will be exited and entered. Kind external implies that the transition, if fired, will exit the composite
(source) state. (External transitions are the most common ones, the other two kinds can be considered as rare cases, possibly not even supported by modeling tools.) We will require that internal and local transitions are always loops i.e., their source and target are the same.

### 1.4 Refinement Formalism

The UML standard discusses state refinement and state containment in a rather informal way, many concepts are used primarily relying on the natural intuition of the reader; although some concepts are underpinned by OCL expressions, the consistent formalization is missing. The section introduces the rigorous formal definitions of concepts, discusses their relations and introduces the notion of state hierarchies and corresponding metaclasses.

#### 1.4.1 Refinement Concepts

Unfortunately not even the state concept is clear in UML: the top state machine is a state-like concept, since it can own any number of regions, but sub-state machines should not be considered as individual state-like concepts since sub-state machines are embedded into a parent state machine by being referred from a state through the role submachine. Taking into consideration this complexity a state can be refined to regions in three ways: (i) instances of the State metaclass can contain regions through the role region, (ii) the top state machine can also contain regions through the role region and (iii), the submachine states refer to state machines that can also contain regions (Fig. 1.5 left side, blue lines). An abstract view of state refinement is introduced in Def. 1

**Definition 1 (Direct Refinement).** A state or the root state machine $s$ is directly refined to region $r$ if (i) $s$ is a state and $r$ is in the $s$-region set or (ii) $s$ is the root state machine $sm_0$ and $r$ is in the $s$-region set or (iii) $s$ is a submachine state that has a submachine $s$-submachine that $r$ is in the $s$-submachine-region set. Below we introduce the function $IsRef^D$ that for an $(s, r)$ pair returns $\top$ if $s$ is directly refined to $r$ (prototype: Eq. 1.1 def.: Eq. 1.4). The relation $\text{Ref}^D$ contains those $(s, r)$ pairs, where $s$ is directly refined to $r$ (prototype: Eq. 1.2 def.: Eq. 1.5). The function $\text{AllRef}^D$ returns all regions refining the state or top state machine $s$ (prototype: Eq. 1.3 def.: Eq. 1.6).

\[
\begin{align*}
\text{IsRef}^D : (\text{State} \cup \{sm_0\}) \times \text{Region} & \to \{\top, \bot\} \quad (1.1) \\
\text{Ref}^D & \subseteq (\text{State} \cup \{sm_0\}) \times \text{Region} \quad (1.2) \\
\text{AllRef}^D : (\text{State} \cup \{sm_0\}) & \to 2^\text{Region} \quad (1.3) \\
\text{IsRef}^D(s, r) &= \begin{cases} 
\top & \text{if } s \in \text{State} \land r \in s\text{-region} \\
\top & \text{if } s = sm_0 \land r \in s\text{-region} \\
\top & \text{if } s \in \text{State} \land \exists sm \in s\text{-submachine : } r \in sm\text{-region} \\
\bot & \text{otherwise} 
\end{cases} \quad (1.4) \\
\text{Ref}^D &= \{(s, r) | s \in (\text{State} \cup \{sm_0\}) \land r \in \text{Region} \land \text{IsRef}^D(s, r)\} \quad (1.5) \\
\text{AllRef}^D(s \in (\text{State} \cup \{sm_0\})) &= \{r | r \in \text{Region} \land \text{IsRef}^D(s, r)\} \quad (1.6)
\end{align*}
\]
A region can contain any number of states through the subvertex role but we have to select the instances of the State metaclass since a subvertex can be a pseudostate or a connection point reference also (Fig. 1.5) left side, orange lines. The notion of direct state containment is introduced by Def. 2.

Definition 2 (Direct Containment). A region \( r \) is said to directly contain a state \( s \) if \( s \) is in the set \( r.\text{subvertex} \). Below we introduce the function \( \text{IsCont}^D \) that for an \( (r, s) \) pair returns \( \top \) if \( s \) is directly contained by \( r \) (prototype: Eq. 1.7 def.: Eq. 1.10). The relation \( \text{Cont}^D \) contains those \( (r, s) \) pairs, where \( s \) is directly contained by \( r \) (prototype: Eq. 1.8 def.: Eq. 1.11). The function \( \text{AllCont}^D \) when called for a region \( r \) returns all the states directly contained by it (prototype: Eq. 1.9 def.: Eq. 1.12).

\[
\begin{align*}
\text{IsCont}^D : & \text{Region} \times \text{State} \to \{\top, \bot\} \\
\text{Cont}^D : & \subseteq \text{Region} \times \text{State} \\
\text{AllCont}^D : & \text{Region} \to 2^{\text{State}} \\
\text{IsCont}^D (r, s) : & \begin{cases} 
\top & \text{if } s \in r.\text{subvertex} \\
\bot & \text{otherwise}
\end{cases} \\
\text{Cont}^D : & \{ (r, s) | r \in \text{Region} \land s \in \text{State} \land \text{IsCont}^D (r, s) \} \\
\text{AllCont}^D (r) : & \{ s | s \in \text{State} \land \text{IsCont}^D (r, s) \}
\end{align*}
\]

Below we introduce the notions related to substate and subregion relations. These derived concepts are illustrated in the middle and right side of Fig. 1.5.

A state \( s_c \) can be considered to be the direct substate of state or root state machine \( s_p \) if the region containing \( s_c \) is a direct refinement of \( s_p \). This concept is formally introduced by Def. 3. The related metaclasses and associations are highlighted by blue lines in the right side of Fig. 1.5.

Definition 3 (Direct Substate). A state \( s_c \) (child state) is said to be a direct substate of state or root state machine \( s_p \) (parent state) if the region containing \( s_c \) is a direct refinement of \( s_p \). Below we introduce the function \( \text{IsSubst}^D \) that for an \( (s_p, s_c) \) pair returns \( \top \) if \( s_c \) is a direct substate of \( s_p \) (prototype: Eq. 1.13 def.: Eq. 1.16). The relation \( \text{Subst}^D \) contains those \( (s_p, s_c) \) pairs, where \( s_c \) is a direct substate of \( s_p \) (prototype: Eq. 1.14 def.: Eq. 1.17). The function \( \text{AllSubst}^D \) when called for a state or root state machine \( s_p \) returns all of its direct substates (prototype: Eq. 1.15 def.: Eq. 1.18).

\[
\begin{align*}
\text{IsSubst}^D : & (\text{State} \cup \{\text{sm}_0\}) \times \text{State} \to \{\top, \bot\} \\
\text{Subst}^D : & \subseteq (\text{State} \cup \{\text{sm}_0\}) \times \text{State} \\
\text{AllSubst}^D : & (\text{State} \cup \{\text{sm}_0\}) \to 2^{\text{State}} \\
\text{IsSubst}^D (s_p, s_c) : & \begin{cases} 
\top & \text{if } \exists r \in \text{AllRef}^D (s_p) : \text{IsCont}^D (r, s_c) \\
\bot & \text{otherwise}
\end{cases} \\
\text{Subst}^D : & \{ (s_p, s_c) | s_p \in (\text{State} \cup \{\text{sm}_0\}) \land s_c \in \text{State} \land \text{IsSubst}^D (s_p, s_c) \} \\
\text{AllSubst}^D (s_p) : & \{ s_c | s_c \in \text{State} \land \text{IsSubst}^D (s_p, s_c) \}
\end{align*}
\]

A region \( r_c \) can be considered to be the direct subregion of the region \( r_p \) if \( r_p \) directly contains a state \( s \) that is directly refined to \( r_c \). This concept is formally introduced by Def. 4. The related metaclasses and associations are highlighted by orange lines in the middle of Fig. 1.5.

Definition 4 (Direct Subregion). A region \( r_c \) (child region) is said to be the direct subregion of the region \( r_p \) (parent region) if \( r_p \) directly contains a state \( s \) that is directly refined to \( r_c \). Below we introduce the function \( \text{IsSubreg}^D \) that for an \( (r_p, r_c) \) pair returns \( \top \) if \( r_c \) is a direct subregion of \( r_p \) (prototype: Eq. 1.19 def.: Eq. 1.22). The relation \( \text{Subreg}^D \) contains those \( (r_p, r_c) \) pairs, where \( r_c \) is a direct subregion of \( r_p \) (prototype: Eq. 1.20 def.: Eq. 1.23). The function \( \text{AllSubreg}^D \) when called for a region \( r_p \) returns all of its direct subregions (prototype: Eq. 1.21 def.: Eq. 1.24).

\[
\begin{align*}
\text{IsSubreg}^D : & (\text{Region} \times \text{Region}) \to \{\top, \bot\} \\
\text{Subreg}^D : & \subseteq (\text{Region} \times \text{Region}) \times \text{Region} \\
\text{AllSubreg}^D : & \text{Region} \to 2^{\text{Region}} \\
\text{IsSubreg}^D (r_p, r_c) : & \begin{cases} 
\top & \text{if } \exists s \in \text{IsCont}^D (r_p) : \text{IsSubst}^D (s, r_c) \\
\bot & \text{otherwise}
\end{cases} \\
\text{Subreg}^D : & \{ (r_p, r_c) | r_p \in \text{Region} \land r_c \in \text{Region} \land \text{IsSubreg}^D (r_p, r_c) \} \\
\text{AllSubreg}^D (r_p) : & \{ r_c | r_c \in \text{Region} \land \text{IsSubreg}^D (r_p, r_c) \}
\end{align*}
\]
Deﬁnition 6 (Transitive Containment).

The recursion in the deﬁnition above always terminates.) Below we introduce the function

Deﬁnition 5 (Transitive Reﬁnement).

The function

Example 1. In Fig. 1.6 the root state machine sm0 contains a submachine state s8 that refers to the state machine sm1; the sets deﬁned above are as follows:

\[
\begin{align*}
\text{Ref}^D &= \{(s_2, r_2), (s_2, r_3), (s_2, r_4), (s_5, r_5), (s_8, r_7), (\text{sm}_0, r_1)\} \\
\text{Cont}^D &= \{(r_1, s_1), (r_1, s_2), (r_1, s_8), (r_2, f_1), (r_2, s_1), (r_3, f_2), (r_3, s_4), (r_4, s_5), (r_5, s_6), (r_6, r_7), (r_7, f_3), (r_7, s_9)\} \\
\text{Subst}^D &= \{(s_2, f_1), (s_2, f_2), (s_2, s_3), (s_2, s_4), (s_2, s_5), (s_5, s_6), (s_8, s_7), (s_8, s_9), (\text{sm}_0, s_1), (\text{sm}_0, s_2), \\
& (\text{sm}_0, s_8)\}
\end{align*}
\]

\[
\begin{align*}
\text{Subreg}^D &= \{(r_1, r_2), (r_1, r_3), (r_1, r_4), (r_1, r_7), (r_4, r_5), (r_4, r_6)\}
\end{align*}
\]

Based on the direct reﬁnement, containment, substate and subregion concepts we introduce below the notion of transitive reﬁnements, containments, substates and subregions.

Deﬁnition 5 (Transitive Reﬁnement). A region r is said to be the transitive reﬁnement of the state s if (i) r is a direct reﬁnement of s or (ii) there is an intermediate region r_1 directly reﬁning s that contains a state s_1 so that r is a transitive reﬁnement of s_1. (Since the reﬁnement hierarchy is ﬁnite the recursion in the deﬁnition above always terminates.) Below we introduce the function IsRef\(^T\) that for an (s, r) pair returns ∨ if s is transitively reﬁned to r (prototype: Eq. 1.30, deﬁn.: Eq. 1.31). The relation Ref\(^T\) contains those (s, r) pairs, where s is transitively reﬁned to r (prototype: Eq. 1.30, deﬁn.: Eq. 1.33). The function AllRef\(^T\) when called for a state or top state machine s returns all the regions transitively reﬁning it (prototype: Eq. 1.31, deﬁn.: Eq. 1.34).

\[
\begin{align*}
\text{IsRef}^T : (\text{State} \cup \{\text{sm}_0\}) \times \text{Region} &\to \{\top, \bot\} \\
\text{Ref}^T &\subseteq (\text{State} \cup \{\text{sm}_0\}) \times \text{Region} \\
\text{AllRef}^T : (\text{State} \cup \{\text{sm}_0\}) &\to \text{Region} \\
\text{IsRef}^T(s, r) &= \begin{cases} 
\top & \text{if } \text{IsRef}^D(s, r) \text{ or } \exists r_1 \in \text{AllRef}^D(s) : \exists s_1 \in \text{AllCont}^D(r_1) : \text{IsRef}^T(s_1, r) \\
\bot & \text{otherwise} \end{cases} \\
\text{Ref}^T &= \{(s, r) | s \in (\text{State} \cup \{\text{sm}_0\}) \land r \in \text{Region} \land \text{IsRef}^T(s, r)\} \\
\text{AllRef}^T(s) &= \{r | r \in \text{Region} \land \text{IsRef}^T(s, r)\}
\end{align*}
\]

Deﬁnition 6 (Transitive Containment). A state s is said to be transitively contained by a region r if (i) r directly contains s or (ii) there is an intermediate state s_1 contained by r that is reﬁned to a region r_1 so that s is transitively contained by r_1. (Since the reﬁnement hierarchy is ﬁnite the recursion in the deﬁnition above always terminates.) Below we introduce the function IsCont\(^T\) that for an (r, s) pair returns ∨ if s is transitively contained by r (prototype: Eq. 1.33, deﬁn.: Eq. 1.35). The relation Cont\(^T\) contains those (r, s) pairs, where s is transitively contained by r (prototype: Eq. 1.35, deﬁn.: Eq. 1.36). The function AllCont\(^T\) when called for a region r returns all the states transitively contained by it (prototype: Eq. 1.37, deﬁn.: Eq. 1.39).

\[
\begin{align*}
\text{IsCont}^T : \text{Region} \times \text{State} &\to \{\top, \bot\} \\
\text{Cont}^T &\subseteq \text{Region} \times \text{State} \\
\text{AllCont}^T : \text{Region} &\to \text{State} \\
\text{IsCont}^T(r, s) &= \begin{cases} 
\top & \text{if } \exists s_1 \in \text{AllCont}^D(r) : \exists r_1 \in \text{IsCont}^D(s_1, r) \\
\bot & \text{otherwise} \end{cases} \\
\text{Cont}^T &= \{(r, s) | r \in \text{Region} \land \exists s \in \text{State} \land \text{IsCont}^T(r, s)\} \\
\text{AllCont}^T(r) &= \{s | s \in \text{State} \land \text{IsCont}^T(r, s)\}
\end{align*}
\]
Definition 7 (Transitive Substate). A state $s_d$ (descendant state) is said to be a transitive substate of the state or root state machine $s_a$ (ancestor state) if (i) $s_d$ is a direct substate of $s_a$ or (ii) there is an intermediate region $r_i$ directly refining $s_a$ and $r_i$ directly contains a state $s_i$ so that $s_d$ is a transitive substate of $s_i$. (Since the refinement hierarchy is finite the recursion in the definition above always terminates.) Below we introduce the function $\text{IsSubst}^T$ that for an $(s_a, s_d)$ pair returns $T$ if $s_d$ is a transitive substate of $s_a$ (prototype: Eq. 1.41 def.: Eq. 1.44). The relation $\text{Subst}^T$ contains those $(s_a, s_d)$ pairs, where $s_d$ is a transitive substate of $s_a$ (prototype: Eq. 1.42 def.: Eq. 1.45). The function $\text{AllSubst}^T$ when called for a state or root state machine $s_a$ returns its all transitive substates (prototype: Eq. 1.43 def.: Eq. 1.46).

\[
\text{IsSubst}^T : (\text{State}(\cup \{s_m\}) \times \text{State}) \rightarrow \{T, \bot\} \\
\text{Subst}^T \subseteq (\text{State}(\cup \{s_m\}) \times \text{State}) \\
\text{AllSubst}^T : (\text{State}(\cup \{s_m\}) \rightarrow 2^{\text{State}}) \\
\]

\[
\text{IsSubst}^T(s_a, s_d) = \begin{cases} 
T & \text{if } \text{IsSubst}^D(s_a, s_d) \text{ or } \exists s_i \in \text{AllCont}^D(r_i) : \exists r_i \in \text{AllRef}^D(s_i) : \text{IsSubst}^T(s_i, s_d) \\
\bot & \text{otherwise} 
\end{cases} \\
\]

\[
\text{Subst}^T = \{(s_a, s_d) | s_a \in (\text{State}(\cup \{s_m\}) \cap s_d \in \text{State} \land \text{IsSubst}^T(s_a, s_d)\} \\
\text{AllSubst}^T(s_a \in (\text{State}(\cup \{s_m\}))) = \{s_d | s_d \in \text{State} \land \text{IsSubst}^T(s_a, s_d)\} \\
\]

Definition 8 (Transitive Subregion). A region $r_d$ is said to be a transitive subregion of $r_a$ if (i) $r_d$ is a direct subregion of $r_d$ or (ii) there is an intermediate state $s_i$ directly contained by $r_a$ that is directly refined to a region $r_i$ so that $r_d$ is a transitive subregion of $r_i$. (Since the refinement hierarchy is finite the recursion in the definition above always terminates.) Below we introduce the function $\text{IsSubreg}^T$ that for an $(r_a, r_d)$ pair returns $T$ if $r_d$ is a transitive subregion of $r_a$ (prototype: Eq. 1.47 def.: Eq. 1.50). The relation $\text{Subreg}^T$ contains those $(r_a, r_d)$ pairs, where $r_d$ is a transitive subregion of $r_a$ (prototype: Eq. 1.48 def.: Eq. 1.51). The function $\text{AllSubreg}^T$ when called for a region $r_a$ returns its all transitive subregions (prototype: Eq. 1.49 def.: Eq. 1.52).

\[
\text{IsSubreg}^T : \text{Region} \times \text{Region} \rightarrow \{T, \bot\} \\
\text{Subreg}^T \subseteq \text{Region} \times \text{Region} \\
\text{AllSubreg}^T : \text{Region} \rightarrow 2^{\text{Region}} \\
\]

\[
\text{IsSubreg}^T(r_a, r_d) = \begin{cases} 
T & \text{if } \text{IsSubreg}^D(r_a, r_d) \\
\bot & \text{otherwise} 
\end{cases} \\
\text{Subreg}^T = \{(r_a, r_d) | r_a \in \text{Region} \land \text{IsSubreg}^T(r_a, r_d)\} \\
\text{AllSubreg}^T(r_a \in \text{Region}) = \{r_d | r_d \in \text{Region} \land \text{IsSubreg}^T(r_a, r_d)\} \\
\]

Example 2. Sets $\text{Ref}^T$, $\text{Cont}^T$, $\text{Subst}^T$ and $\text{Subreg}^T$ in case of Fig. 1.6 are as follows:

\[
\text{Ref}^T = \{(s_2, r_2), (s_2, r_3), (s_2, r_4), (s_2, r_5), (s_2, r_6), (s_5, r_5), (s_5, r_6), (s_8, r_7), (s_m, r_1), (s_m, r_2), (s_m, r_3), (s_m, r_4), (s_m, r_5), (s_m, r_6), (s_m, r_7)\} \\
\]

(1.53)
are formalized and proven by Thm. 4, Thm. 5 and Thm. 6. in App. A.5.

Three important observations can be proven about the refinement hierarchy: (i) if regions \( r_1 \) and \( r_2 \) are not in a refinement relation, the set of their substates is disjoint and (iii) if states \( s_1 \) and \( s_2 \) are not in a refinement relation, the state hierarchies rooting in \( s_1 \) and \( s_2 \) are disjoint. These statements are formalized and proven by Thm. 4 Thm. 5 and Thm. 6 in App. A.5.

**Def. 9 (Concurrent State Pair).** An \((s_1, s_2)\) pair of states is said to be concurrent if there is a state or root state machine \( s \) that is refined to (at least) two regions \( r_1 \) and \( r_2 \) is directly or transitively contained by \( r_1 \) and \( s_2 \) is directly or transitively contained by \( r_2 \). Below we introduce the function \( \text{IsCnrtPair} \) that for an \((s_1, s_2)\) pair of states returns \( \top \) if \( s_1 \) and \( s_2 \) are a concurrent pair (prototype: Eq. 1.57 def.: Eq. 1.58). The relation \( \text{CnrtPair} \) contains all \((s_1, s_2)\) concurrent state pairs, where \( s_1 \neq s_2 \) (prototype: Eq. 1.58 def.: Eq. 1.61). The function \( \text{AllCnrtPair} \) for a state \( s_1 \) returns all \( s_2 \) states that \((s_1, s_2)\) are concurrent pairs (prototype: Eq. 1.59 def.: Eq. 1.62).

\[
\text{IsCnrtPair} : \text{State} \times \text{State} \to \{ \top, \bot \} \\
\text{CnrtPair} \subseteq \text{State} \times \text{State} \\
\text{AllCnrtPair} : \text{State} \to \text{State}^2
\]

\[
\text{IsCnrtPair}(s_1, s_2) = \begin{cases} 
\top & \text{if } \exists s \in (\text{State} \cup \{\text{sm}_0\}) : \exists \{r_1, r_2\} \subseteq \text{AllRef}^D(s) : \\
\bot & \text{otherwise}
\end{cases}
\]

\[\text{AllCnrtPair}(s_1) = \{s_2 | s_2 \in \text{State} \land \text{IsCnrtPair}(s_1, s_2)\} \tag{1.61}\]

**Example 3.** The set of concurrent state pairs in case of Fig. 1.6 are as follows:

\[
\text{CnrtPair} = \{(f_1, f_2), (f_1, s_4), (f_1, s_6), (f_1, s_7), (f_2, f_1), (f_2, s_3), (f_2, s_5), (f_2, s_6), (f_2, s_7), (s_3, f_2), (s_3, s_4), (s_3, s_5), (s_3, s_6), (s_3, s_7), (s_4, f_1), (s_4, s_3), (s_4, s_5), (s_4, s_6), (s_4, s_7), (s_5, f_1), (s_5, f_2), (s_5, s_3), (s_5, s_4), (s_6, f_1), (s_6, f_2), (s_6, s_3), (s_6, s_4), (s_6, s_6), (s_7, f_1), (s_7, f_2), (s_7, s_3), (s_7, s_4), (s_7, s_6)\} \tag{1.63}
\]

**1.4.2 Well-Formedness of Refinement**

Some refinement-related constraints can only be introduced with respect to the higher level refinement formalism introduced above: (i) any state \( s \) is contained by exactly one region (Eq. 1.64), (ii) any region \( r \) refines exactly one state (or the root state machine) (Eq. 1.65) and (iii) the refinement graph is acyclic (Eq. 1.66).

\[
\forall s \in \text{State} : |\{r \in \text{Region} | (r, s) \in \text{Cont}^D\}| = 1 \tag{1.64}
\]

\[
\forall r \in \text{Region} : |\{s \in \text{State} \cup \{\text{sm}_0\} | (s, r) \in \text{Ref}^D\}| = 1 \tag{1.65}
\]

\[\forall r_1, r_2, \ldots, r_n, s_1, s_2, \ldots, s_n : \{r_1, s_1\}, \{r_2, s_2\}, \ldots, \{r_n, s_n\} \subseteq \text{Cont}^D \land \{s_1, r_2\}, \{s_2, r_3\}, \ldots, \{s_{n-1}, r_n\}, \{s_n, r_1\} \subseteq \text{Ref}^D \tag{1.66}\]

**1.4.3 Relations of Refinement Concepts**

Three important observations can be proven about the refinement hierarchy: (i) if regions \( r_1 \) and \( r_2 \) are not in a refinement relation the set of states contained by them is disjoint; (ii) if states \( s_1 \) and \( s_2 \) are not in a refinement relation, the set of their substates is disjoint and (iii) if states \( s_1 \) and \( s_2 \) are not in a refinement relation, the state hierarchies rooting in \( s_1 \) and \( s_2 \) are disjoint. These statements are formalized and proven by Thm. 4 Thm. 5 and Thm. 6 in App. A.5.
1.4.4 State Hierarchies

When reasoning about execution semantics of transitions we have to talk about state hierarchies. We introduce here a simple formalism for expressing state hierarchies in a straightforward way.

A state hierarchy node represents the top state of a state hierarchy and implemented as a metaclass named StateHierarchyNode (Fig. 1.7 left side). A state is associated to each state hierarchy nodes indicating the top of the hierarchy represented by the state hierarchy node (role state). A state hierarchy node may contain any number of sub-state hierarchy nodes (role substates).

Example 4. The object diagram in the callout of Fig. 1.7 represents the state hierarchy entered by the transition in the statechart fragment in the right side of the figure (states a, c, d and h).

Definition 10 (Building a State Hierarchy). Below we present the building a state hierarchy function (prototype: Eq. 1.67, def.: Eq. 1.68) that for a state \( s \) returns a state hierarchy node representing the entire state hierarchy rooting in \( s \) (i.e., \( s \) itself as the top state and all of its substates according to the refinement hierarchy). The arguments given to the StateHierarchyNode constructor in the pseudocode are the objects of state and substates roles respectively.

\[
\text{BuildHierarchy} : \text{State} \rightarrow \text{StateHierarchyNode}
\]

\[
\text{BuildHierarchy}(s \in \text{State}) = \text{StateHierarchyNode}(s, \{\text{BuildHierarchy}(s_c) | s_c \in \text{AllSubst}^D(s)\})
\]

1.5 Compound Transition Structures

A transition is defined by the UML standard (15.3.14) as “a directed relationship between a source vertex and a target vertex. It may be part of a compound transition, which takes the state machine from one state configuration to another, representing the complete response of the state machine to an occurrence of an event of a particular type” (Cit. 13).

Based on this definition the meaning of an instance of the Transition metaclass can not be considered in isolation, since transitions are used for (i) indicating the initial substate of a region (single transition originating in an initial pseudostate), (ii) default substates to be entered in case of various history vertices, (iii) can be used as stand-alone transitions connecting two states or (iv) parts of compound transition paths possibly involving various pseudostates (fork, join, etc.).

Multiple weak points of the standard originate in the fact that UML does not explicitly introduce the concept of compound transitions and use the notion of “transition” even in cases where actually no state transition may occur (e.g., indicating initial states by a transition originating in an initial pseudostate, indicating the default history states by transitions originating in history vertices etc.).

For understanding and specifying the operational semantics of UML statecharts we are definitely in need of better separation of concerns. We will address the issue as follows: first we will discuss the possible application roles of transitions in UML statecharts, then introduce and formally define the concepts of various transition conglomerate classes. The rest of the section defines various concepts related to transition conglomerate classes: we will first define the notions of trigger events and guard predicates in the context of transition conglomerates, then we will introduce concepts related to the source and target states and containing regions, finally we will define priority relations and possible conflicts amongst transition conglomerates.
1.5. COMPOUND TRANSITION STRUCTURES

1.5.1 Possible Roles of Transitions

The UML standard tends to follow the strategy of only defining forbidden relations of metaclass instances (e.g., specifying which vertices may not be sources of transitions etc.) allowing the modeler to build anything that is not explicitly illegal. This loosely defined structure poses a definitely hard problem when having to define a precise formal semantics, since we have to take into consideration all possible ways for connecting vertices by transitions. In order to prevent our discussion from omitting any possible transition application contexts, we will follow a systematic approach: in Fig. 1.8 we collected all possible ways for connecting vertices of a precise statechart by a transition. Below we will investigate all possible cases and discuss whether that specific case is valid or not and if it is valid we will assign it to one of a few number of transition conglomerate classes. A transition conglomerate class notion introduced by us is a formal representation of the informally defined UML compound transition path concept. A transition conglomerate class can be seen as a usage pattern of one or multiple Transition instances possibly connected by some pseudostates. The topmost aspect for classifying transitions is the kind of the transition. Correspondingly the table in Fig. 1.8 contains three columns for local, internal and external transitions respectively.

Invalid Source–Target Pairs

The largest part of the table in Fig. 1.8 corresponds to external transitions and contains a sub-table itself. The internal table contains lines and columns for the specific source and target vertex types (in a precise statechart pseudostates of kind initial, join and fork can be used and states). Although final states (and termination states derived from final states) are states themselves, since their application requires special care they are investigated in separation from states, thus the state symbol in the figure represents the states that are not final states.

Several source–target vertex pairs are forbidden by one or more well-formedness rules (discussed in-depth in App. A.2). These cases are indicated by a red × symbol in the top left corner of the corresponding table cell. The number in parenthesis in the bottom left corner of the cell indicates one of the reasons why that specific construct is invalid (note that a construct may violate multiple well-formedness rules, below we mention only one corresponding to each numbers): (1) – an initial pseudostate may not be the target of any transitions (Eq. A.3); (2) – the single transition originating in an initial pseudostate should target a state (Eq. A.3); (3) – the single transition originating in a join vertex should target a fork pseudostate or a state (Eq. A.7); (4) – the targets of transitions originating in a fork pseudostate should be states (Eq. A.10) and (5) – final states cannot have any outgoing transitions (Eq. A.1). The rest of cases are meaningful constructs, below we will discuss their semantics.
Internal and Local Transitions

Local and internal transitions are always loops i.e., their source and target state is the same (Eq. A.15). We assign local and internal transitions to the transition conglomerate classes $A$ and $B$ respectively (the class is indicated in the figure on the left hand side under the corresponding symbol in Fig. 1.8 in braces, e.g., [a], [b], etc.).

Initial Transitions

A special application of transitions is to identify the initial state of a region. The initial state is targeted by the single transition originating in the initial pseudostate of the region. This kind of constructs are identified by [i] in Fig. 1.8. Below we define three functions for (i) obtaining the initial state of a region indicated by the initial pseudostate and a transition, for (ii) constructing the entire initial state hierarchy of a region and for (iii) constructing the initial state hierarchy of the entire state machine.

**Definition 11 (Initial States and State Hierarchies).** The initial state of a region function (prototype: Eq. 1.69, def.: Eq. 1.71) for an $r \in \text{Region}$ region returns the initial state of $r$. It can be shown that for any $r \in \text{Region}$ the function returns a single initial state: for any region $r$ the corresponding well-formedness rules ensure that (i) there is exactly one initial pseudostate subvertex in $r$ (Eq. A.2), (ii) $i$ has a single outgoing transition (Eq. A.3), (iii) that targets a state $s$ (Eq. A.3); since $s$ is in the same container region as $i$ (Eq. A.3) based on the definition of $\text{AllCont}_D$ (Def. 2) it is obvious that $\forall r \in \text{Region} : |\text{InitialState}^R(r)| = 1$. The initial state hierarchy of a region function (prototype: Eq. 1.70, def.: Eq. 1.72) for an $r \in \text{Region}$ region returns the initial state hierarchy of $r$. The function returns a state hierarchy node whose top state is the initial state of $r$ and its substate hierarchy nodes are the ones returned by recursive calls to regions directly refining the initial state of $r$. The initial state hierarchy of the state machine function (def.: Eq. 1.73) returns the initial state hierarchy of the entire state machine. (The function returns the initial state hierarchy of the single region (see the well-formedness rule Eq. A.14) refining the root state machine $sm_0$.)

\[
\text{InitialState}^R : \text{Region} \rightarrow \text{State} \quad (1.69) \\
\text{InitialHierarchy}^R : \text{Region} \rightarrow \text{StateHierarchyNode} \quad (1.70) \\
\text{InitialState}^R(r \in \text{Region}) = \{ s \in \text{AllCont}_D^D(r) \exists i \in \text{Pseudostate}^{lb} : i \in r.\text{subvertex} \land s = i.\text{outgoing.target} \} \quad (1.71) \\
\text{InitialHierarchy}^R(r) = \text{StateHierarchyNode}(\text{InitialState}^R(r)) \quad (1.72) \\
\text{InitialHierarchy}() = \text{InitialHierarchy}^R(sm_0.\text{region}) \quad (1.73)
\]

Compound Transition Patterns

The remaining source – target pairs in Fig. 1.8 (i.e., the external ones that are not invalid, nor indicating an initial state) can be grouped into four categories:

**Simple transitions** are ones that directly connect states (note that the source and the target state may be the same even in case of external transitions). These constructs are identified by a [c] in Fig. 1.8 (corresponding to the transition conglomerate class C defined below).

**Compound transition paths with a join vertex** are the ones that consist of a join pseudostate, two or more transitions originating in orthogonal states and targeting the join vertex and a single transition originating in the join pseudostate. These constructs are identified by a [d] in Fig. 1.8 (corresponding to the transition conglomerate class D defined below).

**Compound transition paths with a fork vertex** are the ones that consist of a fork pseudostate, a single transition targeting the fork vertex and two or more transitions targeting orthogonal
1.5. COMPOUND TRANSITION STRUCTURES

states and originating in the fork pseudostate. These constructs are identified by a [e] in Fig. 1.8 (corresponding to the transition conglomerate class E defined below).

Compound transition paths with join and fork vertex are the most complicated constructs involving a join vertex j, a fork vertex f, a transition originating in j and targeting f, two or more transitions originating in orthogonal states and targeting j and two or more transitions originating in f and targeting orthogonal states. These constructs are identified by a [f] in Fig. 1.8 (corresponding to the transition conglomerate class F defined below).

1.5.2 Transition Conglomerate Classes

Above we have identified six application cases (marked [a] – [f] in Fig. 1.8) where transitions actually indicate state transition (i.e., not the notation for initial state of a region marked [i] in Fig. 1.8). These cases are collected and symbolically depicted in Fig. 1.9. Below we introduce formal constructs called transition conglomerate classes corresponding to the six application cases.

Definition 12 (Transition Conglomerate Class A). The \( \mathcal{T}_a \) set of local transition conglomerates (\( \mathcal{T}_a \subseteq \text{Transition} \times (\text{State} \setminus \text{FinalState}) \)) contains those \((t, s)\) pairs, where (i) \( t \) is a transition (ii) of kind \( \text{local} \), (iii) the source and the target of \( t \) are the same and (iv) the source (and target) of the transition is a state (but not a final state); the source (and target) state of \( t \) is \( s \) (Eq. 1.74).

Definition 13 (Transition Conglomerate Class B). The \( \mathcal{T}_b \) set of internal transition conglomerates (\( \mathcal{T}_b \subseteq \text{Transition} \times (\text{State} \setminus \text{FinalState}) \)) contains those \((t, s)\) pairs, where (i) \( t \) is a transition (ii) of kind \( \text{internal} \), (iii) the source and the target of \( t \) are the same and (iv) the source (and target) of the transition is a state (but not a final state); the source (and target) state of \( t \) is \( s \) (Eq. 1.75).

Definition 14 (Transition Conglomerate Class C). The \( \mathcal{T}_c \) set of simple transition conglomerates (\( \mathcal{T}_c \subseteq (\text{State} \setminus \text{FinalState}) \times \text{Transition} \times \text{State} \)) contains those \((s_{\text{src}}, t_{\text{in}}, j, t, s_{\text{trg}})\) tuples where (i) \( t \) is a transition (ii) of kind \( \text{external} \), (iii) the source of \( t \) is a state but not a final state and (iv) the target of \( t \) is a state (possibly a final state); the source state of \( t \) is \( s_{\text{src}} \), the target state of \( t \) is \( s_{\text{trg}} \) (Eq. 1.76).

Definition 15 (Transition Conglomerate Class D). The \( \mathcal{T}_d \) set of external transition conglomerates with a single join vertex (\( \mathcal{T}_d \subseteq 2^{(\text{State} \setminus \text{FinalState})} \times 2^{\text{Transition}} \times \text{Pseudostate}^{\text{lo}} \times \text{Transition} \times \text{State} \)) contains those \((S_{\text{src}}, T_{\text{in}}, j, t, s_{\text{trg}})\) tuples where (i) \( t \) is a transition (ii) of kind \( \text{external} \), (iii) the source of \( t \) is a join pseudostate, (iv) the target of \( t \) is a state, for all \( t_{\text{in}} \) transitions targeting the source of \( t \) (the join vertex) holds that (v) \( t_{\text{in}} \) has no guard, (vi) \( t_{\text{in}} \) has no trigger, (vii) \( t_{\text{in}} \) is of kind \( \text{external} \), (viii) the source of \( t_{\text{in}} \) is a state but not a final state and (ix) for each \((t_{\text{in}1}, t_{\text{in}2})\) pairs of transitions targeting the source of \( t \) holds that the sources of \( t_{\text{in}1} \) and \( t_{\text{in}2} \) are concurrent states. The source vertex of \( t \) is \( j \) (a join pseudostate), the transitions targeting \( j \) are in the \( T_{\text{in}} \) set, the sources of transitions in \( T_{\text{in}} \) are in the \( S_{\text{src}} \) set and the target of \( t \) is \( s_{\text{trg}} \) (Eq. 1.77).

Definition 16 (Transition Conglomerate Class E). The \( \mathcal{T}_e \) set of external transition conglomerates with a single fork vertex (\( \mathcal{T}_e \subseteq (\text{State} \setminus \text{FinalState}) \times \text{Transition} \times \text{Pseudostate}^{\text{fo}} \times 2^{\text{Transition}} \times 2^{\text{State}} \)) contains those \((s_{\text{src}}, t, f, T_{\text{out}}, s_{\text{trg}})\) tuples where (i) \( t \) is a transition (ii) of kind \( \text{external} \), (iii) the source of \( t \) is a state but not a final state, (iv) the target of \( t \) is a fork pseudostate, for all \( t_{\text{out}} \) transitions originating in the target of \( t \) (the fork vertex) holds that (v) \( t_{\text{out}} \) has no guard, (vi) \( t_{\text{out}} \) has no trigger, (vii) \( t_{\text{out}} \) is of kind \( \text{external} \), (viii) the target of \( t_{\text{out}} \) is a state (ix) and for each \((t_{\text{out}1}, t_{\text{out}2})\) pairs of

![Figure 1.9: Transition Conglomerate Classes](image-url)
transitions originating in the target of \( t \) holds that the targets of \( t_{out1} \) and \( t_{out2} \) are concurrent states. The target vertex of \( t \) is \( f \) (a fork pseudostate), the transitions originating in \( f \) are in the \( T_{out} \) set, the targets of transition in \( T_{out} \) are in the \( S_{src} \) set and the source of \( t \) is \( s_{src} \) (Eq. 1.78).

**Definition 17 (Transition Conglomerate Class \( F \)).** The \( T\mathcal{C}_f \) set of external transition conglomerates with fork and join vertices \( (T\mathcal{C}_f \subseteq 2^{(\text{State} \setminus \text{FinalState})} \times 2^{\text{Transition} \times \text{Pseudostate}^{\text{src}} \times \text{Transition} \times \text{Pseudostate}^{\text{src}}}) \) contains those \( (S_{src}, T_{in}, j, t, f, T_{out}, S_{trg}) \) tuples where (i) \( t \) is a transition (ii) of kind external, (iii) the source of \( t \) is a join pseudostate, for all \( t_{in} \) transitions targeting the source of \( t \) (the join vertex) holds that (iv) \( t_{in} \) has no guard, (v) \( t_{in} \) has no trigger, (vi) \( t_{in} \) is of kind internal, (vii) the source of \( t_{in} \) is a state but not a final state, (viii) for each \( (t_{in1}, t_{in2}) \) pairs of transitions targeting the source of \( t \) holds that the sources of \( t_{in1} \) and \( t_{in2} \) are concurrent states, (ix) the target of \( t \) is a fork pseudostate, for all \( t_{out} \) transitions originating in the target of \( t \) (the fork vertex) holds that (x) \( t_{out} \) has no guard, (xi) \( t_{out} \) has no trigger, (xii) \( t_{out} \) is of kind external, (xiii) the target of \( t_{out} \) is a state and (xiv) for each \( (t_{out1}, t_{out2}) \) pairs of transitions originating in the target of \( t \) holds that the targets of \( t_{out1} \) and \( t_{out2} \) are concurrent states. The source vertex of \( t \) is \( j \) (a join pseudostate), the transitions targeting \( j \) are in the \( T_{in} \) set, the sources of transitions in \( T_{in} \) are in the \( S_{src} \) set, the target vertex of \( t \) is \( f \) (a fork pseudostate), the transitions originating in \( f \) are in the \( T_{out} \) set and the targets of transitions in \( T_{out} \) are in the \( S_{trg} \) set (Eq. 1.79).

\[
\begin{align*}
\mathcal{T}_C_a &= \{(t, t.\text{source}) \mid t \in \text{Transition} \land \text{t.kind} = \text{local} \land t.\text{source} = t.\text{target} \land t.\text{source} \in (\text{State} \setminus \text{FinalState})\} \quad (1.74) \\
\mathcal{T}_C_b &= \{(t, t.\text{source}) \mid t \in \text{Transition} \land \text{t.kind} = \text{internal} \land t.\text{source} = t.\text{target} \land t.\text{source} \in (\text{State} \setminus \text{FinalState})\} \quad (1.75) \\
\mathcal{T}_C_c &= \{(t.\text{source}, t.\text{target}) \mid t \in \text{Transition} \land t.\text{kind} = \text{external} \land t.\text{source} \in (\text{State} \setminus \text{FinalState}) \land t.\text{target} \in \text{State}\} \quad (1.76) \\
\mathcal{T}_C_d &= \{(t.\text{source}, t.\text{target}, t.\text{source.incoming}, t.\text{source.outgoing}) \mid t \in \text{Transition} \land t.\text{kind} = \text{external} \land t.\text{source} \in \text{Pseudostate}^{\text{src}} \land t.\text{target} \in \text{State} \land \\
&\quad (\forall t_{in} \in t.\text{source.incoming} : t_{in}.\text{guard} = \emptyset \land t_{in}.\text{trigger} = \emptyset \land t_{in}.\text{kind} = \text{external} \land \\
&\quad t_{in}.\text{source} \in (\text{State} \setminus \text{FinalState})) \} \quad (1.77) \\
\mathcal{T}_C_e &= \{(t.\text{source}, t.\text{target}, t.\text{source.outgoing}, \{t_{out.target} \mid t_{out} \in t.\text{target.outgoing}) \mid t \in \text{Transition} \land t.\text{kind} = \text{external} \land t.\text{source} \in (\text{State} \setminus \text{FinalState}) \land t.\text{target} \in \text{Pseudostate}^{\text{src}} \land \\
&\quad (\forall t_{out} \in t.\text{target.outgoing} : t_{out}.\text{guard} = \emptyset \land t_{out}.\text{trigger} = \emptyset \land t_{out}.\text{kind} = \text{external} \land t_{out}.\text{target} \in \text{State}) \} \quad (1.78) \\
\mathcal{T}_C_f &= \{(t.\text{source}, t.\text{target}, t.\text{source.outgoing}, \{t_{out.target} \mid t_{out} \in t.\text{target.outgoing}) \mid t \in \text{Transition} \land t.\text{kind} = \text{external} \land t.\text{source} \in \text{Pseudostate}^{\text{src}} \land t.\text{target} \in \text{Pseudostate}^{\text{src}} \land \\
&\quad (\forall t_{in} \in t.\text{source.incoming} : t_{in}.\text{guard} = \emptyset \land t_{in}.\text{trigger} = \emptyset \land t_{in}.\text{kind} = \text{external} \land \\
&\quad t_{in}.\text{source} \in (\text{State} \setminus \text{FinalState})) \} \quad (1.79) \\
&\quad (\forall t_{out} \in t.\text{target.outgoing} : t_{out}.\text{guard} = \emptyset \land t_{out}.\text{trigger} = \emptyset \land t_{out}.\text{kind} = \text{external} \land t_{out}.\text{target} \in \text{State}) \} \quad (1.79) \\
&\quad (\forall t_{out1}, t_{out2} \mid t_{out1} \neq t_{out2} \land t_{out1}.\text{target}, t_{out2}.\text{target}) \} \}
\end{align*}
\]

**Definition 18 (Set of Transition Conglomerates).** In order to enable unique handling we introduce the \( TC \) set of transition conglomerates as union of sets \( \mathcal{T}_C_a, \mathcal{T}_C_b, \ldots, \mathcal{T}_C_f \):

\[
TC = \mathcal{T}_C_a \cup \mathcal{T}_C_b \cup \mathcal{T}_C_c \cup \mathcal{T}_C_d \cup \mathcal{T}_C_e \cup \mathcal{T}_C_f \quad (1.80)
\]

It is worth noting that each transition conglomerate can be characterized by a single transition of central role: this transition was systematically named \( t \) in all definitions and figures above. For classes \( t \) can be used to derive \( s_{src}, S_{src}, T_{in}, j, f, T_{out}, S_{trg} \) and \( S_{trg} \) (where applicable). These \( t \) transitions can be seen as “real transitions” in the model; all other transitions and join and fork pseudostates should have been introduced as drawing symbols by the standard. Below we will refer to this transition of central role simply as \( t \) for any \( tc \in TC \) transition conglomerates.
1.5. Compound Transition Structures

1.5.3 Triggers and Guards

Transition conglomerates were defined so that only the transition segments may have trigger and guard, therefore the trigger and guard of transition conglomerates are defined by ones of $t$.

Definition 19 (Trigger of Transition Conglomerate). The $\text{TCTrigger} : \mathcal{TC} \rightarrow \text{Trigger}$ function (Eq. 1.81) for a $t \in \mathcal{TC}$ returns the trigger of the transition $t$ if $t$ has a trigger or the special symbol $t_0$ representing the empty trigger ($t_0 \not\in \text{Trigger}$):

$$\text{TCTrigger}(t \in \mathcal{TC}) = \begin{cases} \text{trigger} & \text{if } |t.\text{trigger}| = 1 \\ t_0 & \text{otherwise} \end{cases}$$  \hspace{1cm} (1.81)

Definition 20 (Guard of Transition Conglomerate). The $\text{TCGuard} : \mathcal{TC} \rightarrow \text{Constraint}$ function (Eq. 1.82) for a $t \in \mathcal{TC}$ returns the guard of the transition $t$ if $t$ has a guard or the special symbol $c_0$ representing the empty constraint ($c_0 \not\in \text{Constraint}$):

$$\text{TCGuard}(t \in \mathcal{TC}) = \begin{cases} \text{guard} & \text{if } |t.\text{guard}| = 1 \\ c_0 & \text{otherwise} \end{cases}$$  \hspace{1cm} (1.82)

1.5.4 Sources, Targets and Containers

As transition conglomerates are compound structures possibly involving multiple transitions and pseudostates, when talking about source or target states we have to distinguish multiple concepts:

- The states where join segments (or the transition $t$ in case of classes A, B, C and E) originate will be called as enabling states i.e., these states should be active to enable the transition conglomerate; the states that are targeted by fork segments (or the transition $t$ in case of classes A, B, C and D) will be called as states determined by the transition conglomerate.

- We will introduce the concept of bottom container region of transition conglomerates – this is the formally well-established equivalent of the loosely-defined UML “least common ancestor” (LCA) concept. The state in the bottom container region that contains all enabling states as substates will be called as top source while the one containing all determined states as substates will be called as top target of the transition conglomerate – these concepts are the formally well-established equivalents of the UML “main source” and “main target” concepts.

- Finally we will define the set of states possibly left by a transition conglomerate and the state hierarchies entered by the transition conglomerate.

Definition 21 (Enabling States of a Transition Conglomerate). The $\text{TCEnabling} : \mathcal{TC} \rightarrow 2^{\text{State}}$ function (Eq. 1.83) for a $t \in \mathcal{TC}$ transition conglomerate returns the states that should be active for being $tc$ enabled:

$$\text{TCEnabling}(t) = \begin{cases} \{s\} & \text{if } t \in \mathcal{TC}_a, t = (t, s) \in \mathcal{TC}_b \\ \{s_{\text{src}}\} & \text{if } t \in \mathcal{TC}_c, t = (s_{\text{src}}, t, v_{\text{trg}}) \in \mathcal{TC}_e \setminus \mathcal{TC}_i \\ S_{\text{src}} & \text{if } t \in \mathcal{TC}_d, t = (s_{\text{src}}, t, p_{\text{trg}}) \in \mathcal{TC}_d, t = (s_{\text{src}}, t, p_{\text{trg}}) \in \mathcal{TC}_i \setminus \mathcal{TC}_j \\ \end{cases}$$  \hspace{1cm} (1.83)

Definition 22 (States Determined by a Transition Conglomerate). The $\text{TCDetermined} : \mathcal{TC} \rightarrow 2^{\text{State}}$ function (Eq. 1.84) for a $t \in \mathcal{TC}$ transition conglomerate returns the states that are to be explicitly made active after performing $tc$:

$$\text{TCDetermined}(t) = \begin{cases} \{s\} & \text{if } t \in \mathcal{TC}_a, t = (t, s) \in \mathcal{TC}_b \\ \{s_{\text{trg}}\} & \text{if } t \in \mathcal{TC}_c, t = (s_{\text{trg}}, t, v_{\text{trg}}) \in \mathcal{TC}_e \setminus \mathcal{TC}_i \\ S_{\text{trg}} & \text{if } t \in \mathcal{TC}_d, t = (s_{\text{trg}}, t, p_{\text{trg}}) \in \mathcal{TC}_d, t = (s_{\text{trg}}, t, p_{\text{trg}}) \in \mathcal{TC}_i \setminus \mathcal{TC}_j \end{cases}$$  \hspace{1cm} (1.84)
Definition 23 (Bottom Container of a State Set). The \( \text{BC}^S : 2^{\text{State}} \to \text{Region} \) \( \text{bottom container of a state set function (Eq. 1.85)} \) for an \( S \subseteq \text{State} \) non-empty set of states \((S \neq \emptyset)\) returns the bottommost region that (directly or transitively) contains all states in \( S \). (It can be proven that \( \text{BC}^S \) always returns a single region; see Thm. 7 in App. A.5)

\[
\text{BC}^S(S \subseteq \text{State}) = \{ r \in \text{Region} | S \subseteq \text{AllCont}^T(r) \land (\forall r' \in \text{AllSubreg}^T(r) : S \subseteq \text{AllCont}^T(r')) \}
\] (1.85)

Definition 24 (Bottom Container of a Transition Conglomerate). Based on \( \text{BC}^S \) we below introduce the \( \text{BC}^{TC} : \text{TC} \to \text{Region} \) \( \text{bottom container of a transition conglomerate function (Eq. 1.86)} \) that for a \( tc \in \text{TC} \) transition conglomerate returns the bottom container region of the set of enabling and determined states of \( tc \) (i.e., the bottommost region that contains all sources and targets of the transition conglomerate). Since neither \( \text{TCEnabling}(tc) \) nor \( \text{TCDetermined}(tc) \) are empty sets, \( \text{BC}^S \) is applicable and this also implies that for any transition conglomerate \( \text{BC}^{TC} \) returns a single region: \( \forall tc \in \text{TC} : |\text{BC}^{TC}(tc)| = 1 \).

\[
\text{BC}^{TC}(tc \in \text{TC}) = \text{BC}^S(\text{TCEnabling}(tc) \cup \text{TCDetermined}(tc))
\] (1.86)

Definition 25 (Top Source of a Transition Conglomerate). The \( \text{TCTopSource} : \text{TC} \to \text{State} \) \( \text{top source function (Eq. 1.87)} \) for a transition conglomerate \( tc \) returns that state in the bottom container region of \( tc \) that contains its all enabling states (it can be proven that \( \text{TCTopSource} \) always returns a single state; see Thm. 9 in App. A.5):

\[
\text{TCTopSource}(tc \in \text{TC}) = \{ s | s \in \text{AllCont}^D(\text{BC}^{TC}(tc)) \land \forall s' \in \text{TCEnabling}(tc) : s' \in (\{s\} \cup \text{AllSubst}^T(s)) \}
\] (1.87)

Definition 26 (Top Target of a Transition Conglomerate). The \( \text{TCTopTarget} : \text{TC} \to \text{State} \) \( \text{top target function (Eq. 1.88)} \) for a transition conglomerate \( tc \) returns that state in the bottom container region of \( tc \) that contains all states determined by \( tc \) (it can be proven that \( \text{TCTopTarget} \) always returns a single state; see Thm. 10 in App. A.5):

\[
\text{TCTopTarget}(tc \in \text{TC}) = \{ s | s \in \text{AllCont}^D(\text{BC}^{TC}(tc)) \land \forall s' \in \text{TCDetermined}(tc) : s' \in (\{s\} \cup \text{AllSubst}^T(s)) \}
\] (1.88)

Example 5. The statechart shown in Fig. 1.10 contains three transition conglomerates: (i) \( tc_1 \) is of class E, (ii) \( tc_2 \) is a local transition (class A) and (iii) \( tc_3 \) is of class D. The enabling and determined states, bottom container regions, top source and target states are indicated in callouts.

States Left by a Transition Conglomerate

When firing a transition conglomerate one or more states may have to be exited: (i) in case of local transitions (class A) the source state \( s_{\text{src}} \) of the transition is not left but if \( s_{\text{src}} \) is refined to any number of regions, the active substates of \( s_{\text{src}} \) are to be exited and the initial state hierarchy of \( s_{\text{src}} \) is to be entered (ii) in case of internal transitions (class B) no states are left and in case of external transition
1.5. COMPOUND TRANSITION STRUCTURES

conglomerates (classes C, D, E and F) the top source is to be left recursively. To put together: when firing a transition conglomerate a set of state hierarchies may be left – this set contains as many elements as the number of direct substates of \( s_{src} \) in case of class A, the set is empty in case of class B and the set contains a single hierarchy in case of classes C, D, E and F.

**Definition 27 (Hierarchies of States Possibly Left by a Transition Conglomerate).** Below we formally define the \( \text{TCPossiblyLeftHierarchySet} \) function (Eq. 1.89) that for a transition conglomerate \( tc \) returns a set of state hierarchy nodes corresponding to the states possibly left by the transition conglomerate.

\[
\text{TCPossiblyLeftHierarchySet}(tc \in TC) = \begin{cases} 
\{ \text{BuildHierarchy}(s_c) | s_c \in \text{AllSubst}^D(\text{TCTopSource}(tc)) \} & \text{if } tc \in TC_a \\
\emptyset & \text{if } tc \in TC_b \\
\{ \text{BuildHierarchy}(\text{TCTopSource}(tc)) \} & \text{otherwise}
\end{cases}
\] (1.89)

Note that the hierarchies returned by \( \text{TCPossiblyLeftHierarchySet} \) contain all states possibly left by the transition conglomerate; when firing the transition conglomerate only those of these states are to be left that are actually active.

**Example 6.** The state hierarchies possibly left by transition conglomerates of Fig. 1.10 are indicated in callouts of Fig. 1.11 (note that in case of the local transition conglomerate \( tc_2 \) the set of hierarchies possibly left contains three elements).

**States Entered by a Transition Conglomerate**

This subsection discusses the issues related to state hierarchies entered by transition conglomerates. First we focus on the most common cases, transition conglomerate classes C, D, E and F than investigate the remaining two classes A and B.

**States Entered by Transition Conglomerates of Classes C, D, E and F**

A transition conglomerate of these classes enters a state hierarchy whose top state is the top target of the transition conglomerate and its substates to be entered are to be selected according to the states determined by the transition conglomerate. We present below a straightforward way for building the state hierarchy entered by a transition conglomerate: we will traverse the state hierarchy starting with the top target of the transition conglomerate and for each regions refining a state to be entered we will check whether a state contained by the region is directly or transitively targeted by the transition conglomerate (the state or one of its substates is in the set of states determined by the transition conglomerate) and if such a state is found we will continue the traversal with that state otherwise we will select by default the initial state of the region indicated by the initial pseudostate.
Building the state hierarchy entered by a transition conglomerate is fundamentally based on the
handshaking of two mutually recursive functions: \texttt{EnteredHierarchy}^S and \texttt{EnteredHierarchy}^R. Both functions return a state hierarchy node representing the state hierarchy to be entered when entering
the state or region respectively. These functions have two input arguments: (i) the state \texttt{s} to be entered
(in case of \texttt{EnteredHierarchy}^S) or the region \texttt{r} to be entered (in case of \texttt{EnteredHierarchy}^R) and (ii) the \texttt{Sdet} set of states that are determined by the transition conglomerate. The function \texttt{EnteredHierarchy}^R(\texttt{r}) first collects those states directly contained by \texttt{r} that are either directly or transis-
tively determined by the region \texttt{j} into \texttt{s_{det}} (introducing \texttt{s_{det}} is only a convenience: we define the value and use the symbol \texttt{s_{det}} at two places of the definition resulting in a more compact equation). If \texttt{s_{det}} contains exactly one state (case label \texttt{|s_{det}| = 1}) the recursive traversal continues with that state, otherwise the default initial state is chosen (case label otherwise). The function \texttt{EnteredHierarchy}^S(\texttt{s}) returns a state hierarchy node whose top state is \texttt{s} and whose subnodes are collected by a set of recursive calls to \texttt{EnteredHierarchy}^R for regions refining \texttt{s}.

\textbf{Definition 28.} The definition of \texttt{EnteredHierarchy}^S and \texttt{EnteredHierarchy}^R is presented below. It can be proven that when calling \texttt{EnteredHierarchy}^R in the scheme discussed above the set \texttt{s_{det}} mentioned in the text above and used in the definition as \texttt{s_{det}} = \{s|s \in \texttt{AllCont}^D(\texttt{r}) \land ((s \in S_{det}) \lor (\exists s' \in \texttt{AllSubst}^R(\texttt{s}) : s' \in S_{det}))\} is empty or contains a single state (see Thm. \texttt{[\ref{thm:sdet}] in App. A.5}).

\begin{align}
\texttt{EnteredHierarchy}^S(\texttt{s}, \texttt{s_{det}}) &= \texttt{StateHierarchyNode}(\texttt{s}, \{\texttt{EnteredHierarchy}^R(\texttt{r}, \texttt{s_{det}}) | \texttt{r} \in \texttt{AllRef}^D(\texttt{s})\}) \quad (1.90) \\
\texttt{EnteredHierarchy}^R(\texttt{r}, \texttt{s_{det}}) &= \begin{cases} 
\texttt{EnteredHierarchy}^S(\texttt{s_{det}}, \texttt{s_{det}}) & \text{if } |\texttt{s_{det}}| = 1 \\
\texttt{EnteredHierarchy}^S(\texttt{InitialState}^B(\texttt{r}), \texttt{s_{det}}) & \text{otherwise}
\end{cases} \quad (1.91)
\end{align}

\textbf{States Entered by Transition Conglomerates of Classes A and B} Transition conglomerates of class \texttt{B} are \textit{internal transitions} therefore when firing them no states are left or entered.

Transition conglomerates of class \texttt{A} are \textit{local transitions}. When firing a local transition originating in state \texttt{s} (the source and target state is the same) all currently active substates of \texttt{s} are to be left (but \texttt{s} itself is not) and the initial substate configuration of \texttt{s} is to be entered. In this aspect we can consider local transition conglomerates as ones that have to enter a \textit{set of state hierarchies} and this set contains as many elements as the number of regions refining \texttt{s} (obviously the default initial states of regions are entered).

\textbf{Summary} To put together: taking into consideration that in case of transition conglomerate class \texttt{A} a \textit{set of state hierarchies} is to be entered, in case of class \texttt{B} nothing is to be entered while in case of classes \texttt{C}, \texttt{D}, \texttt{E} and \texttt{F} a \textit{single} hierarchy is to be entered, the unified approach for handling all possible classes should return a set of state hierarchies containing possible multiple hierarchies in case of class \texttt{A}, containing nothing in case of class \texttt{B} and containing exactly a single hierarchy in case of classes \texttt{C}, \texttt{D}, \texttt{E}, and \texttt{F}.

\textbf{Definition 29 (State Hierarchies Entered by Transition Conglomerates).} The definition of the \texttt{TCEnteredHierarchySet} : TC \rightarrow 2^{\texttt{StateHierarchyNode}} \textit{state hierarchies entered by transition con-
glomerates} function (Eq. \ref{eq:tehs}) is shown below.

\begin{align}
\texttt{TCEnteredHierarchySet}(\texttt{tc}) &= \\
&= \begin{cases} 
\{\texttt{InitialState}^B(\texttt{tc}), \texttt{TCDetermined}(\texttt{tc})\} & \text{if } \texttt{tc} \in \texttt{TC}_a \\
\emptyset & \text{if } \texttt{tc} \in \texttt{TC}_b \\
\{\texttt{EnteredHierarchy}^S(\texttt{TCTopTarget}(\texttt{tc})), \texttt{TCDetermined}(\texttt{tc})\} & \text{otherwise}
\end{cases} \quad (1.92)
\end{align}

\textbf{Example 7.} The state hierarchies entered by transition conglomerates of Fig. \ref{fig:transition_conglomerates} are indicated in callouts of Fig. \ref{fig:example7_callouts}.
1.5. COMPOUND TRANSITION STRUCTURES

1.5.5 Priorities and Conflicts

Definition 30 (Transition Conglomerate Priority). If \( tc^l \) and \( tc^h \) are transition conglomerates \( \{tc^l, tc^h\} \subseteq TC \), \( tc^h \) is said to have higher priority than \( tc^l \) (\( tc^l \prec tc^h \)) if (i) for all enabling states \( s^h \in TCEnabling(tc^h) \) of \( tc^h \) holds that \( s^h \) is one of the enabling states of \( tc^l \) or a substate of an enabling state of \( tc^l \) and (ii) there is at least one enabling state \( s^h \in TCEnabling(tc^h) \) of \( tc^h \) that \( s^h \) is one of the enabling states of \( tc^l \) (i.e., all source states of \( tc^h \) are at the same or deeper levels in the state hierarchy and at least one of its sources is explicitly deeper in the state hierarchy than sources of \( tc^l \)), i.e., \( tc^l \prec tc^h \) if:

\[
\forall s^h \in \text{TCEnabling}(tc^h) : \exists s^l \in \text{TCEnabling}(tc^l) : s^h \in (\{s^l\} \cup \text{AllSubst}^T(s^l)) \land
\exists s^h \in \text{TCEnabling}(tc^h) : \exists s^l \in \text{TCEnabling}(tc^l) : s^h \in \text{AllSubst}^T(s^l) \quad (1.93)
\]

It is easy to see that the transition priority relation is transitive (proven by Thm. 8 in App. A.5).

Definition 31 (All Transition Conglomerates of Higher Priority). Below we define a function (prototype: Eq. (1.94) def.: Eq. (1.95)) that for a transition conglomerate \( tc \) returns all transition conglomerates that are of higher priority than \( tc \):

\[
\text{TCAllHigherPriority} : TC \rightarrow 2^{TC} \quad (1.94)
\]

\[
\text{TCAllHigherPriority}(tc) = \{tc^h|tc^h \in TC : tc^l \prec tc^h\} \quad (1.95)
\]

Definition 32 (Conflicting Transition Conglomerates). Two different transition conglomerates \( tc_1 \) and \( tc_2 \) (\( \{tc_1, tc_2\}|tc_1 \neq tc_2 \subseteq TC \)) are said to be in conflict (\( tc_1 \# tc_2 \)) if the intersection of states possibly left by them is non-empty, i.e., \( tc_1 \neq tc_2 \) if:

\[
(\{\text{TCTopSource}(tc_1)\} \cup \text{AllSubst}^T(\text{TCTopSource}(tc_1))) \cap
(\{\text{TCTopSource}(tc_2)\} \cup \text{AllSubst}^T(\text{TCTopSource}(tc_2))) \neq \emptyset \quad (1.96)
\]

Note that the definition takes into consideration the top source and all of its substates even in case of local and internal transitions corresponding to the standard.

Definition 33 (All Conflicting Transition Conglomerates). Below we define the all conflicting transition conglomerates function (prototype: Eq. (1.97) def.: Eq. (1.98)) that for a transition conglomerate \( tc \) returns all transition conglomerates that are in conflict with \( tc \) and the set of all conflicting transition conglomerate pairs (Eq. (1.99)):

\[
\text{TCAllInConflict} : TC \rightarrow 2^{TC} \quad (1.97)
\]

\[
\text{TCAllInConflict}(tc) = \{tc^c|tc^c \in TC : tc \# tc^c\} \quad (1.98)
\]

\[
\text{TCConflict} = \{(tc_1, tc_2)|tc_1 \in TC, tc_2 \in TC : tc_1 \# tc_2\} \quad (1.99)
\]
Example 8. There are four transition conglomerates in Fig. 1.13 (identifiers are simply indicated above the corresponding transitions): (i) \( tc_1 : s_2 \rightarrow s_3 \) (class C), (ii) \( tc_2 : s_1 \rightarrow s_5 \) (class C), (iii) \( tc_3 : \{s_3, s_4\} \rightarrow s_6 \) (class D) and (iv) \( tc_4 : s_5 \rightarrow s_7 \) (class C); both \( tc_1 \) and \( tc_3 \) are of higher priority than \( tc_2 \) (\( tc_2 < tc_1, \) \( tc_2 < tc_3 \)) but there are no priority relations amongst \( tc_1 \) and \( tc_3 \) and amongst \( tc_4 \) and any other transition conglomerates; \( tc_1 \), \( tc_2 \) and \( tc_3 \) are in conflict with each other (\( tc_1 \# tc_2, \) \( tc_2 \# tc_3, \) \( tc_3 \# tc_1, \) \( tc_2 \# tc_3 \) and \( tc_3 \# tc_1 \)) but \( tc_4 \) is not in conflict with any other ones. The sets returned by \( \text{TCAllHigherPriority} \) and \( \text{TCAllInConflict} \) are indicated in callouts.

1.6 Compound Activity Structures

The entire response of a state machine to a trigger may involve the execution of multiple activities (leaving states, performing effects of transitions and entering states). There may be various precedence relations amongst these activities: there are some strict subsequence relations (e.g., the exit activity of a child state should be performed before the exit activity of its parents, etc.) and there are activity pairs that are not in such a relation, i.e., they may be performed even in parallel (e.g., exit and entry activities of states in different regions of an orthogonal state) and when multiple transitions are selected for firing they are performed also in a non-specified order (even in parallel).

Enabling parallel execution is definitely an important feature of any behavioral models since it allows better exploiting of computing resources of modern hardware platforms and operating systems. In spite of the evident need for a well-established compound activity structure concept, the UML standard does not formally introduce such notion at the level of statecharts. According to the observations above, our approach aims at providing the maximal freedom for parallel execution and for this purpose needs a formally well-founded concept that is capable of expressing compound activity structures with subsequence relations and possibilities for parallel execution.

Parallel operation is however not as easy to express as sequential processes: various specification and programming languages provide custom and mostly non-portable language facilities and libraries for doing this. Not even the Microsoft AsmL executable specification language is perfectly flexible in this aspect: one can use the \texttt{forall} and related constructs for expressing that various operations are executed in parallel or at least in a non-deterministic order, but there is no straightforward and easy to understand way for expressing multi-level overlapping of activities (e.g., two transitions are fired in parallel and within these transitions entry activities of concurrent states are also performed in parallel, etc.). In order to overcome these difficulties we will use a high-level, language-independent formalism for expressing compound activity structures and subsequence relations by integrating PERT graphs into the metamodel and semantics of precise statecharts. This section defines PERT graphs and discusses how to describe the compound activity structure to be carried out when entering or leaving a state hierarchies and performing effects of transition conglomerates.

1.6.1 PERT Graphs

PERT graphs (Program Evaluation and Review Technique) have been successfully used in various fields (manufacturing, business, etc.) for expressing dependencies between atomic steps of compound processes. A PERT graph is a directed acyclic graph (DAG) where the nodes are atomic steps of the process and directed edges express the execution dependency between two steps: if an edge originates in \( a \) and targets \( b \) (\( a \rightarrow b \)) it means that \( b \) can only be started after finishing \( a \). We will use PERT graphs
1.6. COMPOUND ACTIVITY STRUCTURES

for expressing the execution dependencies between basic activities building up the entire response of the state machine to a trigger. Our approach does not enforce any scheduling schemes for the execution of activities enabling the implementation to exploit as much of the parallel processing capabilities of the underlying HW/SW platform as possible while meeting the loosely defined subsequence relations.

**Definition 34 (PERT Graphs).** Formally a PERT graph $p = (N, P)$ where $N$ is a set of nodes and $P$ is a set of precedences. A precedence is an ordered pair of nodes: $P \subseteq N \times N$. If $(n_{pre}, n_{post}) \in P$ then the activity represented by node $n_{pre}$ should be performed before $n_{post}$. A PERT graph is a DAG (Eq. 1.100). We will use the $p_{\emptyset}$ symbol for empty graphs: $p_{\emptyset} = (\emptyset, \emptyset)$.

$$\beta\{n_1, n_2, \ldots, n_m\} \subseteq N : \{(n_1, n_2), (n_2, n_3), \ldots, (n_{m-1}, n_m), (n_m, n_1)\} \subseteq P$$  \hspace{1cm} (1.100)

Our metamodel of PERT graphs is shown in the left side of Fig. 1.14 the PERT graph (PERTGraph) contains any number of nodes (PERTNode) and precedence relations (PERTPrecedence) through roles nodes and precedences respectively. Nodes refer to zero or one activity.

**Example 9.** The object diagram in the callout of Fig. 1.14 shows the formal representation of the compound activity structure in the middle of the figure: there are five activities in the graph: $a$, $b$, $c$, $d$ and $e$ and three precedences amongst them: $a \rightarrow b$, $b \rightarrow c$ and $d \rightarrow c$. Activities are represented by instances of Activity and referenced by five PERTNode instances through the role activity. Precedences are represented by instances of PERTPrecedence, e.g., $a \rightarrow b$ is represented by the leftmost PERTPrecedence referring to nodes representing $a$ and $b$ through roles pre and post respectively.

**Definition 35 (First and Last Nodes).** A node is considered to be one of the first nodes of the graph if there are no other nodes that are to be executed before this node (i.e., it represents one of the first activities of the compound operation) (Eq. 1.101); a node is considered to be one of the last nodes if there are no other nodes that are to be executed after this node (i.e., the node represents one of the last activities in the compound operation) (Eq. 1.102).

$$\text{PERTFirstSet}(g) = \{n \in g.nodes | \beta p \in g.precedences : n = p.post\}$$  \hspace{1cm} (1.101)

$$\text{PERTLastSet}(g) = \{n \in g.nodes | \beta p \in g.precedences : n = p.pre\}$$  \hspace{1cm} (1.102)

**Example 10.** There are two PERT graphs shown in the middle of Fig. 1.15 $p_1$ and $p_2$. The set of first nodes in $p_1$ is $\{a, b, d, f\}$, last nodes are $\{d, g, h\}$; the set of first nodes in $p_2$ is $\{i\}$, last nodes are $\{n, o\}$. Note that the sets of first and last nodes need not be disjoint (e.g., node $d$ in $p_1$).

**Definition 36 (Union of PERT Graphs).** The union of $P$ set of PERT graphs (prototype: Eq. 1.105) def.: Eq. 1.104 is a PERT graph containing all nodes and precedences of $p_i \in P$ graphs:
Definition 37 (Appending PERT Graphs). Appending the PERT graph $p_2$ after $p_1$ (prototype: Eq. 1.105 def.: Eq. 1.106) results in a PERT graph containing all nodes and precedences of $p_1$ and $p_2$ and a set of $(p'_1, p'_2)$ precedences that force the first nodes of $p_2$ come after the last nodes of $p_1$:

$$\text{PERTAppend} : \text{PERTGraph} \times \text{PERTGraph} \rightarrow \text{PERTGraph}$$

$$\text{PERTAppend}(p_1, p_2) = \text{PERTGraph}(p_1.\text{nodes} \cup p_2.\text{nodes},$$

$$p_1.\text{precedences} \cup p_2.\text{precedences} \cup \{(p'_1, p'_2)|p'_1 \in \text{PERTLastSet}(p_1), p'_2 \in \text{PERTFirstSet}(p_2)\})$$

Example 11. Unifying the PERT graphs $p_1$ and $p_2$ in the middle of Fig. 1.13 results in the graph shown in the blue callout at the left side of the figure. Appending the PERT graph $p_2$ after the graph $p_1$ results in the graph shown in the orange callout at the right side of the figure.

As the direct precedence relations in PERT graphs imply a more general precedence concept below we define the notion of being two nodes in indirect (transitive) precedence relation possibly involving any number of intermediate nodes then introduce the restrictiveness relation amongst PERT graphs.

Definition 38 (Implied Dependencies). Below we introduce the notion of direct or transitive dependence between nodes of a PERT graph (prototype: Eq. 1.107 def.: Eq. 1.108) by generalizing the information carried by precedence pairs. A node $n^*_{\text{post}}$ of the PERT graph $p$ directly or transitively depends on another node $n^*_{\text{pre}}$ of $p$ (i.e., $n^*_{\text{pre}}$ is to be performed before $n^*_{\text{post}}$) if $n^*_{\text{post}}$ is reachable from $n^*_{\text{pre}}$ through single steps of precedence relations. Next we introduce the function all dependencies (prototype: Eq. 1.109 def.: Eq. 1.110) that calculates all direct and implied dependencies.

$$\text{PERTDepends} : (\text{PERTGraph} \times \text{PERTNode} \times \text{PERTNode}) \rightarrow \{\top, \bot\}$$

$$\text{PERTDepends}(p, n^*_{\text{pre}} \in p.\text{nodes}, n^*_{\text{post}} \in p.\text{nodes}) \leftarrow$$

$$((n^*_{\text{pre}}, n^*_{\text{post}}) \in p.\text{precedences}) \lor \exists n^* \in p.\text{nodes} : (n^*_{\text{pre}}, n^*) \in p.\text{precedences} \land \text{PERTDepends}(p, n^*, n^*_{\text{post}})$$

$$\text{PERTDependencies} : \text{PERTGraph} \rightarrow 2^{\text{PERTNode} \times \text{PERTNode}}$$

$$\text{PERTDependencies}(p) = \{(n^*_{\text{pre}}, n^*_{\text{post}})|\{n^*_{\text{pre}}, n^*_{\text{post}}\} \subseteq p.\text{nodes} \land \text{PERTDepends}(p, n^*_{\text{pre}}, n^*_{\text{post}})\}$$

Example 12. The left side of Fig. 1.16 presents three PERT graphs $p_1$, $p_2$ and $p_3$: the implied dependencies are shown in the right side of the figure with blue arrows. It can be seen that $p_1 \preceq p_2$ but $p_1$ and $p_3$ are not in such relation (i.e., neither $p_1 \preceq p_3$ nor $p_3 \preceq p_1$ holds).
1.6. COMPOUND ACTIVITY STRUCTURES

1.6.2 Entering and Leaving State Hierarchies

When entering a state hierarchy (during initialization or firing a transition conglomerate) multiple entry activities are to be performed in a loosely specified order: the UML standard only requires that the entry activity of state s is to be performed before the entry activity of any of its substates (Cit. [18]). We will use the function EnteringHierarchy that for a state hierarchy node shn returns a PERT graph describing the entry activity structure to be performed and the set of states entered (i.e., the flat set of states in shn). Similarly when leaving a state hierarchy (e.g., during firing a transition) multiple exit activities are to be performed and the UML standard requires only that the exit activities of state s is to be performed after the exit activities of its substates (Cit. [18]). Constructing the PERT graph describing the activity structure to be performed when leaving a state is similar to entering a state hierarchy but while entering a state hierarchy can be calculated independently of the actual configuration, when recursively leaving a state we have to take into consideration the actual configuration of the statechart. We will use the function LeavingHierarchy that for the actual configuration (set of states) and a state hierarchy node shn returns a PERT graph describing the exit activity structure to be performed and the set of states actually left (i.e., the flat set of states in shn that were actually active). The two functions can be easily extended for sets of hierarchies, these functions will be called EnteringHierarchySet and LeavingHierarchySet respectively. The definition of the functions can be found in App. A.3 below we present the prototypes and some illustrative examples:

\[
\text{EnteringHierarchy} : \text{StateHierarchyNode} \rightarrow (\text{PERTGraph} \times 2^{\text{State}}) \tag{1.112}
\]

\[
\text{LeavingHierarchy} : (2^{\text{State}} \times \text{StateHierarchyNode}) \rightarrow (\text{PERTGraph} \times 2^{\text{State}}) \tag{1.113}
\]

\[
\text{EnteringHierarchySet} : 2^{\text{StateHierarchyNode}} \rightarrow (\text{PERTGraph} \times 2^{\text{State}}) \tag{1.114}
\]

\[
\text{LeavingHierarchySet} : (2^{\text{State}} \times 2^{\text{StateHierarchyNode}}) \rightarrow (\text{PERTGraph} \times 2^{\text{State}}) \tag{1.115}
\]

Example 13. Let tc be the transition conglomerate highlighted in Fig. 1.17 let S_L be the (single) state hierarchy left by tc and S_E be the (single) state hierarchy entered by tc. Calling the function LeavingHierarchySet with the configuration \(\{s_1, s_2\}\) and \(S_L\) returns the PERT graph \(p_L\) representing the compound activity structure to be performed and the states actually left \(\{s_1, s_2\}\) (Eq. 1.116); calling the function EnteringHierarchySet with \(S_E\) returns the PERT graph \(p_E\) representing the compound activity structure to be performed and the states actually entered \(\{s_3, s_4, s_6, s_7\}\) (Eq. 1.117). The graphs \(p_L\) and \(p_E\) are shown in callouts in Fig. 1.17 where the symbol \(X_s\) represents the exit activity of state \(s_i\) and \(E_s\) represents the entry activity of state \(s_j\). It is easy to see that the precedence relations correspond to the UML semantics: (i) in case of \(p_L\) the exit activity of the child state is performed before the one of the parent state and (ii) in case of \(p_E\) the entry activity of \(s_3\) is to be performed first, then the entry activities of \(s_4\) and \(s_6\) are performed in an unspecified order (possibly in parallel) and the entry activity of \(s_7\) is to be performed after the entry activity of \(s_6\).

\[
\text{LeavingHierarchySet}(\{s_1, s_2\}; \{S_L|S_L=TCPossiblyLeftHierarchySet(tc)\}) = (p_L, \{s_1, s_2\}) \tag{1.116}
\]

\[
\text{EnteringHierarchySet}(\{S_E|S_E=TCPossiblyLeftHierarchySet(tc)\}) = (p_E, \{s_3, s_4, s_6, s_7\}) \tag{1.117}
\]
1.6.3 Effects of Transition Conglomerates

Since transition conglomerates may consist of multiple transitions that have effects assigned the effect of a transition conglomerate is not as easy to define as triggers or guards. We will use the \( TCEffect : \text{TC} \rightarrow \text{PERTGraph} \) function that for a transition conglomerate \( tc \) returns a PERT graph that contains the effects of transitions involved in the conglomerate as nodes and specify the precedence relations amongst these activities. The UML standard only requires that effects of transition segments in a compound transition path should be performed according to the path starting in the source state(s) and ending in the target state(s) (Cit. 24). In case of local and internal transitions and simple external loop transitions (classes A, B and C) the transition conglomerate consists of a single transition \( t \), obviously the effect of the transition conglomerate is a PERT graph containing a single node, the effect of \( t \). In case of transition conglomerates involving fork or join pseudostates the effect of the transition conglomerate contains the effects of the join segments (if any) the effect of \( t \) and the effects of fork segments (if any). The formal definition of \( TCEffect \) for all transition conglomerate classes can be found in App. A.3. Below we give an illustrative example.

**Example 14.** Let \( tc \) be the transition conglomerate highlighted in Fig. 1.17; \( tc \) is of class E involving a transition \( t \) originating in state \( s_2 \) and targeting a fork vertex \( f \) and two fork segments both originating in \( f \) and targeting states \( s_4 \) and \( s_7 \) respectively. All the three transitions building up \( tc \) have effects assigned (\( e_1 \) for \( t \) and effects \( e_2 \) and \( e_3 \) for the fork segments). The PERT graph in the callout under the transition conglomerate indicates the order of activities: first the effect of \( t \) is to be performed then the effects of fork segments in an unspecified order. The orange callout in the bottom left corner of the figure represents the entire firing of the transition conglomerate appending the PERT graphs representing (i) leaving the source states, (ii) performing the effect and (iii) entering the target states.

1.7 Introduction of New Concepts Into the Metamodel

Our newly defined concepts are inserted into the metamodel as shown in Fig. 1.18. First we introduced the \text{PreciseStatechart} metaclass derived from the UML \text{StateMachine} metaclass representing precise statecharts. The initial state hierarchy of the statechart is represented by the containment relation between the \text{PreciseStatechart} and the \text{StateHierarchyNode} (i.e., the state hierarchy returned by \text{InitialHierarchy}). Although the set of all states in the statechart can be obtained by the traversal of the state refinement hierarchy, for simplicity reasons we introduced a direct association between the \text{PreciseStatechart} and \text{State} (role: \text{states}). The \text{TC} set of transition conglomerates is represented by the \text{TC} abstract metaclass. The transition conglomerates in a statechart are associated to the \text{PreciseStatechart} metaclass through the containment role \text{transitionConglomerates}. \text{Triggers} and \text{guards} of transition conglomerates are represented by association roles \text{trigger} and \text{guard} respectively (i.e., ones returned by functions \text{TCTrigger} and \text{TCGuard}); the empty trigger \( t_0 \) and empty constraint \( c_0 \) are represented by zero association multiplicities. \text{Enabling} and \text{determined} states are
1.8. OPERATIONAL SEMANTICS

represented by associations en and det respectively, the bottom container region is indicated by the bcTc association role, top source and target states are at association roles topSource and topTarget. The hierarchies of states possibly left and entered by a transition conglomerate are represented by association roles possiblyLeftStateHierarchySet and enteredStateHierarchySet respectively. All transition conglomerates of higher priority and all conflicting transition conglomerates are indicated by association roles allHigherPriority and allInConflict. The effect of a transition conglomerate is represented by a PERT graph associated to TC under the role effect. Concrete transition conglomerate metaclasses derived from TC can be introduced on a straightforward way (presented formally in App. A.3).

1.8 Operational Semantics

Having discussed the formal foundations of precise statecharts, introduced formalisms for compound transition and activity structures and introduced the new concepts into the metamodel, we are finally ready to define the operational semantics. This section outlines the representation of variables used in guard predicates or activities, then defines the operation of precise statecharts by a KTS, gives some detailed examples and presents the operation by easy to implement imperative algorithms.

1.8.1 Modeling Extended Variables

Guard predicates of transitions are constraints (Boolean expression) that may refer to various properties e.g., a member attributes of the object, etc. Although the values of these properties contribute to the state space of the state machine, they are not explicitly modeled by states of the statechart, thus they are usually called extended variables. As storing and updating the values of extended variables is out of the scope of statecharts, we will use a straightforward abstraction. Let V be the set of extended variables; an evaluation of extended variables assigns the actual values (at the appropriate point of execution) to v ∈ V variables. We will use the notation e(v) = w for situations when the evaluation e assigns the value w to the variable v.

Let φ be a Boolean expression possibly referencing any number of extended variables (e.g., φ = (v1 < 2) ∧ (v2 = "red") where the extended variables are v1 and v2) and e the actual evaluation of variables. We say that the expression φ holds in the actual evaluation of extended variables e denoted by e |= φ if substituting the actual values into the variables of φ into the expression, φ is true (e.g., if e(v1) = 1 and e(v2) = "red" then φ = (1 < 2) ∧ ("red" = "red") = ⊤). Let the set of all possible evaluations be Eval. We will use the symbol e0 again for representing “empty constraints” that are considered to be always true for any evaluations: ∀v ∈ Eval : e |= e0. The special symbol evala ∈ Eval represents the initial evaluation of variables.

The evaluation of extended variables may be modified by the activities performed during the operation of the statechart. We will use the Issue : (Eval × PERTGraph) → Eval issue an activity structure function that for an evaluation e and a p PERT graph of activities returns the evaluation of variables after performing the modifications described by p on e (e.g., let e represent the following assignments: e(v1) = 1 and e(v2) = 2, let the node n1 of p represent the activity v1 := v2 + 1 and the node n2 represent v2 := v2 × 2 (plain arithmetic add and multiplication operations and assignments) and the single precedence relation of p require that n2 should be performed after n1; this way the evaluation e′ returned by e′ = Issue(e, p) is such that e′(v1) = 3 and e′(v2) = 4). The definition of Issue is implementation-specific thus out of the scope of this discussion.

1.8.2 Definition of the Kripke Transition System

Definition 40 (Kripke Transition System). Let P be a precise statechart. The operational semantics of P (i.e., the behavior of the system modeled by P) is specified by a Kripke transition system BKTS = (S ⊆ L), L) over the AS set of state labels and AT set of transition labels, where (i) S is the set of states, (ii) ⊆ is the labeled transition relation and (iii) L is a composite state labeling function.
States, State Labels and Labeling Functions

The $S$ state set of $B^{KTS}$ corresponds to statuses of the precise statechart $P$. The status concept is a compound representation of (i) the actual configuration of $P$, (ii) the actual evaluation of extended variables and (iii) the actual phase of the operation (defined below). These three concepts are mapped to compound state labels: $A_{S} \subseteq 2^{S} \times \text{Eval} \times \text{Phs}$ (i.e., for a $a_{s} = (C, e, p) \in A_{S}$: the actual configuration is $C$, the actual evaluation extended variables is $e$ and is the actual phase of the operation is $p$). The $I \subseteq S$ set of initial states contains a single state: $I = \{s_{\alpha}\}$. The labeling function $L^{S} : S \rightarrow A_S$ is used for associating the state labels to elements of $S$. For simplicity reasons we will decompose $L^{S}$ into three functions $L_{\text{Cnf}}^{S}$, $L_{\text{Eval}}^{S}$ and $L_{\text{Phs}}^{S}$ and consider $L^{S} : S \rightarrow A_S$ as being composed of the three functions above i.e., $L^{S}(s|s \in S) = (L_{\text{Cnf}}^{S}(s), L_{\text{Eval}}^{S}(s), L_{\text{Phs}}^{S}(s))$ where the three functions are as follows:

- $L_{\text{Cnf}}^{S} : S \rightarrow 2^{S}$ assigns the actual configuration of $P$ (set of active states) to $S$. For the single initial state $s_{\alpha} : L_{\text{Cnf}}^{S}(s_{\alpha}) = \emptyset$, for any other states $\forall s \in (S \setminus I) : L_{\text{Cnf}}^{S}(s) \neq \emptyset$.

- $L_{\text{Eval}}^{S} : S \rightarrow \text{Eval}$ assigns the actual evaluation of external variables to $S$. For the single initial state $s_{\alpha} : L_{\text{Eval}}^{S}(s_{\alpha}) = \text{eval}_{\alpha}$.

- $L_{\text{Phs}}^{S} : S \rightarrow \text{Phs}$ assigns one of four labels to states in $S$ indicating the phase of operation: (i) $\text{Phs}_{\alpha}$ if $P$ is uninitialized, (ii) $\text{Phs}_{S}$ if $P$ is in a stable configuration, (iii) $\text{Phs}_{U}$ if $P$ is an unstable phase of operation (i.e., actually performing a run-to-completion (RTC) step) and (iv) $\text{Phs}_{\omega}$ if $P$ is terminated. For the single initial state $s_{\alpha} : L_{\text{Phs}}^{S}(s_{\alpha}) = \text{Phs}_{\alpha}$.

Transition Labels and Labeled Transition Relation

The transitions of $B^{KTS}$ indicate the possible steps between statuses of $P$ (note that we are talking about the transitions of the KTS and not about the ones of the statechart). Steps are (i) initiated by one or zero triggers, (ii) possibly result in firing some transition conglomerates and (iii) executing an activity structure (specified by a PERT graph). These three concepts are mapped to the set of compound transition labels: $A_{T} \subseteq (\text{Trigger} \cup \{t_{\emptyset}\}) \times 2^{T_{C}} \times \text{PERTGraph}$ where for a $a_{t} = (t, T_{C}, g) \in A_{T}$ $t$ is the trigger, $T_{C}$ is the set of transition conglomerates fired and $g$ is the PERT graph describing the activity structure to be carried out. We use the special symbol $t_{\emptyset} \not\in \text{Trigger}$ as "empty trigger" indicating that no trigger was processed in the corresponding step. The labeled transition relation $\Rightarrow$ indicates the possible steps between states of $B^{KTS}$:

$$\Rightarrow \subseteq (S \times A_{T} \times S) = (S \times \left((\text{Trigger} \cup \{t_{\emptyset}\}) \times 2^{T_{C}} \times \text{PERTGraph}\right) \times S)$$ (1.118)

The members of the tuple represent the following concepts: (i) source state of the step, (ii) the label of the transition (where (ii-a) trigger consumed in the step, (ii-b) the set of transition conglomerates fired in the step, (ii-c) the PERT graph of activities to be performed in the step) and (iii) the target state of the step. For simplicity reasons we will decompose $\Rightarrow$ into six subsets: $\Rightarrow = \Rightarrow_{\text{init}} \cup \text{open} \cup \text{inter} \cup \text{close} \cup \text{term} \cup \text{drop}$. Below we define these relation subsets presenting the transition labels expanded as in Eq. (1.118) for simplicity i.e., in $(s, (t, T_{C}, p), s')$ states $s$ and $s'$ are the source and target states respectively, the tuple $(t, T_{C}, p) \in A_{T}$ is the transition label where $t$ the the trigger (if any), $T_{C}$ is the set of transition conglomerates fired in the step and $p$ is the PERT graph describing the activity structure.

$$\Rightarrow_{\text{init}} = \{(s_{\alpha}, (t_{\emptyset}, \emptyset, p), s_{\alpha}) | s_{\alpha} \in I \} \wedge \exists s_{\epsilon} : (p, S_{\epsilon}) = \text{Init}(\epsilon) \wedge (L^{S}(s_{\alpha}) = (s_{\epsilon}, \text{Issue}(L_{\text{Eval}}^{S}(s_{\alpha}, p), \text{Phs}_{U})))$$ (1.119)
### 1.8. OPERATIONAL SEMANTICS

\[ \text{open} = \{ (s_a, (t, TC_t, p), s_u) \mid \left( L^{s_a}_{\text{PPhs}}(s_a) = \text{Phss} \right) \land \left( TC_t = \text{Fireable}(L^{s_a}_{\text{Cut}}(s_a), L^{s_a}_{\text{Eval}}(s_a), t) \right) \land \left( TC_t \neq \emptyset \right) \land \exists S_t, S_e : ((S_t, p, S_e) = \text{Fire}(L^{s_a}_{\text{Cut}}(s_a), TC_t)) \land \left( L^S(s_a) = \left( (L^{s_a}_{\text{Cut}}(s_a) \setminus S_t) \cup S_e, \text{Issue}(L^{s_a}_{\text{Eval}}(s_a), p), \text{Phss} \right) \right) \} \]  

\[ \text{inter} = \{ (s_u, (t_0, TC_t, p), s_u') \mid \left( L^{s_u}_{\text{PPhs}}(s_u) = \text{Phss} \right) \land \left( \text{TerminationState}(L^S(s_a)) = \emptyset \right) \land \left( TC_t = \text{Fireable}(L^{s_u}_{\text{Cut}}(s_u), L^{s_u}_{\text{Eval}}(s_u), t_0) \right) \land \left( L^S(s_u') = \left( (L^{s_u}_{\text{Cut}}(s_u) \setminus S_t) \cup S_e, \text{Issue}(L^{s_u}_{\text{Eval}}(s_u), p), \text{Phss} \right) \right) \} \]  

\[ \text{close} = \{ (s_u, (t_0, \emptyset, p), s_u) \mid \left( L^{s_u}_{\text{PPhs}}(s_u) = \text{Phss} \right) \land \left( \text{TerminationState}(L^S(s_a)) = \emptyset \right) \land \left( s_u = \text{Fireable}(L^{s_u}_{\text{Cut}}(s_u), L^{s_u}_{\text{Eval}}(s_u), t_0) \right) \land \left( L^S(s_u) = \left( L^{s_u}_{\text{Cut}}(s_u), L^{s_u}_{\text{Eval}}(s_u), \text{Phss} \right) \right) \} \]  

\[ \text{term} = \{ (s_u, (t_0, \emptyset, p), s_u) \mid \left( L^{s_u}_{\text{PPhs}}(s_u) = \text{Phss} \right) \land \left( \text{TerminationState}(L^S(s_a)) = \emptyset \right) \land \left( L^S(s_u) = \left( L^{s_u}_{\text{Cut}}(s_u), L^{s_u}_{\text{Eval}}(s_u), \text{Phss} \right) \right) \} \]  

\[ \text{drop} = \{ (s_u, (t, \emptyset, p), s_u') \mid \left( L^{s_u}_{\text{PPhs}}(s_u) = \text{Phss} \right) \land \left( t = \text{Fireable}(L^{s_u}_{\text{Cut}}(s_u), L^{s_u}_{\text{Eval}}(s_u), t) \right) \land \left( L^S(s_u') = \left( L^{s_u}_{\text{Cut}}(s_u), L^{s_u}_{\text{Eval}}(s_u), \text{Phss} \right) \right) \} \]

**Initialization** \( \text{init} \) represents the initialization of the statechart (Eq. 1.119). Members of the tuple \((s_a, (t, \emptyset, p), s_u)\) indicate that the step (i) starts in the “uninitialized configuration” of the statechart \((s_a)\), (ii) consumes no triggers \((t, \emptyset)\), (iii) fires no transition conglomerates \((\emptyset)\), (iv) results in performing the activity structure \(p\) and (v) finally ends in the unstable phase \(s_u\). The meaning of sub-expressions in the definition are as follows: (i) \(s_a\) represents the initial state of the KTS (corresponding to the “uninitialized configuration” of the statechart), (ii) the activities to be performed during initialization are represented by the PERT graph \(p\), the states entered are in \(S_e\) as returned by the function \(\text{Init}\) and (iii) the labeling of the target state \(s_u\) indicates that \(s_u\) represents the configuration \(S_e\) (states entered during initialization), the extended variables are updated according to \(p\) and \(s_u\) is unstable. (See the definition of \(\text{Init}\) below.)

**Opening a run-to-completion step** \( \text{open} \) represents opening a run-to-completion step (Eq. 1.120). Members of the tuple \((s_s, (t, TC_t, p), s_u)\) indicate that the step (i) starts in the stable state \((s_s)\), (ii) consumes the trigger \(t\), (iii) fires the transition conglomerate set \(TC_t\), (iv) results in performing the activity structure \(p\) and (v) finally ends in the unstable state \(s_u\). The meaning of sub-expressions in the definition are as follows: (i) \(s_s\) is stable, (ii) the set of transition conglomerates that can be fired (with respect to the actual configuration, extended variable evaluation and trigger \(t\)) is \(TC_t\) as returned by the function \(\text{Fireable}\) (iii) \(TC_t\) is a non-empty set, (iv) firing the set \(TC_t\) results in leaving the set of states \(S_t\) performing the activity structure \(p\) and entering the set of states \(S_e\) as returned by the function \(\text{Fire}\) and (v) the labeling of the target state \(s_u\) indicates that it represents the configuration obtained by removing \(S_t\) from the original configuration and adding \(S_e\), the extended variables are updated according to \(p\) and \(s_u\) is unstable. (See the definitions of \(\text{Fireable}\) and \(\text{Fire}\) below.)

**Internal step** We will refer to steps in which we fire transition conglomerates that have no triggers
as internal steps (Eq. [1.121]). \( \text{inter} \) represents an internal step within a RTC step. Members of the tuple \((s_u, (t_\emptyset, TC_I, p), s'_u)\) indicate that the step (i) starts in the unstable state \((s_u)\), (ii) consumes no triggers \((t_\emptyset)\), (iii) fires the transition conglomerate set \(TC_I\), (iv) results in performing the activity structure \(p\) and (v) finally ends in the state \(s'_u\). The meaning of sub-expressions in the definition as follows: (i) \(s_u\) is unstable, (ii) there are no active termination states in the configuration, (iii) the set of transition conglomerates that have no triggers and can be fired (with respect to the actual configuration and extended variable evaluation) is \(TC_I\) as returned by the function \(\text{Fireable}\), (iv) \(TC_I\) is a non-empty set, (v) firing the set \(TC_I\) results in leaving the set of states \(S_I\) performing the activity structure \(p\) and entering the set of states \(S_e\) as returned by the function \(\text{Fire}\) and (vi) the labeling of the target state \(s'_u\) indicates that it represents the configuration obtained by removing \(S_I\) from the original configuration and adding \(S_e\), the extended variables are updated according to \(p\) and \(s'_u\) is still unstable.

**Closing a run-to-completion step** \(\text{close} \) represents closing a run-to-completion step (Eq. [1.122]). Members of the tuple \((s_u, (t_\emptyset, \emptyset, p_\emptyset), s_s)\) indicate that the step (i) starts in the unstable state \((s_u)\), (ii) consumes no triggers \((t_\emptyset)\), (iii) fires no transition conglomerates \((\emptyset)\), (iv) results in performing no activities \((p_\emptyset)\) and (v) finally ends in the stable state \(s_s\). The meaning of sub-expressions in the definition as follows: (i) \(s_u\) is unstable, (ii) there are no active termination states in the configuration, (iii) there are no transition conglomerates that have no triggers and can be fired as returned by the function \(\text{Fireable}\) and (iv) the labeling of the target state \(s_s\) indicates that it represents the same configuration as \(s_u\), the step does not modify the evaluation of extended variables and \(s_s\) is stable.

**Termination** \(\text{term} \) represents termination of the state machine (Eq. [1.123]). Members of the tuple \((s_u, (t_\emptyset, \emptyset, p_\emptyset), s_\omega)\) indicate that the step (i) starts in the unstable state \((s_u)\), (ii) consumes no triggers \((t_\emptyset)\), (iii) fires no transition conglomerates \((\emptyset)\), (iv) results in performing no activities \((p_\emptyset)\) and (v) finally ends in the terminated state \(s_\omega\). The meaning of sub-expressions in the definition are as follows: (i) \(s_u\) is unstable (ii) at least one termination state is active in the current configuration and (iii) the labeling of the target state \(s_\omega\) indicates that it represents the same configuration as \(s_u\), the step does not modify the evaluation of extended variables and in \(s_\omega\), the state machine is terminated.

**Dropping a trigger** \(\text{drop} \) represents dropping a trigger (Eq. [1.124]). Members of the tuple \((s_s, (t, \emptyset, p_\emptyset), s'_s)\) indicate that the step (i) starts in the stable state \((s_s)\), (ii) consumes the trigger \(t\), (iii) fires no transition conglomerates \((\emptyset)\), (iv) results in performing no activities \((p_\emptyset)\) and (v) finally ends in the stable state \(s'_s\). The meaning of sub-expressions in the definition are as follows: (i) \(s_s\) is stable (ii) there are not transition conglomerates that can be fired (with respect to the to the actual configuration, extended variable evaluation and trigger \(t\)) and (iii) the labeling of the target state \(s'_s\) indicates that it represents the same configuration as \(s_s\), the step does not modify the evaluation of extended variables and \(s'_s\) is also stable.

Below we formally define the functions used in the definitions above i.e., (i) \(\text{Init}\) for obtaining the activity structure to be performed and states to be entered during the initialization, (ii) \(\text{Fireable}\) for collecting the transition conglomerates that are to be fired with respect to a configuration, evaluation of extended variables and trigger and (iii) \(\text{Fire}\) for calculating the activity structure to be performed, states to be left and entered when firing a set of transition conglomerates.

**Definition 41 (Initialization).** The function \(\text{Init}\) returns a pair \((p, S_e)\) where \(p\) is a PERT graph describing the activities to be performed during initialization and \(S_e\) is the set of states entered:

\[
\text{Init}() = \text{EnteringHierarchy} (\text{InitialHierarchy}())
\] (1.125)
1.8. OPERATIONAL SEMANTICS

Definition 42 (Collecting Transition Conglomerates to be Fired). The function \texttt{Fireable}(C, eval, t) (Listing [1.1]) returns a set of transition conglomerates that are to be fired when processing the trigger \( t \) in the configuration \( C \), in case of the \texttt{eval} evaluation of extended variables. The function first collects the \texttt{enabled} transition conglomerates (2), selects the subset of them that are \texttt{not overpowered} by another enabled transition conglomerates of higher priority (3) and before returning this set ensures that there are no conflicting pairs in it (4). (If a conflict is found after resolving the priority relations the model is considered to be ill-formed and a runtime error occurs.) In the definition we used functions (i) \texttt{Enabled} for collecting enabled transition conglomerates, (ii) \texttt{NonOverpowered} for deselecting the transition conglomerates that are overpowered by some priority relations and (iii) \texttt{ConflictFree} for checking whether the finally calculated set is free from conflicts.

The function \texttt{Enabled} (Eq. 1.126) for a \( C \) configuration (set of active states), an \( e \) evaluation of variables and \( t \) trigger collects those transition conglomerates whose (i) enabling states are active in \( C \), (ii) trigger is \( t \) and (iii) guard evaluates to true for \( e \).

An enabled transition conglomerate \( tc^l \) in the set \( TC_{en} \) is considered to be \texttt{overpowered} by another \( tc^h \) in the set if \( tc^h \) is of higher priority than \( tc^l \) (\( tc^l \prec tc^h \)). The function \texttt{NonOverpowered} (Eq. 1.127) collects those transition conglomerates from the set \( TC_{en} \) that are not overpowered.

The function \texttt{ConflictFree} (Eq. 1.128) checks the \( TC_{chk} \) set of fireable transition conglomerates and returns \( \top \) if there are no conflicting pairs in \( TC_{chk} \) and \( \bot \) otherwise.

\[
\begin{align*}
\text{Enabled}(C, e, t) &= \{ tc | tc \in TC : (\text{TCEnabling}(tc) \subseteq C) \land (\text{TCTrigger}(tc) = t) \land e \models \text{TCGuard}(tc) \} \quad (1.126) \\
\text{NonOverpowered}(TC_{en}) &= \{ tc | tc \in TC_{en} : \exists tc^h \in TC_{en} : tc^h \prec tc^l \} \quad (1.127) \\
\text{ConflictFree}(TC_{chk}) &= \begin{cases} 
\top & \text{if } |\{tc_1, tc_2\}|\{tc_1, tc_2\} \subseteq TC_{chk} : tc_2 \in TCAllInConflict(tc_1) \\
\bot & \text{otherwise}
\end{cases} \quad (1.128)
\end{align*}
\]

Definition 43 (Firing Transition Conglomerates). Firing a transition conglomerate \( tc \) in the configuration \( C \) (i) starts with recursively leaving the top sources of \( tc \) (ii) then performing the effect of \( tc \) and (iii) finally entering the target hierarchies of \( tc \). The algorithm in Listing [1.2] performs these steps using the functions \texttt{LeavingHierarchySet}, \texttt{TCEffect} and \texttt{EnteringHierarchySet} defined previously and finally appends the PERT graphs according to the precedence scheme. The function returns a \((p_{full}, S_{leave}, S_{enter})\) tuple where \( p_{full} \) is a PERT graph describing the entire activity, \( S_{leave} \) is the set of states left and \( S_{enter} \) is the set of states entered. In Listing [1.3] we extend \texttt{FireSingle} below for \( tcSet \) sets of transition conglomerates fired in parallel (function \texttt{Fire}).

Having defined the KTS and the related functions we have presented the formal operational semantics of UML 2.0 statecharts in a declarative way. The next subsection presents those algorithms that organize the declarative definitions into easy to implement imperative algorithms, but before doing this we present some examples that illustrate the operation by the concepts of the underlying KTS.
The examples describe a scenario of the same statechart. The blue rectangles with the picture of the statechart represent states of $B^{\text{KTS}}$; the name of the actual KTS state is indicated in braces in the bottom left corners (e.g., $S = \{s_a, s_b, s_c, \ldots\}$); the actual configuration of the statechart is indicated by highlighting the active states by blue lines, e.g., in state $s_a \in S$ (Fig. 1.19) the states $s_1$ and $s_2$ are active (note that $\{s_1, s_2\} \subseteq \text{State}$). In order to avoid confusion of KTS and statechart states the states of the KTS are subscripted by letters and the special symbol $\alpha$ (e.g., $s_\alpha, s_\beta, s_\gamma, \ldots$), while the states of the statechart are subscripted by numbers (e.g., $s_1, s_2, \ldots$). The state labels are indicated in orange callouts above states (e.g., in case of $s_a$: $L_\text{Eval}(s_\alpha) = \{s_1, s_2\}$ indicates the actual configuration, $L_\text{Eval}(s_a) = \{s_1, s_2\}$ assigns the value 0 to the single extended variable $v_1$ and the phase of $s_a$ is unstable). Steps of $B^{\text{KTS}}$ are indicated by blue arrows: the type of the step is indicated above the arrow (e.g., the one between $s_a$ and $s_b$ is an initialization step); the transition labels are indicated by the tuple below the arrow (e.g., $(s_\alpha, (t_q, 0, p), s_a)$ between $s_a$ and $s_a$); the PERT graph of activities is indicated in the callout below the tuple using the usual notation (e.g., the $p$ PERT graph of activities to be performed in the step between $s_a$ and $s_b$ consists of three nodes $X_{s_a} \subseteq \text{TC}$ and $E_{s_b}$ corresponding to the exit activity of $s_2$, the effect of $tc_1 \in \text{TC}$ and the entry activity of $s_3$ respectively). The transition conglomerates $\text{TC} = \{tc_1, tc_2, \ldots\}$ are marked by green callouts at $s_a$ in Fig. 1.19. The a single extended variable $v_1$ is referenced in the effect of $tc_3$ and in the guard of $tc_4$. Below we present examples for initialization and trigger processing; examples for dropping a trigger and termination can be found in App. A.7.

Example 15 (Initialization). Fig. 1.19 presents the initialization of the state machine. The uninitialized status of the statechart corresponds to the $s_a$ state of $B^{\text{KTS}}$; no states are active ($L_\text{Eval}(s_\alpha) = \emptyset$), the evaluation of extended variables assigns 0 to $v_1$. In the uninitialized phase only the initialization step may be performed (Eq. 1.119) that takes $B^{\text{KTS}}$ from $s_a$ to $s_a$, not triggered by any triggers ($t_q$), by firing no transition conglomerates ($\emptyset$) and performing first the entry activity of $s_1$ then the entry activity of $s_2$. The resulting $s_a$ state is unstable. In an unstable phase $\xrightarrow{\text{inter}} \xrightarrow{\text{close}} \xrightarrow{\text{term}}$ steps can be performed; since there are no active termination states and there is a fireable transition conglomerate without trigger ($tc_1$) an internal step is performed taking $B^{\text{KTS}}$ to $s_b$ by firing $tc_1$, performing the exit activity of $s_2$, the effect of $tc_1$ and the entry activity of $s_3$. The resulting configuration is $\{s_1, s_3\}$.
and $s_b$ is still unstable. Since there are no active termination states in $s_b$ and there are no fireable transition conglomerates, the run-to-completion step is closed resulting in the stable state $s_c$.

**Example 16 (Processing a Trigger).** Fig. 1.20 presents the entire processing of a trigger $t_1$ received in the state $s_a$ (the last state of the previous example). The reception of the trigger is indicated by the red lightning in the top right corner of $s_c$. Since $t_1$ triggers $t c_2$ a run-to-completion step is opened, taking $B^{KTS}$ to the state $s_d$. As indicated in the callout under the arrow, the step is initiated by the trigger $t_1$, involves the firing of $t c_2$ and the activity structure to be performed is as follows: first the exit activities of states left are performed staring with the one at the higher level of the hierarchy ($X_{s_3}$ and $X_{s_1}$), then the effect of $t c_2$ is performed, then the entry activities of target states starting with $s_4$ then the concurrent substates in parallel ($E_{s_3}$ and $E_{s_2}$). In the $s_d$ unstable state $t c_3$ is fireable but $t c_4$ is disabled by its guard, therefore in the next internal step only $t c_3$ is fired resulting in the state $s_e$. (Note the new value assigned to $v_1$. The effect of $t c_3$ assigned 1 to the single extended variable $v_1$ making $t c_4$ fireable in $s_e$ therefore in the next internal step $t c_4$ is fired resulting in $s_f$. Obviously $s_f$ is still unstable but since there are no more fireable transition conglomerates (and there are no active termination states) the run-to-completion step is closed resulting in $s_g$.

### 1.8.3 High-Level Algorithms

Collecting states of the Kripke transition system into groups by the phase of the operation (i.e., phases uninitialized ($P h s_u$), stable ($P h s_s$), unstable ($P h s_u$) and terminated ($P h s_t$)) results in a high-level view of the operation shown as a state machine in part a of Fig. 1.21. Transitions of the “high-level state machine” representation of the Kripke transition system are triggered by the triggers received by $B^{KTS}$, guards are derived from the definitions of $⇒$ subsets:

\[ g_{open} : \text{Fireable}(L^5_{\text{Cut}}(s_u), L^5_{\text{Eval}}(s_u), t) \neq \emptyset \]  
\[ g_{inter} : ((\text{TerminationState} \cap L^5_{\text{Cut}}(s_u)) = \emptyset) \land \text{Fireable}(L^5_{\text{Cut}}(s_u), L^5_{\text{Eval}}(s_u), t) \neq \emptyset \]  
\[ g_{close} : ((\text{TerminationState} \cap L^5_{\text{Cut}}(s_u)) = \emptyset) \land \text{Fireable}(L^5_{\text{Cut}}(s_u), L^5_{\text{Eval}}(s_u), t) = \emptyset \]
The corresponding step types are indicated as transition effects in the figure, e.g., the subset \( \text{open} \) contains tuples in the form \((s_t, (t, TC_t, p), s_u)\); it is easy to see that the corresponding transition of the high-level state machine is (i) triggered by \( t \), (ii) the source state is in the stable phase \((\text{Phs}_{s})\) as required by the \( L_{\text{Phs}}(s_t) = \text{Phs}_{s} \) sub-expression in the definition, (iii) the guard predicate is \( \text{Fireable}(L_{\text{Cond}}(s_t), L_{\text{Eval}}(s_t), t) \neq \emptyset \) (derived from the definition) and the (iv) effect of the transition represents the assignments in the definition (the effect is indicated as \( \text{open} \) for simplicity).

As an event-triggered system, the high-level state machine can be seen to be waiting for the reception of a trigger and performing a trajectory in the state space as response until a stable state is reached; stable states are the ones from which no transitions originate or only such transitions originate that have explicit triggers. It is easy to see that these states correspond to stable and terminated phases \((\text{Phs}_{s} \text{ and Phs}_{s_{o}} \text{ respectively})\). Two scenarios can be identified representing initialization and trigger processing – these scenarios are the equivalents of the UML run-to-completion concept:

- The initialization of the statechart (i) starts in \( \text{Phs}_{s_{o}} \), then (ii) steps to \( \text{Phs}_{U} \) (step \( \text{init} \rightarrow \)) and performs any number of \( \text{inter} \) steps, finally ends in (iii) \( \text{Phs}_{S} \) (step \( \text{close} \rightarrow \)) if a non-terminal stable configuration is reached or (iv) in the phase \( \text{Phs}_{s_{o}} \) (step \( \text{term} \rightarrow \)) if a terminal configuration is reached. Note that this scenario is not explicitly triggered by any events, only implicitly by initializing the statechart. This scenario is shown in part \( b \) of Fig. 1.21

- The trigger processing step of the statechart (i) starts in \( \text{Phs}_{S} \); the operation is initiated by the reception of a trigger, depending on the number of transition conglomerates found to be fireable the next phase is (ii) \( \text{Phs}_{S} \) again if no transitions can be fired (i.e., dropping the trigger by step \( \text{drop} \rightarrow \)) or (iii) \( \text{Phs}_{U} \) if at least one transition conglomerate can be fired (step \( \text{open} \rightarrow \)). The statechart may perform any number of \( \text{inter} \) steps while remaining in the phase \( \text{Phs}_{U} \) and finally end the scenario in (iv) \( \text{Phs}_{S} \) if a non-terminal stable configuration is reached (step \( \text{close} \rightarrow \)) or (v) in \( \text{Phs}_{s_{o}} \) (step \( \text{term} \rightarrow \)) if a terminal configuration is reached. This scenario is shown in part \( c \) of Fig. 1.21

Definition 44 (Initialization). The imperative algorithm representing initialization of the Statechart (part \( b \) of Fig. 1.21) derived from the definition of the labeled transition relations of the KTS is in Listing 1.4. The algorithm has no input arguments \((1)\). The actual state of \( B^{KTS} \) is maintained in the \text{actState} variable. The algorithm must be started in the uninitialized phase \((2)\): the first statements correspond to the \( \text{init} \rightarrow \) step \((3-7)\) ending in an unstable phase \((7)\). The rest of the algorithm is a loop: while being in the unstable phase \((9)\) the algorithm checks whether a terminal configuration is reached \((10)\) - in this case performs the \( \text{term} \rightarrow \) step \((11-12)\) and the operation terminates; otherwise \((13)\) the fireable transition conglomerates that have no triggers are collected \((14)\); if there are no fireable ones \((15)\) the algorithm performs the \( \text{close} \rightarrow \) step \((16-17)\), otherwise \((18)\) an \( \text{inter} \rightarrow \) step \((19-22)\).
1. InitializationStep()
2. require \( L_{\text{Phs}_{\text{Init}}}^{\text{Init}}(\text{actState}) = \text{Phs}_{\text{Init}} \)
3. // \( \text{init} \)
4. \((p_{\text{Init}}, S_{\text{Init}}) = \text{Init}() \)
5. \( L_{\text{Cnf}}(\text{actState}) = S_{\text{Init}}^{\text{Init}} \)
6. \( L_{\text{Eval}}(\text{actState}) = \text{Issue}(L_{\text{Eval}}(\text{actState}), p) \)
7. \( L_{\text{Phs}}(\text{actState}) = \text{Phs}_{\text{Init}} \)
8. while \( L_{\text{Phs}}(\text{actState}) = \text{Phs}_{\text{Init}} \)
9. \( \text{if} \) \( \emptyset \neq (L_{\text{Cnf}}(\text{actState}) \cap \text{TerminationState}) \)
10. \( \text{// } \text{term} \)
11. \( L_{\text{Phs}}(\text{actState}) = \text{Phs}_{\text{Init}} \)
12. \( \text{else} \)
13. \( \text{TC}_f = \text{Fireable}(L_{\text{Cnf}}(\text{actState}), L_{\text{Eval}}(\text{actState}), t_0) \)
14. \( \text{if} \) \( |\text{TC}_f| = 0 \)
15. \( \text{// } \text{close} \)
16. \( L_{\text{Phs}}(\text{actState}) = \text{Phs}_{\text{S}} \)
17. \( \text{else} \)
18. \( \text{// } \text{inter} \)
19. \((p_{\text{Inter}}, S_{\text{Inter}}, S_{\text{Inter}}) = \text{Fire}(L_{\text{Cnf}}(\text{actState}), \text{TC}_f) \)
20. \( L_{\text{Cnf}}(\text{actState}) = (L_{\text{Cnf}}(\text{actState}) \setminus S_{\text{Inter}}) \cup S_{\text{Inter}} \)
21. \( L_{\text{Eval}}(\text{actState}) = \text{Issue}(L_{\text{Eval}}(\text{actState}), p) \)

Listing 1.4: Pseudocode of the InitializationStep Method

Definition 45 (A Trigger Processing Step). The imperative algorithm representing a trigger processing step (corresponding to the scenario depicted in part c of Fig. 1.21) derived from the definition of the labeled transition relations of the KTS is shown in Listing [1.5]. The single input argument of the algorithm is the actually received trigger \( t \) (1). The actual state of \( B_{\text{KTS}} \) is maintained in the actState variable. The algorithm must be started in a stable or a terminated configuration (2) (but obviously in the latter case no operation will be performed). If the actual state is stable (3) the fireable transition conglomerates are collected that are triggered by \( t \) (4); if there are no fireable ones (5) the trigger is dropped (step \( \text{drop} \)) (6); otherwise (7) an open = step is performed (8–12). The rest of the algorithm is the same loop as the one at the end of the InitializationStep algorithm: while being in the unstable phase (14) the algorithm checks whether a terminal configuration is reached (15) – in this case performs the term = step (16–17) and the operation terminates; otherwise (18) the fireable transition conglomerates that have no triggers are collected (19); if there are no fireable ones (20) the algorithm performs the close = step (21–22), otherwise (23) an inter = step (24–27).

With these two high-level algorithms the operational semantics of precise UML 2.0 statecharts is complete. The imperative approach followed in the definition of algorithms enables the straightforward mapping of behavioral structures to any popular imperative programming languages.

1.8.4 Formal Algorithm Definitions and Algorithmic Complexity

The unambiguous Microsoft AsmL specification of all the algorithms discussed here is presented in App. [A.4] coupled with analysis of algorithmic complexity. Without any details we only mention here that the execution time of both InitializationStep and TriggerProcessingStep is dominated by the number of transition conglomerates in the model, for more details see App. [A.4].

1.8.5 Correspondence to the Informal Semantics of the Standard

Having defined a formal operational semantics it is beneficial to show that our formalism honours the informal semantics discussed of the UML 2.0 standard. Below we will show that the key (informal) requirements of the standard are met by our approach. A similar comparison is presented by Latella et al. for showing that their formal EHA-based semantics \[77, 78\] corresponds to the informal UML semantics.
With respect to the selection of enabled transitions the standard requires that a transition may considered to be enabled if (i) all of its source states are active, (ii) the trigger satisfies the actually received trigger and (iii) guards are true (Cit. [20]). It is easy to see that the definition of Enabled directly implements this selection based on the formally well established background.

Two transitions are considered to be in conflict by the standard if the intersection of the set of states they exit is non-empty; obviously conflicting transitions may not be fired in parallel (Cit. [21]). In our approach the statechart is required to be deterministic thus a pair of conflicting transition conglomerates may not be found to be simultaneously fireable as discussed above. This property is enforced by asserting ConflictFree in Fireable. It is easy to see that the transition conglomerate conflict relation implements UML’s notion of transition conflict.

By the definition of the standard a transition originating from a substate has higher priority than a conflicting transition originating from any of its containing states (Cit. [22]). It is easy to see that our notion of transition conglomerate priority relation implements the same concept.

The set of transitions to be fired is defined by the standard as a maximal set of transitions that satisfies the following conditions: (i) all transitions in the set are enabled, (ii) there are no conflicting transitions within the set and (iii) there is no transition outside the set that has higher priority than a transition in the set (Cit. [23]). Based on the definition of Fireable it is easy to see that (i) only enabled transition conglomerates are considered, (ii) there may be conflicting pairs selected and (iii) based on the definition of NonOverpowered and the transitivity of the transition conglomerate priority relation (proven in Thm. [8] in App. [A.3]) not even the priority rule is violated.

Finally it is important to show that during the entire execution our semantics maintains a consistent configuration where there may be no more than one states simultaneously active in a region and for all regions $r$ refining an active state $s$ there should by exactly one state active in $r$, etc. The detailed discussion is presented in App. [A.5] first we formally define the notion of configuration consistency (Def. [76]), then show that the initial configuration after $init$ is consistent (Thm. [12]), prove that firing a single transition conglomerate or a set of fireable ones preserves the consistency of the configuration (Thm. [13] and Thm. [14] respectively) finally the statement is summarized in Thm. [15].
1.9 Conclusions and Future Work

This chapter has presented a formal operational semantics for UML 2.0 statecharts. The introduction presented an overview on behavior modeling facilities for event-driven state-based finite state-transition systems focusing on statecharts; the discussion concluded that there have been no approaches published in the literature that are based on an unambiguous formalism, support the entire modeling toolkit of UML 2.0 statecharts and are based on the concepts of the metamodel. The rest of the chapter was dedicated for discussing the achievement of our research goals. First we separated the fundamental modeling concepts of statecharts from shorthand notations and decided to present a formal operational semantics for fundamental concepts (models built up of this reduced toolkit are called precise statecharts in our approach) and a denotational semantics for shorthand concepts (i.e., presenting a transformation of shorthand notations to composite structures built up of fundamental elements). The solid foundations of our formalization efforts were established by defining the metamodel of precise statecharts and a set of well-formedness rules in Sec. 1.3 and App. A.2 refinement concepts (state containment, refining states to regions etc.) were formally defined in Sec. 1.4 a formalism for compound transition structures was defined in Sec. 1.5. Our approach puts emphasis on modeling parallel execution by defining compound activity structures based on PERT graphs in Sec. 1.6. The new concepts introduced in our discussion are integrated into the metamodel in Sec. 1.7. The operational semantics is defined by a Kripke transition system in Sec. 1.8 the semantics is finally presented by imperative algorithms enabling this way the straightforward implementation in popular programming languages.

There are multiple appendices attached to the main discussion in App. A. App. A.1 collects some key statements from the standard for unambiguous discussion and underpinning our modeling and design decisions; the formal well-formedness rules related to precise statecharts are presented in App. A.2 the functions and algorithms mentioned in this chapter are presented in the unambiguous notation of the AsmL language in App. A.4 along with analysis of algorithmic complexity; some lengthy definitions and theorems were moved from the main part to App. A.3 and App. A.5 respectively; the transformation of shorthand notations (e.g., history pseudostates, junction vertices etc.) to the modeling toolkit of precise statecharts is outlined in App. A.6 finally App. A.8 presents an example for using the implementation of this semantics in a simulator.

According to our knowledge our approach is the most complete formal operational semantics for UML 2.0 statecharts with such remarkable features as full formalization of state refinement and compound transition structures, explicit modeling of extended variables and concurrency issues, detailed unambiguous specification of algorithms in AsmL and analysis of algorithmic complexity. The achievements discussed in this chapter are summarized by the following thesis.

Thesis 1 ( Formal Operational Semantics for UML 2.0 Statecharts). I elaborated a formal operational semantics for statecharts of the UML 2.0 modeling language. As compared to other approaches published previously in the literature my solution supports the entire set of practically relevant statechart artifacts, based on a well-established formalism for refinement concepts, uses a high-level formalism for representing possibilities for parallel execution and its inherently pragmatic point of view provides a solid formal foundation for various engineering efforts e.g., automatic code generation, runtime error detection, etc. The thesis is built up of the following parts:

(i) I defined a new formalism for state refinement concepts and state hierarchies; as compared to the informal discussion of the UML standard, my approach is based on an unambiguous formalism and a comprehensive set of well-formedness criteria.

(ii) I defined a new formalism for compound transition structures built up of multiple transitions and pseudostate vertices and defined the corresponding well-formedness criteria.

(iii) I defined a new formalism for compound activity structures built up of multiple atomic activities with precedence relations based on PERT graphs, formally defined the corresponding well-formedness criteria and algorithms.
Based on the formalisms above I defined a formal operational semantics for UML 2.0 statecharts by a Kripke transition system.

(v) Finally I mapped the basically declarative notation of the Kripke transition system to imperative algorithms that can be implemented in popular programming languages on a straightforward way.

This semantics is the formal establishment of our code generation and runtime error detection approaches discussed in later parts of the document thus the semantics obviously appears in our publications about code generation and runtime error detection, e.g.: [124, 127] (conference and journal papers about code generation) and [46] (book chapter about runtime error detection). The most recent version of our semantics is outlined in the mathematical introduction part of [115] (book chapter). In addition to the relatively terse discussions mentioned above, the semantics was explained in-depth in detailed technical reports: the first approach for the formal semantics was published in [132] prepared as a research report (150 pages) for the Hungarian Academy of Sciences (MTA/TKI) in the framework of the Embedded Systems Research Group at Budapest University of Technology and Economics, Department of Measurement and Information Systems. The metamodel, the well-formedness rules and transformation of shorthand notations to fundamental concepts were presented in a research report [119] (50 pages, in Hungarian) prepared in the framework of the “GVOP-3.1.1-2004-05-0523/3.0” project; the semantics is discussed in the first part of another research report [136] (about 40 pages from the entire document of 100 pages, in Hungarian). The most recent semantics focusing on precise statecharts only was presented in [135] (technical report of about 40 pages). As this semantics was the basis of multiple tool developments even involving students of the university, the metamodel of precise statecharts with the discussion of some tool interoperability issues was presented in another document [121] (15 pages).

All the algorithms, well-formedness rules and data structures discussed here were implemented in the Microsoft Abstract State Machine Language (AsmL) as a library to be used by simulator, code generator and runtime error detection tools. AsmL is a modern executable specification language developed for the Microsoft .Net platform aimed at specification of dependable software applications, model checking and fast prototyping. The definition of all the algorithms defined or referred to in this chapter are presented as AsmL source code in App. A.4. The final implementation of the semantics as a library is about 6200 of AsmL source (250kB).

The most important application possibilities of achievements are discussed in later parts of this document: the semantics is the formal foundation of automatic code generation discussed in Chp. 2 and runtime error detection approaches presented in Chp. 3. We also built a basic statechart simulator based on the library implementing the semantics. Our future plans include the development of a model-checker tool based on this statechart semantics and a visual simulator integrated into the graphical interface of a UML modeling tool.

The discussion of the semantics identified some points where further research efforts may be focused. Our semantics in its current form does not support deferred triggers and “do activities” attached to states and considers triggers as plain enumerated entities without refinement possibilities. The integration of deferred triggers will require the formalization of event pool handling e.g., by FIFO queues. The notion of do activities seem to be contradictory to other parts of the UML statechart semantics, since the execution of state entry and exit activities and transition effects takes place on trigger processing i.e., during state transitions similarly to the operation of Mealy automata while do activities are bound to states according to a Moore automaton semantics. Trigger refinement is supported by the notion of signal events in UML: a signal event contains a signal instance that is a generalizable element; our semantics can support signal events organized into a refinement hierarchy by a slight extension of the algorithm that selects enabled transitions. Another possible direction of future research can be the the definition of a formal operational semantics for activity diagrams; since activity diagrams are quite close to statecharts, the definition of a similar semantics should require a relatively modest effort.
Chapter 2

Automatic Implementation of UML 2.0 Statecharts

2.1 Introduction

Due to the widespread application of visual modeling languages, naturally emerges the need for automatic implementation of elaborated models. The expected benefits from the substitution of time consuming and labor intensive manual programming with automatic tools are increasing productivity and considerable enhancement of code quality. Most of modern modeling environments provide some support for automatic implementation of static aspects of software models (e.g., automatic generation of data types, classes, interfaces or function signatures in various programming languages, definition of database schemes etc.) – these facilities are obviously based on the interpretation of data models (class, package or deployment diagrams). Support for implementation of dynamic models (activity, collaboration or statechart diagrams) is provided only by a very few number of professional tools probably because while there is a one-to-one correspondence between data modeling concepts and object-oriented programming features, the implementation of behavior is actually writing programs i.e., a complex, creative process for which nearly infinite number of (correct) solutions can be found. Despite of the difficulties the automatic implementation of visual behavioral models is an important and up-to-date problem, since the manual implementation of complex event-driven state-based behavior is a highly labor-intensive error-prone task and even a minor modification in the model may require significant modification of the source code.

This chapter presents a method for implementation of behavior specified by UML 2.0 statecharts. Our approach is focused on resource constrained embedded platforms where both the processing power and the available memory may be seriously restricted. Our proposal is organized according to the Model Driven Architecture initiative of the Object Management Group as we clearly separate platform-independent modeling (PIM), platform-specific modeling (PSM) and implementation steps. This process involves the two step transformation of the original statechart semantics through an intermediate representation to source code and proving the correctness of these transformations.

As the implementation of complex behavior specified by statecharts is definitely a non-trivial issue, multiple solution proposals has been published in the literature or implemented in commercial tools. First we present an overview including the most fundamental techniques, an introduction to capabilities of commercial solutions and the best-known approaches published in the literature.

The two best-known fundamental techniques for implementation of traditional state machines (Mealy or Moore automata without state refinement) are the nested switch statement and the state table techniques. The nested switch implementation is based on two branches in the control flow according to the actual state and the actually received trigger. Both branches are usually implemented by the switch construct of the ANSI-C language, hence the name of the pattern (see App. B.1 for a detailed example). States and triggers are represented by identifiers typically implemented by ANSI-C
**enum** constructs. The solution does not provide any explicit means for representation of state hierarchies: the pattern was intended for implementing flat state machines; state hierarchies are to be flattened for being handled by this pattern and the possibly complex activities to be carried out in case of transitions that involve leaving and entering multiple states are to be implemented manually within appropriate **case** blocks of the internal **switch** statements – unfortunately this is on one hand a complex error-prone task on the other hand may result in exponential code growth and repeated implementations of activities of composite states. The solution is used in the I-Logix Rhapsody tool.

The **state table** pattern translates the state machine into a state-transition table: the rows of the table correspond to states, the columns correspond to triggers; a cell of the table corresponding to state $s_i$ and trigger $t_j$ represents the response of the state machine to the reception of $t_j$ in state state $s_i$; the cell contains a pointer to a function implementing the activity to be carried out (may be empty) and the identifier of the state to be stepped to (may be the original state if no transition occurs). The state-transition table is typically implemented by a two dimensional array (number of states $\times$ number of triggers), see App. B.1 for a detailed example. Similarly to the nested switch approach the state table solution does not provide any explicit means for representation of state hierarchies either; the same workaround may be applied as discussed above (flattening the hierarchy, implementation of multiple activities in functions referenced by table entries etc.). According to [165] a similar approach is used in the commercial tool VisualSTATE from IAR Systems.

Although the capabilities of the nested switch and the state table patterns are about the same there are some benefits and drawbacks to be highlighted with respect to the two solutions. The nested switch approach typically requires less memory since in the direct implementation as **switch** constructs the programmer has enough control to implement only the necessary routines (e.g., if a trigger is not handled in a specific state, the corresponding internal **case** edge needs not be written) but the performance of the **switch** structures depend on the number of states and triggers. The **state table** approach features very fast constant time selection of the appropriate table cell (a single indirect addressing) but the memory required for storing the table can get too large, especially if the number of states is high (large table) but there are relatively few relevant state–trigger pairs (transitions, activities etc.) – in this case the table will become large but mostly unused. Another interesting difference to be highlighted is the relation of code and data sections: the nested switch approach implements the state machine by explicitly writing **code** (**switch** structures), while the state table solution uses a **generic interpreter** that processes the input according to the state table stored as initialized constant **data**.

The third approach considered to be another **fundamental technique** is the **State design pattern** proposed by the “Gang of four” (Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides) in their well-known book about software design patterns [47]. The State design pattern does not provide more features than the previous two approaches but aims at wrapping the solution into a more object-oriented structure: states are mapped to classes derived from the same abstract base (e.g., **AbsState** base and **WaitingState** derived from it); the derived classes are typically singletons. Trigger processing activities are mapped to member functions declared in the base class and implemented in derived classes (e.g., **AbsBase.onKeyPress**). The context class maintains its state by a pointer referring to the actual state (e.g., **AbsState *myState = WaitingState::instance();**). The behavioral polymorphism (responding to various events according to the actual state) is achieved by calling functions declared in the abstract base (e.g., **myState -> onKeyPress()**) thus the state-dependent concrete behavior is selected by the virtual member function resolution mechanism [42]. Unfortunately the State design pattern does not provide explicit means for handling state refinement either (see App. B.1 for a detailed example).

Thanks to the popularity of abstract modeling languages and OMG’s Model Driven Architecture initiative [52] the automatic implementation of visual models has gained an increasing attention recently and multiple more or less elaborated **commercial solutions** have been offered. Although most of UML modeling tools provide some sort of automatic code generation, in most cases this is restricted to mapping class structures to various programming languages or database tables. An older version
of the Poseidon for UML modeling tool from Gentleware contained an optional “Statechart-to-Java” plug-in that was capable of adding functions signatures to the declarations of classes corresponding to the triggers possibly received (similarly to the State design pattern) but without the actual implementation of any control logic. The iUML modeling environment from Kennedy Carter provides full automatic implementation of xUML (executable UML) models, even statecharts – unfortunately the xUML version of “statecharts” are practically flat state machines without any hierarchy.

According to our knowledge there are two fully elaborated commercial solutions for automatic implementation of software specified by statecharts: VisualSTATE from IAR Systems and the Rhapsody family from I-Logix. As discussed in [165] VisualSTATE uses an implementation strategy based on flattening state hierarchies and implementing them as a set of parallel Mealy machines using a pattern similar to the state table approach. Rhapsody and Statemate implement statecharts using a strategy similar to the nested switch solution. Obviously as both these tools are commercial environments their exact code generation method is not public.

The difficulty of implementing statecharts has also caught the attention of the research community and multiple solutions have been proposed in the literature. Blech, Glesner and Leitner published a paper [17] aiming at formal verification and Java code generation based on UML statecharts using the Fujaba modeling environment and the Isabelle theorem prover [108]; unfortunately their discussion is restricted to implementation of flat state machines by the nested switch pattern.

Duby [36] presented a slight improvement of the state table solution by proposing an idea for implementing do-activities (activities to be performed while being in a state) but unfortunately the paper does not solve any issues of the original approach (missing support for state refinement, etc.).

Sane and Campbell [146] presented detailed discussion on incremental construction techniques for achieving code reuse in the State design pattern; the technique uses subclassing, composition, delegation and generalization to incrementally modify and combine simpler machines but still does not support state refinement. Ali and Tanaka [3] discusses an extension of the State design pattern with state refinement, history mechanism and concurrency; unfortunately the implementations of ever complex artifacts are introduced in a more and more intuitive manner without providing unambiguous formal algorithms for translating statechart to source code. The most important drawback of the solution is that it does not handle entry and exit activities according to the standard (e.g., only the exit and entry activities of the innermost states are performed when firing transitions).

There were proposals published by the model checking community primarily aiming at proven correct implementation of formally analyzed models. Sekerinski and Zurob presented a detailed discussion [149] on the implementation possibilities of statecharts and illustrated their approach by translating statechart to the Abstract Machine Notation of the B method [1]. Their support for non-determinism and state refinement are remarkable features of the approach but still lacks the history mechanism, junctions and choices etc. and totally neglects the implementation of any activities. In [15] Björklund, Lilius and Porres outlines a method for translating statecharts to the SMDL intermediate language (Statechart Description Language [14]); the idea of the paper is to discover the control flow graph of the intermediate SMDL representation and implement the graph in ANSI-C or even directly in assembly. The authors claim that the high-level optimization possibilities on the control flow graph are superior to any compiler level optimizations and allow very efficient implementation; unfortunately the paper is mainly about SMDL, its translation to the control flow graph, the optimization and the efficient implementation are not obvious from the discussion. Knapp and Merz published a paper [71] about model checking and code generation for UML state machines and collaboration diagrams. Their approach uses the Hugo/RT model translator [84] for generating Java code; the method is claimed to support all UML modeling constructs but unfortunately the paper does not discuss any detail of the implementation strategy and even the authors admit that the method is “not aimed to produce product-quality, optimized code”.

The key value of proposals originating from the model checking domain is the declared goal of

---

constructing proven correct code ensuring the semantic equivalence of the model and the resulting application. Unfortunately the resulting “code” is typically not of a well-known programming language but expressed in languages aimed to be used primarily for model checking therefore the wide gap between the formally well established modeling languages and the practical programming languages seriously limits the applicability of these approaches.

There are practical approaches that are capable of supporting state refinement even preserving the hierarchy in the runtime representation of the statechart (i.e., without flattening or similar transformation). Behrmann, Kristoffersen and Larsen [11] outlined their proposal for using the state table approach by organizing multiple tables into a refinement hierarchy; unfortunately the paper does not discuss any details of the solution. Wasowski presented an in-depth discussion [165] on a highly efficient representation of state hierarchies in a tree structure (the statechart semantics used in the paper is the variant used by the VisualSTATE tool [166]). His approach supports state refinement, concurrency and history mechanisms; the drawback of the paper that the main focus is on the efficient encoding of the state hierarchy, the actual algorithms that process these structures are not presented.

The most frequent negative observation about the papers mentioned above was the lack of necessary detail of discussion; clearly: a conference paper of 8–20 pages is simply too short for discussing the semantics in enough detail, unambiguous definition of the implementation method, proving the correctness and presenting the results of an experimental evaluation campaign. This severe limitation does not apply to the Quantum Hierarchical state machine (QHsm) pattern proposed by Miro Samek since the author first outlined the idea in an on-line journal article [145] with detailed code samples attached then dedicated an entire book [144] for the discussion. Below we take a deeper look at this pattern because we will use it in comparisons for evaluating our approach.

The QHsm pattern is illustrated on the example of a watch [145] in Fig. 2.1. The watch can show the actual time or date (substates time and date respectively in state display) and can be set (state setting) – the actual month, day, hour and minute are to be set individually (substates month, day, hour and minute respectively). There are two buttons on the watch: mode and set. In display state the mode button is used for switching between showing the time or the date while during setting the watch the mode button is used for increasing the value of the actually adjusted value by one. The set button is used for switching to setting mode and circulating between the four values to be adjusted. The third trigger used in the example is the tick event emitted by the internal clock generator; this event triggers the internal transition in state display.

The base class of the QHsm pattern is the QHsm abstract class; QHsm is responsible for maintaining the actual state of the state machine, initialization, dispatching triggers and performing transitions (member variable myState, and methods init, dispatch and tran respectively). Actual state machine classes (e.g., Watch) are derived from QHsm, states are represented by member functions (called “state functions” here) of derived classes (e.g., display, setting, time, etc. corresponding to states display, setting, time, etc. respectively); the type of pointers to state functions is QState. Processing triggers is implemented by handshaking of state functions: a state function returns NULL if the actual trigger was successfully processed, otherwise the address of the state function corresponding to its parent state,
2.1. INTRODUCTION

```c
QState Watch::time(QEvent const *e) {
    switch (e->sig) {
    case MODE_SIG:
        Q_TRAN(&Watch::date);
        return static_cast<QState>(0);
    }
    return (QState)&Watch::display;
}
```

Listing 2.1: Implementation of the Time State

```c
QState Watch::display(QEvent cont *e) {
    switch (e->sig) {
    case Q_INIT_SIG:
        Q_INIT(&Watch::time);
        return static_cast<QState>(0);
    case Q_ENTRY_SIG:
        show();
        return static_cast<QState>(0);
    case SET_SIG:
        Q_TRAN(&Watch::setting);
        return static_cast<QState>(0);
    }
    return (&Watch::top);
}
```

Listing 2.2: Implementation of the Display State

e.g. the implementation of state function `time` is shown in Listing 2.1: it is easy to see that the state directly handles the trigger `mode` (4) resulting in a state transition to `mode` (5) and returns NULL (6), in any other cases the address of the parent state handler function (state `display`) is returned (8).

Bodies of state functions are implemented typically by `switch` constructs branching according to the type of the trigger received (stored in the `sig` field of class `QEvent`) as shown in the implementation of state `display` (Listing 2.2). There are four reserved trigger types: (i) `Q_INIT_SIG` is used for initialization: the function has to call the `Q_INIT` macro with the address of the state function corresponding to the initial substate of the state in case of compound states (4–6), (ii) `Q_ENTRY_SIG` and `Q_EXIT_SIG` are used for performing the entry and exit activities of states respectively (e.g., displaying the date or time in the entry activity of state `display` (7–9)) and (iv) the unnamed 0 value (an invalid trigger type) is used for forcing the state function to return the address of the parent state (14).

The convenience macro `Q_TRAN` is used for performing state transitions, e.g., from state `display` to `setting` (11) in Listing 2.2. The macro is responsible for (i) discovering the LCA of the source and target states by providing the invalid 0 value as trigger type to the state functions forcing them to return the parent states until the first common parent state is found, (ii) discovering the implicitly entered states (initialization) by providing `Q_INIT_SIG` to the target state function and (iii) providing `Q_EXIT_SIG` and `Q_ENTRY_SIG` signals to the appropriate functions for performing exit and entry activities.

As performing a transition this way requires quite a lot of runtime hierarchy discovery and calling multiple functions, the QHsm pattern provides a cached version of `Q_TRAN` that after performing a transition stores the pointers of functions called (i.e., the hierarchy discovered) in a static pointer array and at the next call no runtime discovery will be performed.

The QHsm pattern is extended to an entire programming framework (the Quantum Framework [144]) and integrated to multiple embedded operating systems. The pattern was reported to be successfully applied in various embedded environment and is widely referred in publications presenting practical approaches. Indeed: the QHsm pattern is by far the most practical and best elaborated proposal for implementation of statecharts in ANSI C/C++ languages with modest resource consumption and relatively large supported modeling toolkit. The drawbacks of the pattern are the lack of direct support for concurrent operation, history mechanism, branching pseudostates, local transitions and
slightly non-standard transition semantics (transition effect can be performed before the first entry activity or after the last exit activity but not between them as required by the standard).

Due to the success of the pattern, multiple papers have been published aiming at adding support for missing constructs or tuning some implementation aspects. A similar approach was published by Babitsky [10] by the generalization of the State design pattern using modern C++ features. Another recent solution was proposed by Heinzmann [61] that reduces the runtime overhead of transition discovery by compile-time inlining and optimization achieved using the most advanced template programming features of up-to-date optimizing C++ compilers. One of our first papers on the implementation of statecharts [127] also suggested some improvements of the QHsm pattern for supporting the history mechanism and standard-compliant implementation of transition effects.

Above we have discussed the fundamental methods and advanced approaches published in the literature about the implementation of statecharts and presented a short overview on the capabilities of commercial tools. Our observations can be summarized as follows: due to the complexity of the UML statechart formalism there were no proposals published in the literature that support the entire modeling toolkit. There are some very practical solutions that were successfully applied even in resource-constrained environments (e.g., the QHsm pattern) but these approaches lack the well-established semantical background and their correctness has not been proven formally. Solutions of I-Logix and IAR Systems are capable of supporting the entire modeling toolkit and their code generation method is probably validated but being commercial environments their implementation method is not public. For the proven correct automatic implementation of statecharts we have to elaborate a systematic mapping of data structures and algorithms specified in Chp. 1 to source code taking into consideration the key properties of resource constrained embedded platforms. We will present our work organized according to the Model Driven Architecture initiative of the Object Management Group.

According to OMG’s MDA scenario [52] the implementation of a system consists of three key steps: (i) in the platform-independent modeling (PIM) step the abstract model of the systems is prepared focusing only on the goal to be achieved without taking into consideration any constraints enforced or any supporting features of the target platform; (ii) in the platform-specific modeling (PSM) step the abstract concepts of the PIM model are mapped to artifacts of the target platform (e.g., GUI elements of the target windowing system, collection types of the target language or library, etc.) and (iii) in the implementation step the actual implementation of the platform-specific model is prepared. The process can be considered at two abstraction levels: (i) the higher one is the metamodel level where the modeling languages of the specific steps reside, (e.g., the standard UML metamodel in the PIM step, an UML profile in the PSM step and the metamodel of the implementation in the final step); (ii) the lower abstraction level corresponds to the models in specific steps, i.e., instances of the corresponding metamodels (e.g., a UML model in the PIM step, the same UML model using the profile of the target platform in the PSM step and the source code of the system in the final step).

Figure 2.2: Implementation of Statecharts in the MDA Framework

Considering the implementation of statecharts for resource constrained embedded environments the three steps correspond to (i) modeling the behavior by a precise statechart (PIM), (ii) mapping the
2.2 Overview on Resource Constrained Embedded Systems

This section presents an overview on resource constrained embedded systems, identifies the characteristic architectural properties that have an impact on software development and indicates those parts of the platform-independent specification of statechart semantics (data representation, algorithm specifications) that should be fitted to the chosen platform.

The discussion can also be seen as a sequence of explanations for backtracking the reasons for algorithms to the available resources (PSM) and (iii) the actual source-code level implementation.

The organization of this chapter is illustrated in Fig. 2.3 by zooming the corresponding box of Fig. 1. This introduction (Sec. 2.1) has presented an overview on fundamental techniques for implementation of state based behavior, outlined the capabilities of commercial solutions and discussed the approaches published in the literature. The discussion concluded that due to the complexity of the UML statechart formalism there were no proposals published in the literature that support the entire modeling toolkit hence for the proven correct automatic implementation of statecharts we have to elaborate a novel solution. In order to achieve this in Sec. 2.2 we present an overview on resource-constrained embedded environments where we will identify those parts of the platform-independent metamodel and algorithms that need to be fitted to the restricted resources available on the target platform. We will use the platform-independent syntax and semantics discussed in Chap. 1. The platform-specific metamodel and algorithms are presented in Sec. 2.3. We will introduce a compact representation for state sets, state hierarchies, state hierarchy sets and transition conglomerate sets. The implementation issues are discussed in Sec. 2.4 by presenting the annotated metamodel indicating which ANSI-C constructs are to be used for implementation of various concepts along with the corresponding data type declarations and source code fragments. An overview about the code generation tool based on this theory is presented in Sec. 2.5. Having discussed our method for implementation of statecharts for resource-constrained embedded systems we present the experimental evaluation of performance and resource consumption in Sec. 2.6. We will compare multiple benchmark applications of various complexity built according to our solution and the QHsm pattern. Finally Sec. 2.7 concludes the discussion, highlights the new scientific achievements and outlines the directions of future research.
having to modify the platform-independent semantics: we will see which are the modified parts in the semantics, why were we forced to do these modifications and which are those fundamental requirements in economical and dependability aspects that necessitated the entire mapping effort.

2.2.1 Application Area, Hardware and Implied Architecture

The application area of embedded systems and the related economical and dependability requirements imply a typical hardware structure that is significantly different than the resources available in modern high-end systems. The relatively low processing performance, small memory and dependability requirements result in a set of architectural properties, programming styles etc. that are characteristic attributes of software development for embedded systems.

Embedded systems are special-purpose computers typically applied as intelligent controllers of electronic, mechanic, thermodynamic, chemical etc. processes completely encapsulated in the controlled device and living in direct interaction with their environment. Embedded systems are built into low-cost, mass-produced devices, therefore the cost of embedding a computer in the context environment should be as low as possible in order to reduce the manufacturing cost of the entire system. In case of portable embedded systems or ones that should operate for long time in hard to access physical areas the low energy consumption is also an important requirement. If an embedded system is applied for controlling dangerous physical or chemical processes or should operate for long time without maintenance, several dependability requirements also emerge.

According to the economical considerations above, hardware resources are chosen to be as low as possible: the main computing resource of an embedded system is typically an FPGA, a microcontroller or a low-cost embeddable microprocessor. Relatively slow, small memory is used for reducing the manufacturing cost and energy consumption, typically integrated into the device.

Taking into consideration the physical properties of the environment, various dependability requirements can be defined including resistance to mechanical, chemical, thermal stress or internal errors addressed by fault tolerance mechanisms.

Although the application of FPGAs, microcontrollers and embeddable processors coupled with cheap memory modules considerably reduces the manufacturing costs and energy consumption, the relatively simple circuits have a significant negative impact on the processing performance delivered by the system: the low processing performance and lack of sophisticated memory management renders the hardware incapable of providing advanced multiprocessing services like the ones available in desktop systems. The runtime overhead implied by modern programming languages (Java, C#, etc.) and the corresponding virtual machines (JVM, .Net framework, etc.) typically can not be accepted therefore in most cases only assembly languages and the ANSI-C programming language is available.

Software aspects of dependability are error detection, fault tolerance and real-time operation properties. We focus here on deterministic resource consumption by requiring deterministic memory consumption (heap and stack) and deterministic execution time of algorithms.

The relation of application area, economical requirements, their impact on the hardware and dependability requirements and the implied architecture as discussed above is shown in Fig. 2.4.
As a representative of real-life low-cost embedded systems, the mitmót modular embedded platform [158] was chosen as target platform for our prototype implementation; the hardware consists of an Atmel Mega-128 8-bit microcontroller running at 6 MHz equipped with 4 KByte RAM and 128 KByte flash ROM. The platform has no operating system, the basic GNU tool-chain is available (C compiler, assembler and linker) and a reduced implementation of the standard ANSI-C library. The platform was developed for fast prototyping and education purposes by the Embedded Information Technology Research Group at Budapest University of Technology and Economics.

2.2.2 Design Decisions

The discussion above identified the key characteristics of the platform; below we will enumerate those design decisions that were drawn for meeting the special requirements implied by the architecture.

The organization of data in the memory has an important impact on both the processing power and memory requirements: compact, non-redundant data structures occupy relatively low amount of memory but their run-time processing may be slower; storing data in a redundant, beneficially byte or word aligned organization, application of advanced caching mechanisms etc. can make the run-time data access easier and faster but requires more memory. In case of resource-constrained embedded systems both the processing power and the available memory is significantly constrained therefore a combined approach seems to be most viable: we will use a compact organization scheme for storing read-only data that is accessed at a low frequency, while intensively accessed read-write data will be organized into a CPU-effective high-performance structure.

A common method for reducing the execution time of complex algorithms is the pre-calculation of complex routines where applicable; we will apply this idea in cases where the extra memory required for storing pre-calculated data is not significant.

As multiprocessing will not be available, there is no use of building PERT graphs during run-time, it is enough for us to calculate a single valid execution sequence of compound activity structures.

As object-oriented language features will not be available at the implementation step, the application of inheritance and other object-oriented features should be avoided wherever possible in algorithm specifications. To be more precise, in our case this means that the type of objects should not be used for directly carrying information e.g., in some algorithms termination states are handled specially but they can only be identified by their type in the platform-independent specification; this is a non-ideal data representation since this way we would have to implement type information manually in a non-object-oriented language.

Dynamically allocated objects are stored in the heap area of the memory. As heap-management algorithms are complex pieces of software, their execution time is hard to predict and may even be influenced by the applications’ actual behavior (memory fragmentation etc.) the avoidance of dynamic memory allocation is a usual requirement against software that has to meet hard real-time constraints hence we will put emphasis on the avoidance of dynamic memory allocation in our approach.

Stack memory is used for passing parameters to functions, storing the return address and for automatic storage class variables. The stack requirements of recursive or mutually recursive functions are often hard to predict. Additionally if easily predictable execution times are required recursive organization is also discouraged. In our approach we will eliminate recursion from the algorithm specifications and present deterministic, imperative algorithm specifications that are easy to analyze and implement. The relation of architectural properties and design decisions are shown in Fig. 2.3.

2.3 Platform-Specific Behavior Modeling

This section systematically applies the modifications motivated by architectural properties of resource-constrained embedded systems to the platform-independent statechart semantics finally resulting in the platform-specific semantics. The discussion is structured as follows: first we identify the modeling
2.3.1 Modeling Concepts Affected by Design Decisions

This subsection identifies those concepts in the platform-independent statechart semantics (metamodel and algorithms) that are affected by our design decisions. As mentioned above we will use compact data structures for storing read-only data. This consideration applies to state sets (e.g., states enabling a transition conglomerate), state hierarchies (initial state hierarchy of the statechart), state hierarchy sets (e.g., hierarchy sets possibly left and entered by transition conglomerates) and transition conglomerate sets (priority and conflict relations). Similarly we will use high performance data structures for runtime manipulation of state sets, state hierarchies and transition conglomerate sets.

The sequential implementation of compound activity structures affects those dynamic parts of the behavioral semantics where PERT graphs were built and used: the most obvious cases are transition conglomerate effects – we will substitute general PERT graphs with activity sequences that represent a valid sequential scheduling of the compound activity structure. Although may not be evident for the first sight, even state hierarchies may be affected by the lack of parallel processing capabilities: we store state hierarchies for being traversed by algorithms like \textit{EnteringHierarchy} or \textit{LeavingHierarchy} that build PERT graphs describing the compound activity structure to be performed when entering or leaving the state hierarchy. It is easy to see that it would be useless to build a general PERT graph that would have to be converted to a sequential equivalent in the subsequent step. We will follow a more effective approach by not even building the general PERT graph, just a valid sequential equivalent of it. The most important observation here is the following: although we can not pre-calculate all activity structures related to traversals of state hierarchies (e.g., for properly leaving a state hierarchy we need the actual configuration), we can considerably reduce the information redundantly stored in state hierarchies by understanding that \textit{we do not need the entire hierarchy information} for building valid activity sequences, it is enough for us to have an \( S = (s_1, s_2, \ldots) \) sequence of states for which if \( s_d \) is a descendant of \( s_a \) in the refinement hierarchy we can be sure that \( s_a \) occurs before \( s_d \) in \( S \) \((S = (s_1, s_2, \ldots, s_0, \ldots, s_d, \ldots))\). Having states organized this way, storing a (simplified) hierarchy is the same as storing a set: let \( H \subseteq \text{State} \) be a set representing a (simplified) state hierarchy and we would like to enter this hierarchy, i.e., perform entry activities of states in \( H \) in a valid order i.e., parent states before their descendants. To achieve this, we only have to iterate through \( S \) and if \( s_i \in H \) we will perform the entry activity of \( s_i \). It is easy to see that we will not violate any precedence rules.

Due to lack of object-oriented programming language features we decided not to use the type of objects for explicitly holding information. Taking a look at the platform-independent metamodel (Fig. 1.18) it is easy to see that there are two such constructs in the metamodel: (i) final states are derived from states and (ii) termination states are derived from final states. The algorithms of the formal operational semantics do not handle final states (that are not terminal states) specially, they
are only distinguished by some well-formedness criteria therefore in the platform-specific metamodel we do not have to distinguish them from ordinary states. In contrast, when termination states are entered the execution of the state machine terminates, therefore it is important to explicitly distinguish termination states from non-terminal ones: the simplest solution for this is to introduce an extra association between the PreciseStatechart and State metaclasses indicating terminal states.

To put together the discussions above: (i) transition conglomerate effects are to be represented by sequences of activities instead of generic PERT graphs (role effect of association between TC and Activity), (ii) the initial state hierarchy of the statechart is to be expressed by an ordered set of states as outlined above (role initialStateHierarchy of association between PreciseStatechart and State) (iii) the state hierarchy possibly left and entered by a transition conglomerate are to be expressed by an ordered set of states (roles possiblyLeftStates and enteredStates respectively of associations between TC and State) and (iv) final and terminal state concepts are eliminated, terminal states are explicitly distinguished from ordinary states by an association role (i.e., not only by their type). The differences are indicated in Fig. 2.6 (to be compared to Fig. 1.18). (Note that Fig. 2.6 is not yet the elaborated platform-specific metamodel, just a conceptual illustration.)

The remaining design decisions do not explicitly affect the metamodel but the specifications of algorithms. As we decided to avoid dynamic memory allocation, we will modify some function signatures to store results of computations in output arguments instead of dynamically allocating an object and returning it. The recursive implementation of LeavingHierarchy and EnteringHierarchy functions is to be modified for a non-recursive organization.

The impact of design decisions on the concepts of behavioral modeling is highlighted in Fig. 2.7.
As discussed above we need data structures for effective storage of state sets, state hierarchies, state hierarchy sets and transition conglomerate sets. The entire hierarchy information need not be carried performed when entering or leaving a hierarchy. Taking into consideration the memory consumption – CPU requirement tradeoff outlined above, we have to provide two implementations for data structures: (i) a compact non-redundant representation for storing read-only data and (ii) a redundant but easy to process organization for intensively accessed read-write data instances.

This subsection will introduce a data structure that is capable of storing sets of instances (states and transition conglomerates) withreserving enough hierarchy information for meeting our goals. First we will discuss the idea and the mathematics behind in an abstract style without taking into consideration low-level data organization aspects, then we will present specialized implementations of the core idea for memory-effective storage facilities and CPU-effective runtime data instances.

### Bit Masks

First we introduce a straightforward method for representing $S$ subsets of an $E$ set by vectors of Boolean values. This organization will be called bit masks.

**Definition 46 (Bit Masks).** Let $E$ be a set and $I = \{0, 1, \ldots, (|E| - 1)\}$ is a set of identifiers. We will assign a unique identifier for all $e \in E$ elements by the $L : E \rightarrow I$ function. $L$ is bijective, i.e., establishes a one-to-one correspondence between elements of $E$ and $I$: $\forall i \in I : |\{e | L(e) = i\}| = 1$. Let $L^{-1} : I \rightarrow E$ be the inverse of $L$ i.e., $(L(e) = i) \leftrightarrow (L^{-1}(i) = e)$; obviously $L^{-1}$ is also bijective. Let $V$ be a Boolean array of length $|E|$. We will use the notation $V[i]$ for accessing the Boolean value at index $i$ where $0 \leq i < |E|$. We will use this vector for representing $S \subseteq E$ sets. Let $V^S[i] = \top$ if and only if the element whose identifier is $i$ is in $S$: $V^S[L(e)] \leftrightarrow e \in S$. We will call a Boolean array wrapped in a class as a bit mask: let $S$ be a subset of $E$: $S \subseteq E$. We say that a bit mask represents the set $S$ if the $V^S$ Boolean vector wrapped in the object contains true values only on those indexes that correspond to elements of $S$ i.e., $V^S[L(e)] = \top$ if $e \in S$ and $V^S[L(e)] = \bot$ otherwise. We will use the notation $\text{BitMask}^S$ for indicating that the bit mask represents the set $S$.

**Example 17.** Let the set $E$ contain the following elements: $E = \{a, b, c, d, e, f\}$, let the set of identifiers be $I = \{0, 1, 2, 3, 4, 5\}$ correspondingly (i.e., $I = \{0, 1, \ldots, (|E| - 1)\}$), let the function $L : E \rightarrow I$ assign the identifiers to elements of $E$ according to their alphabetical order: $L(a) = 0$, $L(b) = 1$, etc. Let the $S$ subset of $E$ be the following: $S = \{b, d, e\}$. Let the bit mask $\text{BitMask}^S$ represent the set $S$. In this case the contents of the $V^S$ Boolean array encapsulated in $\text{BitMask}^S$ is as follows: $V = [\bot, \top, \bot, \top, \bot, \bot]$, i.e., storing $\top$ only at indexes of elements $b$, $d$ and $e$.

### Operations on Bit Masks

Having defined the representation of $S \subseteq E$ sets by bit masks, various operations can be defined. Due to space restrictions we discuss here only those operations that are used in the algorithm examples in later parts of this chapter. Detailed definitions with proof of correctness can be found in [134].

**Definition 47 (Obtaining a Value).** Let a bit mask $\text{BitMask}^S$ represent the set $S \subseteq E$. The member function $\text{GetValue}(i)$ returns the value stored at the index $i (0 \leq i < |E|)$ of the wrapped Boolean array (i.e., $V^S[i]$): $\text{BitMask}^S, \text{GetValue}(i) = V^S[i]$.

**Definition 48 (Setting a Value).** Let a bit mask $\text{BitMask}^S$ represent the set $S \subseteq E$. The member function $\text{SetValue}(i, b)$ sets the value stored at the index $i (0 \leq i < |E|)$ of the wrapped Boolean array to the Boolean value $b$ (i.e., $V^S[i] := b$).
**Definition 49 (Superset or Equivalence Relation).** Let a bit mask $\text{BitMask}^{S_1}$ represent a set $S_1 \subseteq E$ and the bit mask $\text{BitMask}^{S_2}$ represent a set $S_2 \subseteq E$. Below we define the member function $\text{IsSupersetEqOfMask}$ for deciding whether $S_1 \subseteq S_2$ holds:

$$\text{BitMask}^{S_2} . \text{IsSupersetEqOfMask}(\text{BitMask}^{S_1}) =$$

$$\neg (\exists i \in I : \text{BitMask}^{S_1} . \text{GetValue}(i) \land \neg \text{BitMask}^{S_2} . \text{GetValue}(i)) \quad (2.1)$$

**Theorem 1.** Let $S_1 \subseteq E$ and $S_2 \subseteq E$ be sets, $\text{BitMask}^{S_1}$ a bit mask representing $S_1$ and $\text{BitMask}^{S_2}$ a bit mask representing $S_2$. The member function $\text{IsSupersetEqOfMask}$ evaluates $S_1 \subseteq S_2$: $S_1 \subseteq S_2 \iff \text{BitMask}^{S_2} . \text{IsSupersetEqOfMask}(\text{BitMask}^{S_1})$

**Proof.**

$$\text{BitMask}^{S_2} . \text{IsSupersetEqOfMask} \text{(BitMask}^{S_1}) =$$

$$\forall i \in I : \text{BitMask}^{S_1} . \text{GetValue}(i) \land \neg \text{BitMask}^{S_2} . \text{GetValue}(i) =$$

$$\forall e \in E : \text{BitMask}^{S_1} . \text{GetValue}(L(e)) \land \neg \text{BitMask}^{S_2} . \text{GetValue}(L(e)) =$$

$$\forall e \in E : -(e \in S_1) \land -(e \in S_2) = \forall e \in E : -(e \in S_1) \lor (e \in S_2) = \forall e \in E : e \notin S_1 \lor e \in S_2 \quad \square$$

The usual set operations (union, intersection, minus, etc.) can be defined on bit masks also; all of these operations are defined formally coupled with proofs of correctness can be found in [134].

**Implementations of Bit Masks**

We will use **compact bit masks** for storing read-only data. A compact bit mask stores the values in $V^S$ in a compact one bit per value alignment i.e., eight Boolean values in a byte. This is a very compact representation that is beneficial in memory requirement aspects but not ideal for the CPU since it has to perform multiple instructions to extract a value from a byte (accessing the word, selecting the byte, masking the selected bit and possibly even shifting the single bit to a position).

We will use **fat bit masks** for storing runtime data. A fat bit stores the values in $V^S$ in a word-aligned organization i.e., one Boolean value in an entire word. This organization wastes a lot of space in the memory (e.g., on a 32-bit platform 31 bits are unused in each words) but ideal for the CPU since it can access a value with a single instruction.

We will use the CompactMask$^{S_i}$ and FatMask$^{S_j}$ notations for referring to compact or fat masks representing $S_i \subseteq E$ and $S_j \subseteq E$ respectively.

**Representation of Transition Conglomerate Sets**

Compact and fat masks were introduced without any restriction about the sets represented. Obviously our goal is to use them for storing state sets, state hierarchies, state hierarchy sets and transition conglomerate sets. Storing transition conglomerate sets is relatively simple: we only have to assign identifiers for all transition conglomerates in the model.

Let $I^{TC} = \{0, 1, \ldots, |TC| - 1\}$ be the set of identifiers and assign identifiers to transition conglomerates by the $L^{TC} : TC \rightarrow I^{TC}$. The only requirement against $L^{TC}$ is that it should be bijective i.e., establish a one-to-one correspondence between transition conglomerates and identifiers (such a function can be trivially constructed by establishing an arbitrary order of transition conglomerates).

**Example 18.** The statechart in Fig. 2.8 contains six transition conglomerates: $tc_1 \ldots tc_6$ as indicated in green callouts. Let $L^{TC}$ assign the identifier $(i - 1)$ to $tc_i$ transition conglomerates (blue callout). Let the set $TC_t = \{tc_3, tc_4\}$ be represented by $\text{BitMask}^{TC_t}$. The encapsulated $V^{TC_t}$ Boolean array is as follows: $V^{TC_t} = [\bot, \bot, T, \bot, \bot, \bot]$ (i.e., $T$ only at the identifiers of $tc_3$ (2) and $tc_4$ (3)).

For didactic reasons we introduce compact and fat bit masks for transition conglomerates (classes CompactTCMask and FatTCMask respectively) directly derived from CompactMask and FatMask respectively.
Figure 2.8: Examples for Identifier Assignment

![Diagram of statechart and identifier assignment](image)

Listing 2.3: AsmL Specification of the Traverse Hierarchy Function

```plaintext
1 public TraverseHierarchy(actLayerSet as Set of State) as Seq of State
2 if actLayerSet.Count = 0 then
3   return []
4 else
5   actLayerSeq as Seq of State = ToSeq(actLayerSet)
6   nextLayerSet as Set of State = BigUnion({AllSubstD(s) | s in actLayerSet})
7   return actLayerSeq + TraverseHierarchy(nextLayerSet)
```

Representation of State Sets, State Hierarchies and State Hierarchy Sets

Assigning identifiers to states is not as easy as in case of transition conglomerates since we have to capture enough information about state hierarchy that enables us to build valid activity sequences when leaving or entering state hierarchies.

As outlined in the introduction, the idea is to assign identifiers to states on a way that the values of identifiers carry some information about the level of the hierarchy at which the state resides. To put together: we need an algorithm that is capable of organizing a State set of states into an \( S = (s_0, s_1, \ldots, s_{|State|-1}) \) sequence for which it holds that for any \( s_a \) state its all \( s_d \in \text{AllSubst}^T(s_a) \) descendants, \( s_a \) comes before \( s_d \): \( S = (s_0, s_1, \ldots, s_a, \ldots, s_d, \ldots, s_{|State|-1}) \). Defining such an algorithm is easy: we have to simply traverse the state refinement hierarchy and collect states at the same layer of hierarchy (i.e., at the same distance from the topmost states) and assign identifiers such way that states in lower layers should get higher identifiers. Having the sequence prepared, the identifier of a state \( s_i \) is its place at the sequence (starting with 0). Below we first present an example and define the function TraverseHierarchy that implements the identifier assignment.

Example 19. Let us consider the statechart shown in Fig. 2.8 the state refinement hierarchy is presented in the orange callout (inheritance arrows indicate direct substate relations, e.g., the direct substates of \( s_1 \) are \( s_2 \) and \( s_3 \)). It is easy to see that there are three states at the roots of the refinement hierarchy: \( s_1, s_4 \) and \( s_{11} \) and there are three “layers of hierarchy”: \( (s_1, s_4, s_{11}), (s_2, s_3, s_5, s_6, s_8, s_{10}) \) and \( (s_7, s_9) \). A possible identifier assignment is also presented in the callout (the state sequence is \( S = (s_1, s_4, s_{11}, s_2, s_3, s_5, s_6, s_8, s_{10}, s_7, s_9)) \). Obviously the order of states within a layer is of no importance, e.g., \( S = (s_4, s_1, s_{11}, s_3, s_2, s_8, s_6, s_5, s_{10}, s_9, s_7)) \) would be another valid sequence.

Definition 50 (Assigning Identifiers to States). The function TraverseHierarchy (Listing 2.3) receives a layer of states (i.e., a set of states representing a layer e.g., \{s_2, s_3, s_5, s_6, s_8, s_{10}\} in Fig. 2.8) as single input argument and recursively traverses downwards the refinement hierarchy and builds the corresponding tail of the state sequence (since sequences and sets play a central role in the algorithm we directly present the AsmL definition instead of the usual pseudocode notation).

The operation is quite simple: if the actual layer is empty (2), the traversal finished, the function returns an empty list (3), otherwise the set of states in the actual layer is converted to a sequence (the exact sequence of states within a layer is undefined and irrelevant as mentioned above) (5), then the next layer of states is calculated by unifying all direct substates of all states in the actual layer into a single set (6) and the sequence returned is constructed by appending the tail of the list returned by a recursive call for the next layer after the actually constructed sub-sequence (7).
If the function is called with the first layer of states: \( \{ s | s \in \text{AllSubst}^D(sm_0) \} \) it returns a sequence of states that we are looking for. Note that the sequence returned by \( \text{TraverseHierarchy} \) is not fully defined: the exact order of states in sub-sequences corresponding to layers is non-defined.

**Theorem 2 (Transitive Substates in the Sequence).** Let \( S_i = (s_{i0}, s_{i1}, \ldots, s_{i \text{State}[1]}) \) be a sequence returned by \( \text{TraverseHierarchy}(\{ s | s \in \text{AllSubst}^D(sm_0) \}) \) and \( s_a \) and \( s_d \) be states so that \( s_a \) is at the index \( i_k \) and \( s_d \) is at the index \( i_l \) in the sequence (i.e., \( s_a = s_{i_k} \) and \( s_d = s_{i_l} \)). It is easy to see that if \( s_{i_0} \) is a transitive substate of \( s_a \) \( (\text{IsSubst}^D(s_a, s_{i_0})) \) then \( k < l \) because of the definition of \( \text{TraverseHierarchy} \). This observation implies two consequences:

- Let \( E \) be a set of states to be entered and \( S_i = (s_{i0}, s_{i1}, \ldots, s_{i \text{State}[1]}) \) a sequence returned by \( \text{TraverseHierarchy}(\{ s | s \in \text{AllSubst}^D(sm_0) \}) \). If we forward-iterate through the sequence \( S_i \) (i.e., \( s_{i_0}, s_{i1}, \ldots, s_{i \text{State}[1]} \)) and perform the entry activity of \( s_{i_j} \) if \( s_{i_j} \in E \) then we do not violate the sequence required by the standard (i.e., parent states are entered before child states).

- Let \( X \) be a set of states to be left and \( S_i = (s_{i0}, s_{i1}, \ldots, s_{i \text{State}[1]}) \) a sequence returned by \( \text{TraverseHierarchy}(\{ s | s \in \text{AllSubst}^D(sm_0) \}) \). If we backward-iterate through the sequence \( S_i \) (i.e., \( s_{i \text{State}[1]}, s_{i \text{State}[2]}, \ldots, s_{i_0} \)) and perform the exit activity of \( s_{i_j} \) if \( s_{i_j} \in X \) then we do not violate the sequence required by the standard (i.e., child states are left before parent states).

**Example 20 (Representation of State Sets).** Let us consider the statechart and the state identifier assignment shown in Fig. 2.8 The state set \( S_1 = \{ s_2, s_5, s_{11} \} \); \( S_1 \) can be represented by the bit mask \( \text{BitMask}^S_1 \) where the encapsulated Boolean vector \( V^{S_1} \) is as follows: \( V^{S_1} = [\perp, \perp, T, T, T, T, T, T, T] \) (i.e., \( T \) only at the identifiers of \( s_2 (3), s_5 (5) \) and \( s_{11} (2) \)).

**Example 21 (Representation of State Hierarchies).** Let us consider the statechart and the state identifier assignment shown in Fig. 2.8 The hierarchy of states possibly left by \( tc_5 \) contains states \( s_4, s_5, s_6, s_7, s_8, s_9 \) and \( s_{10} \). Let the bit mask \( \text{BitMask}^S \) represent the hierarchy, this way the \( V^S \) Boolean vector encapsulated in \( \text{BitMask}^S \) is as follows: \( V^S = [\perp, T, \perp, \perp, \perp, T, T, T, T, T] \) (i.e., \( T \) only at the identifiers of \( s_4 (1), s_5 (5), s_6 (6), s_7 (9), s_8 (7), s_9 (10) \) and \( s_{10} (8) \)). It is easy to see that when having to leave this hierarchy in the configuration \( C = \{ s_4, s_6, s_7, s_8, s_9 \} \), we can simply backward-iterate through \( V^S \) (i.e., \( i \in 10 \ldots 0 \)) and if the state corresponding to the actual identifier is active and \( V^S[i] = T \) the exit activity of the state identified by \( i \) is to be performed; in this example the sequence of exit activities is as follows: \( s_9, s_7, s_8, s_6, s_4) \) and this is a valid sequence of exit activities.

Using our identifier assignment scheme, bit masks are capable of representing even sets of non-overlapping state hierarchies: let the set of hierarchies to be represented be \( H = \{ h_1, h_2, \ldots, h_n \} \) and let the state sets in the hierarchies be \( S_1, S_2, \ldots, S_n \); since the \( h_i \) hierarchies do not overlap (\( \forall \{i, j\} \in 1 \ldots n : S_i \cup S_j = \emptyset \)), the \( S_i \) sets are disjoint; this way \( H \) can be represented by storing the \( S^U = (S_1 \cup S_2 \cup \cdots \cup S_n) \) set in the usual way. Furthermore, the beneficial observations above also apply presenting a straightforward way for calculating the activity sequence to be performed when leaving or entering state hierarchy sets.

**Example 22 (Representation of State Hierarchy Sets).** The local transition conglomerate \( tc_5 \) in Fig. 2.8 enters two state hierarchies (corresponding to the two regions refining \( s_4 \); let the set of states in the hierarchies be \( S_1 = \{ s_5 \} \) and \( S_2 = \{ s_8, s_9 \} \) respectively. These two hierarchies can be represented by the bit mask \( \text{BitMask}^{S_1 \cup S_2} \) where the encapsulated \( V^{S_1 \cup S_2} \) Boolean vector is as follows: \( V^{S_1 \cup S_2} = [\perp, \perp, \perp, \perp, \perp, T, \perp, T, T, T] \) (i.e., \( T \) only at the identifiers of \( s_5 (5), s_8 (7) \) and \( s_9 (10) \)). It is easy to see that when having to enter this set of hierarchies, we can simply forward-iterate through \( V^{S_1 \cup S_2} \) (i.e., \( i \in 0 \ldots 10 \)) and if \( V^{S_1 \cup S_2}[i] = T \) the entry activity of the state identified by \( i \) is to be performed; in this example the sequence of entry activities is as follows: \( s_5, s_8, s_9 \) and this is a valid sequence of entry activities.
Chapter 2. Automatic Implementation of UML 2.0 Statecharts

Listing 2.4: Pseudocode for Conversion of PERT Graphs to Activity Sequences

For didactic reasons we introduce compact and fat bit masks for states (classes \texttt{CompactStateMask} and \texttt{FatStateMask} respectively) directly derived from \texttt{CompactMask} and \texttt{FatMask} respectively.

Bit Masks in the Platform-Specific Metamodel

This subsection has introduced a smart data structure for effectively storing transition conglomerate sets, state sets, state hierarchies and state hierarchy sets. This method will be used for implementation of the following modeling concepts (highlighted in Fig. 2.6): (i) the initial state hierarchy of the statechart (a single state hierarchy to be represented by a state mask); (ii) state hierarchies entered and possibly left by a transition conglomerate (sets of state hierarchies to be represented by state masks); (iii) conflicting transition conglomerates and ones of higher priority (transition conglomerate sets to be represented by transition conglomerate masks) and (iv) states enabling a transition conglomerate (a state set to be represented by a state mask).

2.3.3 Representation of PERT Graphs by Activity Sequences

Effects of transition conglomerates are described by PERT graphs in the platform-independent model. Mapping to the embedded platform should involve the transformation of generic PERT graphs to valid activity sequences. As PERT graphs are acyclic, there is always a sequence of activities that does not violate any direct or implied dependencies of the PERT graph; below we define such a conversion.

Definition 51 (Conversion of PERT Graphs to Activity Sequences). Listing 2.4 presents a straightforward algorithm for constructing a sequence of PERT graph nodes that is a valid sequential implementation of the behavior specified by the original PERT graph. The idea is quite simple: the function maintains a set of nodes already processed (argument \texttt{processed}); if all nodes are processed returns the empty list and terminates (6–7) otherwise identifies the nodes still to be processed (9) and selects those nodes whose dependencies are met i.e., there is no precedence relation in the PERT graph that requires a non-processed node to be the predecessor of the actual one (10–13) and returns the sequence constructed by converting the selected subset of nodes to a sequence and appending the sequence tail returned by the recursive call (15). Obviously the entry point of the algorithm is calling the main function with an empty set of already processed nodes (17–18).

It is easy to see that since we select nodes of the graph only if their precedence relations are met, the resulting sequence of nodes represents a valid sequence of activities to be performed (starting from the head of the sequence). Note that the resulting PERT node sequence can also be considered as a PERT graph where no concurrent execution may occur. For consistency reasons we will not explicitly
Figure 2.9: Conversion of PERT Graphs to Activity Sequences

replace PERT graphs (metaclass \texttt{PERTGraph}) with activity sequences in the metamodel but we will modify the implementation of the metaclass to simply contain a sequence of activities.

**Example 23.** Let us consider the generic PERT graph $p$ in the left side of Fig. 2.9: a valid sequential equivalent of $p$ is $p'$. The right side of the figure presents implied dependencies of $p$ and $p'$: it is easy to see that all direct and implied dependencies of $p$ are also required by $p'$.

As highlighted in Fig. 2.6 the effects of transition conglomerates are to be expressed by activity sequences in the platform-specific metamodel.

2.3.4 Platform-Specific Metamodel

Having defined data structures for effective representation of transition conglomerate sets, state sets, state hierarchies, state hierarchy sets and sequential PERT graphs we can introduce these new constructs into the platform-specific metamodel (Fig. 2.10). It is beneficial to compare metamodels shown in Fig. 1.3 (the original metamodel of state machines presented in the standard), Fig. 1.4 (the metamodel containing the core concepts and the newly introduced termination state and precise statechart metaclasses), Fig. 1.18 (metamodel after the introduction of transition conglomerates, PERT graphs, state hierarchies and related concepts), Fig. 2.6 (figure highlighting those parts of the platform-independent metamodel that need to be fitted to resource-restricted platforms) and Fig. 2.10 (the resulting platform-specific metamodel) for better understanding the evolution of the metamodel.

As mentioned above and indicated in Fig. 2.6 the key differences between the platform-independent metamodel (Fig. 1.18) and the platform-specific mapping (Fig. 2.10) are as follows: (i) the initial state hierarchy of the statechart is represented by a state mask; (ii) state hierarchies entered and possibly left by a transition conglomerate are represented by state masks; (iii) conflicting transition conglomerates and ones of higher priority are represented by transition conglomerate masks (iv) states enabling a transition conglomerate are represented by a state mask and (v) effects of transition conglomerates are represented by sequential PERT graphs. Furthermore the set of termination states is indicated by a state mask and triggers of transition conglomerates are considered to be plain enumerated types (represented by an integer in the metamodel).

The last difference between the platform-independent and the platform-specific metamodel is the explicit representation of ordered state and transition conglomerate containment: these are the metaclasses whose instances are referenced by their identifiers therefore it is essential to be able to access an element by its identifier; we followed the most straightforward approach by storing states and transition conglomerates in arrays according to the identifiers i.e., state or transition conglomerate with identifier
i is at the \( i \)th element of ordered collections (arrays) \( \text{StateSeq}.\text{values} \) or \( \text{TCSeq}.\text{values} \) respectively. The introduction of metaclasses \( \text{StateSeq} \) and \( \text{TCSeq} \) serves mainly didactic reasons to unambiguously indicate that the containment is ordered.

The base and utility metaclasses related to the platform-specific metamodel are presented in Fig. 2.11: most of the metaclasses are derived from the metaclass \( \text{Element} \) (similarly to the organization of the platform-independent metamodel); bit masks are derived from the abstract \( \text{BitMask} \) metaclass that contains an ordered sequence of Boolean values; this containment is implemented in a storage-effective way in case of compact masks and CPU effective way in fat masks; finally PERT graphs were modified to contain node sequences instead of arbitrary node-precedence sets.

Having presented the platform-specific metamodel and discussed the differences between the platform-independent and specific metamodels the static part of platform-specific behavioral modeling is ready (detailed AsmL definition of the entire metamodel can be found in [134]). Our next step will be the adaptation of operation algorithms to the platform-specific metamodel.

### 2.3.5 Platform-Specific Mapping of Algorithms

In order to present a platform-specific mapping of algorithms we have to define the platform-specific equivalents of all the functions and algorithms specified in Chp. 1 and we have to prove the equivalence of the function pairs. Due to space restrictions we can not present the entire mapping here, we will only focus to some interesting cases. The formal definition of all the platform-specific algorithms, the proofs of equivalence and the related analysis of algorithmic complexity can be found in [134].

The key differences between the platform-independent and platform-specific algorithms were collected in Fig. 2.7; we would like to illustrate the following issues: (i) representation of state and transition conglomerate sets by bit masks, (ii) non-recursive implementation of the originally recursive functions and (iii) application of return arguments instead of returning dynamically allocated sets. These modifications aim at achieving the following design goals: (i) using high-performance data structures for being stored in RAM, (ii) avoidance of recursive or mutually recursive operation for better predictability of execution time and stack consumption and (iii) reduction of dynamic memory allocation for better predictability of heap consumption. Having taken into consideration these goals the functions \textbf{Enabled} and \textbf{LeavingHierarchy} were chosen as illustrative examples for mapping the algorithms to the target platform. We will present AsmL specifications and prove the equivalence of the functions to the platform-independent ones defined in App. A.4.

The platform-specific algorithms maintain actual state of the Kripke-transition system in the \texttt{actState} variable: the configuration is accessed as \texttt{actState.LConf} (a fat state mask) and the evaluation of extended variables in accessed as \texttt{actState.LEval} (an \texttt{Eval} object for which the evaluation of a predicate \( p(e \models p) \) is obtained as \texttt{Eval.Models(p)}). For more detailed discussion see [134].

The platform-specific mapping of the function \textbf{Enabled} is shown in Listing 2.5. We have to prove that the algorithm specified in Listing A.12 (PIM) is correctly implemented by Listing 2.5 (PSM). Below we will discuss the differences and prove the equivalence of the functions:

**Function signature:** The PSM mapping uses the output argument \texttt{retFatTCMask} instead of dynami-
cally allocating and returning a set, the actual configuration and evaluation of extended variables need not be specified in the argument list since they are stored in the labeling of the actual state as mentioned above and since as trigger types are referred to by numeric identifiers the PSM mapping receives an integer argument instead of a string as in case of the PIM specification.

Block 16–17 (Listing A.12) compared to 16–21 (Listing 2.5): as (i) \( i \) iterates over transition conglomerate identifiers, (ii) the configuration is represented by \( \text{actState}.LConf \), (iii) enabling states are accessed through the role .en and (iv) \( \text{IsSupersetEqOfMask} \) implements the \( \subseteq \) set operation, the PIM and PSM implementations are equivalent.

Block 19–20 (Listing A.12) compared to 23–31 (Listing 2.5): it is easy to see that after executing the block \( \text{retFatTCMask}.\text{GetValue} \) will return \( \top \) only for those \( i \) indexes, where (i) \( \text{retFatTCMask}.\text{GetValue}(i) \) was initially true (i.e., \( tc \in \text{TCAct} \)) and (ii) triggers match.

Block 22–23 (Listing A.12) compared to 33–44 (Listing 2.5): it is easy to see that after executing the block \( \text{retFatTCMask}.\text{GetValue} \) will return \( \top \) only for those \( i \) indexes, where (i) \( \text{retFatTCMask}.\text{GetValue}(i) \) was initially true \( tc \in \text{TCT} \) and (ii) the transition conglomerate has no guard (representing the \( c_0 \) empty constraint) or the guard evaluates to true.

The platform-specific mapping of \textit{LeavingHierarchy} is shown in Listing 2.6. We have to prove that the algorithm specified in Listing A.16 (PIM) is correctly implemented by Listing 2.6 (PSM):

Function signature: The PSM mapping uses the output arguments \( \text{retPERTGraph} \) and \( \text{retLeftFatStateMask} \) as accumulators instead of dynamically allocating and returning objects. The input argument \( \text{shn} \) is a state hierarchy node in the PIM specification while it is a compact state mask representing the same hierarchy in the PSM mapping (as discussed above). The actual configuration needs not be specified in the argument list of the PSM function since it is stored in the labeling of the actual state \( \text{actState}.LConf \) as mentioned above).

Non-recursive behavior: The PSM mapping is a non-recursive implementation of the original PIM specification: as discussed previously, collecting the states actually left and building the corresponding sequential PERT graph of exit activities is simply a backward iteration through the state mask and checking whether the actual state is active in the current configuration.

Having outlined the “manual” proof of correctness of our algorithms, naturally emerges the question, whether this proof could be carried out automatically or at least supported by theorem provers etc. The usual approach for proving the equivalence of two algorithms consists of constructing some automaton model of both implementations and proving that the two automata can simulate each other. The aspect to be highlighted here is the following: we are not proving that PIM and PSM versions of our algorithms are fully equivalent since we have intentionally modified the PIM semantics for seamless fitting to the target platform; this way in case of an automatic proof of correctness our design decisions would also have had to be integrated into the model e.g., by modifying the automaton describing the PIM semantics. With respect to the relatively high complexity and low number of algorithms the manual proof of correctness seemed to be the most straightforward approach.

2.4 Implementation

Having reached this point of discussion we have specified the platform-specific metamodel that can be easily mapped to ANSI-C data structures and algorithms whose ANSI-C implementation is nearly just a syntactic rewriting. The first part of this section is focused on the implementation of the static structure (Sec. 2.4.1), the second part discusses the implementation of algorithms (Sec. 2.4.2).
private Enabled(t as Integer, retFatTCmask as FatTCMask)

require retFatTCMask.Length = psc.NumTCs

// Transition conglomerates with active enabling states
// This implementation directly stores enabled TCs in retFatTCMask
step foreach i in psc.TCIndexSeq
    val as Boolean =
        actState.LConf.IsSupersetEqOfMask(psc.transitionConglomerateSeq(i).en)
        retFatTCMask.SetValue(i, val)

// Active transition conglomerates that are triggered by the trigger
// This implementation directly stores triggered TCs in retFatTCMask
step foreach i in psc.TCIndexSeq
    val as Boolean =
        if retFatTCMask.GetValue(i) then
            (psc.transitionConglomerateSeq(i).trigger = t)
        else
            false
        retFatTCMask.SetValue(i, val)

// Triggered transition conglomerates whose guard (if any) evaluates to true
// This implementation directly stores enabled TCs in retFatTCMask
step foreach i in psc.TCIndexSeq
    val as Boolean =
        if retFatTCMask.GetValue(i) then
            if psc.transitionConglomerateSeq(i).guard = null then
                true
            else
                actState.LEval.Models(psc.transitionConglomerateSeq(i).guard)
            else
                false
        retFatTCMask.SetValue(i, val)

Listing 2.5: AsmL Implementation of the Enabled Function (PSM)

2.4.1 Implementation of the Static Structure

In order to present the implementation of the static structure specified by the platform-specific metamodel in Fig. 2.10 we have to discuss how to map the modeling artifacts (metaclasses, associations, containment etc.) to ANSI-C language constructs. The mapping is presented as an annotated implementation metamodel in Fig. 2.12.

The most apparent difference between the platform-specific and the implementation metamodel is the naming convention that was fitted to the target language: instead of the UML (MOF, Java, etc.) style where the first characters of words are capitalized, in the ANSI-C mapping lowercase names are used with underscores between the words (e.g., precise_statechart instead of PreciseStatechart) and member attributes are indicated in C-style using the internal data types of the language (e.g., int count instead of count: Integer).

We used a few straightforward stereotypes on metaclasses and relations for indicating the corresponding ANSI-C language feature to be used for their implementation: (i) <<structure>> indicates that the corresponding metaclass is to be implemented by an ANSI-C struct data type (applied to precise statecharts, transition conglomerates, transition conglomerate sequences, states, state sequences and compact state and transition conglomerate masks), (ii) <<fn>> indicates that the corresponding
2.4. IMPLEMENTATION

Listing 2.6: AsmL Implementation of the LeavingHierarchy Function (PSM)

metaclasse (state entry and exit activities, effects and guards of transition conglomerates) are to be implemented by functions, (iii) the <<array>> stereotype highlights that the UML ordered collection concept is to be implemented by an array and (iv) <<fn-ptr>> indicates that the associations targeting metaclasses mapped to functions should be implemented by function pointers.

We used the data type effect_t instead of sequential PERT graphs in order to avoid confusion: although a PERT graph contains a sequence of activities, in the ANSI-C implementation the entire activity sequence is implemented by a single function (i.e., not by multiple functions).

As shown in the figure, the Boolean vector encapsulated in compact masks are implemented by arrays of bytes (unsigned characters) where all bits are used for storing Boolean values i.e., for storing \( n \) Boolean values \( \lfloor n/8 \rfloor \) bytes are used.

Due to space restrictions we can not present the entire implementation of the static structure here, only show the implementation of transition conglomerates (metaclasse TC in Fig. 2.10 and tc_t in Fig. 2.12) as an illustrative example. (The entire implementation of all metaclasses and discussion of bit mask implementation is presented in [134].)

As shown in the listing and indicated in the DoxyGen-Java style comments, states that are possibly left (7–8) and entered (10–11) by the transition conglomerate (state hierarchy sets) and the set of states enabling it (13–14) are implemented by compact state masks, transition conglomerates of higher priority (16–17) and conflicting ones (19–20) are implemented by transition conglomerate masks; the effects (22–23) and guards (28–29) are implemented as function pointers and the type of the trigger (25–26) is indicated by an integer member variable.

2.4.2 Implementation of the Operation Algorithms

As we put emphasis on the elimination of those constructs from the platform-specific model that would be hard to implement in the target ANSI-C language, the implementation of the operation algorithms is nearly just a syntactic rewriting of the AsmL specification to ANSI-C. The only difference is related to issuing activities and evaluation of guards: activities and constraints are abstract entities in the platform-specific model that are handled mainly as symbolic entities: activities were “issued” and
constraints were “evaluated” without actually executing real-life code; in the implementation activities and guards are mapped to real functions that are to be called for evaluation of guard predicates or performing some real task. Obviously our implementation really calls these functions implementing this way the abstract “evaluation of a guard” or “execution of an activity” concepts. Taking into consideration this difference there is no use for first calculating a valid sequence of activities to be performed and finally calling the corresponding functions according to the sequence: the implementation can call the functions (guards, activity implementations) where most suitable, we only have to ensure that the valid sequence is not violated.

In [134] we present the entire implementation and prove the equivalence of the platform-specific algorithm specification (AsmL listings) and the ANSI-C implementation. Due to space restrictions we can only present here the implementation and proof of correctness for the Enabled and LeavingHierarchy functions (ones whose platform-specific mappings were also discussed).

The ANSI-C implementation of the function Enabled is shown in Listing 2.8. Comparing the implementation to the AsmL specification (Listing 2.5) line-by-line as shown in the comments it is easy to see that the ANSI-C implementation differs only in some minor, language-related aspects.

```
1 typedef struct {
2      /* States that are possibly left by the transition conglomerate. */
3      compact_state_mask_t possibly_left_states;
4      /* States entered by the transition conglomerate. */
5      compact_state_mask_t entered_states;
6      /* Set of states enabling the transition conglomerate. */
7      compact_state_mask_t en;
8      /* Transition conglomerates that are of higher priority than the actual one. */
9      compact_tc_mask_t all_higher_priority;
10     /* Transition conglomerates that are in conflict with the actual one. */
11     compact_tc_mask_t all_in_conflict;
12     /* Effect associated to the transition conglomerate. */
13     effect_t effect;
14     /* Identifier of the trigger type triggering this transition conglomerate. */
15     int trigger;
16     /* Guard predicate associated to the transition conglomerate. */
17     constraint_t guard;
18   } tc_t;
```

Listing 2.7: ANSI-C Implementation of Transition Conglomerates
2.4. IMPLEMENTATION

```c
unsigned i;
p precise_statechart_t *psc = this -> psc;
/* [AsmL]
 * require retFatTCMask.Length = psc.NumTCs
 */
assert (fat_mask__length(ret_fat_tc_mask) == precise_statechart__num_tcs(this -> psc));

/* [AsmL]
 * step foreach i in psc.TCIndexSeq
 * val as Boolean =
 * if retFatTCMask.GetVal
 * then (psc.transitionConglomerateSeq(i).trigger = t)
 * else
 * false
 * retFatTCMask.SetValue(i, val)
 */
for (i = 0; i < precise_statechart__num_tcs(psc); i += 1) {
    bool_t val =
        fat_mask__is_superset_eq_of_mask(this -> act_state.l_conf,
                                        psc->transition_conglomerate_seq.values[i].en);
    fat_mask__set_value(ret_fat_tc_mask, i, val);
}

/* [AsmL]
 * step foreach i in psc.TCIndexSeq
 * val as Boolean =
 * if retFatTCMask.GetVal
 * then
 * else
 * false
 * retFatTCMask.SetValue(i, val)
 */
for (i = 0; i < precise_statechart__num_tcs(psc); i += 1) {
    bool_t val =
        fat_mask__get_value(ret_fat_tc_mask, i) &&
            (psc -> transition_conglomerate_seq.values[i].trigger == t);
    fat_mask__set_value(ret_fat_tc_mask, i, val);
}

/* [AsmL]
 * step foreach i in psc.TCIndexSeq
 * val as Boolean =
 * if retFatTCMask.GetVal
 * then
 * else
 * false
 * retFatTCMask.SetValue(i, val)
 */
for (i = 0; i < precise_statechart__num_tcs(psc); i += 1) {
    bool_t val =
        fat_mask__get_value(ret_fat_tc_mask, i) &&
            !((constraint_t) psc -> transition_conglomerate_seq.values[i].guard);
    fat_mask__set_value(ret_fat_tc_mask, i, val);
}
```

Listing 2.8: ANSI-C Implementation of the Enabled Function

The ANSI-C implementation of the function `LeavingHierarchy` is shown in Listing 2.9. As discussed above, there is a single conceptual difference between the implementation and the AsmL specification (Listing 2.6): the ANSI-C function does not build a PERT graph of activities but directly calls the corresponding functions. It is easy to see that the order of calling them is exactly the same as activities would be stored in the sequential PERT graph – this way we can be sure that the activity subsequence is issued in a valid order. Obviously in global terms we will have to take into consideration that the function `LeavingHierarchy` is to be called at appropriate points of execution (i.e., before
performing transition effect and state entry activities) – this is shown in [134].

```c
static void
kts_leaving_hierarchy(kts_t *this, compact_state_mask_t shn,
fat_state_mask_t ret_left_fat_state_mask) {
    int i;
    precise_statechart_t *psc = this -> psc;
    /* [AsmL]
     * require retLeftFatStateMask.Length = psc.NumStates
     */
    assert (fat_mask__length(ret_left_fat_state_mask) ==
            precise_statechart__num_states(psc));
    /* [AsmL]
     * // Collect states actually left
     * step foreach i in psc.StateIndexSeq
     * val as Boolean = (actState.LConf.GetValue(i) and shn.GetValue(i))
     * retLeftFatStateMask.SetValue(i, val)
     */
    for (i = 0; i < precise_statechart__num_states(psc); i += 1) {
        bool_t val = (fat_mask__get_value(this -> act_state.l_conf, i) &&
                      compact_mask__get_value(shn, i));
        fat_mask__set_value(ret_left_fat_state_mask, i, val);
    }
    /* [AsmL]
     * // Build the PERT graph
     * retPERTGraph.nodes := [] as Seq of PERTNode
     * step foreach i in Reverse(psc.StateIndexSeq)
     * if retLeftFatStateMask.GetValue(i)
     * retPERTGraph.nodes := retPERTGraph.nodes + [ new PERTNode(psc.stateSeq(i).exit) ]
     */
    /* In the C implementation activities are called directly, no PERT graphs are built. */
    for (i = precise_statechart__num_states(psc) - 1; i >= 0; i -= 1)
        if (fat_mask__get_value(ret_left_fat_state_mask, i))
            if (((activity_t) 0) != psc -> state_seq.values[i].exit)
                ((activity_t) (psc -> state_seq.values[i].exit))
                (this -> context, this -> act_state.l_conf);
}
```

Listing 2.9: ANSI-C Implementation of the LeavingHierarchy Function

### 2.5 Code Generation

The mapping to platform-specific semantics and the implementation steps were realized as an automatic code generator. The tool processes models drawn in a UML modeling environment and exported to the XML Metadata Interchange Format (XMI) [109], performs the subsequent transformation steps and outputs ANSI-C source. Since even the intermediate platform-specific representation is executable, the code generator is also capable of printing Microsoft AsmL source implementing the platform-specific semantics to be used as a simulator of PSM semantics.

As bit mask operations have a significant impact on the performance of the generated applications, their implementation is available in two formats. The basic implementation is practically a direct rewriting of AsmL sources to ANSI-C; this approach handles bit masks as arrays of Boolean values and uses the usual array subscripting and iteration constructs for accessing and manipulating data. The performance enhanced implementation stores Boolean values compacted in words of memory and realizes the bit mask functions by some Boolean logical operations (practically written in assembly). The current limitation of the performance enhanced version, that the sets represented by the bit masks may not be larger than the width of the largest integer supported by the architecture and the C compiler (e.g., 64 bits in case of a PC using the (non-ANSI) long long int type of the GNU C compiler). App. B.3 presents an overview on the code generation process.
2.6 Experimental Evaluation

This section presents the experimental evaluation of applications generated according to our approach in performance and memory consumption aspects. The discussion goes far beyond the pure presentation of numeric results by comparing the experienced measures to the corresponding attributes of application built according to the QHsm approach, the best elaborated solution presented in the literature. Unfortunately the comparison can not be completely fair to our approach since the QHsm solution does not support such important constructs as concurrent state refinement, transitions without triggers, history mechanism, termination etc. that are fully supported by our approach (obviously the models used in our experiments may not contain such UML constructs that are not supported by QHsm). Since the support for these features introduces some overhead into our solution that does not apply to the less complex operation implemented by the QHsm approach, the main goal of the comparison is to ensure that the support for more complex features does not introduce unacceptable overhead.

As it will be shown below, although for some simple, non-hierarchical models the QHsm pattern proves to be better, for more complex models the gap caused by the overhead is not only eliminated but our solution proves to be of higher performance with significantly lower memory consumption.

2.6.1 Memory Layout and Experiment Setup

As an application may be composed of any number of statecharts (e.g., various controller algorithms implemented for the same microcontroller) and for each statecharts any number of instances (objects) may reside in the memory (e.g., multiple physical processes controlled by the same algorithm), the entire memory used by an object can be divided into four parts:

**Static part** Both approaches are based on some library that is available as an off-the-shelf component to be linked to the application. This library contains the code of the QHsm base class and some related functions in case of the QHsm pattern and the generic interpreter routines in case of our approach. We will call these parts of programs as static part. Note that independently of the number of statecharts implemented by the application the library is linked to the binary in one instance. Since the static part is purely code in both cases, this component can reside in read-only code segments in the RAM in case of general purpose PCs or in the cheap read-only memory (e.g., flash memory) in case of embedded systems.

Obviously the size of the static part is the same for all benchmark experiments: this was calculated simply by compiling the corresponding source files and identifying the size of the text segments in the object files by a disassembler.

**Model-specific part** The statecharts implemented by the application are represented by classes derived from the QHsm base class in case of the QHsm pattern and an initialized constant data structure in case of our pattern. We will call this part of programs as model-specific part. Note that independently of the number of instances operating according to the statechart, the model specific part is stored in one instance per statechart. Since the model-specific part is purely code in case of the QHsm pattern and a constant (read-only) data structure in case of our approach, this component can reside in read-only code segments in the RAM in case of general purpose PCs or in the cheap read-only memory (e.g., flash memory) in case of embedded systems.

The size of the model-specific part in case of the QHsm pattern this was calculated by compiling the source file containing the derived class only and identifying the size of the text segment. In case of our approach the size of the initialized data structure was calculated by investigating the source code containing the initialized data structure (since we know the exact sizes of pointers and various primitive types doing this calculation was easy.

**Per-instance part** Both approaches require some bytes of memory for storing the actual configuration of an instance: this is a single pointer (myState) in case of the QHsm pattern and a bit mask
in case of our pattern (one bit per state in the statechart of the object). The QHsm pattern uses another member variable \texttt{(mySource)} for storing a the source state during a transition (used like a global variable in order to achieve some performance benefit).

Furthermore the QHsm pattern uses a sophisticated caching mechanism for storing the activities to be performed when firing a transition. This cached data is implemented as a static array of pointer to member functions (the length of this array is set to 8 in the source of the QHsm base class) and an unsigned integer (at least 16 bits long) for storing some bits of information corresponding to the pointer to member function array. (Note that a pointer to member function construct consists of two pointers: one for the object another one for the function, i.e., in case of a 32 bit architecture a plain pointer is 4 bytes, a pointer to member function is 8 bytes.) The QHsm pattern does not provide means for setting the size of the cached pointer to member array even by the programmer – to modify this value the source code of the QHsm base class has to be adjusted. The caching mechanism can be switched off but this results in significantly slower operations. We will present experiment results with both caching enabled and disabled.

To put together, for storing an instance specified by a statechart of \(N_{\text{State}}\) states and \(N_{\text{TC}}\) transition conglomerates in the memory of an architecture where plain pointers are represented in \(p\) bytes and pointers to member functions are represented in \(m\) bytes, we will need \(2p + N_{\text{TC}}(8m + 2)\) bytes in case of the QHsm pattern with caching enabled and \(\lceil N_{\text{State}} / 8 \rceil\) bytes in case of our pattern (this means that for practical cases our approach needs much less memory for storing an instance). We will call the memory required for storing an instance as \textit{per-instance part} of memory consumption (note however that the per-instance memory consumption is determined by the statechart of the instance). The per-instance data is intensively read and written therefore needs to be stored in RAM. The size of the per-instance memory consumption was calculated in our experiments according to the formulae above.

\textbf{Runtime work variables} Obviously the functions of all implementations need some stack for argument transfer and local variables. As none of the implementations use extremely deep function call chains or large local variable set, we will not discuss this aspect of memory consumption.

Since the static and model-specific parts can be stored in ROM we will present the sum of these areas as \textit{read-only} part in the experiments below.

Execution times were measured by providing a sufficiently long trigger sequence to the implementation by a manually written \texttt{main} function. Activities (state entry and exit, transition effects etc.) were left empty this way the performance data presented here represents the execution of the statechart engine only. We will present execution times for the QHsm pattern with both caching enabled and disabled and for our pattern using both the basic and the performance enhanced implementations. Obviously the exact sizes and execution times depend on the HW/OS architecture, compiler and compiling options etc. but investigating the two implementations on the same architecture, compiler family using the same compiling options enables a fair comparison. For better controllability of measurements (initialization of applications, providing input triggers and measuring execution time) the experiments were carried out on an ordinary PC (Pentium IV at 2.8GHz clock frequency with the Debian GNU/Linux system version 3.1 “Sarge” running the Linux kernel version 2.4.29) instead of directly on an embedded device. It is important to highlight that since both solutions provide pure ANSI C/C++ source code without any assembly-level optimization taking into consideration any specific properties of the target platform, the performance degradation experienced when porting them to a resource constrained embedded platform (e.g., a microcontroller-based version of the \textit{mitmőt} system mentioned previously) is expected to be the same for both approaches. The applications were compiled by the GNU C/C++ compiler family version 3.3.5. The compiler was configured to perform all supported optimizations that do not involve a space-speed tradeoff (option \texttt{-O2}).

In our experiments the static part of the QHsm pattern took 5032 bytes, the static part of our implementation pattern took 4600 bytes.
2.6. EXPERIMENTAL EVALUATION

2.6.2 Experiments

We will use four scalable models for discovering the impact of the model complexity with respect to the number of states, number of transitions and depth of the state hierarchy on the performance measures: (i) a loop of $n$ states (states $s_0$, $s_1$, ..., $s_{n-1}$ are organized into a loop connected by $s_i \rightarrow s_{(i+1)}$ transitions, the number of transitions is $n$ as shown in part a of Fig. 2.13); (ii) a full mesh of $n$ states (states $s_0$, $s_1$, ..., $s_{n-1}$ are organized into a full mesh i.e., a state $s_i$ is connected to $n-1$ states $s_0$, $s_1$, ..., $s_{i-1}$, $s_{i+1}$, ..., $s_{n-1}$, the number of transitions is $(n \times (n-1))/2$ as shown in part b of Fig. 2.13); (iii) a hierarchy of $n$ states with a self-transition originating in the topmost state (states $s_0$, $s_1$, ..., $s_{n-1}$ are organized into a hierarchy where $s_i$ is a direct substate of $s_{i-1}$, there is a single transition originating in and targeting $s_0$ as shown in part c of Fig. 2.13) and (iv) a pair of hierarchies connected by two interlevel transitions ($2 \times n$ states organized into two hierarchies with $n$ states in both hierarchies where $s_i$ is a direct substate of $s_{i-1}$ and $s'_i$ is a direct substate of $s'_{i-1}$ and there are two transitions originating in $s_{n-1}$ and targeting $s'_{n-1}$ and in the opposite direction, originating in $s'_{n-1}$ and targeting $s_{n-1}$ as shown in part d of Fig. 2.13). The four scalable models will be called loop($n$), mesh($n$), hierarchy($n$) and interlevel($n$) respectively.

Expected Results

Taking into consideration the differences between the two implementations the following a priori expectations can be assumed with respect to the memory consumption and execution time:

ROM consumption: The QHsm implementation constructs a function for each state in the model while states in our approach are represented by relatively compact structures; representation of transitions in the QHsm solution is very compact (a single function call in the state function) while our solution has to store a large amount of data about transitions that are practically unused in case of non-concurrent models (e.g., priority relations, conflicts, multiple enabling states etc.). To put together, with respect to the ROM consumption, our solution is sensitive to the number of transitions while the QHsm pattern is sensitive to the number of states.

RAM consumption: The RAM consumption of our approach is always very low since the only data stored in RAM is the configuration and the $[N_{State}/8]$ bytes needed in case of $N_{State}$ states is nearly negligible for practical number of states and probably fit into a single integer. The QHsm pattern stores two pointers (myState and mySource) and the contents of transition caches in RAM: the pointers require two words while a single transition cache structure need eight pointers to member functions and an integer of at least two bytes resulting in $2p + N_{TC}(8m + 2)$ bytes of RAM consumption as discussed above (where $p$ is the size of a pointer in bytes, $N_{TC}$ is the number of transitions and $m$ is the size of a pointer to member function in bytes); note that in case of 10 transitions on a 32-bit architecture ($p = 4$, $m = 8$) this results in 668 bytes for storing a single instance. To put together the RAM consumption of our solution is always low while the caching mechanism of the QHsm pattern is very sensitive to the number of transitions.

![Figure 2.13: Synthetic Benchmark Models](image-url)
Table 2.1: Memory Consumption (“Loop” and “Mesh” Models)

<table>
<thead>
<tr>
<th>Model</th>
<th>Read only data [byte]</th>
<th>Per-instance data [byte]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Static QHsm</td>
<td>Our</td>
</tr>
<tr>
<td>Loop (5)</td>
<td>5032</td>
<td>4600</td>
</tr>
<tr>
<td>Loop (10)</td>
<td>5032</td>
<td>4600</td>
</tr>
<tr>
<td>Loop (15)</td>
<td>5032</td>
<td>4600</td>
</tr>
<tr>
<td>Loop (20)</td>
<td>5032</td>
<td>4600</td>
</tr>
<tr>
<td>Mesh (5)</td>
<td>5032</td>
<td>4600</td>
</tr>
<tr>
<td>Mesh (10)</td>
<td>5032</td>
<td>4600</td>
</tr>
<tr>
<td>Mesh (15)</td>
<td>5032</td>
<td>4600</td>
</tr>
<tr>
<td>Mesh (20)</td>
<td>5032</td>
<td>4600</td>
</tr>
</tbody>
</table>

![Figure 2.14: Read-Only Memory Consumption (“Loop” and “Mesh” Models)](image)

**Execution time:** The most expensive algorithms in our pattern are used in the transition selection mechanism, priority resolution and conflict checking. These issues do not even occur in case of models supported by the QHsm pattern since in case of non-concurrent statechart always at most one transition may fire and this selection is implemented very effectively in the QHsm pattern: the compiler is likely to be able to translate the virtual function call (selection of the actual state) and handling the trigger (switch statement in the state function) to some indirect jump instructions. The runtime discovery of state hierarchy in case of the QHsm pattern can be considered to be an expensive operation involving multiple function calls and quite complex routines; this runtime discovery is practically eliminated by the caching mechanism but caching is limited to a certain depth of hierarchy and consumes a large amount of memory and mentioned above. To put together the execution time of QHsm implementations can be sensitive to the depth of the state refinement hierarchy especially if the seriously memory consuming transition caching is switched off while algorithms are not sensitive to the depth of the hierarchy.

**“Loop” and “Mesh” Models**

First we will focus on *loop* and *mesh* models. These models are completely flat (states are not organized into a hierarchy), the number of states is relatively low but the number of transitions is proportional to the square of number of states in case the *mesh* models resulting in a large number of transitions for large models.

The memory consumptions of implementations according to the QHsm pattern and our approach are compared in Tab. 2.1 and visually highlighted in Fig. 2.14 (ROM) and Fig. 2.15 (RAM).

The ROM consumption of applications built from the *loop* models (Fig. 2.14 left side) justifies our assumption that the QHsm pattern is somewhat more sensitive to the large number of states than our approach: since QHsm implementations require more memory (e.g., in case of 15 states QHsm requires 6468 bytes while our pattern uses only 5264 bytes). Due to the large number of transitions in case of
2.6. EXPERIMENTAL EVALUATION

Figure 2.15: RAM Consumption ("Loop" and "Mesh" Models)

Table 2.2: Execution Times ("Loop" and "Mesh" Models)

mesh models (Fig. 2.14, right side) our method required somewhat more read-only memory (e.g., in case of 15 states the QHsm implementation requires 11268 bytes while our pattern uses 12416 bytes) as discussed above (note that the very large number of transitions (105) in this model is non-realistic.)

As predicted above the RAM consumption of the caching mechanism of the QHsm pattern was proven to be very sensitive to the number of transitions: in case of loop models of 15 states storing a single instance in memory requires 998 bytes (actually a large piece of memory in case of an embedded system) and the RAM consumption even increases to 14858 bytes in case of mesh models (more than three times of the entire RAM of the 8-bit microcontroller module of the mitmôt system). When switching off the caching mechanism the RAM consumption drops to 8 bytes. The per-instance RAM consumption of our approach is even smaller (between 1 and 3 bytes for these models).

The execution times for performing 100000 transitions in cases of the two patterns are compared in Tab. 2.2 and visually highlighted in Fig. 2.10

In case of loop models we can observe that these models are so trivially simple that the cached version of the QHsm pattern performs actually weaker than the same pattern with caching disabled (e.g., in case of 15 states the execution took 0.8 seconds with caching and 0.25 without caching). The execution times of our basic implementation was significantly higher (3.8 seconds for the same model) but the performance enhanced version performed close to the cached version of the QHsm pattern (0.84 seconds for the same case). The impact of large number of transitions clearly appears in case of mesh models: as the QHsm pattern is not sensitive to the number of transitions the execution times remain between 0.8 and 0.25 seconds while our approach exposed a definitely poor performance e.g., 34.37 seconds for the basic implementation in case of 15 states while the performance enhanced solution was not applicable due to the very large ((15 × 14)/2 = 105) number of transitions. The explanation for the poor performance as discussed above is the high execution time of transition selection, priority resolution and conflict checking algorithms.

The observations above can be summarized as follows: in case of flat models without any state hierarchy (i) if there are relatively few transitions the read-only memory consumption of our approach is smaller than the QHsm solution while (ii) if there are a large number of transitions the ROM consumption of our approach is higher than in case of the QHsm solution; (iii) the RAM consumption
of the QHsm pattern with caching enabled can get so extremely high that disables its application in a resource-constrained environment and (iv) our solution requires even less RAM than the QHsm solution with disabled caching; (v) the execution time of QHsm implementations is significantly smaller than in case of the basic implementations our approach and about the same as the one of our performance-enhanced implementation (if applicable). To put together: in cases of these simple models without state hierarchy the QHsm pattern typically outperforms our pattern in most aspects. This phenomenon can be explained by the fact that our method has to carry out such extra tasks that appear as useless overhead in case of non-concurrent, non-hierarchical models.

“Hierarchy” and “Interlevel” Models

In hierarchy and interlevel models states are organized into a state hierarchy, firing transitions requires performing multiple activities i.e., leaving and entering multiple states. We will see that in cases of these complex models our approach performs significantly better than QHsm implementations. Note that these models still do not involve such constructs that can not be implemented by the QHsm approach, the performance gain and reduction of resource consumption is achieved with still having to perform such “overhead” tasks (checking for transition conflicts, capability for firing multiple transitions etc.) that are not needed for non-concurrent models.

The memory consumptions corresponding to the two patterns are compared in Tab. 2.3, the ROM consumption is visually highlighted in Fig. 2.17 (the RAM consumption exposes similar trend as in the previous comparison).

Our first observation is the fact that QHsm implementations require significantly more read-only memory for both models than our approach (e.g., in case of 15 states the QHsm implementation requires 6020 and 7012 bytes for hierarchy and interlevel models respectively while our pattern uses only 4792 and 4959 bytes for the same cases), the explanation for this phenomenon is the same as in the previous case: states are implemented by more compact structures in our solution and there are few transitions in the model.

Since the caching mechanism of the QHsm pattern supports by default only transitions that do not leave more than 8 states the caching mechanism had to be switched off for \( n > 5 \) models. Obviously with caching disabled the RAM consumption of the QHsm pattern is relatively small (74 bytes for all
The execution times for performing 100000000 transitions (we used 10× higher workload for easier comparison) for the two patterns are compared in Table 2.4 and visually highlighted in Fig. 2.18. In case of hierarchy models our both approaches perform better than the QHsm implementation without caching (e.g., in case of 15 states the execution took 18.78 seconds for the QHsm implementation without caching while 11.66 seconds for our basic implementation and only 2.8 for our performance enhanced version). Caching was available for the n = 5 case only in case of the QHsm pattern, in this case its execution time was 2.55 seconds that is less than the execution time of our basic implementation (6.23 seconds) but the performance-enhanced version outperforms the QHsm approach even in this case (1.88 seconds). Similar trends can be observed in cases of interlevel models: although QHsm implementations perform somewhat better than our basic implementations for smaller models, the execution time of our approach increases slower, in case of the n = 20 model the execution time of our basic implementation is even somewhat lower than the QHsm pattern’s one. Our performance-enhanced implementation proves to be the fastest again (e.g., the QHsm implementation requires 21.31 seconds for n = 15 case, while our implementation uses 4.66 seconds only), even faster than the cached version of the QHsm pattern (note again: caching was available for case n = 5 only).

### 2.6.3 Summary

This section has presented the results of an experiment series where we used models that do not exploit the advanced features of our pattern therefore we were able to implement them even using the QHsm pattern. According to our observations for *simple models* where states are not organized into a hierarchy (actually flat state machines) the QHsm pattern was found to be more beneficial (about the same memory consumption but lower execution time) but in case of complex models with deep state hierarchies our approach consumed significantly less memory and delivered much higher performance.
2.7 Conclusions and Future Work

This chapter has presented a method for automatic source-code level implementation of behavior specified by UML 2.0 statecharts. In Sec. 2.1 we discussed previous approaches and concluded that on one hand there were no full-featured solutions published in the literature, on the other hand the theories that elaborated commercial solutions are built on are not public. Our goal was to propose a mapping of the semantics specified in Chp. 1 to resource-constrained embedded systems. Our approach is fitted to the Model Driven Architecture initiative of the OMG. Our design decisions were built on the overview on resource-constrained embedded environments presented in Sec. 2.2. Based on these observations the platform-specific mapping of data structures (metamodel) and algorithms was outlined in Sec. 2.3. The final ANSI-C implementation was presented in Sec. 2.4. The code generator built on these mapping steps was outlined in Sec. 2.5. The subsequent transformation steps are proven to be correct i.e., the PSM mapping corresponds to the PIM specification and the implementation correctly realizes the PSM specification. Finally in Sec. 2.6 our solution was experimentally evaluated by comparing to the best-known approach published in the literature (QHsm); the comparison concluded that although for some very simple models the QHsm pattern resulted in more compact and faster implementation, for really complex problems our approach not only provides support for much more features (concurrency, history mechanism etc.) but the resulting code is even smaller and faster than the ones prepared according to QHsm.

Some detailed discussions were moved to appendix App. B. the detailed AsmL specification of PSM algorithms is presented in App. B.2 and an example for automatic code generation is discussed in App. B.3.

According to our knowledge the solution presented here is the most complete approach for automatic source-code level implementation of UML 2.0 statecharts even supporting resource constrained embedded systems. The remarkable features of our method are as follows: (i) the solution corresponds to the steps of the Model Driven Architecture i.e., conceptually different tasks (identification of target platform’s features, abstract mapping to the available resources and the implementation) are clearly separated; (ii) a significant effort was laid on proving the correctness of subsequent transformation steps; (iii) being based on the semantics presented in Chp. 1 our solution supports the most complete toolkit of the most recent version of the UML standard, (iv) the mapping to resource constrained platforms does not restrict the set of usable modeling concepts and (v) for sufficiently complex models the performance and size of the resulting applications is even more beneficial as compared to the ones built according to less sophisticated approaches supporting much less modeling features. The achievements discussed in this chapter are summarized by the following thesis.

**Thesis 2 (Implementation of Statecharts on Resource Constrained Systems).** I elaborated an **effective source code-level implementation of behavior specified by UML 2.0 statecharts.** The low memory and processing power requirements of my solution enable its approach even in resource constrained embedded systems. The thesis is built up of the following parts:

(i) According to OMG’s Model Driven Architecture initiative, I identified the typical resource conditions of embedded platforms (low performance CPU, lack of support for parallel execution,
2.7. CONCLUSIONS AND FUTURE WORK

heavily constrained memory) and based on these observations I elaborated a platform-specific mapping of data structures and algorithms specified in Chp. 1.

(ii) I implemented the intermediate platform-specific representation in the ANSI-C programming language that seamlessly fits to the target platform. This solution is the most complete published approach for source code level implementation of systems specified by statecharts. The correctness of the solution was shown by systematic comparison of algorithms and data structures at corresponding transformation steps.

(iii) In order to prove the viability of my approach I carried out an experimental evaluation of memory and processing power requirements comparing applications built according to my solution and the QHsm pattern published previously in the literature. The experiments have shown that for sufficiently complex models my solution does not only support much more modeling concepts but the resource consumption of the resulting applications is also lower.

Our achievements on source code-level implementation of statecharts were discussed in several publications. The need for the implementation of systems specified by statecharts first occurred during my early research activities: not having a formal operational semantics for statecharts that time (2001-2002) our first solutions were restricted to non-hierarchical statecharts (practically pure state diagrams); these preliminary efforts were documented in [116] and [118] (Scientific Student Conference), [117] (Master's Thesis) and [137] (technical report of about 50 pages). The formally well-established next-generation approach was based on a semantics defined by extended hierarchical automata (EHA) published in [124] (conference paper) and [127] (journal article) and discussed in-depth in the technical report [125]; a short introductory overview was presented in [126]. This approach was the direct ancestor of the final version outlined in this document. Unfortunately due to space restrictions we cannot fully present the solution in this paper; a much more detailed discussion is presented as a technical report in [134] (95 pages). Another publications in the subject of model-driven development with my participation were [97] and [96] (conference papers).

The theory discussed above was implemented as a code generator that processes a UML model exported from a modeling tool to the XML Metadata Interchange Format (XMI) and outputs ANSI-C source code. Using the XMI format enables the application of practically any UML 2.0 compliant modeling environments. The generated code was successfully compiled and deployed to the hardware of the mitmol modular embedded platform [158]. Being built on a library implementing the statechart semantics discussed in Chp. 1, the source of the code generator was relatively compact: 3700 lines of AsmL source (170kB) coupled with code fragments of about 6600 lines (230kB) of AsmL, and ANSI-C source. The code generator supports both the basic and the performance enhanced implementation of state masks and is also capable of implementing the PSM representation as executable AsmL source. (The previous, EHA-based implementation of the code generator mentioned above consisted of 1500 lines of core application built on a library of 14000 lines written in C++.)

The application possibilities of our approach are quite obvious: the automatic implementation of complex behavioral models significantly increases the productivity by reducing the development time and enhancing code quality. Automatic implementation of behavioral models can be also be beneficial for fast prototyping.

There are multiple points both in the theory and the implementations where further research can be focused. This chapter aimed at mapping statecharts to resource constrained systems where such important facilities are missing as concurrent execution, object-oriented languages and modern middlewares, i.e., we illustrated the viability of our approach by bridging the widest gap that may occur between the fully abstract world of visual modeling and concrete implementations. Similarly to our approach the statechart semantics can be mapped to more advanced platforms and languages (e.g., the Java language with the JVM platform, the C# language and the .Net platform, etc.) where the algorithms specified in Chp. 1 can be implemented practically without any modification, i.e., choosing an object-oriented language for the implementation, the PIM – implementation mapping can be even
more straightforward than in case of the ANSI-C language. Our actual implementation can be further enhanced itself: we put primary emphasis on the correctness, hence the implementation is practically the syntactic rewriting of the AsmL (PSM) specifications to ANSI-C; although this approach makes proving the equivalence easier, multiple minor performance fine tuning possibilities were left open. In the experimental evaluation we found that for simple models where most of advanced features are not used (flat statecharts without state hierarchy etc.) the more sophisticated algorithms in our solution pose a significant overhead; a further enhancement possibility would be to prepare multiple variants of the operation algorithms that support narrower modeling toolkits and provide dedicated solutions for those subsets of problems.
Chapter 3

Error Detection in UML 2.0 Statechart Implementations

3.1 Introduction

As the use of both general-purpose and specialized computers pervasively extends to virtually all facets of life, the dependence of the society on their correct behavior leads to growing impact of these systems’ dependability. In the past, the term “safety critical” was primarily used for referring to such evident cases as flight control systems or nuclear power plant operation etc. At present, the wide variety of applications ranging from onboard embedded microcontrollers in cars to the use of high performance servers driving the Internet infrastructure all constitute “critical” systems whether for their clear impact on human health or financial conditions [30].

Below we provide the exact definitions of dependability-related concepts primarily relying on the widely accepted terminology established by J. C. Laprie in his fundamental work [76] (a slightly extended version of the terminology [9] was published in the first issue of IEEE’s new journal, Transactions on Dependable and Secure Computing).

The general concept dependability is defined as the trustworthiness of a computer system such that reliance can justifiably be placed on the services it delivers [23].

Depending on the application of the system dependability may be viewed according to various different but complementary attributes [76]: (i) availability means readiness for usage: this is the measure of correct service delivery with respect to the alternation of correct and incorrect service; (ii) reliability means continuity of service: this is the measure of continuous correct service delivery i.e., the time to failure; (iii) safety means non-occurrence of catastrophic failures: this is the measure of continuous delivery of correct or incorrect service after benign failure, i.e., the time to catastrophic failure; (iv) security means the prevention of unauthorized access or handling information.

A system failure occurs when the delivered service no longer complies with the specification. An error is that part of the system state which is liable to lead to a failure: an error affecting the service is an indication that a failure occurs or has occurred. The adjudged or hypothesized cause of an error is a fault. The scenario in which a fault in the system causes an error in its state space finally propagating to the output resulting in a failure is called the basic fault – error – failure chain.

Faults can be classified according to three viewpoints: nature, origin and persistence (Fig. 3.1):

- According to the nature of faults one can distinguish (i) accidental faults which appear or are created fortuitously and (ii) intentional faults which are created deliberately.

- Origins of faults can be decomposed according to three viewpoints: phenomenological causes, system boundaries and phase of creation [8]. In phenomenological aspects one can distinguish (i) physical faults which are due to some physical phenomena and (ii) human-made faults which result from human imperfections. With respect to the system boundaries one can distinguish (i)
internal faults which are those imperfect parts of the system whose utilization in a computation may lead to an erroneous system state and (ii) external faults which result from interference with the physical environment (electromagnetic perturbations, radiation, temperature, vibration etc.) or human users. According to the faults’ phase of creation one can distinguish (i) design faults which result from imperfections arising during the development of the system or subsequent modifications and (ii) operational faults which appear during the system’s exploitation.

- With respect to temporal persistence of faults one can distinguish (i) permanent faults whose persistence cannot be bound to a point or interval of time while (ii) temporary faults are present for a limited amount of time.

Errors were defined to be liable to lead to subsequent failure. Whether or not an error will actually lead to a failure depends on three major factors: the composition of the system, the system activity and the user’s viewpoint on the notion of failure. The (i) composition of the system may include such intentional or unintentional redundancy that prevents an error from leading to a failure. Due to the (ii) actual system activity the erroneous state of the system may be overwritten before causing a failure. Finally the (iii) user’s point of view is also important since what is a failure for a given user it may be a bearable nuisance for another one. The creation and manifestation of faults, error and failures can be summarized according to the following fault pathology [76]:

- A fault is active when it produces an error. An active fault is either (i) an internal fault which was previously dormant and has been activated by the computation or (ii) an external fault.

- An error may be latent or detected. An error is (i) latent when it has not been recognized as such; (ii) an error is detected by a detection algorithm or mechanism. An error may disappear before being detected. An error may, and in general does propagate and create other new errors. During operation, the presence of active faults is determined only by the detection of error.

- A failure occurs when an error passes through the user interface and affects the service delivered.

As in a system built up of multiple components the failure of a low-level component may be viewed as a fault in another component or subsystem containing it, the basic fault → error → failure chain can be completed as . . . failure → fault → error → failure → fault → . . .

The key observation to be highlighted is the following. The goal of dependability-related activities is to reduce the possibility of the system to expose such behavior that deviates from the specification i.e., the occurrence of a failure. Origins of failures are faults which can not be in general totally eliminated from the system. Faults themselves can not be “detected”, only the errors caused by them. The process of determining the (possible) cause of an error is called fault diagnosis.

Taking into consideration the observations above and the reasoning about the fault – error – failure chain, two approaches can be distinguished aiming at the reduction of the system’s possibility for exposing a failure: (i) the avoidance of faults and (ii) if a fault has already occurred in the system its tolerance by some redundancy measures. The avoidance of faults means how to “aim at a fault-free system” in general [76] and includes the actual prevention of introducing faults and removal of faults if they were still introduced before putting the system in operation. Fault avoidance clearly relates to
3.1. INTRODUCTION

general system engineering even in case of non critical systems. *Fault tolerance* is achieved by error detection, error processing and fault treatment.

- **Error detection** is the activity of identifying that a system state is erroneous.

- **Error processing** aims at removing errors from the computational state by some *recovery activities* (i.e., in backward recovery the system state is restored to a previously saved error-free state while in forward recovery the system state is explicitly moved to an error free state from which the system can continue operation, frequently in degraded mode) or *error compensation* (i.e., using some built-in redundancy to deliver error-free service even in the erroneous system state).

- The first step of **fault treatment** is **fault diagnosis** which consists of determining the causes of errors in terms of both location and nature. Then come the actions aiming at fulfilling the main purpose of fault treatment: preventing the faults from being activated again thus making it passive, i.e., *fault passivation*.

The conceptual introduction above has considered all human-made faults in the software (in its model, source code or binary representation) as design faults. With respect to the subsequent steps of software development, design faults can be distinguished further. We will consider the scenario as described in the V-Model of software development as shown in Fig. 3.2.

The left side of the V-Model corresponds to development activities (ellipses) and the corresponding documents to be prepared (rectangles) starting with (i) requirement analysis and involving (ii) specification, (iii) architecture design, (iv) module design until the (v) implementation step resulting in (i) requirement documentation, (ii) system specification, (iii) design models, (iv) module models and (v) the actual implementation (i.e., source code) respectively. This process can be seen as *iterative refinement of the system’s models* to ever more detailed models finally resulting in the implementation. With respect to our terminology above, in these steps the developers aim at preventing faults from being introduced into the software under development.

The right side of the V-Models corresponds to integration, verification and validation steps. Verification is the formal comparison of two models in order to prove that the more refined one holds specific properties defined in the context of the more general one enabling this way the identification of model refinement faults – in this aspect the concept of verification is also bound to the left side of the V-Model; the figure indicates verification steps in the integration chain mainly corresponding to integration steps where it should be proven that no dependability properties of modules are violated by some negative interactions. The final validation step proves that the finally built system really meets the original user requirements. Sziray presented a unified test model for both verification and validation of HW/SW systems in [156]. According to our discussion above this process aims at removal of faults possibly introduced in the development process.
CHAPTER 3. ERROR DETECTION IN UML 2.0 STATECHART IMPLEMENTATIONS

Theoretically when using formal models, automatic model transformation and code generation coupled with model checking techniques a fault-free implementation could be synthesized. Unfortunately model checking is a complicated task and in case of large systems often infeasible, therefore mostly testing techniques are applied for reducing the number of faults remaining in the implementation. Obviously testing can not guarantee the removal of all faults during the development therefore the fault tolerance measures built in to the system have to be prepared for not only physical faults in the underlying hardware and various external faults but even for internal ones.

This chapter presents two error detection techniques primarily aiming at the detection of control flow errors that originate from (i) model refinement faults or (ii) implementation bugs. According to the terminology introduced above: we will focus on faults which are accidental by nature, with respect to their origin, they are human made (phenomenological cause), internal (within system boundaries) and created during the design phase (phase of creation) finally with respect to their persistence they can be considered permanent since they reside in the implementation already at the deployment of the system (Fig. 3.1). Both techniques discussed here aim at detecting errors in the control flow of the system whose behavior is specified by UML 2.0 statecharts:

- The first approach is targeted for detecting those errors that are caused by such model refinement faults where a more detailed behavioral model violates some dependability properties defined in the context of a more general model. We will define a temporal logic language for statecharts (PSC-PLTL) and propose a high performance method for evaluating temporal dependability requirements during the execution of the system.

- Our second technique is targeted for detecting control flow errors caused by faults in the implementation of the statechart-based behavior. Based on the statechart semantics introduced in Chp. 1 we will define a watchdog structure (PSC-WD) that uses the fully elaborated statechart model of a class for runtime checking its behavior.

Additionally we will highlight that our solutions are capable of detecting errors that are caused by those low-level (physical) faults whose effect appears as violation of some temporal correctness criteria or deviation from the correct control flow specified by the statechart of the system.

Development activities, documents prepared and the possible fault classes are highlighted in Fig. 3.3. Software models of high abstraction level (draft statecharts) are prepared in the specification phase; these statecharts are the context of temporal correctness criteria to be checked during runtime by our first technique. In the design phase draft statecharts are iteratively refined finally resulting in fully elaborated statechart models; these statecharts will be used as reference information by the watchdog in our second technique. In the implementation step the source code of the application is prepared and binaries are built. With respect to the fault domain, design faults (e.g., invalid refinement, omitted transition, etc.) are addressed by checking temporal requirements defined in the context of specification models and implementation faults (misunderstanding the statechart semantics,
coding bugs) are addressed by comparing the actually exposed behavior to the one specified by design models. Both error detection approaches require some runtime information about the actual behavior of the application, i.e., a trace of execution. This runtime information is compared to references and an error is signaled on the violation of any requirements.

The organization of this chapter is illustrated in Fig. 3.4 by zooming the corresponding box of Fig. 1. This introduction (Sec. 3.1) has presented an overview on dependable software development according to the V-Model; we have introduced the key concepts of dependability, discussed the phases of development and the corresponding models and documents, finally we declared the faults to be addressed by our solutions.

The first part of the chapter is dedicated to the approach aiming at the detection of errors caused by model refinement faults (middle left part of Fig. 3.4). Sec. 3.2 presents an introduction to temporal logic languages by defining Kripke structures, the notion of execution traces, outlines the classification of temporal logic languages and presents a propositional linear temporal logic language as example. Sec. 3.3 defines our temporal logic variant developed for UML 2.0 statecharts (PSC-PLTL): first we will transform the Kripke transition system defined in Chp. 1 to a Kripke structure, then present the syntax and semantics of Boolean and temporal operators finally define the atomic predicates that bound statechart artifacts to the temporal logic language. Sec. 3.4 and Sec. 3.5 are focused on the evaluation of temporal logic expressions on execution traces: Sec. 3.4 outlines the previously published approaches and identifies their weaknesses; in Sec. 3.5 we present a novel solution for evaluating propositional linear temporal logic formulae on finite execution traces with significantly lower resource requirements than any previously proposed approaches.

The second part of the chapter is focused on detecting errors caused by implementation faults (middle right part of Fig. 3.4). Sec. 3.6 presents an overview on watchdog processors and related concepts: defines the notion of control flow graphs and the meaning of runtime and reference information. Our watchdog scheme developed for UML 2.0 statechart implementations (PSC-WD) is defined in Sec. 3.7 by deriving an accepting automaton from the Kripke transition system defined in Chp. 1 and formally defining the semantic checks to be performed on steps taken by the application.

The error detection capabilities of the two solutions are experimentally evaluated by a fault injection campaign in Sec. 3.8; finally Sec. 3.9 concludes the discussion and outlines our future research.
3.2 Background on Temporal Logic Languages

This section introduces the basic concepts related to temporal logics, presents a classification of temporal logic languages, highlights that the best suited languages for runtime detection are the propositional linear temporal logic (PLTL) languages and presents the syntax and semantics of such a language.

3.2.1 Basic Concepts

Temporal logic languages were originally suggested by Pnueli [139] in 1977 for reasoning about concurrent programs. Since then several researchers have used temporal logic variants for checking and proving temporal correctness of software, communication protocols and hardware devices.

The core concept of checking temporal criteria is to define a finite state-transition system representing an abstraction of the system and check that specific propositions hold for execution traces of the system. This abstraction is usually presented by a Kripke-structure. A Kripke-structure (KS) can be seen as a very simple state machine consisting of simple (i.e., not reﬁneable, non-concurrent, etc.) states, transitions and labels assigned to states.

Definition 52 (Kripke-Structure). Having a label set $P$ a Kripke-structure is a three-tuple: $K = (S, I, T, L)$ where $S$ is the set of states, $I \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is the state transition relation and $L$ is the labeling function that assigns labels to states: $L : S \rightarrow 2^P$.

Definition 53 (Finite Trace and Trace Suffix). A $\Pi = (s_0, s_1, \ldots, s_n-1)$ finite trace of the Kripke-structure is a sequence of states connected by the state transition relation: $s_0, s_1, \ldots, s_{n-1} \in S$, $n > 0$ and $\forall 0 \leq i < n(s_i-1, s_i) \in T$. A $\Pi^j = (s_j, s_{j+1}, \ldots, s_{n-1})$ trace suffix is obtained from the trace $\Pi = (s_0, s_1, \ldots, s_{n-1})$ by removing the first $j$ steps ($j < n$). By definition: $\Pi^0 \equiv \Pi$.

Example 24. Let us consider a server that may be idle, or working. While working the server may be receiving a request from a client, processing or delivering the result of the operation. In this simple example the server has four states that correspond to four stages: (i) idle, (ii) receiving a request, (iii) processing the request and (iv) delivering the result. Obviously while not being idle, the server is considered to be working. The five attributes above (idle, working, receiving a request, processing and delivering the result) can be mapped to a $P = \{idle, working, receiving, processing, delivering\}$ set of labels. A possible KS representing the server $K_{Server} = (S, T, L)$ may be as follows:

\[
S = \{s_0, s_1, s_2, s_3\}, \ I = \{s_0\} \tag{3.1}
\]

\[
T = \{(s_0, s_1), (s_1, s_2), (s_2, s_3), (s_3, s_1)\} \tag{3.2}
\]

\[
L(s_{j<3}) = \begin{cases} 
\{idle\} & \text{if } s = s_0 \\
\{working, receiving\} & \text{if } s = s_1 \\
\{working, processing\} & \text{if } s = s_2 \\
\{working, delivering\} & \text{if } s = s_3 
\end{cases} \tag{3.3}
\]

It is easy to see that the states of the Kripke-structure are organized into a loop ($s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_0$) and $s_0$ represents the idle status, $s_1$ represents the reception of a request, $s_2$ represents processing the request and $s_3$ represents the delivery of the results. Since while both the reception of a request, processing and delivery of the result the server is considered to be working, states $s_1$, $s_2$ and $s_3$ have two labels, one is working and the another one corresponding to the actual phase. The Kripke-structure is shown visually in the left side of Fig. 3.3, the states are represented by circles, labels are listed in callouts and transitions are arrows.

Example 25. A possible finite execution trace of $K_{Server}$ above is $\Pi = (s_0, s_1, s_2, s_3, s_0, s_1, s_2, s_3, s_0)$; examples for suffixes of $\Pi$ may be: $\Pi^0 = (s_0, s_1, s_2, s_3, s_0, s_1, s_2, s_3, s_0)$, $\Pi^1 = (s_1, s_2, s_3, s_0, s_1, s_2, s_3, s_0)$, $\Pi^5 = (s_1, s_2, s_3, s_0)$, etc. (right side of Fig. 3.3).
3.2. BACKGROUND ON TEMPORAL LOGIC LANGUAGES

Temporal logic languages can be classified according to various criteria: (i) with respect to the operator set a temporal logic may be propositional (temporal and classic propositional logic operators may be used) or first order (temporal, propositional and even first order operators may be used); (ii) the evaluation of temporal operators may be focused to atomic points of time or to time intervals; (iii) with respect to the granularity of time there are discrete time and continuous time temporal logics; (iv) the notion of subsequence may be simply interpreted over a linear timeline or, in branching time temporal logics, the life of a system is considered as a tree of states where states may have multiple subsequent states and (v) the temporal operators may refer to the future of the system or the past or both.

As the goal of our approach is to define a language for runtime detection of errors in UML statechart implementations, the following observation can be taken: (i) as we can observe only the actual single execution of the system, we need a propositional language; (ii) as statecharts operate in run-to-completion steps, temporal operators are to be evaluated at atomic points of time and (iii) obviously the granularity of time is a discrete; (iv) as the black box nature of the approach prevents us from reasoning about all execution traces therefore we will use of linear timeline semantics; (v) the fact that temporal operators refer to the past or the future is of secondary importance, we choose the future-time syntax similarly to most approaches. We can summarize the results of the classification and discussion above as follows: we need a propositional linear temporal logic language of discrete time semantics for reasoning about the execution of UML 2.0 statechart implementations.

3.2.3 A Propositional Linear Temporal Logic Language

Below we present an inductive definition of the syntax of PLTL formulae and introduce the notion of a formula \( \phi \) being true at a suffix \( \Pi^i \) denoted by \( \Pi^i \models \phi \):

- \( \Pi \models p \) iff \( p \in L(s_0) \) (Atomic propositions) (3.4)
- \( \Pi \models \neg e \) iff \( \Pi \models e \) is not true (Boolean not) (3.5)
- \( \Pi \models e_1 \land e_2 \) iff \( \Pi \models e_1 \) and \( \Pi \models e_2 \) (Boolean and) (3.6)
- \( \Pi \models Xe \) iff \( n > 1 \land \Pi^i \models e \) (Temporal next) (3.7)
- \( \Pi \models e_1 U e_2 \) iff \( \exists 0 \leq i < n, \Pi^i \models e_2 \) and \( \forall 0 \leq j < i, \Pi^j \models e_1 \) (Temporal until) (3.8)

Informally: An atomic proposition is true if the corresponding label is a member of the labeling set of the first step of the suffix. The application of Boolean operators is obvious, \( X \) refers to the next suffix of the trace, \( e_1 U e_2 \) requires the condition \( e_1 \) to hold until \( e_2 \) becomes true. We will also use the usual Boolean constants (\( \top \) and \( \bot \)), the or (\( \lor \)) and implication (\( \rightarrow \)) operators and the finally \( F \) and globally \( G \) temporal operators. The meaning of Boolean operators is obvious, \( F e \) represents the proposition that there is a suffix of the trace for which \( e \) holds (possibly even starting with the actual state or only sometime in the future) and \( G e \) represents that \( e \) holds for all suffixes of the trace:

- \( \Pi \models e_1 \lor e_2 \) iff \( \Pi \models \neg(e_1 \land \neg e_2) \) (Boolean or) (3.9)
- \( \Pi \models e_1 \rightarrow e_2 \) iff \( \Pi \models \neg(e_1 \lor e_2) \) (Boolean implication) (3.10)
- \( \Pi \models F e \) iff \( \Pi \models \top U e \) (Temporal finally) (3.11)
- \( \Pi \models G e \) iff \( \Pi \models \neg F \neg e \) (Temporal globally) (3.12)
CHAPTER 3. ERROR DETECTION IN UML 2.0 STATECHART IMPLEMENTATIONS

The fact that we are presenting a finite trace semantics can be captured in the definition of the $\mathcal{X}$ next-time and $\mathcal{U}$ operators: (i) for $\mathcal{X}e$ being true at a trace suffix $\Pi$ we require that the trace is longer than 1 i.e., that the trace actually has a next state (Eq. 3.7) and (ii) for $\mathcal{U}e$ being true we require that $e$ should evaluate to true before the end of the trace. Finally we present some examples.

**Example 26.** In case of a bus system we can require that “whenever setting line $A$ to high, line $B$ should be low in the next cycle”. Having a label set $P = \{(l^A_{High}), (l^B_{Low})\}$ representing the high state of $A$ and the low state of $B$ respectively, the requirement can be formalized as: $G(l^A_{High} \rightarrow X(l^B_{Low}))$.

**Example 27.** Let us consider the server above (Example 24). Let the label set $P = \{i, w, r, p, d\}$ represent idle, working, receiving a request, processing and delivering the result respectively. For the correct operation we may require that “whenever the server receives a request from the client it should be processing it without interruption until the delivery”. This can be formalized as: $G(r \rightarrow (pU d))$.

### 3.3 A Temporal Logic Language for Statecharts

This section defines PSC-PLTL, a temporal logic language for precise statecharts. According to our knowledge there were only a few proposals published in the literature about mapping temporal logic languages to artifacts of statechart diagrams. Sowmya and Ramesh in [153] proposed the FNLOG language based on predicate calculus and temporal logic for specifying and proving various dependability criteria on statecharts primarily aiming at model checking. Drusinsky proposed TLCharts [34, 35] that are extensions of statecharts allowing temporal expressions to be used in guard predicates primarily to be used in his Temporal Rover [33] commercial tool.

In this section we will define a temporal logic language for UML 2.0 statecharts by providing concrete meaning to atomic predicates of PLTL in the context of statecharts.

For mapping a temporal logic language for UML 2.0 statecharts we have to identify the finite state-transition model of the language (i.e., a Kripke structure) and introduce a set of atomic propositions that refer to artifacts of statecharts (states, transition conglomerates, activities etc.).

Since the operational semantics of UML 2.0 statecharts was defined by a Kripke transition system (KTS), we do not have to build the Kripke structure (KS) model of the language from scratch, our only task is to transform the Kripke transition system into a Kripke structure.

The transformation is nearly just a syntactic rewriting as follows. Let a KTS $S^{KTS}$ be specified by the tuple $S^{KTS} = (S, \Rightarrow, I)$ over the AP set of state labels and Act set of transition labels where $S$ is the set of states, $\Rightarrow \subseteq S \times Act \times S$ is the labeled transition relation and $I : S \rightarrow 2^{AP}$ the state labeling function. As we would like to construct a KS that represents the same semantics as the KTS, we have to move the information on KTS transition labels into the KS state labels by using $(s, a_i)$ pairs as states where $s$ is an original state of the KS and $a_i$ is the label of a transition targeting $s$. This way the KS is specified by the tuple $S^{KS} = (S \times Act, R, I')$ over AP $\cup$ Act state labels where the transition relation is $R = \{(s, a_i), (s', a'_i)\} | (s, a_i, s') \in \Rightarrow$ and the state labeling function is $I'(s, a_i) = I(s) \cup \{a_i\}$. The rewriting is highlighted in Tab. 3.1.

As the transition labeling information was moved into the state labeling function $L$, state labels are now six-tuples. Similarly to the definition of the KTS we decompose the composite labeling function to six sub-functions for obtaining the (i) configuration (Eq. 3.13), (ii) evaluation of variables (Eq. 3.14)
3.3. A TEMPORAL LOGIC LANGUAGE FOR STATECHARTS

and (iii) phase of operation (Eq. 3.15) from the original state labeling and (iv) the trigger that initiated the step (Eq. 3.16), (v) the set of transition conglomerates fired in the step (Eq. 3.17) and (vi) the activity structure performed in the step (Eq. 3.18) from the original transition labeling.

\[
\begin{align*}
&\mathcal{L}_{\text{Cnf}}^S : \hat{S} \rightarrow 2^{\text{State}} \\
&\mathcal{L}_{\text{Eval}}^S : \hat{S} \rightarrow \text{Eval} \\
&\mathcal{L}_{\text{Phs}}^S : \hat{S} \rightarrow \text{Phs} \\
&\mathcal{L}_{\text{Trigger}}^\Delta : \hat{S} \rightarrow \text{Trigger} \cup \{a\} \\
&\mathcal{L}_{\text{TC}}^\Delta : \hat{S} \rightarrow 2^{\text{TC}} \\
&\mathcal{L}_{\text{PERTGraph}}^\Delta : \hat{S} \rightarrow \text{PERTGraph}
\end{align*}
\]

\[
\begin{align*}
&\mathcal{L}_{\text{Cnf}}^S((s, (t, TC_t, p))) = \mathcal{L}_{\text{Cnf}}^S(s) & \text{(Configuration)} \\
&\mathcal{L}_{\text{Eval}}^S((s, (t, TC_t, p))) = \mathcal{L}_{\text{Eval}}^S(s) & \text{(Evaluation of variables)} \\
&\mathcal{L}_{\text{Phs}}^S((s, (t, TC_t, p))) = \mathcal{L}_{\text{Phs}}^S(s) & \text{(Phase of operation)} \\
&\mathcal{L}_{\text{Trigger}}^\Delta((s, (t, TC_t, p))) = t & \text{(Trigger)} \\
&\mathcal{L}_{\text{TC}}^\Delta((s, (t, TC_t, p))) = TC_t & \text{(Transition conglomerates)} \\
&\mathcal{L}_{\text{PERTGraph}}^\Delta((s, (t, TC_t, p))) = p & \text{(Activity structure)}
\end{align*}
\]

Definition of a temporal logic language consists of the definition of the Boolean and temporal operators used and the atomic propositions. We use the Boolean not and and operators and the temporal next-time and until operators according to their usual syntax and semantics as discussed in Sec. 3.2 (Eq. 3.5, Eq. 3.6, Eq. 3.7, and Eq. 3.8 respectively).

Atomic propositions are derived from state labels of the KS: (i) the predicate \( \mathcal{P}_{\text{Cnf}}(C) \) is true if all states in \( C \) are active in the actual configuration (Eq. 3.15), (ii) \( \mathcal{P}_{\text{Phs}}(p) \) is true if the actual phase of operation is \( p \) (Eq. 3.20), (iii) \( \mathcal{P}_{\text{Trigger}}(t) \) is true if the trigger \( t \) in the actual state was initiated by the trigger \( t \) (Eq. 3.21), (iv) \( \mathcal{P}_{\text{TC}}(TC_t) \) is true if the step ending in the actual configuration of the statechart all transition conglomerates in \( TC_t \) were fired (Eq. 3.22) and (v) \( \mathcal{P}_{\text{PERTGraph}}(g) \) is true if the PERT graph of activities performed during the step ending in the actual configuration is not less restrictive than \( g \) (i.e., the subsequence relations required by \( g \) were met) (Eq. 3.23).

Having discussed the language syntax and semantics we have defined the temporal logic language for precise statecharts. We will refer to the language as PSC-PLTL for simplicity. For convenience we will also use the shorthand operators defined in Sec. 3.2 for expressing logical or by \( \lor \) (Eq. 3.9), logical implication by \( \rightarrow \) (Eq. 3.10), logical true by \( \top \), logical false by \( \bot \), temporal finally by \( F \) (Eq. 3.11) and temporal globally by \( G \) (Eq. 3.12).

**Example 28.** Finally we present some examples for PSC-PLTL expressions in the context of a valve controller’s the statechart in Fig. 3.7. Since the initial configuration of the statechart is \( \{s_{\text{ValveOpen}}, s_{\text{Closing}}\} \), the expression \( \mathcal{P}_{\text{Cnf}}(\{s_{\text{ValveOpen}}, s_{\text{Closing}}\}) \) is true. It is obviously true that “at any time one of states \( s_{\text{ValveClosed}} \) or \( s_{\text{ValveOpen}} \) is active”: \( G(\mathcal{P}_{\text{Cnf}}(\{s_{\text{ValveClosed}}\}) \lor \mathcal{P}_{\text{Cnf}}(\{s_{\text{ValveOpen}}\})) \). We may require that “when being in the state \( s_{\text{ValveOpen}} \) the state \( s_{\text{ValveClosed}} \) should finally become active” (i.e.,
the execution may not end in while being the valve open: \( G(\mathcal{P}_{\text{Caf}}(s_{\text{ValveOpen}})) \rightarrow \mathcal{F}(\mathcal{P}_{\text{Caf}}(s_{\text{ValveClosed}})) \). Finally we can require that “whenever receiving a trigger danger in the state \( s_{\text{ValveOpen}} \) (regardless its active substates), the safe state \( s_{\text{ValveClosed}} \) should be active”: \( G((\mathcal{P}_{\text{Caf}}(s_{\text{ValveOpen}})) \land X \mathcal{P}_{\text{Trigger}}(\text{danger})) \rightarrow \mathcal{F}(\mathcal{P}_{\text{Caf}}(s_{\text{ValveClosed}})) \).

3.4 Previous Approaches for PLTL Evaluation

Effective evaluation of linear temporal logic formulae on execution traces is definitely a non-trivial issue. Below we will highlight the drawbacks of naive approaches, discuss the advanced methods proposed in the literature and highlight the benefits and drawbacks of various approaches. Since our temporal logic language (PSC-PLTL) defined in Sec. 3.3 is specific to statecharts only in the semantics of atomic propositions, below we will discuss evaluation of PLTL expressions in general without restricting the observations and achievements to PSC-PLTL.

3.4.1 Naive Methods

The most obvious naive approach is based on iterative decomposition of formulae: when having an expression in the form \( e_1 U e_2 \) to be evaluated on the trace suffix \( \Pi^i \) the original expression is to be rewritten as \( e_1 U e_2 = e_2 \lor (e_1 \land X(e_1 U e_2)) \) (see Eq. 3.35 later), while the evaluation of \( \Pi^i \models X e \) is semantically equivalent to the evaluation of \( \Pi^{i+1} \models e \).

Example 29. During the evaluation of \( pU d \) over the trace \( \Pi \) (i.e., \( \Pi \models pU d \)), the original until sub-expression is rewritten as: \( \Pi \models pU d \rightarrow (d \lor (p \land X(pU d))) \), and the next-time operator can be substituted as: \( \Pi \models pU d \rightarrow (d \lor (p \land (\Pi^1 \models pU d))) \). The \( \Pi^1 \models pU d \) sub-expression can be rewritten in the second iteration similarly: \( \Pi \models pU d \rightarrow (d \lor (p \land (d \lor (p \land (\Pi^2 \models pU d))))), \) etc. The evaluation of \( r \rightarrow (pU d) \) is similar, the only difference is in the first step i.e., \( \Pi \models r \rightarrow (pU d) \) is to be rewritten as \( \Pi \models r \rightarrow (pU d) \rightarrow r \land \Pi^0 \models pU d \) and the rewriting on the until sub-expression is the same: \( \Pi \models r \rightarrow (pU d) \rightarrow r \land (d \lor (p \land (\Pi^1 \models pU d))) \) The iterative decomposition of the two formulae is shown in parts (a) and (b) of Fig. 3.7. It is easy to see that the highlighted parts of the trees representing the evaluations of the expressions are repeated in all steps of the trace therefore the \( N_{\text{Expr}} \) number of nodes and \( L_{\text{Expr}} \) number of leaves (atomic propositions to be evaluated) of expression \( Expr \) for the two expressions above in case of an \( n \) steps trace are as follows: \( N_{pU d} = 6n \), \( L_{pU d} = 2n \), \( N_{r \rightarrow (pU d)} = 6n + 4 \) and \( L_{r \rightarrow (pU d)} = 2n + 1 \).

This iterative decomposition is usable for simple expressions where temporal until sub-expressions \( (e_1 U e_2) \) regenerate themselves in one instance only during the subsequent steps but in case of non-trivial expressions until sub-expressions may be generated in more than one instance, resulting in an ever-growing tree of expression to be evaluated.

Example 30. Showing such an example is very easy, e.g., expressions in the form \( G(e_1 U e_2) \) are such cases: when rewriting the globally construct according to the definition (Eq. 3.12) two until sub-expressions will occur:

\[
\begin{align*}
(\Pi \models G(e_1 U e_2)) & \rightarrow (\Pi \models \neg F(\neg(e_1 U e_2))) \rightarrow (\Pi \models \neg (T U \neg(e_1 U e_2))) \\
(\Pi \models \neg (e_1 U e_2) \lor (T \land X(T U \neg(e_1 U e_2)))) & \rightarrow (\Pi \models \neg (e_1 U e_2) \lor (T \land \Pi^1 \models (T U \neg(e_1 U e_2))))
\end{align*}
\]

The naive iterative evaluation of \( G(r \rightarrow (pU d)) \) is shown in part (e) of Fig. 3.7. For easier understanding we used the \( (\Pi \models G e) \rightarrow (\Pi \models e \land X G e) \) rewriting:

\[
\begin{align*}
G e = \neg F \neg e = \neg (T U \neg e) = \neg \neg e \lor (T \land X(T U \neg e)) = e \land \neg (T \land X(T U \lor e)) = e \land \neg X(T U \neg e) = e \land X \neg F \neg e = e \land X G e = e \land X(G e)
\end{align*}
\]
3.4. PREVIOUS APPROACHES FOR PLTL EVALUATION

3.4.1 Evaluation of LTL Formulae

FIGURE 3.7: Naive Evaluation of LTL Formulae

It is easy to see that the evaluation of $G(r \rightarrow p Ud)$ starts the evaluation of $r \rightarrow p Ud$ in all steps of the trace (sub-trees left from the $\land$ symbol under $G(r \rightarrow p Ud)$), therefore the $N_{G(r \rightarrow p Ud)}^n$ number of nodes and $L_{G(r \rightarrow p Ud)}^n$ number of leaves can be derived from $N_{r \rightarrow p Ud}^n$ and $L_{r \rightarrow p Ud}^n$:

$$N_{G(r \rightarrow p Ud)}^n = \left( \sum_{i=1}^{n} N_{r \rightarrow p Ud}^i \right) + 2n = \left( \sum_{i=1}^{n} N_{r \rightarrow p Ud}^i \right) + 2n = \left( \sum_{i=1}^{n} (6i + 4) \right) + 2n = 3n^2 + 9n \quad (3.26)$$

$$L_{G(r \rightarrow p Ud)}^n = \sum_{i=1}^{n} L_{r \rightarrow p Ud}^i = \sum_{i=1}^{n} (1 + 2i) = n^2 + 2n \quad (3.27)$$

Since we will use the latter expression in another example we present the “canonical” form of evaluation using the core LTL operators only (i.e., without $G$, $\lor$ and $\rightarrow$) in Fig. 3.8.

As summary we can observe that even for such simple expressions the number of nodes and leaves of the naive evaluation tree is a proportional to the second power of the length of the trace.

3.4.2 Method Proposed in the Literature

Evaluation of LTL formulae over infinite traces is a typical model checking problem. The idea of solution is as follows: (i) let $\phi$ be the formula expressing the property to be proven about the system; (ii) first the formula is negated ($\neg \phi$), then (iii) the $A_{\neg \phi}$ Büchi automaton is constructed that accepts sentences that meet $\neg \phi$ i.e., violate the requirement $\phi$, next (iv) the automaton $A_P$ of the $P$ program to be checked is constructed; finally (v) the $A_{\neg \phi \times P}$ multiplication of two automata $A_{\neg \phi}$ and $A_P$ is constructed that accepts those executions of $A_P$ that violate $\phi$: the correctness of the program $P$ is proven by proving that $A_{\neg \phi \times P}$ accepts the empty language. The construction of $A_{\neg \phi}$ is based on some variants of the algorithm published by Gerth, Peled, Vardi and Wolper in [49]. The well-known model checker SPIN by Holzmann is based on this approach, and there were multiple improvements proposed in the literature by Vardi [29], Holzmann [44] and others [48, 152].
As the infinite trace semantics of automaton theoretic approaches mentioned above can not be applied to finite traces [59], checking finite traces has received growing attention recently. Havelund and Roşu presented a solution based on an efficient formula rewriting technique [59, 141] using the Maude term-rewriting system [24] and applied the achievements in their PathExplorer tool [60].

Some code generation approaches have also been proposed: Finkbeiner and Sipma discussed the implementation of alternating automata in Java [45], Rodriguez, Fabre and Arlat defined a method [142] for developing wrappers for real-time systems based on temporal logic specifications. The Temporal Rover tool by Drusinsky [33] is capable of translating temporal formulae into executable assertions for various programming languages and hardware description formats (being a commercial tool the method of evaluating formulae in the generated code is not public).

In spite of the diversity of verification methods and tools mentioned above the approaches share a common problem: they are not directly applicable in low-cost embedded systems where the processing power and memory constraints heavily restrict the applicable technology. The formula-rewriting technique discussed by Havelund and Roşu in [59, 60, 141] requires the heavy-weight Maude term-rewriting engine; the Java VM required for executing the code generated in the automaton-theoretic approach from Finkbeiner and Sipma [45] is typically not available in embedded systems. The C code generated in the wrapping approach may be very effective both in terms of processing power requirements and memory consumption but the specification language in [142] heavily restricts the expressive power of the linear temporal logic by allowing the usage of some important temporal operators in special cases only (the \( \mathcal{U} \) “until” operator may only be used for defining the \( \mathcal{G} \) “globally” and \( \mathcal{F} \) “finally” constructs).

To put together: since naive approaches do not scale to long execution traces while the resource
requirements of previously proposed approaches prevent their application in resource constrained embedded systems, we have to develop a novel solution for evaluation of PLTL formulae over finite traces. Our proposal is discussed in Sec. 3.5.

### 3.5 A Novel High-Performance PLTL Evaluation Method

This section presents a novel, high-performance method for evaluating propositional linear temporal logic formulae over finite execution traces. As contrary to formula-rewriting and automaton-theoretic approaches our solution is very close to the engineering practice: the idea is introduced on the basis of a straightforward hardware analogy that is transformed into data structures and algorithms to be implemented in conventional imperative programming languages like C, C++ or even Java, C#, etc.

#### 3.5.1 Overview of the Idea

The core idea of our approach is illustrated by a straightforward hardware analogy in Fig. 3.9. The left side of the figure presents the naive evaluation of $G(r \rightarrow p Ud)$ (similarly to Fig. 3.8). First, we will identify the drawbacks of the naive implementation and present the idea of our solution illustrated in the right side of Fig. 3.9. The naive iterative decomposition builds a tree of PLTL formulae:

- The root of the tree is the expression to be evaluated ($G(r \rightarrow (p Ud))$) that is rewritten to $(\neg (r \land (\neg d \land (\neg (p \land X(p Ud))))) \land \neg (\neg (\neg (d \land X(TU(r \land (\neg (p Ud))))))))$ using the definition of the $G$ operator and the $E_1 \cup E_2 \equiv \neg (E_2) \lor (E_1 \land X(E_1 \cup E_2))$ equivalence.

Figure 3.9: Overview of the Idea
The atomic propositions \( (r, d, p, \top) \) in the leaves of the tree are to be evaluated on the appropriate suffix (i.e., state) of the trace that is indicated on the left side.

When encountering a next-time expression \( X(e) \) to be evaluated over \( \Pi^i \) the \( e \) sub-expression is taken out from the scope of \( X \) and it is evaluated over \( \Pi^{i+1} \), e.g., in the example \( X(\top U(r \land \neg(pUd))) \) to be evaluated over \( \Pi^0 \) yields \( \top U(r \land \neg(pUd)) \) to be evaluated over \( \Pi^1 \). The next-time expressions rewritten this way are highlighted by white rounded boxes, the resulting expressions are in orange-yellow rounded boxes, rewriting is indicated by blue arrows.

Iteratively rewriting the formulae and taking out the expressions from the scopes of the \( X \) operators results in a tree with multiple layers corresponding to a trace suffixes. The layers are separated by dashed tan lines in the example, suffixes are indicated by blue signs in the middle of the figure (i.e., the topmost layer corresponds to suffix \( \Pi^0 \), the middle layer to \( \Pi^1 \), etc.).

The most important drawback of the naive method clearly appears on the tree: since the approach does not recognize the need for systematically collecting the expressions to be evaluated over specific suffixes it tends to evaluate the same expression over multiple times over the same suffix e.g., the \( pUd \) expression is evaluated once over \( \Pi^1 \), twice over \( \Pi^2 \), three times over \( \Pi^3 \) etc. because \( pUd \) re-generates itself during the rewriting and \( \top U(r \land \neg(pUd)) \) generates a new \( pUd \) expression in each step (this is the reason for having \( N^3_{G(r \rightarrow pUd)} = 3n^2 + 9n \) tree nodes in Example 30).

The goal of our approach is to introduce a straightforward data structure for systematically collecting and maintaining the evaluation of expressions generated during the rewriting steps preventing this way the explosion of the formula set that renders naive approaches unusable for non-trivial traces. Havelund and Rosu proposed a similar idea in [59, 60, 141] based on introducing a formula transforming function (called “event consumption” in their terminology).

Our proposal is closer to engineers’ way of thinking: we use a straightforward hardware analogy and present a code generation method for automatically synthesizing a software implementation that is effective both in the terms of processing power requirements and memory consumption.

The idea of our solution is shown on the right side of Fig. 3.9, subsequent steps of the evaluation correspond to “logic circuits” that encapsulate a piece of evaluation logic. The enclosed logic is similar to corresponding part of the evaluation trees on the left side. The “logic circuits” have three interfaces:

- **Top interfaces** contain outputs representing the expressions to be evaluated on the corresponding suffix: when an expression gets evaluated on a specific suffix its value appears on the top interface of the corresponding “logic circuit”. In Fig. 3.9 the single expression on the top interface of the topmost node is \( \neg(\top U(r \land \neg(pUd))) \) (i.e., \( G(r \rightarrow (pUd)) \) after substituting the shorthand operators), the expressions on the top interfaces of the two other nodes are: \( pUd \) and \( \top U(r \land \neg(pUd)) \). The top interface can be seen as the specification of the circuit: the entire internal logic and the other interfaces can be derived from this expression set.

- **Left interfaces** contain inputs representing the atomic propositions to be evaluated on the first state of the corresponding suffix. All “logic circuits” in the example have the \( r, d \) and \( p \) propositions on their left interfaces.

- **Bottom interfaces** contain inputs representing the next-time expressions that can only be evaluated with respect to some future behavior. All “logic circuits” in the example have the \( X(pUd) \) and \( X(\top U(r \land \neg(pUd))) \) expressions on their bottom interfaces.

The structure prevents the anomaly of naive implementations by representing all next-time expressions only once on the bottom interface regardless how many times they were generated (e.g., the middle and the bottom circuits have only one \( X(pUd) \) inputs on their bottom interfaces although this value is connected to two gates in the internal logic).

In the following we will imagine “logic circuits” that they have registers on their inputs and outputs. The left interface contains binary registers since atomic propositions can always be evaluated during
3.5. A NOVEL HIGH-PERFORMANCE PLTL EVALUATION METHOD

runtime by investigating the labels of the corresponding state (e.g., after the observed application enters the second state of the trace $s_1$, the binary registers on the left interface of the middle circuit in the example can be loaded appropriately). The registers on the two other interfaces can not be implemented so simply: since during runtime the value of next-time expressions may be unknown until observing some future behavior, the top and the bottom interfaces should be able to reflect this behavior by operating in ternary logic. (Operators of ternary logic ($\neg_3$ and $\land_3$) are discussed later.) We will use the $\top$ and $\bot$ symbols for the true and false Boolean values and the $?$ symbol for the unknown value of the ternary logic.

As mentioned above the top interface completely defines the internal logic and the other two interfaces of “logic circuits”: we will introduce a normal form that expressions on the top interfaces are to be converted to and this normal form will define the internal logic. Having specified the internal logic, the definition of the left and the bottom interface is just the collection of atomic propositions and next-time sub-expressions that are used in the evaluation.

It is important to highlight that the hardware analogy is just a straightforward illustration of the idea: from software engineering point of view the “logic circuits” and their internal structure correspond to data types and algorithms. In the following sections we will first discuss how to model these constructs independently of the concrete programming language using UML class diagrams and pseudo-code, then we will outline our implementation targeting the C++ language. We will refer to classes representing the “logic circuits” as evaluation nodes and the entire interconnected structure of evaluation nodes as evaluation chain.

3.5.2 Iterative Decomposition of Formulae

As mentioned above, “logic circuits” are completely defined by the expressions on their top interfaces: we know what we have to evaluate, we need to define how to do this (internal logic) and identify the information we need for the evaluation (left and bottom interfaces). This subsection introduces the mathematical foundations for building the internal logic and interfaces of evaluation nodes.

The implementation of Boolean operations does not need much discussion, atomic propositions and next-time expressions are to be represented on the left and bottom interfaces respectively, only the until temporal operator ($\mathcal{U}$) requires special handling, since it refers to both the present state and to the future behavior. Expressions built using this operator should be transformed in order to move the $\mathcal{U}$ operator in the argument of an $\mathcal{X}$ operator. As the first step we introduce a normal form for expressions that contain the $\mathcal{U}$ operator only in the argument of an $\mathcal{X}$ operator (Def.

**Definition 54 (Connection Normal Form).** A PLTL expression is in connection normal form (CNF) if the $\mathcal{U}$ operator appears only in the argument of an $\mathcal{X}$ operator.

**Theorem 3 (Conversion to Connection Normal Form).** For any PLTL formula $f$ there exists a PLTL formula $c$ in connection normal form that for any trace $j = f_i^j = c$.

**Proof.** Below we prove the theorem by introducing the $\text{To}^{\text{CNF}}(e)$ notation as the “conversion of the LTL expression $e$ to CNF” and extend the conversion to $E$ sets of expressions for simplicity ($\text{To}^{\text{CNF}}_{\text{Set}}$): the only non-trivial transformation is $e_1 \mathcal{U} e_2$ is shown in Eq. (3.35).

\[
\begin{align*}
\text{To}^{\text{CNF}}(p_{p \in P}) &= p \\
\text{To}^{\text{CNF}}(\top) &= \top \\
\text{To}^{\text{CNF}}(\neg e) &= \neg \text{To}^{\text{CNF}}(e) \\
\text{To}^{\text{CNF}}(\mathcal{X} e) &= \mathcal{X} e \\
\text{To}^{\text{CNF}}(e_1 \land e_2) &= \text{To}^{\text{CNF}}(e_1) \land \text{To}^{\text{CNF}}(e_2) \\
\text{To}^{\text{CNF}}(e_1 \mathcal{U} e_2) &= \text{To}^{\text{CNF}}(\neg(\neg e_2 \land \neg(e_1 \land \mathcal{X}(e_1 \mathcal{U} e_2)))) \\
\text{To}^{\text{CNF}}_{\text{Set}}(E) &= \bigcup_{e \in E} \{\text{To}^{\text{CNF}}(e)\}
\end{align*}
\] (3.28) (3.29) (3.30) (3.31) (3.32) (3.33) (3.34)
Chapter 3. Error Detection in UML 2.0 Statechart Implementations

This means that when having a root expression set \( e_1 \), taking out the arguments of the expression sets \( X \) resulting in the top interface of the next evaluation node, etc.

Example 31. In the example of Fig. 3.9 the internal logic of evaluation nodes was constructed by converting the formulae on the top interfaces to CNF:

\[
T_0^{\text{CNF}}(p \land d) = \neg((d) \land \neg(p \land X(p \land d)))
\]

\[
T_0^{\text{CNF}}(\neg(r \land \neg p)) = \neg((r \land \neg d) \land \neg(p \land X(p \land d)))
\]

\[
T_0^{\text{CNF}}((\exists t \land (r \land \neg(p \land d))) = \neg((r \land \neg d) \land \neg(p \land X(p \land d)))
\]

Having converted the expression to be evaluated (top interface) to CNF we can define the left and bottom interfaces simply by collecting the atomic propositions and next-time sub-expressions (Def. 55).

Definition 55 (Atomic and Next-Time Subexpressions). The operators for collecting the atomic and next-time subexpressions of a CNF expression are defined below iteratively as \( \text{Collect}^A \) and \( \text{Collect}^X \) respectively. For convenience we extend the operators for \( \text{sets of expressions} \) (\( \text{Collect}^A_{\text{Set}} \) and \( \text{Collect}^X_{\text{Set}} \) respectively).

\[
\text{Collect}^A(p \in r) = \{p\}
\]

\[
\text{Collect}^A(\top) = \emptyset
\]

\[
\text{Collect}^A(e_1 \land e_2) = \text{Collect}^A(e_1) \cup \text{Collect}^A(e_2)
\]

\[
\text{Collect}^A(\neg e) = \text{Collect}^A(e)
\]

\[
\text{Collect}^A(X e) = \emptyset
\]

\[
\text{Collect}^A_{\text{Set}}(E) = \bigcup_{e \in E} \text{Collect}^A(e)
\]

Example 32. In Fig. 3.9 the left and bottom interfaces of evaluation nodes were constructed by converting the formulae on the top interfaces to CNF and collecting the atomic propositions and next-time sub-expressions (for CNF forms see Example 31):

\[
\text{Collect}^A_{\text{Set}}(T_0^{\text{CNF}}(\exists t \land (r \land \neg(p \land d)))) = \{r, p, d\}
\]

\[
\text{Collect}^X_{\text{Set}}(T_0^{\text{CNF}}(\exists t \land (r \land \neg(p \land d)))) = \{X (p \land d), X(T \land U(r \land \neg(p \land d)))\}
\]

The bottom interface of an evaluation node defines the top interface of the next node in the chain. This means that when having a root expression set \( R \) to be evaluated we can specify all “logic circuit types” to be used during the evaluation by iteratively converting the set of expressions on the actual top interface to CNF, collecting the next-time sub-expressions, taking out the arguments of the \( X \) operators resulting in the top interface of the next “circuit” etc.

Definition 56 (Deriving expressions). The \( D = \text{Drv}^N(R) \) notation defined below indicates that the expression set \( D \) can be derived from the root expression set \( N \) steps. The set of expression sets that can be derived in any steps from the set \( R \) is \( D^* = \text{Drvs}(R) \).

\[
\text{Drv}^0(R) = R
\]

\[
\text{Drv}^{N+1}(R) = \text{Drv}^N(\text{Collect}^X_{\text{Set}}(R))
\]

\[
\text{Drvs}(R) = \{D | \exists N : D = \text{Drv}^N(R) \land D \neq \emptyset\}
\]
Instances

<table>
<thead>
<tr>
<th>Types</th>
<th>Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{EN}_{\neg(\top U(r \wedge \neg(pU d)))}$</td>
<td>$\mathcal{EN}^{\mathbb{T}}_{\neg(\top U(r \wedge \neg(pU d)))}$</td>
</tr>
<tr>
<td>$\mathcal{EN}_{(pU d, \top U(r \wedge (pU d)))}$</td>
<td>$\mathcal{EN}^{\mathbb{T}}<em>{(pU d, \top U(r \wedge (pU d)))}, \mathcal{EN}^{\mathbb{B}}</em>{(pU d, \top U(r \wedge (pU d)))}$</td>
</tr>
</tbody>
</table>

Table 3.2: Evaluation Node types in the Example

It is easy to see that $\text{Drvs}(R)$ is a finite set by considering that $E_1 U E_2$ constructs re-generate themselves and any next-time expressions in the root set are extracted by the formula deriving process. After a sufficient number of deriving steps the resulting expression set will be empty (in trivial cases where there were no $E_1 U E_2$ constructs in the root set) or re-generate itself (e.g., as in Fig. 3.9).

Example 33. In the example of Fig. 3.9 the top interfaces of the “logic circuits” were derived from the root set containing a single expression $\neg(\top U(r \wedge \neg(pU d)))$ (note that (i) $\neg(\top U(r \wedge \neg(pU d)))$ is $\mathcal{G}(r \rightarrow (pU d))$ using core operators only and (ii) Drv$^1$ functions expect and return expressions in CNF; since the CNF forms of expressions in our example are long and complex (see Example 3.1) we used non-CNF formats below and the $\text{To}_{\mathbb{CNF}}$ symbol for readability purposes):

\[
\begin{align*}
\text{Drv}^0(\text{To}_{\mathbb{CNF}} ((\neg(\top U(r \wedge \neg(pU d)))))) &= \text{To}_{\mathbb{CNF}} ((\mathcal{G}(r \rightarrow (pU d)))) \quad (3.52) \\
\text{Drv}^1(\text{To}_{\mathbb{CNF}} ((\neg(\top U(r \wedge \neg(pU d)))))) &= \text{To}_{\mathbb{CNF}} ((pU d, \top U(r \wedge \neg(pU d)))) \quad (3.53) \\
\text{Drvs}(\text{To}_{\mathbb{CNF}} ((\mathcal{G}(r \rightarrow (pU d)))) &= (\text{To}_{\mathbb{CNF}} ((\mathcal{G}(r \rightarrow (pU d)))), \text{To}_{\mathbb{CNF}} ((pU d, \top U(r \wedge \neg(pU d)))))) \quad (3.54)
\end{align*}
\]

3.5.3 Evaluation Node Types and Instances

The notations above define a complete toolkit for specifying “logic circuits”: the Drvs method returns the expressions on the top interfaces of all “circuits” while using the Collect$^\mathbb{A}$ and Collect$^\mathbb{X}$ utilities we can define the left and bottom interfaces. Based on them we introduce the evaluation node type abstraction (Def. 5.7) as the mathematical formalism corresponding to “logic circuits”: evaluation nodes are defined by the $E$ set of expressions to be evaluated by the node and contain three functions corresponding to the top, left and bottom interface respectively and a pointer $P$ referring to the previous node in the chain.

Definition 57 (Evaluation Node Types). Having an $E$ set of CNF expressions the corresponding evaluation node type $\mathcal{EN}_E$ is a four-tuple: $\mathcal{EN}_E = (P, T, L, B)$, where: $P$ is a reference to an evaluation node instance or the 0 empty symbol, $T$, $L$ and $B$ (top, left and bottom) are functions:

\[
\begin{align*}
T : E &\rightarrow \{\top, \bot, ?\} \\
L : \text{Collect}^\mathbb{A}_{\text{Sat}}(E) &\rightarrow \{\top, \bot\} \\
B : \text{Collect}^\mathbb{X}_{\text{Sat}}(E) &\rightarrow \{\top, \bot, ?\}
\end{align*}
\]

We will use the following notation for referring to the values of the $P$, $T$, $L$ and $B$ fields: $\mathcal{EN}_E : \text{P}$ (“previous instance”), $\mathcal{EN}_E : \text{L}[a] \in \text{Collect}^\mathbb{A}_{\text{Sat}}(E)$ (“value of atomic proposition $a$ on the left interface”), $\mathcal{EN}_E : \text{T}[e] \in E$ (“value of expression $e$ on the top interface”) and $\mathcal{EN}_E : \text{B}[x] \in \text{Collect}^\mathbb{X}_{\text{Sat}}(E)$ (“value of next-time expression $x$ on the bottom interface”). We will refer to an instance of the evaluation node type $\mathcal{EN}_E$ belonging to the trace suffix $\Pi'$ as $\mathcal{EN}^{\Pi'}_E$.

Example 34. In Fig. 3.9 there are two evaluation node types and the chain consists of three instances as shown in Tab. 3.2 (expressions were not converted to CNF for readability purposes).

Having defined the concept of evaluation nodes we are practically ready to present the method for constructing the implementation-independent static structure of the runtime error detection system. The basic data types corresponding to the $\mathcal{G}(r \rightarrow (pU d))$ (i.e., $\neg(\top U(r \wedge \neg(pU d)))$) example are shown in the left part of Fig. 3.10. The binary and ternary values used during the evaluation of formulae
are represented by the Binary and Ternary enumerated types (their possible values are true, false and unknown corresponding to the unknown value notion of the ternary logic).

While dealing with PLTL formulae we have to catch somehow the notion of various expressions (top interface expressions, atomic propositions, next-time sub-expressions etc.). These expressions are introduced in the left part of Fig. 3.10 as enumerated types TopExpr, LeftExpr and BottomExpr respectively. (The enumeration values are constructed by appending the expression as a subscript to a prefix string TE, LE and BE respectively).

These enumerated types can be automatically constructed from the root expression set $R$ by deriving all the top interfaces using $D = Drvs(R)$ for constructing TopExpr and using the Collect$^X_A$ and Collect$^X_{Set}$ methods for constructing LeftExpr and BottomExpr respectively.

Evaluation node types are obviously mapped to classes in the static structure model (Fig. 3.10 right side). In order to facilitate the benefits of uniform handling and behavior reuse we introduce an abstract base class $EN$. Names of classes representing evaluation node types are prefixed with $EN$ and indicate the expressions on the top interface in subscript. The fields of evaluation node types are mapped to artifacts of the UML model: the $P$ reference to the previous instance is a self-association in the base class. The multiplicity of the association is “0..1” representing the fact that the $P$ field may contain the 0 empty symbol (zero multiplicity).

The three interfaces are represented as functions $t$, $l$ and $b$ in the derived classes (top, left and bottom respectively). The $\mathcal{D}$ domain of the functions is derived from the set of expressions on the top interface, e.g., in case of the $EN_{\neg(\top l(r \wedge \neg(p \land d)))}$ class:

\begin{equation}
\mathcal{D}(EN_{\neg(\top l(r \wedge \neg(p \land d)))}) : t = \{TE_{\neg(\top l(r \wedge \neg(p \land d)))}\} \tag{3.58}
\end{equation}

\begin{equation}
\mathcal{D}(EN_{\neg(\top l(r \wedge \neg(p \land d)))}) : l = \{LE_x, LE_y, LE_z\} \tag{3.59}
\end{equation}

\begin{equation}
\mathcal{D}(EN_{\neg(\top l(r \wedge \neg(p \land d)))}) : b = \{BE_x(p \land d), BE_y(\neg(\top l(r \wedge \neg(p \land d)))\} \tag{3.60}
\end{equation}

As the static structure model can not restrict the assignments made by the $t$, $l$ and $b$ functions according to the semantics we introduce the notion of valid evaluation node instances as a validity assertion about the fields of nodes needed for proving the correctness of algorithms.

**Definition 58 (Valid Evaluation Node).** An evaluation node is said to be valid if it does not hold inconsistent data i.e., its left interface is filled according to the corresponding step of $\Pi$ and its top and bottom interfaces hold valid Boolean values or the ? unknown value; formally $EN^\Pi_E$ is valid if the followings apply:

- $EN^\Pi_E : P$ contains the 0 empty symbol if the node belongs to the first step of the trace, otherwise it points to another valid node $EN^{\Pi^{i-1}}_E$ where $E = Drv^1(F)$.

- $EN^\Pi_E : T$ assigns the ? uncertain value to $e \in E$ or indicates the validity of $\Pi^{i} \models e$ (i.e., $EN^\Pi_E : T[e] = \top$ if $\Pi^{i} \models e$ and $\bot$ if $\Pi^{i} \not\models e$) (Eq. 3.61 – Eq. 3.64).
3.5.4 Basic Operations on Evaluation Nodes

This subsection presents the basic operations on evaluation nodes. First we will define methods for (i) creating the evaluation node instance corresponding to the first step of the trace (Head, Def. 59) and (ii) for copying instances that are to be connected after an already-existing one (Next, Def. 60). These methods just choose the appropriate evaluation node type, load the left interface according to the labeling of the actual state and initialize the “registers” on the top and bottom interfaces to uncertain values. Next we define information propagation methods for (i) evaluating top-interface expressions within nodes with respect to the values stored in left and bottom interfaces (IntProp, Def. 62) and (ii) for copying values from the top interface of a node to the bottom interface of the previous node in the chain i.e., between nodes (ExtProp, Def. 63).

Definition 59 (Head(\(\Pi^0, R\))). The “head node” procedure returns an instance \(\text{EN}^{\Pi^0}_R\) corresponding to the first step of trace \(\Pi\) and root expression set \(R\) where (i) \(\text{EN}^{\Pi^0}_R :: \text{P} \) contains the 0 empty symbol (Eq. 3.69), (ii) \(\text{EN}^{\Pi^0}_R :: \text{T} \) assigns the uncertain value to all \(r \in R \) (Eq. 3.70), (iii) \(\text{EN}^{\Pi^0}_R :: \text{L} \) holds the labels of \(s_0\) (first step of \(\Pi\) (Eq. 3.71) and (iv) \(\text{EN}^{\Pi^0}_R :: \text{B} \) assigns the uncertain value to all \(x \in \text{Collect}_{\text{Set}}^X(R)\) expressions (Eq. 3.72):

\[
\begin{align*}
\text{EN}^{\Pi^0}_R :: \text{P} & = 0 \quad (3.69) \\
\text{EN}^{\Pi^0}_R :: \text{T}[r] & \leftarrow R \quad (3.70) \\
\text{EN}^{\Pi^0}_R :: \text{L}[a] & = \begin{cases} T & \text{if } a \in L(s_0) \\ \perp & \text{if } a \notin L(s_0) \end{cases} \quad (3.71) \\
\text{EN}^{\Pi^0}_R :: \text{B}[x] & \leftarrow \text{Collect}_{\text{Set}}^X(R) \quad (3.72)
\end{align*}
\]

Definition 60 (Next(\(\text{EN}^{\Pi^0}_F\))). The “next node” procedure when called with a valid \(\text{EN}^{\Pi^0}_F\) instance returns \(\text{EN}^{\Pi^1}_F\) corresponding to the suffix \(\Pi^1\) where \(F = \text{Drv}^1(E) \neq \emptyset\) and (i) \(\text{EN}^{\Pi^1}_F :: \text{P} \) refers to \(\text{EN}^{\Pi^0}_F\) (the node corresponding to the previous step) (Eq. 3.73), (ii) \(\text{EN}^{\Pi^1}_F :: \text{T} \) assigns the uncertain value to all \(f \in F \) (Eq. 3.74), (iii) \(\text{EN}^{\Pi^1}_F :: \text{L} \) holds the labels of \(s_{i+1}\) (first step of \(\Pi^{i+1}\) (Eq. 3.75) and (iv) \(\text{EN}^{\Pi^1}_F :: \text{B} \) assigns the uncertain value to all \(x \in \text{Collect}_{\text{Set}}^X(F)\) expressions (Eq. 3.76):

\[
\begin{align*}
\text{EN}^{\Pi^1}_F :: \text{P} & = \text{EN}^{\Pi^0}_F \quad (3.73) \\
\text{EN}^{\Pi^1}_F :: \text{T}[f] & \leftarrow R \quad (3.74)
\end{align*}
\]
Table 3.3: Truth Table of Ternary Logic Operators

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>a ∧₃ b</th>
<th>¬₃ a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊤</td>
</tr>
<tr>
<td></td>
<td>{T, ⊥, ?}</td>
<td>⊥</td>
<td>⊥</td>
<td>⌦</td>
</tr>
<tr>
<td>1</td>
<td>⊥</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>{T, ?}</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

The truth table of the ternary logic operators

\[
\begin{align*}
\text{EN}^{π+1}_F : L[a]_{|a \in \text{Collect}^F_\text{Set}(E)} &= \begin{cases} 
T & \text{if } a \in L(s_{i+1}) \\
\bot & \text{if } a \notin L(s_{i+1})
\end{cases} \\
\text{EN}^{π+1}_F : B[x]_{|a \in \text{Collect}^x R_\text{Set}(E)} &= ?
\end{align*}
\]

(3.75) (3.76)

From a programming point of view Head and Next are constructors or factory methods of the corresponding EN₀ class: after initialization the object is ready for usage. In the next step we will define the IntProp procedure (Def. 62) that attempts to evaluate the expressions on the top interface of a node that are still uncertain i.e., the informal semantics of the procedure is: “Walk through the expressions on the top interface and try to re-evaluate the uncertain ones (\(\text{?} \) value); if the evaluation succeeds (i.e., the result is not uncertain any more) store this value in the top interface.” As the evaluation involves operations on ternary logic values, below we introduce first the Eval₃ utility function (Def. 61) and define the truth table of ternary logic operators.

**Definition 61 (Eval₃(EN₃₁, e)).** The “evaluate ternary” utility function evaluates the top-interface expression \(e\) according to the semantics of ternary logic in the context of EN₃₁:

\[
\begin{align*}
\text{Eval}_3(\text{EN}^{π}_E, a_{|a \in \text{Collect}^A_\text{Set}(E)}) &= \text{EN}^{π}_E :: L[a] \\
\text{Eval}_3(\text{EN}^{π}_E, T) &= T \\
\text{Eval}_3(\text{EN}^{π}_E, e \wedge f) &= \text{Eval}_3(\text{EN}^{π}_E, e) \wedge_3 \text{Eval}_3(\text{EN}^{π}_E, f) \\
\text{Eval}_3(\text{EN}^{π}_E, \neg e) &= \neg_3 \text{Eval}_3(\text{EN}^{π}_E, e) \\
\text{Eval}_3(\text{EN}^{π}_E, \forall \, e) &= \text{EN}^{π}_E :: B[\forall \, e]
\end{align*}
\]

(3.77) (3.78) (3.79) (3.80) (3.81)

The truth table of the ternary logic operators \(\wedge_3\) and \(\neg_3\) is given in Tab. 3.3. The semantics behind the definition is as follows: (i) for the \(\neg_3\) operator: “we may say that an expression \(\neg_3 e\) is true if we are sure (i.e., not uncertain) that \(e\) is false and we may say that an expression \(\neg_3 e\) is false if we are sure that \(e\) is true, otherwise we are unsure” and (ii) for the \(\wedge_3\) operator: “we may say that an expression \(e_1 \wedge_3 e_2\) is true if we are sure that both \(e_1\) and \(e_2\) are true and we may say that an expression \(e_1 \wedge_3 e_2\) is false if we are sure that any of \(e_1\) or \(e_2\) is false, otherwise we are unsure”.

**Definition 62 (IntProp(EN₃₁)).** The “internal propagate” utility procedure modifies the EN₀ function of the valid instance EN₃₁ to EN₃₁ :: T' that will assign the value returned by Eval₃ to the expressions \(e \in E\). The procedure returns the set of expressions (Updated) where the assignment was made certain (i.e., changed from \(\text{?}\) to \(T\) or \(\bot\)).

\[
\text{EN}^{π}_E :: T'[e] = \text{Eval}_3(\text{EN}^{π}_E, e)
\]
\[
\text{Updated} = \{ e \in E | \text{EN}^{π}_E :: T'[e] = ? \wedge \text{EN}^{π}_E :: T'[e] \in \{ T, \bot \} \}
\]

(3.82) (3.83)

The information propagation between nodes is implemented by the ExtProp method: we copy the certain values from the top interface of EN₃₁ to the bottom interface of EN₃₁. A special kind of propagation is to be performed when the trace terminates: the procedure TermTrace fills the bottom interface of the corresponding instance with \(\bot\) values according to the semantics of the \(\forall\) operator.
3.5. A NOVEL HIGH-PERFORMANCE PLTL EVALUATION METHOD

Listing 3.1: Pseudocode of the RecProp Method

Definition 63 (ExtProp(EN\textsuperscript{II}_E, d|_{d \in \text{Drv}\textsuperscript{I}(E)}, v|_{v \in \{T, \bot\}})). The “external propagate” utility procedure modifies the EN\textsuperscript{II}_E :: B function of the valid evaluation node instance EN\textsuperscript{II}_E to EN\textsuperscript{II}_E :: B' that will assign v to ancestors of d:

\[ EN\textsuperscript{II}_E :: B'[x(e)] = \begin{cases} 
  v & \text{if } d \text{ is } T_0^{\text{CNF}}(e) \\
  EN\textsuperscript{II}_E :: B[X(e)] & \text{otherwise} 
\end{cases} \tag{3.84} \]

Definition 64 (TermTrace(EN\textsuperscript{II}_E^{n-1})). The “terminate trace” utility procedure modifies the EN\textsuperscript{II}_E :: B function of the valid evaluation node instance EN\textsuperscript{II}_E^{n-1} corresponding to the last state of the trace to EN\textsuperscript{II}_E^{n-1} :: B' that will assign the \( \bot \) value to all next-time sub-expressions \( x \) of the \( E \) set:

\[ EN\textsuperscript{II}_E^{n-1} :: B'[x] = \bot \]

Definition 65 (IsReady(EN\textsuperscript{II}_E)). An evaluation node is said to be ready (IsReady(EN\textsuperscript{II}_E) returns \( T \)) if all expressions on the top interface of the node have a certain value, (i.e., not \( ? \)):

\[ \text{IsReady}(EN\textsuperscript{II}_E) = \begin{cases} 
  T & \text{if } \forall e \in E : EN\textsuperscript{II}_E :: T[e] \in \{T, \bot\} \\
  \bot & \text{otherwise} 
\end{cases} \tag{3.85} \]

3.5.5 Algorithm of Evaluation

Finally we will present the top-level algorithms RecProp and CheckTrace for encapsulating the utility methods above.

Definition 66 (RecProp(EN\textsuperscript{II}_E)). The RecProp method encapsulates the activity to be carried out after observing a next step in the trace and inserting the corresponding instance into the evaluation chain. The informal semantics of the method is as follows: “walk through the chain starting with the last node and try to make certain as many expressions on top interfaces as possible and propagate these values until the head element is reached or the recursive propagation gets stuck; in the return value indicate whether we are ready or not” (the procedure returns \( T \) if the recursive propagation reaches the head of the chain and all the expressions on its top interface are certain). The definition of RecProp(EN\textsuperscript{II}_E) is presented in Listing 3.1. When called with EN\textsuperscript{II}_E, the algorithm first attempts to evaluate the expressions on the top interface that are still uncertain (2). If EN\textsuperscript{II}_E is the head instance (3) returns the readiness of the node (4). If EN\textsuperscript{II}_E is not the head instance (5) investigates whether there were expressions made certain in step 1: if there are values to be propagated (6) the algorithm copies these values to the bottom interface of the previous node (7, 8) and recursively continues this operation on the previous node (9). When there were nothing to be propagated in step 5 (10) the operation terminates and the method returns \( \bot \) indicating that we are not yet ready (11).

Definition 67 (CheckTrace(R, II)). The topmost-level function CheckTrace defined in Listing 3.2 encapsulates the entire evaluation: the algorithm evaluates \( r \in R \) expressions over the finite trace II.
CHAPTER 3. ERROR DETECTION IN UML 2.0 STATECHART IMPLEMENTATIONS

Listing 3.2: Pseudocode of the CheckTrace Method

After the execution the top interface of $EN_{head}$ will indicate whether $\Pi \models e$ or not. The algorithm maintains three variables: $EN_{head}$ (2) and $EN_{tail}$ (3) referring to the head element of the chain and to the actually last element respectively and the $i$ trace index (4). Until the end of the trace is reached (5) or the $RecProp$ procedure returns $\top$ indicating that all root expressions were evaluated (8) the algorithm iteratively inserts a next element at the end of the chain using the $Next$ procedure (9). If the end of the trace is reached before evaluating all root expressions, the bottom interface of the last evaluation node instance is filled with $\perp$ values using $TermTrace$ (10) and $RecProp$ is called (11).

3.5.6 Correctness and Algorithmic Complexity

It can be proven (App. C.1) that CheckTrace constructs and maintains valid nodes throughout its entire execution (i.e., there will not appear inconsistent values at the top interface of the head node), always terminates and evaluates all expressions in the top interface of the head node (i.e., all $r \in R$ root expressions will be evaluated). Furthermore it can be proven (App. C.2) that the execution time of the algorithm is a linear function of the trace length i.e., of $\mathcal{O}(n)$ algorithmic complexity.

3.5.7 Implementation Considerations

Based on the theoretical background discussed above we have implemented a code generator that generates C++ source implementing the algorithms and data structures based on a set of expressions to be evaluated. In our implementation the enumerated types in Fig. 3.10 (Binary and Ternary data types and the $LeftExpr$, $BottomExpr$ and $TopExpr$ expression identifiers) are mapped to enum constructs evaluation node classes in Fig. 3.10 are mapped to C++ classes, the reference to the previous instance (field $p$) is implemented by a pointer and “registers” on interfaces are implemented by arrays addressed by the enumeration identifiers mentioned above.

In our implementation the $Head$ procedure is a static method of the $EN$ base class, $Next$ is implemented by virtual member functions in derived classes. A key performance issue is the implementation of the ternary logic evaluation ($Eval_3$) used by $IntProp$: these expressions are translated to preprocessor macros that are expanded into inline expressions and compiled to few machine code instructions by the compiler. $IntProp$, $ExtProp$, $TermTrace$ and $IsReady$ are also implemented by virtual member functions in classes derived from $EN$. The $RecProp$ method is a static function of $EN$, while the $CheckTrace$ top-level algorithm is implemented as a stand-alone function. A detailed example for code generation is presented in App. C.3.

It is easy to see that the key considerations on the mapping of abstract data structures and algorithms to source code are not specific to programming languages, practically all modern object-oriented languages (C++, Java, C#) can be used, even including the ANSI C language.

3.5.8 Memory Consumption

The memory consumed by our approach for evaluating an $R$ set of root expressions on a $\Pi$ trace of $n$ steps involves the code implementing our algorithms (ROM consumption) and the memory used for
storing evaluation node instances (RAM consumption). Since our algorithms can be implemented in some lines of code obviously the RAM usage dominates the memory consumption. Below we will focus on the RAM consumption (memory needed for storing evaluation node instances) in the context of our implementation; although some calculations will be specific to the C++ language, the results are applicable to another language ports of our approach with minor modifications.

An evaluation node instance maintains the values stored on its interfaces and the pointer to the previous instance. The “registers” on the interfaces of evaluation node instances can be implemented by bit vectors: storing the values of atomic expressions on the left interfaces require a single bit per expression while ternary values on the top and bottom interfaces need at least two bits per expression (storing bit patterns is supported by C bit-fields, the C++ bitset data type template, etc.). The pointer to the previous instance is obviously implemented as a programming language level pointer or reference construct.

This way the \( I_{\text{EN}} \) number of bytes required for storing the raw data represented by an evaluation node instance of class \( \text{EN} \) is theoretically the sum of number of bytes required for storing (i) two bits per expressions on the top interface (ternary logic), (ii) a single bit per expressions on the left interface (binary logic), (iii) two bits per expressions on the bottom interface (ternary logic) and (iv) the \( p \) size of a pointer in bytes:

\[
I_{\text{EN}} = \left\lceil \frac{2|E|}{8} \right\rceil + \left\lceil \frac{\text{Collect}_{\delta_{\text{Set}}}(E)}{8} \right\rceil + \left\lceil \frac{2|\text{Collect}_{\text{Set}}(E)|}{8} \right\rceil + p
\]  

(3.86)

Added to the raw data size above, the programming language may add some housekeeping data for each instances, that may increase the memory required for storing an instance. This overhead may include type information, metadata etc. In case of the C++ programming language chosen for our prototype implementation, no explicit metadata or type information is stored, only in case of classes with virtual functions a single pointer to the virtual function pointer table (some of our data propagations methods are naturally implemented by virtual functions). Below we present this “C++–speciﬁc mapping” of the expression above where \( p \) is the size of a pointer (note that for speed optimization purposes the compiler may decide to word-align data members etc.; these minor compiler and architecture speciﬁc deviations were not taken into consideration):

\[
I_{\text{EN}}^{\text{C++}} = p + \left\lceil \frac{2|E|}{8} \right\rceil + \left\lceil \frac{\text{Collect}_{\delta_{\text{Set}}}(E)}{8} \right\rceil + \left\lceil \frac{2|\text{Collect}_{\text{Set}}(E)|}{8} \right\rceil + p
\]  

(3.87)

\textbf{Example 35.} Let us consider a 32 bit architecture where a pointer is 4 bytes long. In this case the number of bytes required for the evaluation of the expression \( G(r \rightarrow (p \text{U} d)) \) in the example of Fig. 3.9 on a trace of 1000 steps using \( I_{\text{EN}} \) is as follows:

\[
I_{\text{EN}}(G(r \rightarrow (p \text{U} d))) + 999(I_{\text{EN}}(p \text{U} d)) + 8) = (4 + 1 + 1 + 4) + 999(4 + 1 + 1 + 4) = 11000
\]

(3.88)

Unfortunately we can not directly compare the memory consumption of our approach to the ones published in the literature since neither Havelund and Roșu, nor Finkbeiner and Sipma presented similar analysis in their corresponding papers. Anyway, according to our experiments, Havelund and Roșu’s implementation run out of memory for such traces that were seamlessly processed by our approach (Finkbeiner and Sipma’s implementation was not available) and in asymptotic aspects we are quite sure that none of the other approaches can reduce the memory consumption below a linear function.

### 3.5.9 Experimental Evaluation of Performance

We have compared the performance of our implementation with the approaches proposed by Havelund and Roșu in \[59\] (efficient term-rewriting using the Maude engine) and the one proposed by Finkbeiner
Table 3.4: Comparison of Execution Times with Havelund and Roșu’s Approach ($\phi_1$ and $\phi_2$)

<table>
<thead>
<tr>
<th>Trace length</th>
<th>Formula: $\phi_1 = G(b \rightarrow F c)$</th>
<th>Execution time [ms]</th>
<th>Speedup</th>
<th>Formula: $\phi_2 = F(\neg G(b \rightarrow F c))$</th>
<th>Execution time [ms]</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>H&amp;R</td>
<td>Our</td>
<td></td>
<td></td>
<td>H&amp;R</td>
<td>Our</td>
<td></td>
</tr>
<tr>
<td>1500</td>
<td>20</td>
<td>0.53</td>
<td>37×</td>
<td>110</td>
<td>0.62</td>
<td>177×</td>
</tr>
<tr>
<td>3000</td>
<td>40</td>
<td>1.10</td>
<td>36×</td>
<td>220</td>
<td>1.25</td>
<td>176×</td>
</tr>
<tr>
<td>4500</td>
<td>60</td>
<td>1.64</td>
<td>36×</td>
<td>320</td>
<td>1.87</td>
<td>171×</td>
</tr>
<tr>
<td>6000</td>
<td>80</td>
<td>2.20</td>
<td>36×</td>
<td>420</td>
<td>2.64</td>
<td>159×</td>
</tr>
<tr>
<td>7500</td>
<td>100</td>
<td>2.70</td>
<td>37×</td>
<td>530</td>
<td>3.29</td>
<td>161×</td>
</tr>
<tr>
<td>9000</td>
<td>120</td>
<td>3.70</td>
<td>32×</td>
<td>640</td>
<td>3.84</td>
<td>166×</td>
</tr>
<tr>
<td>10500</td>
<td>140</td>
<td>4.00</td>
<td>35×</td>
<td>760</td>
<td>4.47</td>
<td>170×</td>
</tr>
<tr>
<td>12000</td>
<td>160</td>
<td>4.50</td>
<td>35×</td>
<td>860</td>
<td>5.50</td>
<td>156×</td>
</tr>
<tr>
<td>13500</td>
<td>180</td>
<td>5.10</td>
<td>35×</td>
<td>970</td>
<td>6.00</td>
<td>161×</td>
</tr>
<tr>
<td>15000</td>
<td>200</td>
<td>5.60</td>
<td>36×</td>
<td>1100</td>
<td>6.60</td>
<td>166×</td>
</tr>
</tbody>
</table>

Table 3.5: Comparison of Execution Times with Havelund and Roșu’s Approach ($\phi_4$)

<table>
<thead>
<tr>
<th>Trace length</th>
<th>Formula: $\phi_4 = G(((a \land X b) \lor (b \land X c)) \lor (c \land X d) \lor (d \land X c)) \cup (d \land X c))$</th>
<th>Execution time [ms]</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>H&amp;R</td>
<td>Our</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15000</td>
<td>1660</td>
<td>2.4</td>
<td>340×</td>
</tr>
<tr>
<td>30000</td>
<td>2430</td>
<td>7.2</td>
<td>338×</td>
</tr>
<tr>
<td>45000</td>
<td>3700</td>
<td>10.3</td>
<td>359×</td>
</tr>
</tbody>
</table>

and Sipma in [45] (Java source code generation based on alternating automata). In order to reduce the possibility of choosing benchmark cases for which an implementation behaves especially good or bad we have used the formulae and traces suggested in the corresponding papers.

Comparison with Havelund and Roșu’s Approach

Havelund and Roșu present the elegant and compact implementation of their system in [59]: we have installed the Maude system, copied the implementation from the paper and carried out performance experiments. There are three benchmark formulae suggested in [59] as follows: $\phi_1 = G(b \rightarrow F c)$, $\phi_2 = F(\neg G(b \rightarrow F c))$ and $\phi_3 = G(((a \land X b) \lor (b \land X c)) \cup (a \land X c))$. Havelund and Roșu also suggest a synthetic benchmark trace for evaluation of formulae: the trace is to be created by iterative repetition of the $(a, b, a, b, a, c, a, a, b, g, f, h, c, b, a)$ fragment. It is easy to see that for the resulting II traces $\Pi \not\models \phi_1$, $\Pi \models \phi_2$ and $\Pi \not\models \phi_3$. The evaluation of $\phi_1$ and $\phi_2$ requires the analysis of the entire trace while $\phi_3$ fails after a few steps, this way we will focus on formulae $\phi_1$ and $\phi_2$ only. In Tab. 3.4 present the execution times for evaluating the two formulae on various trace lengths (the measurements were carried out on a Pentium IV machine running at 2.2GHz, equipped with 512 MBytes of DDR-233MHz memory).

As indicated in the table, our implementation was about 36× faster in case of the simpler $\phi_1$ formula while about 160× faster in case of the more complex $\phi_2$ formula. The impressive speedup can be explained by the fact that in a term-rewriting system the evaluation of formulae requires several formal term rewriting steps while in our solution these critical operations are compiled to few machine code instructions (see the discussion about the implementation of Eval3 above). This observation induces that the more complicated the formulae the faster is our solution as compared to the term-rewriting approach. In order to prove this statement we have investigated the behavior of the two implementations in case of the more complex $\phi_4 = G(((a \land X b) \lor (b \land X c)) \lor (c \land X d) \lor (d \land X c)) \cup (d \land X c)$ formula and a similarly created trace (iterative repeating of $(a, b, c, d)$ fragment) and we observed that our implementation was about 340× faster (Tab. 3.5).
3.5. A NOVEL HIGH-PERFORMANCE PLTL EVALUATION METHOD

### Table 3.6: Comparison of Execution Times with Finkbeiner and Sipma’s Approach

<table>
<thead>
<tr>
<th>Formula</th>
<th>Trace length</th>
<th>Execution time [ms]</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>depth-first</td>
<td>breadth-first</td>
</tr>
<tr>
<td>$\phi_1$</td>
<td>1000</td>
<td>1733</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td>2000</td>
<td>8402</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>3000</td>
<td>21940</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>4000</td>
<td>44185</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>5000</td>
<td>76899</td>
<td>123</td>
</tr>
<tr>
<td>$\phi_2$</td>
<td>1000</td>
<td>40</td>
<td>82</td>
</tr>
<tr>
<td></td>
<td>2000</td>
<td>79</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>3000</td>
<td>112</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>4000</td>
<td>222</td>
<td>94</td>
</tr>
<tr>
<td></td>
<td>5000</td>
<td>247</td>
<td>117</td>
</tr>
<tr>
<td>$\phi_3$</td>
<td>1000</td>
<td>45</td>
<td>876</td>
</tr>
<tr>
<td></td>
<td>2000</td>
<td>87</td>
<td>1660</td>
</tr>
<tr>
<td></td>
<td>3000</td>
<td>185</td>
<td>2377</td>
</tr>
<tr>
<td></td>
<td>4000</td>
<td>217</td>
<td>3244</td>
</tr>
<tr>
<td></td>
<td>5000</td>
<td>298</td>
<td>4034</td>
</tr>
</tbody>
</table>

Comparison with Finkbeiner and Sipma’s Approach

Since in case of the approach by Finkbeiner and Sipma the implementation was not available, we built a similar configuration as used by them and carried out the same experiments as presented in [45]. This comparison is especially interesting since two code generation approaches were compared.

Finkbeiner and Sipma present three algorithms in their paper based on (i) depth-first search, (ii) breadth-first search strategies and (iii) an algorithm based on the reverse traversal of the trace. The third algorithm was indicated to be of highest performance in [45]; unfortunately this algorithm requires the entire trace to be available for evaluating the formulae therefore not applicable for runtime error detection where always only a prefix of the trace is available that has been observed until that point.

Finkbeiner and Sipma suggested formulae $\phi_1 = G F z$, $\phi_2 = G F a$ and $\phi_3 = G (b \rightarrow \neg a U (a U (\neg a U a)))$ for experimental evaluation of the performance and define a trace format by specifying the average number of states with labels $a$, $b$, $c$, $d$. In order to ensure the satisfaction of formulae and easier comparison, a state with label $z$ was added to the end of the trace in case of $\phi_1$ and a state with label $a$ for $\phi_2$ and $\phi_3$. We used the same formulae and generated benchmark traces as specified above. As the implementation was not available, we could only compare the two approaches for the quite short traces ($n = 1000 \ldots 5000$) discussed in [45] (execution times for Finkbeiner and Sipma’s implementations were taken from the same paper). The comparison is presented in Tab. 3.6. The last two columns indicate the comparison of performance delivered by our solution as compared to Finkbeiner and Sipma’s (i) actually fastest algorithm (note that the fastest solution may not be applicable for runtime error detection) and (ii) the fastest algorithm that was actually usable for runtime error detection (columns Fastest and Applicable respectively).

As shown in the tables, the differences in the execution times were not as high as in the previous comparison: in case of the $G F z$ and $G F a$ formulae our implementation was about $30 - 40 \times$ faster than the fastest algorithm that is applicable for runtime error detection while in case of the more complicated $G (b \rightarrow \neg a U (a U (\neg a U a)))$ formula our implementation was about $40 - 50 \times$ faster.

Although the performance difference between the Java Virtual Machine and the native application are not negligible, this does not explain the $30 - 40 \times$ and $40 - 50 \times$ speedup; the explanation can be again the highly effective implementation of sub-expression evaluation and the inherently programming-oriented nature of our approach: the entire solution is targeted for code generation, the data structures and algorithms seamlessly fit to modern programming languages resulting in a straightforward code generation step and a very high performance application. (The preliminary Java implementation of our approach is about $3 - 5 \times$ faster in case of $G F z$ and $G F a$ while $3 - 8 \times$ faster in case of the $G (b \rightarrow \neg a U (a U (\neg a U a)))$ formula when compared to the fastest applicable algorithm in [45].)
3.5.10 Summary

Based on the PLTL evaluation method discussed here we implemented a tool for evaluating PSC-PLTL expressions on execution traces of precise statechart implementations. As PSC-PLTL is bound to statechart artifacts only by its atomic propositions, the source code synthesized by our code generator was usable for evaluating PSC-PLTL formulae over execution traces of statechart implementations after a simple preprocessing phase.

3.6 Background on Watchdog Processors

The detection of errors caused by faults in the implementation of a UML 2.0 statechart is practically the comparison of an abstract behavior specification (the fully elaborated statechart model) and the actually exposed behavior.

The behavior of a program can be described by the control flow graph (CFG) whose nodes are the stable states of the program, and directed edges are the possible steps between program states. The notion of a program state depends on the abstraction level of modeling: (i) when describing an algorithm at the level of machine instructions, a node in the CFG is a sequence of instructions that are executed one after another without branches, while directed edges correspond to branch instructions; (ii) when describing a program by a high-level programming language, nodes are subsequent branch free statements, edges appear as function calls, conditional execution (e.g., if, switch etc. constructs), various loops organization schemes (e.g., for, while, etc.) or explicit jump instructions (e.g., goto); (iii) when describing a program by a statechart, nodes of the CFG correspond to state configurations of the statechart, while edges are steps (as described by \( \xrightarrow{\text{init}} \), \( \xrightarrow{\text{open}} \), etc. in Chp. 1) of the statechart.

Various successful approaches has been proposed in the literature for detecting control flow errors caused by faults in the underlying hardware using the machine instruction level or statement level description of programs as reference information. Obviously when aiming at detection of errors caused by implementation faults, we can not use the implementation itself as reference, we have to rely on the more abstract statechart model.

Solutions aiming at detection of control flow errors are usually called watchdogs for historical reasons. Watchdog circuits attached to interfaces of devices were originally used for detecting such lethal conditions as loss of clock signal, power outage or lack of some life signals within a period of time etc. – in these cases the watchdog circuit was responsible for resetting the CPU or switching to a backup system. **Watchdog processors (WP)** \(^{[91]}\) are able to perform more sophisticated checks on the operation of the main processor: the WP observes the actual behavior of the main processor and checks it against the correct control flow graph. The description of the correct control flow is called the reference information while the actually observed behavior is called the runtime information.

Runtime information is represented by compacted pieces of observed behavior called signatures (e.g., address of the actually fetched instruction without the actual operation code and arguments). The WP can obtain the runtime signatures in two ways: (i) observing the system buses (called derived signatures) or (ii) explicitly receiving messages from the main processor (called assigned signatures). The benefit of using derived signatures is the inherently non-intrusive nature of the approach since the application running on the main processor needs not be modified and obtaining the signatures does not pose any extra overhead on the bus communication; the drawback of this solution is the fact that the operation of most modern microprocessors can not be observed because of predictive prefetching techniques, caching mechanisms, pipeline organization, out of order execution etc. therefore derived signatures are applicable now only in case of simple microcontrollers. The assigned signature technique is based on modifying the program running on the main processor to explicitly transmit signatures similarly to sending data to any external device; the obvious drawback is the need for modifying the main program but this seems to be the only applicable solution for making the execution observable on modern processors.
3.6. BACKGROUND ON WATCHDOG PROCESSORS

There were three solutions proposed in the literature for the representation of the reference information: (i) storing the correct CFG in a database in the WP, (ii) executing a watchdog program on the WP that accepts correct runs of the main program or (iii) the reference information can be embedded in signatures themselves. The stored database approach uses a traditional method (adjacency matrix or adjacency list) for storing the directed control flow graph in the memory of the WP; unfortunately this approach may result in significant memory consumption (e.g., large, sparse adjacency matrix) or time-consuming lookup operations (e.g., searching for an edge in the adjacency list). The watchdog program approach considers the valid executions of the main program as sentences of a language where the words are signatures of execution; the watchdog program is language parser that accepts valid sequences of signatures i.e., valid executions of the main program; it is easy to see that having built the valid CFG of the main program, it is relatively easy to automatically synthesize such an automaton that accepts valid runs. The embedded signatures approach uses composite signatures whose first part identifies the actual point of execution (similarly to plain signatures used in other two approaches) and the second part identifies the set of signatures that may follow the actual one in a correct execution.

There were proposals for all combinations of representing runtime and reference information mentioned above. Derived signatures were used together (i) with stored database in Eifert and Shen’s Asynchronous Signatured Instruction Stream approach [41], (ii) with watchdog program by Michel, Leveugle and Saucier in [103] and (iii) with embedded signatures in Path Signature Analysis approaches of Namjoo [99] and Shridar and Thatte [154]. Assigned signatures were used together (i) with stored database in Michel and Hohl’s Extended Structural Integrity Checking approach [102], (ii) with watchdog program in Lu’s Structural Integrity Checking method [83] and (iii) with embedded signatures in the Signature Encoded Instruction Stream (SEIS) approach from Majzik et al. [92, 94, 95, 112].

The actual implementation of the watchdog processor can be (i) a real physical processor, but there were approaches proposed for (ii) utilizing the unused resources of the main processor for emulating the WP, e.g., Schuette and Shen’s method for using instruction level parallelism for transparent integrated control-flow monitoring [148] and (iii) integrating watchdog processors into multiprocessor systems, e.g., Shen’s Roving Monitoring Processor [151] or the Checker by Madeira [86].

Having introduced the key concepts related to watchdog processors, we can reason about choosing the most appropriate solutions for representation of runtime information, reference information and the implementation of a watchdog for detecting errors caused by faults of the implementation of a statechart. To put together: we would like to construct a watchdog that observes the execution of an application and checks whether the actual behavior honors the statechart specification, e.g., after the reception of a trigger a valid set of transition conglomerates is selected to be fired, state exit and entry activities, transition effects are performed in a valid order and the resulting configuration is calculated correctly. It is easy to see that due to the complexity of statechart semantics the representation of runtime and reference information will be much more complex than in case of any previous approaches. Below we discuss the relation of control flow graphs to the statechart semantics, how to represent runtime and reference information and how to implement the watchdog.

Control Flow Graph The concepts of the control flow graph seamlessly map to the Kripke transition system specified in Chp. 1: nodes of the CFG are states of the KTS (corresponding to the composite representation of the configuration, evaluation of external variables and phase of operation as defined in Def. 10) and edges of the CFG are labeled transitions of the KTS (e.g., $\text{init} = \Pixel{}$, $\text{open} = \Pixel{}$, etc. as defined in Eq. 1.119, Eq. 1.120, etc. respectively).

Runtime Information Using derived signatures as runtime information does not seem to be viable since the fully non-intrusive external observation of such a complex behavior is nearly impossible, therefore we will use assigned signatures that carry the relevant information of $\text{init}$, $\text{open}$, etc. tuples, i.e., the source and target configuration, the trigger processed, the set of transition conglomerates fired and the PERT graph of activities performed.

It is important to highlight that the tuples transmitted to the watchdog are not necessary mem-
bers of the $\Rightarrow$ set; in contrary: we would like to detect those cases where the tuple $\Rightarrow_x$ describing the actual step violates a semantic rule i.e., $\Rightarrow_x \notin \Rightarrow$. As an illustration: in traditional watchdog schemes the application sends a signature $s$ to the watchdog to check whether the state represented by $s$ is a valid successor of the previous state; in our approach the application sends a tuple $\Rightarrow_x$ to the watchdog to check whether $\Rightarrow_x \in \Rightarrow$.

The signature sent to the watchdog may not carry the entire information content of a KTS step since strictly speaking the $\Rightarrow_x = (s_{\text{src}}, (t, TC_f, p), s_{\text{trg}})$ tuple contains the entire labeling of source and target state ($s_{\text{src}}$ and $s_{\text{trg}}$ respectively) including the $L_{\text{Conf}}$ configuration, the $L_{\text{Eval}}$ evaluation of extended variables and the phase of operation; obviously the evaluation of extended variables probably can not be transmitted to the watchdog since in extreme cases this would require the transmission of the entire memory image of the observed application. We will present a largely flexible scheme where the information sent to the watchdog is configurable and the watchdog performs as in-depth checks as possible with respect to the available information, e.g., without the evaluation of extended variables the guard conditions of all transition conglomerates are considered to be true. The lack of full observability appears also in case of traditional watchdog approaches: as the WP observes only the control flow can not check whether conditional branches are taken correctly, i.e., if there are multiple edges $e_1, e_2, \ldots$ originating in a state $s_{\text{src}}$ of the CFG, all target states of $e_i$ edges are accepted as subsequent states without checking the correctness of the data dependent branch.

**Reference Information** Storing the reference information in a database like an adjacency matrix or adjacency list is surely not viable since this would require the flattening of the statechart and pre-calculating all possible steps; due to the possible orthogonal state decomposition the size of the flattened statechart would be an exponential function of the original size itself but taking into consideration all the information stored in states of the KTS (evaluation of variables, phase of operation) and the information to be pre-calculated for all possible edges (all possible valid set of transition conglomerates fired, all valid PERT graphs not violating the specification, etc.) would definitely result in reference models of astronomical size.

The application of embedded signatures is not an option again since embedding a “fragment of possible future behavior” seems to be semantically even more complicated than the definition of the entire statechart semantics itself. Thus we will present an approach that is closest to watchdog program approaches: we will synthesize an automaton from the definition of the KTS that accepts correct runs of the application with respect to the statechart. The watchdog automaton will run in synchrony with the observed application, and check the validity of steps.

**Implementation** Since our watchdog has to perform quite complex checks, we will implement is purely in software. The software implementation can run as a stand-alone application connected to the observed application by some IPC mechanisms or embedded in the observed application in a self-checking manner as mentioned above in the context of the traditional watchdog approach [148]. Due to space restrictions we will not discuss any aspects of implementing the IPC mechanism between the main application and the watchdog, but we presented and evaluated a high-performance IPC mechanism explicitly designed for sending signatures to a watchdogs in [116, 117]. For simplicity reasons we will focus on checking a single object specified by the statechart but our approach can be extended to checking any number of objects specified by any number of statecharts as discussed in our papers [46, 128].

To put together: we are building a watchdog processor that uses assigned signatures as runtime information, a watchdog program-like solution for reference information and is implemented in software.

According to our knowledge there were no elaborated solutions proposed in the literature for runtime error detection of statechart implementations. Majzik, Jávoroszky, Pataricza and Selényi suggested a combination of their SEIS approach with a statechart-based high level specification in [93]. In their
approach statecharts are flattened and the reference information is stored in an adjacency matrix; state exit and entry activities are considered to be implemented by functions and the executions of these functions is checked by the SEIS watchdog scheme. The paper uses statecharts for checking the correctness of traversal in the state space i.e., ensuring that functions (that are checked by the SEIS approach) are called in a valid order. Unfortunately due to the lack of a formal semantics for statecharts their approach can not exploit the most of the description power of statecharts, the proposed solution is primarily an extension of the SEIS approach than a watchdog for statechart.

3.7 A Watchdog Architecture for UML 2.0 Statecharts

In our PSC-WD approach we will first construct an automaton \( W \) that accepts the correct runs of a statechart implementation. We will assume that the \( \Rightarrow_i \in \Rightarrow \) steps of the implementation can be observed (through a monitor interface and by instrumentation). The task of building this automaton can be characterized as a computer linguistics problem: we would like to build an automaton that accepts valid sentences of a language: (i) the grammar of the language is specified by the actual statechart and the underlying Kripke structure with respect to the semantics discussed in Chp. II (ii) words of the language are the \( \Rightarrow_i \in \Rightarrow \) steps of the implementation, and (iii) sentences are the \( S = (\Rightarrow_0, \Rightarrow_1, \ldots) \) step sequences. For a step sequence \( S \) we have to decide whether \( S \) is valid according to the statechart and the operation semantics i.e., \( S \) is a valid sentence of the language or not.

Let us imagine the implementation as an automaton shown in part a of Fig. 1.21 in this aspect we can consider the implementation as an automaton that writes the \( \Rightarrow_i \in \Rightarrow \) words to its output as indicated by the labels on arrows (e.g., writing \( \Rightarrow_i \in \text{init} \Rightarrow \) in case of “Initialization” etc.).

Initially let us consider that \( W \) can only see the type of the step taken by the implementation (i.e., whether \( \Rightarrow_i \) should be in \( \text{init} \Rightarrow \) or \( \text{open} \Rightarrow \), etc.), but does not investigate the internal semantics of \( \Rightarrow_i = (s, (t, TC_i, p), s') \) tuples. Let us assign observation states in \( W \) for phases of operations in the implementation (e.g., \( \text{Obs}_a \) for \( \text{Phs}_a \), \( \text{Obs}_U \) for \( \text{Phs}_U \) etc.). Constructing an automaton that observes the actual phase of operation in the implementation is easy since a source phase – step type pair unambiguously determines the target phase (the observer automaton should be obviously started from the \( \text{Obs}_a \) state corresponding to the initial phase \( \text{Phs}_a \)).

Until this point our watchdog automaton \( W \) can only indicate those abnormal situations when the implementation performs a step whose type is invalid in the actual phase (e.g., opening an RTC step in the uninitialized phase). The next step is adding more intelligence to \( W \) by enabling it to check the internal semantics of \( (s, (t, TC_i, p), s') \in \Rightarrow \) tuples. The automaton extended this way is shown as a UML statechart in Fig. 3.11 \( W \) has two top states: Observing (the watchdog is currently observing behavior of the implementation and has not detected any inconsistencies) and Error (an error was detected). The observer automaton is implemented as substrates and transitions in the Observing state. The predicates that check the internal consistency of \( (s, (t, TC_i, p), s') \in \Rightarrow \) tuples are implemented as guard predicates. Note that according to priority relations, transitions between Observing and Error are only fireable if (i) a step of invalid type was performed (e.g., \( \text{open} \Rightarrow \) in \( \text{Phs}_a \) or (ii) a guard predicate evaluates to false indicating a step consistency error.

From this point our remaining work is only to specify the guard predicates that check whether a \( \Rightarrow_i = (s, (t, TC_i, p), s') \) tuple is a valid instance of the corresponding step type according to the
semantics. The six guard predicates as derived from the semantics (Eq. [3.89]– Eq. [3.94]) are presented in Eq. [3.89]– Eq. [3.94]. (In the definition of InitOK we used the symbols $S_{\text{initial}}$ and $p_{\text{initial}}$ where $(p_{\text{initial}}, S_{\text{initial}}) = \text{Init}()$ and in the definition of OpenOK and InterOK we used the symbols $TC_{pf}$, $S^r_{pf}$, $p^r_{pf}$ and $S^s_{pf}$; $TC_{pf} = \text{PossiblyFireable}(L^s_{\text{Eval}}(s), t)$ and $(S^r_{pf}, p^r_{pf}, S^s_{pf}) = \text{Fire}(L^s_{\text{Eval}}(s), TC_t)$).

The predicates systematically check the consistency of transition labels ($t$ and $TC_t$), source and target configurations and (where applicable) check that the set of transition conglomerates fired ($TC_t$) is valid i.e., priority relations are not violated and $TC_t$ is free from conflicts.

\[
\text{InitOK}((s, (t, TC_t, p), s')) = (t = t_0) \land (TC_t = \emptyset) \land (L^s_{\text{Eval}}(s) = \emptyset) \land (p.\text{nodes} = p^0_{\text{initial}}, \text{nodes}) \land (p \leq p^0_{\text{initial}}) \land (L^s_{\text{Eval}}(s') = S_{\text{initial}}) \land (L^s_{\text{Eval}}(s') = \text{Phs}_s) \quad (3.89)
\]

\[
\text{OpenOK}((s, (t, TC_t, p), s')) = (t \in \text{Trigger}) \land (TC_t \neq \emptyset) \land (L^s_{\text{Eval}}(s) \neq \emptyset) \land (L^s_{\text{Eval}}(s) \cap \text{TerminationState} = \emptyset) \land (TC_t \subseteq TC_{pf}) \land (\forall t'c \in TC_t : \beta_{t'c} \subseteq TC_t : tc < tc') \land (\beta_{tc_1, tc_2} \subseteq TC_t : tc_1 \neq tc_2) \land (p.\text{nodes} = p^r_{\text{nodes}}) \land (p \leq p^r_{\text{nodes}}) \land (L^s_{\text{Eval}}(s') = \text{Phs}_s) \land (L^s_{\text{Eval}}(s') \subseteq S^r_{\text{Eval}} \cup S^s_{\text{Eval}}) \land (L^s_{\text{Eval}}(s') = \text{Phs}_s) \quad (3.90)
\]

\[
\text{InterOK}((s, (t, TC_t, p), s')) = (t = t_0) \land (TC_t \neq \emptyset) \land (L^s_{\text{Eval}}(s) \neq \emptyset) \land (L^s_{\text{Eval}}(s) \cap \text{TerminationState} = \emptyset) \land (TC_t \subseteq TC_{pf}) \land (\forall t'c \in TC_t : \beta_{t'c} \subseteq TC_t : tc < tc') \land (\beta_{tc_1, tc_2} \subseteq TC_t : tc_1 \neq tc_2) \land (p.\text{nodes} = p^r_{\text{nodes}}) \land (p \leq p^r_{\text{nodes}}) \land (L^s_{\text{Eval}}(s') = \text{Phs}_s) \land (L^s_{\text{Eval}}(s') \subseteq S^r_{\text{Eval}} \cup S^s_{\text{Eval}}) \land (L^s_{\text{Eval}}(s') = \text{Phs}_s) \quad (3.91)
\]

\[
\text{CloseOK}((s, (t, TC_t, p), s')) = (t = t_0) \land (TC_t = \emptyset) \land (p = p_0) \land (L^s_{\text{Eval}}(s) \neq \emptyset) \land (L^s_{\text{Eval}}(s) \cap \text{TerminationState} = \emptyset) \land (L^s_{\text{Eval}}(s') = \text{Phs}_s) \land (L^s_{\text{Eval}}(s') \subseteq S^r_{\text{Eval}} \cup S^s_{\text{Eval}}) \land (L^s_{\text{Eval}}(s') = \text{Phs}_s) \quad (3.92)
\]

\[
\text{TermOK}((s, (t, TC_t, p), s')) = (t = t_0) \land (TC_t = \emptyset) \land (p = p_0) \land (L^s_{\text{Eval}}(s) \neq \emptyset) \land (L^s_{\text{Eval}}(s) \cap \text{TerminationState} = \emptyset) \land (L^s_{\text{Eval}}(s') = \text{Phs}_s) \land (L^s_{\text{Eval}}(s') \subseteq S^r_{\text{Eval}} \cup S^s_{\text{Eval}}) \land (L^s_{\text{Eval}}(s') = \text{Phs}_s) \quad (3.93)
\]

\[
\text{DropOK}((s, (t, TC_t, p), s')) = (t \neq t_0) \land (TC_t = \emptyset) \land (p = p_0) \land (L^s_{\text{Eval}}(s) \neq \emptyset) \land (L^s_{\text{Eval}}(s) \cap \text{TerminationState} = \emptyset) \land (L^s_{\text{Eval}}(s') = \text{Phs}_s) \land (L^s_{\text{Eval}}(s') \subseteq S^r_{\text{Eval}} \cup S^s_{\text{Eval}}) \land (L^s_{\text{Eval}}(s') = \text{Phs}_s) \quad (3.94)
\]

\[
\text{PossiblyFireable}(C, t) = \{tc \in TC \mid (\text{TCEnabling}(tc) \subseteq C) \land (\text{TCTrigger}(tc) = t)\} \quad (3.95)
\]

When defining the predicates, we decided not to expect the implementation to be able to transmit the evaluation of variables ($L^s_{\text{Eval}}(s)$) as discussed above thus we can not calculate the exact set of transition conglomerates to be fired, only a superset of it i.e., the ones that are triggered and whose enabling states are active (this way all guards are considered to be enabled) – this set is calculated by the “possibly fireable” function (Eq. [3.95]).

**Initialization** The predicate InitOK (Eq. [3.89]) represents the semantic check of the *initialization*. The meaning of sub-expressions in the definition are as follows: (i) the step did not consume any trigger, (ii) there were no transition conglomerates fired, (iii) the source configuration was the “uninitialized configuration”, (iv) the activities performed during the initialization step were the ones to be actually performed (i.e., the nodes in the PERT graph of the activities performed in the step are the same as in the reference PERT graph), (v) no subsequence relations amongst activities were violated, (vi) the resulting configuration is the initial configuration and (vii) the resulting phase is unstable.

**Opening a run-to-completion step** The predicate OpenOK (Eq. [3.90]) represents the semantic check of *opening a run-to-completion step*. The meaning of sub-expressions in the definition are as follows: (i) the step consumed a trigger (i.e., not the empty trigger), (ii) there were one or more transition conglomerates fired, (iii) the source configuration was not empty, (iv) there were no termination states in the source configuration, (v) the transition conglomerates fired in the step were in the set of possibly fireable ones, (vi) none of the transition conglomerates fired in the step were disabled by priority relations, (vii) there were no conflicting transition conglomerate pairs fired, (viii) the activities performed during the initialization step were the
3.8. **EXPERIMENTAL EVALUATION**

ones to be actually performed, (ix) no subsequence relations amongst activities were violated, (x) the resulting configuration is derived from the source configuration by removing $S^e_{ref}$ and adding $S^e_{ref}$ and (xi) the resulting phase is unstable.

**Internal step** The predicate **InterOK** (Eq. 3.91) represents the semantic check of an *internal step*. The meaning of sub-expressions in the definition are as follows: (i) the step did not consume any trigger, (ii) there were one or more transition conglomerates fired, (iii) the source configuration was not empty, (iv) there were no termination states in the source configuration, (v) the transition conglomerates fired in the step were in the set of possibly fireable ones, (vi) none of the transition conglomerates fired in the step were disabled by priority relations, (vii) there were no conflicting transition conglomerate pairs fired, (viii) the activities performed during the initialization step were the ones to be actually performed, (ix) no subsequence relations amongst activities were violated, (x) the resulting configuration is derived from the source configuration by removing $S^e_{ref}$ and adding $S^e_{ref}$ and (xi) the resulting phase is unstable.

**Closing a run-to-completion step** The predicate **CloseOK** (Eq. 3.92) represents the semantic check of closing a *run-to-completion step*. The meaning of sub-expressions in the definition are as follows: (i) the step did not consume any trigger, (ii) there were no transition conglomerates fired, (iii) there were no activities performed, (iv) the source configuration was not empty, (v) there were no termination states in the source configuration, (vi) the configuration was not modified and (vii) the resulting phase is stable.

**Termination** The predicate **TermOK** (Eq. 3.93) represents the semantic check of *termination*. The meaning of sub-expressions in the definition are as follows: (i) the step did not consume any trigger, (ii) there were no transition conglomerates fired, (iii) there were no activities performed, (iv) the source configuration was not empty, (v) the configuration was not modified and (vii) the resulting phase is terminated.

**Dropping a trigger** The predicate **DropOK** (Eq. 3.94) represents the semantic check of *dropping a trigger*. The meaning of sub-expressions in the definition are as follows: (i) the step consumed a trigger (i.e., not the empty trigger), (ii) there were no transition conglomerates fired, (iii) there were no activities performed, (iv) the source configuration was not empty, (v) there were no termination states in the source configuration, (vi) the configuration was not modified and (vii) the resulting phase remained stable.

The statechart of the watchdog coupled with formal definitions of guard predicates above fully define the operation of our watchdog, PSC-WD. As mentioned above, the checks performed are configurable with respect to the available information since if an attribute of $\Rightarrow_{i} = (s, (t, TC_{i}, p), s')$ tuples is not available, the affected sub-expressions in the guards are to be substituted by the Boolean $\top$ constant, e.g., if the PERT graph of activities is not sent to the watchdog, sub-expressions referring to $p$ are considered to be true.

We implemented a prototype of the PSC-WD discussed here as a stand-alone application that reads the statechart model of the observed application (from an XMI file), processes a trace of execution and signals an error on the violation of the behavior specification.

### 3.8 Experimental Evaluation

In order to illustrate the error detection capabilities of our two methods we carried out a fault injection experiment involving both model refinement, implementation and physical faults. This section defines the basic concepts of fault injection, outlines the measurement setup, the process of collecting experiment data and presents the results of experiments.
3.8.1 Background on Fault Injection

The effectiveness of error detection techniques and fault tolerance mechanisms can be evaluated by investigating the actual behavior of the computer system in the presence of faults. As faults become active relatively rarely, waiting for faults to become active in a fully passive and uncontrolled way would be unpractical and very time-consuming during the development and evaluation of software. In order to increase the frequency of fault activation hence reducing the time needed for evaluating the software, various fault injection techniques were developed.

By definition, fault injection corresponds to the artificial insertion of faults into a real target system. Fault injection experiments can be aimed at achieving three benefits: (i) understanding of the effects of real faults and thus of the related behavior of the target system, (ii) possible enhancement and correction of the fault tolerance mechanisms and (iii) assessing the efficiency (coverage) provided by the fault tolerance mechanisms.

Fault injection experiments are characterized by the input and the output domains. The input domain corresponds to (i) the $F$ set of injected faults, called the faultload and (ii) the $W$ set of activities the system has to perform during the experiment, called the workload. The output domain corresponds to the $R$ set of readouts that are collected to characterize the target system behavior in the presence of faults. The outcome of fault injection experiments are called the measures that are derived from the analysis and processing of the $F$, $W$ and $R$ sets.

Most of fault injection solutions proposed in the literature aim at modeling physical faults (with respect to the phenomenological cause of the fault). Moreover, supported by the fault statistics available, transient faults have dominated this effort over permanent faults. Dedicated tools were developed to flip bits at the pins of integrated circuits, (e.g., MESSALINE by Arlat et al. [6], RIFLE by Madeira et al. [89], AFIT by Martinez et al. [98]), alter the power supply or even bomb the system or chips with electromagnetic interference or heavy ions [69]. The main drawbacks of applying physical fault injection are the high cost (caused by possible physical damage of the tested device), the intrusiveness of the approach and the questionable repeatability of experiments.

It has been pointed out that what really matters when evaluating the behavior of the system in the presence of faults is less the faults themselves than the consequences (errors) caused by them while the system is processing the workload. This observation was the main driver of the emergence of software implemented fault injection techniques (SWIFI). Typically a SWIFI tool flips bits in the memory or registers of the processor simulating this way the consequences of transient physical faults. There have been multiple SWIFI tools proposed in the literature, e.g., FERRARI [68], Mafalda [143], Xception [21], Balista [66], NFTAPE [155], GOOFI [2], etc.

Injecting software faults seems to be a more difficult task. Although significant work has been carried out dealing with software mutation [157, 162], software fault injection has gained intensive attention only in recent years. Durães et al. presented an analysis of common software faults by the investigation of bug databases of open source software [39] and defined a set of code mutation operators to be used in software fault injection experiments aiming at characterization of operating system behavior in the presence of faulty drivers [37]. SWIFI techniques were not only proven to be capable of generating errors similar to the ones resulting from physical fault injection but it has been pointed out that they are to some extent able to simulate software faults as well [57].

In the experimental evaluation of our error detection approaches we will investigate the behavior of an example application in the presence of model refinement, implementation and physical faults. As fault injection is not in the main focus of this thesis, the techniques used here represent our best efforts on the field and are admitted to require more in-depth investigation in our future research.

Goal Our experiments aim at assessing the error detection capabilities of the PSC-PLTL checker and PSC-WD with respect to model refinement, implementation and physical faults.

Observed application Choosing the observed application was definitely a non-trivial issue because of the multiple aspects to be taken into consideration. We wanted to use a model that is relatively
3.8. EXPERIMENTAL EVALUATION

easy to understand, but is complex enough to enable general conclusions. As we need to illustrate model refinement we had to choose an application whose model can naturally result from refinement of an initial statechart. The initial statechart should be usable for defining temporal requirements to be checked in the context of the elaborated statechart. On the other hand the application should perform some easy to capture computation and deliver some obvious results since we needed the notion of service for reasoning about failures and fault masking. The model we have chosen is a slightly enhanced version of the desktop calculator presented in [144].

Faultload Unfortunately according to our knowledge there have not been in-depth analyses published in the literature about typical model refinement faults concerning UML statecharts, neither were corresponding “modeling fault injector tools” proposed hence the faulty model refinement scenario presented below is primarily an illustration: we will present an initial statechart of the desktop calculator whose refinement is the actual elaborated statechart. Model refinement faults will be illustrated by removing a transition from the correct statechart.

With respect to implementation faults we decided to focus on the faulty implementation of two functions playing key role in statechart-based behavior: Enabled and FireSingle. According to our original plans we wanted to apply the methods and tools developed by Durães et al. but due to practical reasons (porting from Windows) and primarily lack of time we decided to postpone these experiments to subject of future work. The simulation of faulty implementation was carried out by manually preparing faulty implementations of the two functions above (at the level of the source code). The faulty version of Enabled was modified to return an invalid set of enabled transition conglomerates by first calculating the correct set than removing one from it and inserting one that was not found to be enabled originally. The faulty version of FireSingle was modified to return an invalid set of states left by adding a state to the correctly calculated set that should not be left. Both faulty implementations were prepared such way that the faulty behavior is exposed only once during the execution after a sufficiently long correct operation (e.g., the first ten executions of the functions were performed correctly, the eleventh was faulty and the rest of executions were correct again). This approach aims at simulating context sensitive implementation faults i.e., ones that are activated infrequently for a low number of inputs.

Physical faults in the underlying hardware were simulated by a SWIFI tool developed for our experiments: the tool was used to flip bits in the data segment of the application and in the code segment containing the implementations of functions Enabled and FireSingle. The fault injection was exhaustive with respect to the regions mentioned above i.e., all bits of the data segment and all bits of the code segment corresponding to the two functions were flipped once (obviously for a single experiment run only a single bit was flipped); this resulted in an experiment campaign of over 180000 experiment runs (see below).

Workload We used three computation scenarios (simulating buttons typed on the calculator) as workload in the experiments (see more detailed discussion later).

Readouts and measures The readouts obtained directly were as follows: outputs written to the standard output and error streams (containing the result of computation, error messages etc.), the execution trace of the application (sequence of assigned signatures) and errors detected by the operating system and the execution environment (signals). The following measures were derived by processing the readouts: occurrences of failures, assertion violations, errors detected by the PSC-PLTL checker and PSC-WD. (See more detailed discussion later.)

In order to prevent the correlation of effects caused by various faults exactly one fault was injected in each experiment runs, e.g., an application is generated from a faulty model or (exclusive or) implemented faulty or executed in the presence of a simulated physical fault.
3.8.2 Experiment Setup

Initial Model and Definition of Temporal Requirements

The initial statechart model of the desktop calculator and an illustration interface plan is shown in Fig. 3.12. The states of the statechart correspond to typing in the operands and operators. The triggers found in the statechart may be numbers 1–9 (trigger `num19Event`), the zero number (`num0Event`), the decimal point (`pointEvent`), the percent sign (`percentEvent`), operators “+”, “-”, “x” and “/” (`opEvent`), the “=” button (`equalsEvent`), the “clear entry” button (text: “CE”, trigger `ceEvent`), the “clear computation” button (text: “C”, trigger `deleteEvent`) and switching off the calculator is represented by the `cancelEvent` trigger. Since the number buttons 1–9 and operators are mapped to two trigger types only (`num19Event` and `opEvent` respectively) the actual number value or the actual operator is stored in the `keyId` attribute of the trigger instance.

The calculator is `ready` if it was just switched on or displays the result of the most recent computation. The calculator performs operations on two operands; the result of the most recent operation can be used again (as the first operand of the next operation). Since the calculator has no explicit negative prefix button the “-” button is used for indicating that the next operand is negative. The negative prefix of operands is represented by states `negated1` and `negated2`. Operands are real numbers; construction of operands is encapsulated in states `operand1` and `operand2` respectively; the initial statechart does not discuss the internals of states `operand1` and `operand2`. It is easy to see that the calculator enters the states `operand1` and `operand2` after pressing any of the 0–9 buttons or the decimal point in the `ready`, `opEntered` or one of `negated` states. The state `opEntered` represents the fact that the first operand and the operator to be used was typed (note that the trigger `opEvent` takes the calculator from `ready` or one of the `operand` states to `opEntered`). Pressing the “=” button after typing the second operand the calculator enters the `ready` state and displays the result. The percent button is used similarly to traditional basic calculators, e.g., “10 + 50% = 15”, this way pressing the percent sign after the second operand the calculator enters to the `ready` state again.

Accumulation of operations i.e., using the result of the previous operation as first operand is implemented by the transition originating in `operand2`, triggered by `opEvent` and targeting `opEntered`. Resetting the calculator is implemented by the loop transition attached to the topmost `calc` state triggered by `cEvent`. Switching off is represented by the transition originating in `calc`, triggered by `cancelEvent` and targeting a final state.

We defined four `temporal correctness criteria` in the context of the initial statechart: (i) “the
3.8. EXPERIMENTAL EVALUATION

The elaborated statechart of the calculator is shown in Fig. 3.13. The refined model presents the internal details of (i) state ready (by distinguishing between beginning of the computation and displaying the result) and (ii) states operand1 and operand2 (by embedding simple parsers for real numbers).

Two elaborated models were prepared: (i) the correct one presented in Fig. 3.13 and (ii) one with a model refinement fault: the faulty statechart is nearly the same as the one in Fig. 3.13 but the transition originating in operand2, triggered by opEvent and targeting opEntered is missing – it is easy to see that the lack of this transition prevents the calculator from being able to accumulate operations since after typing the second operand all other operands will be dropped.

The calculator should correctly handle operators” i.e., pushing an operator button in any of states operand1 or operand2 should take the calculator to opEntered (Eq. 3.96); (ii) “the user should be able to switch off the calculator in any (non-final) state” (Eq. 3.97); (iii) “if the calculator is switched on, it can be reset at at any time” (Eq. 3.98) and (iv) “the clear entry button can be used for deleting the (mistyped) operand or the negative prefix of operands” (i.e., taking the calculator from (a) the operand states to ready or opEntered and (b) from the negated states to ready or opEntered) (Eq. 3.99). The four requirements are formalized as φoper, φoff, φreset and φce below:

\[
\begin{align*}
\phi_{\text{oper}} &= \mathcal{G}\left(\left(\mathcal{P}_{\text{Calc}}(\text{operand}_1) \land \mathcal{P}_{\text{Calc}}(\text{operand}_2)\right) \land \mathcal{X} \mathcal{P}_{\text{Trigger}}(\text{opEvent}) \rightarrow \mathcal{X} \mathcal{P}_{\text{Calc}}(\text{opEntered})\right) \\
\phi_{\text{off}} &= \mathcal{G}\left(\left(\mathcal{P}_{\text{Calc}}(\text{calc}) \land \mathcal{X} \mathcal{P}_{\text{Trigger}}(\text{cancelEvent})\right) \rightarrow \mathcal{X} \mathcal{P}_{\text{Calc}}(\text{final})\right) \\
\phi_{\text{reset}} &= \mathcal{G}\left(\left(\mathcal{P}_{\text{Calc}}(\text{calc}) \land \mathcal{X} \mathcal{P}_{\text{Trigger}}(\text{cEvent})\right) \rightarrow \mathcal{X} \mathcal{P}_{\text{Calc}}(\text{calc}, \text{ready})\right) \\
\phi_{\text{ce}} &= \mathcal{G}\left(\left(\mathcal{P}_{\text{Calc}}(\text{operand}_1) \land \mathcal{X} \mathcal{P}_{\text{Trigger}}(\text{ceEvent})\right) \rightarrow \mathcal{X} \mathcal{P}_{\text{Calc}}(\text{ready})\right) \land \left(\left(\mathcal{P}_{\text{Calc}}(\text{negated}_1) \land \mathcal{X} \mathcal{P}_{\text{Trigger}}(\text{ceEvent})\right) \rightarrow \mathcal{X} \mathcal{P}_{\text{Calc}}(\text{ready})\right) \land \left(\left(\mathcal{P}_{\text{Calc}}(\text{negated}_2) \land \mathcal{X} \mathcal{P}_{\text{Trigger}}(\text{ceEvent})\right) \rightarrow \mathcal{X} \mathcal{P}_{\text{Calc}}(\text{opEntered})\right)
\end{align*}
\]
Source Code Generation, Building Binaries and Implementation Faults

We used our code generator for synthesizing the sources both in basic mode and with performance enhanced implementation of bit mask operations (as mentioned in Sec. 2.5). Applying the two patterns to the two input models (the correct one and the faulty one) resulted in four sets of sources.

As mentioned above the simulation of implementation faults were carried out by manually modifying the sources of functions `Enabled` and `FireSingle` in the sources generated from the correct model, finally resulting in eight source file sets: two correct implementations of the faulty model, four faulty implementations of the correct model (i.e., two patterns and two faulty functions) and two correct implementations of the correct model.

The eight source sets were built using four different optimization options of the C compiler\(^1\): `-O0` (no optimization), `-O1` (reduction of code size and execution time without performing expensive optimizations), `-O2` (performing nearly all supported optimizations that do not involve a space-speed tradeoff) and `-O3` (nearly all optimization options – as we needed to distinguish functions in the binary image the compiler was explicitly prevented from inlining functions). Applying the four compilation options to the eight source sets resulted in 32 binary executables (Tab. 3.7).

Execution and Physical Faults

The correct implementations of correct models (eight binaries) were executed in the presence of physical faults simulated by a SWIFI tool, the remaining 24 binaries (ones model refinement or implementation faults) were executed without physical fault injection. The fault injection was targeted to the data segment of the binaries and those regions of the text (code) segment that contained the binary image of functions `Enabled` and `FireSingle`. Obviously due to the various implementation patterns and compiler options, the size and exact position of these regions were specific to the individual binaries: the exact intervals in the binary images to be targeted by fault injection was identified using the usual UNIX debuggers and tools like `objdump` and `nm`. The number of physical faults to be injected was 46696 (Tab. 3.7). Both experiments were performed for four workloads `tiny`, `small`, `medium` and `complex`:

Tiny workload: contains a single trigger switching off the calculator.

Small workload: contains three accumulated additions: \(1 + 10 + 100 + 1000 = \) (the expected result of the computation is obviously 1111). The workload consists of 14 triggers.

Medium workload: contains an addition, percent calculation and a multiplication even involving negative numbers: \(-0.123 + 4.56\% \times -23.45 = \) (the expected result of the computation consists of two printed results (one after pressing the percent sign another one after pressing the equation sign): \(-0.128609\) and \(3.015881\) respectively). The number of triggers processed at this workload is 20. Note that however there are two results printed, the workload does not accumulate operations by performing the transition missing from the faulty model.

Complex workload: contains a multiplication and addition with a multiple clear entry button presses: \(+ - 0[CE] - .12[CE] - 0101.02 \times -0[CE] - .1[CE] - 2[CE]0.08 + 2.00 = \) where `[CE]` indicates pressing the clear entry button. The excepted result is \(-6.0816\). The number of triggers processed at this workload is 38.

Entire Database of Experiments

The preparation of models, sources, and binaries with indicating the physical fault injection applied and workload processed is highlighted in Tab. 3.7.

---

\(^1\)We used the GNU project C compiler, gcc, version 3.3.5 on a Debian GNU/Linux 3.1 “Sarge” system running the Linux kernel version 2.4.29.
3.8. EXPERIMENTAL EVALUATION

<table>
<thead>
<tr>
<th>Refinement faults</th>
<th>Implementation</th>
<th>Building</th>
<th>Physical</th>
<th># Experiments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>Normal</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Perf. enh.</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>Normal</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Perf. enh.</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Perf. enh.</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Perf. enh.</td>
<td>1</td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Faults</th>
<th>Pattern</th>
<th>#</th>
<th>%</th>
<th>Experiments (4 workloads)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Normal</td>
<td>1</td>
<td>-00 ... -03</td>
<td>4 x 1184 = 4736 text (5208 bits) 4 x 5208 = 20832</td>
</tr>
<tr>
<td></td>
<td>Perf. enh.</td>
<td>1</td>
<td>-01</td>
<td>4 x 1184 = 4736 text (3528 bits) 4 x 3528 = 14112</td>
</tr>
<tr>
<td></td>
<td>Perf. enh.</td>
<td>1</td>
<td>-02</td>
<td>4 x 1184 = 4736 text (3840 bits) 4 x 3840 = 15360</td>
</tr>
<tr>
<td></td>
<td>Perf. enh.</td>
<td>1</td>
<td>-03</td>
<td>4 x 1184 = 4736 text (3840 bits) 4 x 3840 = 15360</td>
</tr>
<tr>
<td></td>
<td>Perf. enh.</td>
<td>1</td>
<td>-05</td>
<td>4 x 3072 = 12288 text (3768 bits) 4 x 3768 = 15072</td>
</tr>
<tr>
<td></td>
<td>Perf. enh.</td>
<td>1</td>
<td>-01</td>
<td>4 x 3072 = 12288 text (2832 bits) 4 x 2832 = 11328</td>
</tr>
<tr>
<td></td>
<td>Perf. enh.</td>
<td>1</td>
<td>-02</td>
<td>4 x 3072 = 12288 text (3328 bits) 4 x 3328 = 13312</td>
</tr>
<tr>
<td></td>
<td>Perf. enh.</td>
<td>1</td>
<td>-03</td>
<td>4 x 3072 = 12288 text (3328 bits) 4 x 3328 = 13312</td>
</tr>
</tbody>
</table>

| Sum    | 8       | 32      | 186880              |

Table 3.7: Overview of Models, Sources and Experiments

For each 186880 experiment runs we recorded the following readouts: (i) everything written to the standard output (containing the result of computation), (ii) everything written to the standard error stream (possibly containing the error messages written in case of assertion failures), (iii) the entire execution trace to be processed by the watchdog and the temporal logic checker and (iv) errors detected by the operating system or the executor environment (result of the `execve` system call for executing the binary, occurrence of timeout and signals delivered to the process).

Based on the raw readouts the following higher-level measures were derived: (i) failures delivered by the system (by comparing the content of the standard out with the expected results of the computation), (ii) assertion failures (by searching for the corresponding messages in the standard error stream), (iii) the errors detected by checking the PSC-PLTL checker and (iv) the errors detected by PSC-WD (based on analyzing the execution trace by our solutions).

The results of all experiments and analyses were archived in files for making any phases repeatable. The entire experiment campaign took about two weeks and consumed over 10GB of disk space. The information stored in files was processed and stored in a relational database.

3.8.3 Experiment Results

Based on the experiment results shown in Tab. 3.8 the following observations were drawn:

**Refinement faults:** We were expecting the refinement faults to be hidden from both PSC-WD (since the implementation of the faulty model is correct) and low-level mechanisms like assertions or the OS error detection routines but we hoped that having defined temporal correctness criteria in the context of the initial model, the PSC-PLTL checker would detect them by evaluating the corresponding expressions on the execution trace. Furthermore, obviously we can detect errors caused by refinement faults only if the workload actually forces the fault to be activated as a behavior error and the temporal logic formula actually investigates that aspect of correct behavior that may be violated.

It is easy to see that the possibly missing transition (originating in `operand2`, triggered by `opEvent` and targeting `opEntered`) is to be taken in case of workloads that involve accumulated

---

2We used the IBM DB2 version 8.1 database manager for storing the experiment data.
CHAPTER 3. ERROR DETECTION IN UML 2.0 STATECHART IMPLEMENTATIONS

Table 3.8: Results of the Fault Injection Campaign

<table>
<thead>
<tr>
<th>Faultload</th>
<th>Workload</th>
<th>Observed phenomena [%]</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Failure</td>
<td>PSC-PLTL Checker</td>
<td>PSC-WD</td>
<td>Assertion</td>
<td>Signal</td>
</tr>
<tr>
<td></td>
<td>$\phi_{\text{off}}$</td>
<td>$\phi_{\text{oper}}$</td>
<td>$\phi_{\text{exact}}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Refinement</td>
<td>tiny</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>small</td>
<td>100.0</td>
<td>0.0</td>
<td>0.0</td>
<td>100.0</td>
</tr>
<tr>
<td></td>
<td>medium</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>complex</td>
<td>100.0</td>
<td>0.0</td>
<td>0.0</td>
<td>100.0</td>
</tr>
<tr>
<td>Impl. of</td>
<td>tiny</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Enabled</td>
<td>small</td>
<td>100.0</td>
<td>0.0</td>
<td>0.0</td>
<td>100.0</td>
</tr>
<tr>
<td></td>
<td>medium</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>complex</td>
<td>100.0</td>
<td>0.0</td>
<td>0.0</td>
<td>100.0</td>
</tr>
<tr>
<td>Impl. of</td>
<td>tiny</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>FireSingle</td>
<td>small</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>medium</td>
<td>100.0</td>
<td>0.0</td>
<td>0.0</td>
<td>100.0</td>
</tr>
<tr>
<td></td>
<td>complex</td>
<td>12.5</td>
<td>0.0</td>
<td>0.0</td>
<td>100.0</td>
</tr>
<tr>
<td>Physical in</td>
<td>tiny</td>
<td>0.09</td>
<td>1.83</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>data</td>
<td>small</td>
<td>10.76</td>
<td>0.0</td>
<td>3.62</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>medium</td>
<td>16.21</td>
<td>0.0</td>
<td>3.52</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>complex</td>
<td>16.87</td>
<td>3.71</td>
<td>3.67</td>
<td>0.0</td>
</tr>
<tr>
<td>Physical in</td>
<td>tiny</td>
<td>0.04</td>
<td>9.97</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Enabled</td>
<td>small</td>
<td>68.40</td>
<td>0.0</td>
<td>0.91</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>medium</td>
<td>80.12</td>
<td>0.0</td>
<td>0.84</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>complex</td>
<td>79.77</td>
<td>1.32</td>
<td>1.33</td>
<td>0.0</td>
</tr>
<tr>
<td>Physical in</td>
<td>tiny</td>
<td>0.53</td>
<td>3.78</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>FireSingle</td>
<td>small</td>
<td>84.47</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>medium</td>
<td>85.66</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>complex</td>
<td>85.77</td>
<td>0.16</td>
<td>0.0</td>
<td>0.07</td>
</tr>
</tbody>
</table>

computations i.e., workloads small and complex. Similarly with respect to the temporal correctness criteria we can see that only $\phi_{\text{oper}}$ investigates accumulated computations.

The corresponding cells of Tab. 3.8 (highlighted by blue fonts) clearly indicate that all errors caused by refinement faults (that were forced to be activated by the workload) were detected by the PSC-PLTL checker. We have to mention that despite of the 100% success of $\phi_{\text{oper}}$, none of the other temporal logic expressions found any errors, indicating the importance of rigorously defining temporal correctness criteria for most important aspects of the behavior.

**Implementation faults:** We were expecting implementation faults that do not cause serious behavior errors to be mainly hidden from temporal correctness checking and, since the faults are introduced into the source code (i.e., the compilation, building, memory organization, etc. are probably correct), we were also expecting low error detection ratio for built-in mechanisms of the HW/OS system; in contrary, we hoped that PSC-WD would detect most of the implementation faults. As discussed above the fault in the implementation is activated at the tenth execution of the corresponding function; hence in case of the tiny faultload that contains a single trigger no fault injection was performed.

The corresponding cells of Tab. 3.8 (highlighted by orange fonts) clearly indicate that a very high percentage of errors caused by implementation faults (that were forced to be activated by the workload) were detected by PSC-WD (87.5% of errors in case of the small workload and the faulty implementation of the FireSingle function and 100% of all other errors).

It is worth to note that some assertion failures also occurred, since the basic sanity checks built into the application detected some obviously inconsistent conditions, e.g., leaving inactive states, etc. As the assert ANSI-C macro on UNIX systems is implemented by an if statement that evaluates the asserted expression and sends an abortion signal (SIGABORT) to the process on the failure of the assertion: we also observed signals in case of assertion failures.
Physical faults: As our solutions were developed for detecting errors caused by refinement and implementation faults, we were expecting low error detection potential from both the PSC-PLTL checker and PSC-WD. Indeed: the corresponding cells of Tab. 3.8 indicate that the evaluation of temporal correctness criteria detected less than 4% of errors even in the best cases and the error detection ratio of the watchdog also fall into the $[1.83\% \ldots 11.8\%]$ interval.

It is remarkable however that (i) the watchdog seems to be somewhat more effective in case of faults in the data segment and (ii) HW/OS mechanisms detected high number of errors caused by faults in the code segment. The explanation of these phenomena is quite straightforward: (i) the data segment (that contains the static initialized data representation of the statechart) contains interpreted data and relatively few pointers (whose corruption may result in invalid memory operations) this way faults are more likely to result in logical errors detected by the watchdog; (ii) on the other hand bit inversions in the code section are likely to result in invalid operation codes or illegal memory addresses detected by the CPU or the MMU respectively.

The observations derived from the experimental evaluation of our error detection techniques have proven our preliminary expectations. The PSC-PLTL checker was found to be effective in detecting errors caused by model refinement faults if the faulty behavior actually violates a temporal correctness criterion and PSC-WD detected most of errors caused by faults in the implementation of behavior logic. The experiments also indicated that our solutions do not replace previously proposed error detection approaches: the detection of errors caused by physical faults are typically not detected by our methods, these issues are to be addressed by traditional WPs and low-level HW/OS mechanisms.

It is important to highlight however that most of errors caused by physical faults immediately resulted in error signals of some low-level HW/OS mechanisms interrupting the execution and preventing our higher-level solutions even from observation of faulty behavior thus we can not present an in-depth reasoning about the error detection potential of our two techniques in case of physical faults when running the experiments and the fault injection on an ordinary PC. Another important aspect to be stressed that the sophisticated low-level error detection solutions like memory page protection by the MMU, instruction sanity checking in the CPU/FPU, protection of privileged operations and all the OS-level mechanisms built on then are available on high-end platforms only: in resource constrained environments where the central processing unit is typically an FPGA or a microcontroller, most of these services are not available, the errors caused by physical faults are not caught immediately but they are likely to propagate to higher abstraction levels of behavior. In these cases user-level solutions like the ones proposed by us are expected to deliver higher error detection ratio (not being preceded by low-level mechanisms) and coupled with traditional WP solutions may be considered as the only possibilities for detecting errors caused by physical faults.

3.9 Conclusions and Future Work

This chapter has presented two techniques for runtime detection of behavioral errors in UML statechart implementations. The introduction established the foundations for reasoning about dependability impairments and fault tolerance, associated various dependability-related activities to the corresponding phases of software development and identified two aspects of runtime error detection where open issues were found. These unsolved problems were targeted by our related research: we addressed (i) model refinement faults by introducing a propositional linear temporal logic language for statecharts (PSC-PLTL) and (ii) faults in the implementation of statechart models by a statechart-level watchdog solution (PSC-WD). A background on temporal logic languages was presented in Sec. 3.2. PSC-PLTL was defined in Sec. 3.3. In Sec. 3.4 we presented an overview of previously proposed approaches for evaluation of PLTL formulae on finite execution traces and concluded that none of them is directly usable in resource constrained embedded environments, hence in Sec. 3.5 we specified a novel high performance method for evaluation of PLTL formulae; we illustrated the viability of our approach by
analytical and experimental evaluation and concluded that our solution was significantly faster than any previous approaches and consumed less memory. The second main part of the chapter was dedicated to detecting implementation faults: Sec. 3.6 presented an overview on watchdog processors and defined the concepts related to runtime and reference control flow representation; our statechart-level watchdog solution (PSC-WD) was formally presented in Sec. 3.7. The error detection capabilities of the two solutions were experimentally evaluated in Sec. 3.8; the fault injection campaign concluded that PSC-WD detected the majority of errors caused by implementation faults and the PSC-PLTL checker successfully addressed model refinement faults that were not detected by any other solutions.

Some lengthy discussions and examples were moved to appendix App. C: an example of automatic code generation for PLTL evaluation is presented in App. C.3 and the proof of correctness and algorithmic complexity of PLTL evaluation are discussed in App. C.1 and App. C.2 respectively.

The key contributions of this chapter are as follows: PSC-PLTL was presented as a temporal logic language explicitly aimed at defining temporal correctness criteria over statechart implementations; according to our knowledge the method for evaluation of PLTL formulae over finite execution traces is by far the fastest and most memory effective solution published in the literature and PSC-WD is the first and proven successful approach for detecting logical errors in the implementation of complex behavior specified by statecharts. Since our method for evaluation of PLTL formulae is not bound to the PSC-PLTL language, our achievements are usable in much broader range of applications. The achievements discussed in this chapter are summarized by the following thesis.

**Thesis 3 (Error Detection in UML 2.0 Statechart Implementations).** I elaborated novel methods for detecting errors in implementations of UML 2.0 statecharts. The thesis is built up of the following parts:

(i) I defined a **propositional linear temporal logic language** fitted to the statechart syntax and semantics introduced in Chp. I. The language (PSC-PLTL) enables the definition of temporal correctness criteria referring to various statecharts artifacts (states, transitions, etc.) and activities performed during execution. Due to the high abstraction level of the language essential temporal correctness criteria can be defined even in early phases of the development when only draft versions of statecharts are available.

(ii) I defined a **novel high performance method for evaluation of propositional linear temporal logic expressions over finite execution traces**. According to my analysis and experimental evaluation the solution delivers significantly higher performance than previous approaches while its low processing power requirements allow its usage in resource constrained embedded environments.

(iii) I elaborated a **watchdog solution for UML 2.0 statechart implementations** (PSC-WD) fitted to the semantics defined in Chp. I. The solution is capable of detecting errors related to violation of the specification with respect to maintaining state configuration, firing transitions and execution of compound activity structures.

(iv) I carried out a fault injection campaign for **experimental evaluation of error detection capabilities** of the solutions presented in this chapter. The experiments indicated that the PSC-PLTL checker was effective for detecting errors caused by model refinement faults, PSC-WD was successfully used for detecting errors caused by implementation faults while physical faults are to be addressed by traditional watchdogs and low-level HW/OS mechanisms.

It is important to highlight that although this chapter has introduced our approaches in the context of dependable software development according to the V-model, our key achievements can be beneficially applied in any development processes, e.g., in the context of the Rational Unified Process (RUP) [72, 73]; our error detection solutions can be used for testing in the construction phase, the implementation task can be supported by our code generation method etc.
3.9. CONCLUSIONS AND FUTURE WORK

Our error detection techniques were presented in multiple publications including conference papers, book chapters and detailed technical reports. The need for a statechart-level watchdog solution first occurred during my early research activities: not having a formal operational semantics for statecharts at that time (2001-2002) our first solutions were restricted to non-hierarchical statecharts (practically pure state diagrams); these preliminary efforts were documented in [116] and [118] (Scientific Student Conference), [117] (Master’s Thesis) and [137] (technical report). The formally well-established next-generation approach for the statechart watchdog was based on a semantics defined by extended hierarchical automata (EHA) the solution was first published in a conference paper [128] and a book chapter [46]. This approach was the direct ancestor of the final version presented in this document. The most recent solution (PSC-WD) was derived from the EHA-based version by fitting the corresponding concepts to the new semantics (presented in Chp. 1). The first version of our temporal logic language for statecharts was presented in [46]. Our solution for evaluation of PLTL formulae on finite execution traces was presented in [131]. In [129] (conference paper) we presented an analysis on the impact of statechart implementation techniques on the effectiveness of error detection techniques: we performed fault injection experiments on statechart implementations generated using two different patterns and evaluated the effectiveness of a statechart-level watchdog and low-level methods; this experiment can be considered as the antecedent of the experimental evaluation presented at the end of this chapter. The two error detection solutions in their present form were presented in the second part of [136] (research report of about 100 pages in Hungarian prepared in the framework of the “GVOP-3.1.1-2004-05-0523/3.0” project). Overviews on our error detection-related research were also presented in some short papers [120, 133]. In [130] (conference paper) we presented a lightweight solution for representing the signals emitted by various error detection solutions in the statechart of the application itself, enabling this way the modeler to define not only the normal behavior of the application by high abstraction level visual models but even the recovery activities (this paper was also selected for being published as a book chapter in [53]). Finally our two error detection techniques in their present up-to-date form built on the most recent version of the UML 2.0 statechart semantics were published as a book chapter [115].

Our method for evaluation of PLTL formulae over finite execution traces was implemented in a code generator that for a set of formulae synthesizes C++ or Java code implementing the evaluation node classes and corresponding algorithms. The code generator was implemented in 2500 lines of Prolog source and some lines of C++/Java source fragments. The main reasons for having chosen the Prolog language were its effective pattern matching and formula rewriting capabilities that were essential for the implementation of deriving formulae (connection normal form, top interfaces, etc.) and decomposition of formulae (left and bottom interfaces). The PSC-PLTL checker used in our experiments was built on a core library synthesized by this code generator coupled with some lightweight preprocessing routines. A prototype of PSC-WD was implemented in the Microsoft Abstract State Machine Language (AsmL). As reference information is directly read from XMI files using the library implementing the statechart semantics as discussed in Chp. 1 the prototype implementation is practically just the realization of the watchdog automaton and the guard predicates as discussed above resulting in a compact AsmL source of some hundred lines of code (21kB). (The previous, EHA-based implementation of the watchdog mentioned above consisted of 3600 lines of core application built on a library of 14000 lines written in C++.)

The targeted application areas of our approaches were discussed in the introduction of the chapter: our solutions are aimed at the runtime detection of errors in dependable applications caused by model refinement and implementation faults; furthermore in case of resource constrained systems where sophisticated low-level HW/OS error detection mechanisms are not available our solutions can be used even for detecting errors caused by physical faults. These error detection facilities can be the fundamental building blocks for the composition of fault tolerant solutions that are capable of initiating fault confinement and error removal activities on the detection of various errors in order to provide continuous service even in the presence of faults.
There are multiple points in the theories and prototype implementations where further research can be focused. The application possibilities of PSC-PLTL have not yet been fully exploited by its usage in runtime error detection: on the foundations provided by the statechart semantics and the PSC-PLTL language a novel model checking solution can be based that is capable of proving that various temporal correctness criteria defined in the context of initial statechart models holds for all intermediate steps and the final fully elaborated models throughout the entire development. Although the performance of our novel solution for the evaluation of PLTL formulae over finite execution traces has been proven to be superior of previously published approaches, neither the theory nor the implementation have been pushed to their limits, e.g., the performance could be increased and the memory requirements could be reduced by merging multiple subsequent evaluation nodes into larger blocks. Considering runtime error detection in broader aspect, a possible direction of future research can be the development of error detection approaches for activity and communication diagrams of UML; being activity diagrams quite close to statecharts the former goal could be achieved with a relatively modest effort while runtime error detection based on communication diagrams would require the elaboration of a corresponding formal semantics, similarly to the one of ITU’s Message Sequence Charts (MSC) [63] or their more advanced form, Live Sequence Charts (LSC) [28].
Chapter 4

Supporting Experiment Evaluation by Data Mining

4.1 Introduction

The previous chapters addressed various aspects of developing dependable software: Chp. 1 established the semantic foundations for reasoning about operation specified by UML statecharts, Chp. 2 presented a method for implementation of statecharts in order to reduce the possibility of coding faults and to increase productivity, finally Chp. 3 introduced two techniques for runtime error detection. Throughout the entire research we put explicit emphasis on proving the viability of our approaches by presenting prototype implementations and performing experimental evaluation where applicable. This final chapter focuses on processing data obtained from dependability evaluation experiments.

The reason for having introduced experimental dependability evaluation approaches (e.g., fault injection, robustness testing, etc.) is the recognition that model checking and simulation approaches are not capable of taking into consideration the entire complexity of large HW/SW systems. Multiple tools and environments were proposed for supporting experimental evaluation as discussed previously in Sec. 3.8. In spite of the big diversity of techniques available, all approaches share a common problem: they tend to produce a large amount of raw data that have to be processed to obtain the desired dependability measures or to get useful information on how the target systems behave in the presence of faults. Very often the analysis of the experimental data is quite complex, as it has to take into account many aspects of the experiment setup such as the target system architecture and configuration, the workload, the type of faults involved, the environmental aspects, etc. Surprisingly although researchers published various achievements on the development of fault injection and robustness testing tools and on the mitigation of problems such as experiment representativeness, intrusiveness and portability of solutions, the problem of coping with the large size of the experimental data sets and the high complexity of the analysis has received less attention: most of fault injection tools either provide rudimentary means to analyze data or, more frequently, just store the raw results in a spreadsheet format. Although this approach can be acceptable for very specific and simple analysis, it is clearly not enough when the analysis required is complex or when the amount of raw data is very large.

A recent newcomer to the bag of experimental dependability evaluation techniques is the dependability benchmarking family. This new approach actually represents an attempt to standardize experimental techniques with the goal of comparing dependability features of different systems or components. This research effort has already caught the attention of companies such as Sun Microsystems [168], IBM [80] and Intel [25], and lead to many dependability benchmark proposals, covering domains such as transactional systems [20, 100, 161], web servers [40], and operating systems [67]. Even topics such as human faults [19] or hardware maintenance [168] have already been subject of dependability benchmark proposals. As a representative of such efforts the DBench project (IST-2000-25425) of the European Commission defined a conceptual framework and an experimental environment for bench-
marking the dependability of COTS and COTS-based systems. Dependability benchmarks represent a new and important source of raw experimental data but the problem of analyzing that data has been even more neglected than in traditional fault injection and robustness testing. In fact, dependability benchmarks rely on a typically small set of measures and the data collected during the benchmark runs is just used to calculate the measures defined in the benchmark specification.

Furthermore, fault injection and dependability benchmarking rely on a priori assumptions about what are the measures we would like to improve (response time, throughput, availability etc.) and the benchmark performer should know what are the infrastructural attributes that determine these measures (e.g., CPU performance, disk bandwidth, operating system) being aimed at tuning the system to deliver the best performance and dependability. Although this approach has been beneficially applied for improving systems of relatively low complexity it does not scale well to complex systems actually used in real applications. In order to overcome this issue an automated mechanism is needed that supports the identification of key infrastructural factors by highlighting the potentially interesting phenomena in the large experiment database. On one hand this approach eliminates the need for a priori human knowledge; on the other hand it avoids some bias coming from some human belief.

This chapter proposes a novel approach for identifying the key infrastructural factors determining the behavior of systems in the presence of faults by the application of intelligent data processing methods that have already been successfully applied in the business field for extracting previously unknown information from large databases. The key idea of our approach is to perform benchmarking experiments on multiple configurations by applying different implementations of the same COTS component (e.g., different hardware setups, operating systems) and record as much information as possible about the infrastructure, delivered performance and dependability attributes. On the basis of this information data mining experiments are carried out to identify which infrastructural factors were really relevant enabling the developers to improve the system without a priori assumptions. We illustrate the solution by the analysis of the data set obtained from the DBench-OLTP dependability benchmarking experiments; this chapter also presents the analysis of the data set obtained from the fault injection campaign carried out for the experimental evaluation of our error detection techniques as discussed in Chp. 3.

The organization of this chapter is illustrated in Fig. 4.1 by zooming the corresponding box of Fig. 1. In Sec. 4.2 we introduce the basic concepts of data mining, outline the three families of techniques (predictive modeling, database segmentation and link analysis) and present an example for applying the classification method from the predictive modeling family to identify those attributes of records that can be used for predicting the value of another attributes.

Sec. 4.3 presents our proposal for automatic identification of those key infrastructural factors that determine the behavior of systems in the presence of faults. The approach is illustrated by the analysis of data obtained from the DBench-OLTP experiments as case study. Out method consists of three steps: (i) preparing input data, (ii) building logical decision trees by the data miner tool over the database of experiment results and (iii) explanation of phenomena observed by field expertise. The DBench-OLTP case study illustrates the approach step-by-step: first we present the experiment setup in Sec. 4.3.1, Sec. 4.3.2 discusses how to prepare data for investigation and how to exploit the benefits of data mining for identifying dominant factors that determine the behavior of the system under benchmarking in presence of faults; finally in Sec. 4.3.3 we present how to identify interesting phenomena by the classification method and how to explain these phenomena by field expertise.

The second part of the chapter (Sec. 4.4) presents a similar data mining analysis on the data set collected during the experimental evaluation of our error detection techniques in Chp. 3 in this second experiment we aim at identifying those factors (code generation method, compilation options, workload, faultload, etc.) that have an impact on the effectiveness of our error detection techniques this way we apply the achievements on the field of automatic data analysis to the outcomes of our primary research.
4.2 Background on Data Mining

Data mining is usually defined as an interdisciplinary field bringing together techniques from machine learning, pattern recognition, statistics, databases, and visualization to address the issue of extracting previously unknown, valid and actionable information from large databases to be used for making crucial business decisions [54]. Our approach aims at porting data mining from the business field to the dependable computing domain for exploiting its benefits for automatic identification of key factors that determine specific performance and dependability attributes of systems in presence of faults. Methods of data mining can be grouped in three families:

- **Predictive modeling** [104] resembles the human learning experience, where we learn how to classify real-world objects into abstract categories by identifying the essential underlying characteristics of phenomena amongst the possibly high number of less important attributes. For example, young children learn how to classify animals as cats and dogs by realizing that however animals are characterized by very large number of attributes (size, color, body structure, etc.) and many of them are not specific to any classes (e.g., there are cats and dogs of the same color, size) there are some key factors that can be used for assigning them to classes (e.g., the voice, body structure). The goal of predictive modeling is to build similar models by analyzing teaching data set and identifying the attributes and their relations that represent the key factors for classifying database records using statistical methods. A typical business situation for building predictive models is when a company is interested in understanding the key aspects of customer behavior (e.g., which customers are going to leave the company) by identifying the dominant attributes (age, purchased products etc.) [64].

- **Database segmentation** [13] aims at partitioning a database in segments of similar records i.e., ones that share a number of properties and so are considered to be homogeneous. A typical business application of database segmentation is the identification of typical customer groups (e.g., highly paid urban women, male university students) to be addressed appropriately [64].
CHAPTER 4. SUPPORTING EXPERIMENT EVALUATION BY DATA MINING

The goal of link analysis is to establish links (associations) between individual records or sets of records in the database. In business applications, link analysis is typically used for identifying products that tend to be sold together (market basket analysis) or for understanding long-term customer behavior for planning timely promotions etc.

Since our goal is to automatically identify key factors that determine specific attributes of systems in presence of faults, we selected the classification method, which is an implementation of the predictive modeling technique. Classification aims at establishing a specific class for each record in a database. The class must be one from a finite set of possible and predetermined class values. The input of the method is a teaching data set that presents the correct answer for some already solved cases; the output is a decision tree where leaves are the predicted classes and the internal nodes are atomic decisions (i.e., very simple predicates involving a single attribute, e.g., “the color of the person’s hair is black”, “the age of the person is below 14”). The key attributes identified by the algorithm are this way in the predicates of the atomic decision nodes.

A typical business application example is depicted in Fig. 4.2: an insurance company interested in understanding the increasing rates of customer attrition. A predictive model has determined that the two attributes of interest are: the length of time the client has been with the company (Tenure) and the number of services that the client uses (Services). The decision tree presents the analysis in an intuitive way. Having built the decision tree on the basis of previously experienced behavior, the company can use it to predict the future behavior of its current customers and try to convince the ones who are likely to leave to stay with the company with special advertisement campaigns etc.

It is important to highlight that although the usual business application of the classification method is to predict future behavior, our goal is somewhat different: we don’t want to predict anything (i.e., to use the algorithm built by the method) but we are interested in the classification algorithm itself. We will use the tree to recognize dominant factors (i.e., the attributes in the decision nodes).

Technically speaking, a decision tree corresponds to a graph of SQL SELECT commands on the teaching set. Each SELECT divides the actual data set in two subsets. Classes are assigned to the resulting subsets by identifying the most frequently occurring value of the predicted attribute (e.g., performing the SELECT corresponding to the “Services < 3” predicate on the data set selected by the “Tenure ≤ 2.5” predicate results in two subsets i.e., the one where the customers use less than three services and the one where use three or more; since most of the customers in the first subset left the company, the data miner assigned the subset to the LEAVE class etc. (Fig. 4.2). The quality of the classification can be measured by the homogeneity of the subsets i.e., how many records are in the subset that do not belong to the majority. The data miner tool uses statistical methods to find the SELECTs that result in as homogeneous subsets as possible with respect to the predicted attribute, identifying this way the attributes that play key role in determining the value of the predicted attribute.

The maximal depth of the tree can be restricted by the data analyst. Specifying high maximal depth enables the data miner to define more complex queries resulting in more homogeneous subsets (but representing smaller subsets of the teaching data set). This feature can be used as a zooming facility: when restricting the maximal depth to a low value only the most important factors are visible.
enabling the data analyst to draw general conclusions, while building a sophisticated tree allows an in-depth investigation into the relations of attributes. Since the data miner does not understand the semantics of the attributes care should be taken on the appropriate selection of the teaching set:

- When two attributes (columns) are semantically related that may represent redundant information that may become misleading for the analysis (for simplicity reasons we call this misleading information in the paper). This typically arises when two columns represent some kind of refinement relation e.g., the car manufacturer and the model (e.g., “VW” and “Golf”, “VW” and “Passat”, “Renault” and “Clio”). In this situation the model typically determines the manufacturer, since manufacturers do not copy model names (i.e., the model “Golf” determines that the manufacturer of the car is “VW”). From the database point of view selecting all VW products can be carried out by simply selecting the “VW” value of the manufacturer column or by selecting all VW models (Golf, Passat etc.) using the model column. Note that the selected subsets are totally the same in both cases, but the second query is more difficult to understand: if you do not recognize that all the models are from VW you will not realize that essentially we were focusing on the manufacturer. Since the data miner does not understand the semantics of columns it is subject to defining this kind of hard to understand queries i.e., defining hard to understand atomic decisions.

- When the teaching set is statistically inappropriate the decisions drawn by the data miner may be wrong (with the human learning analogy when a child sees only black cats and white dogs she/he may get to the wrong conclusion that the most important attribute for classifying animals as dogs or cats is the color). This missing information situation can arise when processing data obtained from a non-exhaustive experiment campaign.

4.3 Analysis of Data from DBench-OLTP Experiments

Having discussed basic concepts of data mining the rest of the chapter is dedicated to the explanation and illustration of our method for identifying the key infrastructural factors determining the behavior of complex HW/SW systems in the presence of faults. This section introduces the idea in the context of our data mining analyses performed on the data set obtained during the DBench project focusing on On-Line Transaction Processing (OLTP) systems.

4.3.1 Experiment Setup

DBench-OLTP \cite{96,161} uses the workload and the general approach of the industry standard TPC-C performance benchmark \cite{26} and adds two new components: (1) a faultload to emulate faults and upsets experienced by OLTP systems in the field (in the experiments used in our study the faultload

\footnote{See: \url{http://www.laas.fr/DBench} and \url{http://www.criticalsoftware.com/DBench}}
includes operator faults) and (2) a set of new measures meant to characterize the behavior of the system in the presence of the artificially introduced faults. These new measures include service (TPC-C transactions per minute) in the presence of faults, TPC-C cost related measures in the presence of faults and availability measures. The most important components of the DBench-OLTP benchmark are the experiment setup, the workload, the faultload, the measures obtained and the benchmark procedure.

**Experiment setup** The main elements of the experiment setup (Fig. 4.3) are the System Under Benchmark (SUB) and the Benchmark Management System (BMS) that emulates the client applications and records the data needed for calculating the benchmark measures. Several hardware-software configurations have been benchmarked with the DBench-OLTP: multiple hardware platforms, various operating systems (Windows 2000, Windows XP, SuSE Linux 7.3) and database management systems (DBMS) (Oracle 8i, 9i, and PostgreSQL).

**Workload** The workload represents the work that the system must perform during the benchmark run. In the DBench-OLTP approach the workload of the TPC-C performance benchmark was used. TPC-C represents a business where a wholesale supplier has a number of warehouses and their associated sale districts and where users submit transactions that include entering and delivering orders, recording payments, checking the status of orders etc.

**Faultload** The faultload represents the set of faults and stressful conditions that emulate real faults experienced by OLTP systems in the field. The DBench-OLTP approach focused on operator faults (i.e., mistakes of the DBMS administrator). The fault types injected during the experiments were as follows: abrupt operating system shutdown, abrupt DBMS shutdown, killing a set of user sessions, dropping a database table used by the workload, deleting the entire user schema, removing single data files from disk, removing sets of files from disk and removing all files from the disk. Since in several cases operator faults can only be detected by the system administrator (i.e., another operator) the fault detection time is not related to a system feature but an attribute of the faultload in this case. In DBench-OLTP experiments typical fault detection times were estimated taking into account the nature of the fault and field experience in OLTP system administration.

**Measures** The measures obtained characterize the performance and dependability of the SUB in presence of the faultload while executing the workload. The performance related measures include (i) the TPMC value (transactions executed per minute) calculated according to TPC-C (i.e., the total number of transactions per slot divided by the elapsed time) and (ii) the price per transaction (€/TPMC, a ratio between the price and the performance of the SUB calculated according to TPC-C pricing rules). The dependability related measures include (i) the TPMC and €/TPMC in the presence of the faultload (measure the impact of the faults on the service provided by the OLTP system, (ii) the number of data errors detected by consistency checking mechanisms measuring the impact of faults on the data integrity, (iii) the unavailability from the SUB point of view (the SUB is considered to be available from its own point of view if it is able to respond to at least one client within the maximal response time), and (iv) the aggregated unavailability from the clients’ point of view (the SUB is considered to be unavailable from the clients’ point of view if it is unable to respond within the maximal response time or returns an error). The data mining analysis presented in this paper focuses on two measures: the number of transactions executed during a slot (this raw performance attribute is the input for calculating the TPMC) and the aggregated unavailability from the clients’ point of view.

**Benchmark procedure** The benchmark procedure consists of fault injection slots (Fig. 4.4). At the beginning of a slot the state of the SUB is explicitly restored. Measurements are performed with the system in a steady state condition i.e., after a given time (steady state time) that is enough for the system to achieve its maximum throughput (e.g., filling data caches). Having achieved the steady state the fault is injected after a certain amount of time (injection time). As discussed above the fault detection time is artificially defined (taking into account the nature of the fault
4.3. ANALYSIS OF DATA FROM DBENCH-OLTP EXPERIMENTS

and field experience in OLTP system administration). After detecting the fault diagnostic and recovery procedures are initiated (the time needed for this is the recovery time). Having completed the recovery the workload is continued for a while (keep time) to measure the system’s speedup after recovery. At the end of the slot data integrity test are performed.

4.3.2 Applying Data Mining to Identify Dominant Factors

This subsection discusses how to apply data mining for identification of dominant factors that determine the behavior of the SUB in presence of faults. The idea of our approach is to process performance and dependability measures obtained during the fault injection experiments by building classification trees and identifying the key factors by investigating the tree built by the data miner. The steps of our approach are as follows: (1) preparing the input data set for making it appropriate to be analyzed by the data miner; (2) running the data miner for building decision trees (i.e., applying the classification method) and finally (3) analyzing the trees for collecting the dominant factors and explaining the phenomena.

Preparing Input Data

The goal of the data preparation is to transform the legacy data obtained during the experiments of the DBench project to a format that is appropriate for using as a teaching set in data mining. This process involves (i) the re-organization of individual tables where the experimental data is stored for eliminating column correlations that may represent misleading information, and joining tables into a single view (required by the data miner tool), (ii) eliminating the effects of missing information and (3) defining classes for the classification. Although the discussion below describes these steps as a human process for simplicity reasons, most of the steps may be facilitated by an automatic process.

Re-Organization of Tables and Defining a View

The goal of re-organizing the individual tables is to eliminate the semantic relation of attributes (columns) that may represent misleading information. In case of the DBench-OLTP data we identified some attribute pairs that represent some kind of refinement relation, e.g., the version of the operating system (e.g., “2000”, “XP”, “7.3”) can be considered as a refinement of the OS family (“Windows”, “SuSE”): e.g., since SuSE has never released “SuSE 2000” the version attribute determines the family. The misleading information was eliminated by introducing a new operating system attribute instead of the individual family and version attributes containing the valid combinations (i.e., “Windows 2000”, “Windows XP” and “SuSE 7.3”).

Technically speaking, when reorganizing the individual tables we have to answer the following question: “Are there two columns A and B that we can define a SELECT operation above A and one above B resulting in the same subsets?” If there are ones, these columns may represent misleading information and should be reorganized e.g., by merging them as presented above. Note that this step requires expertise neither in database management nor in data mining: the question can be answered by the designer of the experiments (i.e., the benchmark expert) on the basis of a priori knowledge about the semantics of the columns.

Since the data miner expects the teaching set as a single database table or view the tables of the legacy table structure of the DBench data was joined into a single view.
Dealing with Missing Information Since we are using a legacy data set that was not collected for data mining purposes we have to take care of the statistical relevance of the teaching set, i.e., we should avoid drawing wrong conclusions resulting from missing information. For example the original goal of DBench-OLTP was to compare DBMSs in a benchmarking style (e.g., compare the two Oracle versions 8i and 9i on a Windows platform, compare Oracle and PostgreSQL on the SuSE platform) this way experiments were not carried out on some operating system – DBMS-OLTP combinations (e.g., PostgreSQL was not benchmarked on Windows). This way, considering the entire data set as a whole it should not be used as a teaching set since the missing information may obscure the observations.

Obviously the best solution would be to provide information by carrying out the missing experiments, but unfortunately this approach is not viable in several cases e.g., since the PostgreSQL is available for the Linux platform only. In order to eliminate the effect of the missing information we focused on a subset of data that was exhaustive with respect to some restriction (e.g., by focusing on different DBMSs on the SuSE platform).

Technically speaking, when searching for missing information we have to answer the following question: “Are there experiments that were not carried out on specific platforms?” If there are ones, the missing experiments should be performed or data subsets have to be selected that are exhaustive with respect to some restrictions, e.g., focusing only on those platforms where all experiments were carried out. Note that this step requires expertise neither in database management nor in data mining: the question can be answered by the expert who performed the experiments on the basis of a priori knowledge about the experiments loaded into the database.

Defining Classes for Numeric Measures Since the measures obtained in the experiments were numeric variables (e.g., server unavailability experienced by the clients in seconds) and the classification method requires that the predicted attribute is from a finite set of possible predetermined class values we had to define classes and assign numeric values to the corresponding classes in the teaching set. This was carried out by dividing the intervals of the obtained measure in sub-intervals (i.e., corresponding to classes like “very low unavailability”, “low unavailability”). For example, in case of a data subset selected in the previous step the attribute “total number of transactions executed per slot in presence of faults” was between 0 and 70000, this way we introduced seven sub-intervals of the width of 10000 (e.g., [0-9999], [10000-19999],... ) for classification purposes. Since the chosen class granularity (i.e., the number of sub-intervals) was mainly intuitive we carried out a validation experiment to identify the impact of class granularity to the trees (see the discussion at the end of this section).

Mining Data

Having prepared the input data configuring the data miner is a few steps. First we have to select the data mining method (classification from the predictive modeling family); next we define the data source to be processed (i.e., selecting the view containing the prepared data) finally we configure the classification by (i) selecting the input columns (i.e., the ones that possibly contain interesting factors: all columns should be selected that describe some aspects of the measurement setup, hardware-software infrastructure, workload, faultload etc.) and (ii) the attribute to be predicted (i.e., the performance or dependability measure to be analyzed e.g., “server unavailability experienced by the clients in seconds”, “TPMC in presence of faults”). Analyzing our aggregated data sets containing some thousand records by the classification method took a few seconds on an ordinary desktop PC. The result of the operation is presented as an interactive decision tree: sub-trees can be closed and opened, leaf data distribution can be displayed etc. Although interactivity supports the analysis, screenshots are not suitable for being inserted into a paper (because they miss some details and occupy a lot of space), thus the figures presented here were redrawn and annotated in a drawing tool.
4.3.3 Discussion of Results and Analysis

This subsection presents how to identify interesting phenomena by the classification method and how to explain the phenomena by field expertise. The discussion is organized as follows:

- Data mining aims at automatically identifying phenomena. In this discussion we will (i) first textually formulate a question, (ii) configure the data miner accordingly (iii) explain the notation of the figures (decision trees) and (iv) textually formulate the atomic decisions chosen by the miner and identify the key factors determining the value of the specific predicted attribute.

- Having automatically identified the phenomena the explanation requires field expertise i.e., in our case knowledge about the internal operation of databases, fault injection etc. In this discussion we present the explanation of some phenomena identified by the data miner that probably would not have been identified when relying barely on human intuition.

Identifying Phenomena by Classification Tree Analysis

In cases of the two experiments presented here we were using a data subset that was exhaustive with respect to restricting the OS to Microsoft Windows 2000. Since PostgreSQL was not available for Windows we had data about two Oracle versions (8i and 9i) set up according to two inherently different configurations this way we could investigate the effects of the DBMS version (8i and 9i), the DBMS configuration and the faultload on the performance and dependability measures. (For more analyses see [122].)

In the first experiment presented here (Fig. 4.5) we aimed at identifying the key factors determining the total number of transactions per slot successfully performed during a fault injection slot. Informally, we asked the data miner to answer the following question: “What are the dominant factors that determine the performance of Oracle databases running on Windows 2000 as expressed by the total number of successful transactions per slot?” Obviously questions like that naturally emerge when planning databases that should deliver high performance even in presence of faults. Answering this question will suggest which DBMS version is to be used, how to configure it and which faults are the most serious ones in performance aspects.

The data miner was configured according to the discussion above: the input columns were the version and configuration of the DBMS and the attributes of the faultload (type, injection and detection time, etc.). The predicted attribute was the class of the “total number of successful transactions per slot” measure. Since the total number of transactions per slot fall in the [0-70000] interval in this subset of the teaching set we defined seven classes ([0-9999], [10000-19999]... etc.).

The notation of Fig. 4.5 is quite straightforward: the predicted classes are visualized by highlighting the appropriate sub-interval in the small graphs in the leaves (where the class homogeneity was considered to be low multiple sub-intervals were highlighted). The numbers under the small graphs in the leaves indicate the size of the corresponding data subsets (e.g., “When using configuration Conf-A and the fault injection time is above 870 (stepping always to the No edge from the root) the total number of transactions per slot was between 40000 and 70000 (very high) and this selection corresponds to 54 records in the teaching set.”).

The atomic decisions are textually formalized in the callout texts in Fig. 4.5. The key factors that determine the total number of successful transactions per slot in presence of faults can be easily identified since the most important one is in the root of the tree; the less important ones are ever closer to the leaves. This way the data miner determined that the most important factor determining the total number of successful transactions per slot in presence of faults in case of the Oracle databases running on the Windows 2000 platform was the configuration (Conf-A was found to deliver higher performance); another attribute of high impact is the time of fault injection (the later we inject the fault, the higher the total number of transactions per slot), and the actual fault load has somewhat smaller impact (dropping a table is a more serious fault in this aspect than removing files from the disk.)
or abruptly shutdown). Note that these observations were drawn only by investigating the tree without a priori knowledge about the system under benchmarking—we have only identified some phenomena until now, but we have not yet explained them. The explanation of the phenomena requires the knowledge of an OLTP expert.

In the second experiment presented here (Fig. 4.6) we aimed at identifying the key factors determining the server unavailability experienced by the clients (i.e., the sum of seconds while the individual clients found the server to be unavailable). Informally, we asked the data miner to answer the following question: “What are the dominant factors that determine the unavailability of Oracle databases running on Windows 2000 from the clients’ point of view?” Obviously questions like that naturally emerge when planning databases that should be highly available even in presence of faults. Answering this question will suggest which DBMS version is to be used, how to configure it and which faults are the most serious ones in availability aspects.

The data miner was configured similarly to the previous experiment but the predicted attribute was obviously the class of the “number of seconds clients experienced server unavailability” measure. Since the number of seconds fall in the [0-80000] interval in this subset of the teaching set we defined eight classes ([0-9999], [10000-19999], .. etc.).

The atomic decisions are textually formalized in the callout texts in Fig. 4.6. The data miner determined that the most important factor that determines the unavailability of Oracle databases running on Windows 2000 in presence of faults is the type of the faults injected. The tree highlights that there are less serious faults like abruptly shutting down the DBMS or the operating system or killing the user session, while inadvertently dropping tables results in high unavailability time. Configuration of the DBMS is also an important attribute: we can achieve higher availability by using Conf-A under the same circumstances. Note that although there were experiments where the newer Oracle DBMS (version 9i) was proven to deliver higher performance and expose better dependability characteristics, in case of the two experiments presented here there were no important differences between the two versions. This observation can also be important when considering software upgrade for improving system performance or availability.

Explaining the Phenomena by Field Expertise

Although data mining provides a powerful support by automatically identifying key factors determining the values of performance and dependability attributes, the phenomena should be explained by a field expert for turning the observations into financial, performance or dependability benefits. Below we
provide explanations for the impact of configuration, faultload and the fault injection time.

The impact of the configuration can be explained by the fact that Conf-A is an effective well-tuned configuration while Conf-B restricts the size of the redo log files to 1MB (very low size preventing the DBMS from exploiting the performance benefits of postponed write operations and buffering) and sets the checkpoint interval to 4 seconds (very low time forcing the DBMS to synchronize the memory image and the data files uselessly often). The performance benefits (higher number of successful transactions per slot in case of Conf-A) are this way obvious. Concerning availability, although Conf-B should favor fast recovery (because the high checkpointing rate) this is true for some types of faults only. Globally, the best performance of the well tuned Conf-A also favors availability.

Both experiments highlighted the important impact of the faultload: dropping tables proved to be more serious than other faults (removing files, abrupt shutdown etc.) in both performance and availability aspects (i.e., results in serious degradation of the total number of successful transactions per slot and availability). These phenomena can be explained by the recovery mechanism of the DBMS: removed data files can be restored by copying back the latest backup file and re-executing the transactions involving the data file that were performed until the creation of the backup. Since during the reconstruction of files the DBMS can accept transactions that are not related to the file deleted (note that, in some cases a single table was partitioned by four files) the clients may not experience unavailability at all. Restoring inadvertently dropped tables is more complicated since dropping a table is a valid database operation, hence the only solution is to restore the latest backups of all the data files and re-execute all the transactions that were performed since the latest backup. This way restoring dropped tables is not only more time consuming but prevents the DBMS from accepting transactions during the recovery process resulting in serious degradation of the availability.

The impact of the fault injection time is probably one of the most interesting observations. The two phenomena identified by the miner may seem to be contradictory for the first sight: the later we inject the fault the higher the total number of successful transactions per slot but the lower the availability. In order to explain these phenomena we have to take into consideration the performance graph of the DBMS (Fig. 4.7). Before injecting the fault the server is operating at the baseline performance; obviously the later we inject the faults the greater the total number of transactions per slot performed during this period. After the fault injection the server may be unavailable during the fault detection and recovery time (depending on the fault type). Although the function is probably not linear it is quite obvious that the greater the total number of transactions per slot executed before the fault injection the longer the time needed for recovery this way we transitive explained the observation that the
later we inject the fault the lower the availability. Having finished the recovery the DBMS is fully functional again but the situation is similar to the one during the steady state time: the DBMS needs some time to achieve its maximum throughput again, this way the delivered performance is below the baseline performance for a while. If this time is high, the total number of successful transactions per slot is dominated by the ones performed before fault injection (the total number of transactions per slot is proportional to the size of the area under the performance graphs in Fig. 4.7). This explains the observation that the later we inject the fault the higher the number of successful transactions.

The phenomena identified by the miner and explained by field experts can be used for improving the performance and the dependability of the systems: e.g., by configuring the DBMS effectively, introducing fault tolerance measures against the most serious faults.

Evaluating the Impact of Class Granularity

Since the definition of classes (sub-intervals) enabling the classification tree analysis was mainly intuitive (i.e., how many sub-intervals to be introduced) we were interested in evaluating the impact of the class granularity (number of classes) on the resulting decision trees. We set up different experiments focusing on the same measure but defining classes of different granularity. For example the “TPMC in presence of faults” performability attribute fall in the [0-2000] interval in the case of selecting all database managers (two Oracle versions and PostgreSQL) running on the SuSE platform; this way we set up experiments with classes corresponding to sub intervals of width of 1000, 500, and 250 respectively (the trees corresponding to the first two granularity levels are presented in Fig. 4.8).

According to our observations the role of class granularity is similar to restricting the dept of the classification tree i.e., can be seen as a zooming facility. Low interval granularity (wide sub-intervals, low number of classes) results in small trees that enable general observations: in case of the “TPMC in presence of faults” attribute the tree built after dividing the entire interval into two sub-intervals highlights the DBMS in use to be the most important factor determining the performance of the system and indicates Oracle versions more performable (Fig. 4.8 left side).

Choosing ever higher granularity (narrower sub-intervals, higher number of classes) the tree will get ever larger enabling in-depth investigation of the phenomena (Fig. 4.8 right side): in case of the “TPMC in presence of faults” attribute the tree built after dividing the entire interval into four sub-intervals still highlights the DBMS in use to be the most important factor determining the performance but also indicates that Oracles are not only faster but are better in fault-tolerance aspects: while in case of PostgreSQL only the “killing the user session” fault can be considered as less serious (i.e., resulting in low degradation of the performance) in case of Oracles all the “killing the user session”, “abruptly shutting down the DBMS” and “removing single files” can be considered as less serious.

4.4 Analysis of Data from Evaluation of Error Detection Techniques

This section applies method discussed above to the data set obtained during the experimental evaluation of our runtime error detection techniques (discussed in Sec. 3.8) connecting this way our achievements on automatic experimental data analysis to the first three parts of the document. Below we will present three analyses in which we aim at identifying those factors that (i) determine whether the user
experiences a failure or not (Sec. 4.4.1), and affect the effectiveness of (ii) PSC-WD (Sec. 4.4.2) and (iii) low-level mechanisms (Sec. 4.4.3).

First we briefly recall the experiments where the data came from: the system under investigation was a desktop calculator application generated from a statechart model; we used four different workloads representing calculations of various complexity (called tiny, small, medium and complex); the faultload consisted of model refinement, implementation and physical faults and the measures obtained included whether the user experienced a failure and whether any of the PSC-PLTL checker, PSC-WD or some low-level mechanisms detected an error etc. The observed application was built with various code generation and compilation options: (i) we used both the basic and the performance enhanced implementation modes supported by our code generator (called pattern in figures below for simplicity) and (ii) the generated sources were compiled using four levels (0, 1, 2 and 3) of optimization supported by the C compiler. The database contained the details of fault injection as well: for implementation faults we specified the faulty function (i.e., Enabled or FireSingle) and for physical faults simulated by the SWIFI tool we specified the target segment (i.e., text or data) and the exact address of faulty bit (i.e., the byte and bit in the binary that was inverted); furthermore in case of physical fault injection in the text segment the function hit by the fault injection was also given. To put together (i) the factors that possibly affected the behavior of the system were as follows: the pattern used for generating the source code of the application, the compiler options used, the workload processed during the experiment, the type of fault injection, the segment, the function and the exact byte and bit hit by the fault injection (where applicable); (ii) similarly to the case of data from DBench-OLTP experiments we were interested in discovering how these factors affect the user experience (failures) and the effectiveness of error detection techniques (PSC-PLTL checker, PSC-WD and low-level mechanisms). In the analyses discussed below we will process the entire database of experiment results or restrict the investigations to some subset of data as indicated at the database icon below the figures.

4.4.1 Failures Experienced

Our first analysis campaign aims at identifying those factors that determine whether the user will experience a failure or not. In the first data mining setup (Fig. 4.9 a), the entire database was processed: the data miner was provided with the workload, the type of the faultload, the implementation pattern and the C compiler options used. As indicated in the callouts of the figure: (i) the data mining indicated that in case of the tiny workload the user typically did not experience any failures but (ii) since the homogeneity of another leaves was low (50.3% and 56.1%) we could not draw more observations and decided to divide the database into smaller parts for more in depth analysis: we established three subsets of measurement data by the type of the faults injected, i.e., groups refinement faults, implementation faults and physical faults.

The results of the analysis focusing on refinement faults is shown in part b of Fig. 4.9: the tree contains a single decision and 100% homogeneous leaves indicating that refinement faults in case of complex and small workload always result in failures.

Part c of Fig. 4.9 presents the tree built for implementation faults: we can see that (i) in case
CHAPTER 4. SUPPORTING EXPERIMENT EVALUATION BY DATA MINING

Figure 4.9: Failures Experienced (All Data, Refinement, and Implementation Faults)

Figure 4.10: Failures Experienced (Physical Faults)

of tiny and complex workload the user typically did not experience any failures, (ii) otherwise the faults injected into the enabled function always resulted in failure and (iii) faults of the fire_single function resulted only in failures only in case of the medium workload.

In the first analysis of the physical faults (Fig. 4.10 a) the data miner was provided with the workload, segment of the fault injection (text or data), the implementation pattern and the C compiler options used but the exact address of the inverted byte and bit was not used as input parameter. As indicated in the callouts (i) the tiny workload was again unlikely to result in failure and otherwise (ii) faults of the text segment were more likely to result in failures than the ones of the data segment. As the decision tree built by the miner was not really interesting we re-run the experiment by providing the exact address of the inverted bit to the miner (Fig. 4.10 b), furthermore we carried another data mining runs explicitly focusing on faults in the text and data segments respectively; the aggregated observations (involving the trees not presented in the figure also) are shown in the callout at the bottom of Fig. 4.10 b. The text and data segments were divided into four parts by the data miner as shown in the figure (e.g., the four parts of the text segment expressed by the o offset of the byte in which the fault was injected are o < 1449, 1450 ≥ o < 1514, 1515 ≥ o < 1912 and 1913 ≥ o). Unfortunately the data miner did not include either the implementation pattern or the compiler option in the queries therefore the pure fault offsets are practically meaningless: there is no such thing that “1513rd byte of the code segment” since the database contains results about eight binaries (two implementation patterns and
four compiler optimization levels) and the binary alignments are specific for each binaries i.e., the “1513rd byte of the code segment” points to different instructions, values etc. in different binaries. To put together: in case of physical faults the data miner was unable to draw semantically meaningful observations other than the clearly appearing role of the workload; probably more information (relative offsets of faults within functions, etc.) should be provided for achieving more relevant results.

The explanations of the phenomena observed above are as follows:

- The analysis of refinement faults (Fig. 4.9 b) is a very nice indication of the fact that only the small and complex workloads contained aggregated calculations (i.e., ones that actually should have traversed the transition that was not implemented due to the faulty refinement), therefore in case of tiny and medium workloads the user did not experience any failure in the computation. To put together: we experienced here that if the workload does not exploit the faulty part of the system the error will not be activated.

- It is quite easy to understand that in case of implementation faults (Fig. 4.9 c) the tiny workload does not result in failure since this workload contains a single trigger and the fault injection is timed to be performed after the fourth call for the target function i.e., in case of the tiny workload there was no fault injection at all. The more interesting phenomenon is the fact that even in case of the complex workload there were very few failures experienced: this virtual contradiction can be explained by taking a look at the computation performed by the complex workload: in this case there are a lot of triggers provided to the state machine but the most of them does not correspond to any actual computation since the numbers typed in the keyboard are mostly cleared by pressing the CE button, therefore the fault injection results in overwritten error i.e., the faults typically do not propagate to the output. This is another nice example demonstrating the effects of overwritten faults without investing a priori knowledge in the analysis.

- The explanation of fact that implementation faults of the enabled function are more likely to result in failures than faults of the fire single function (Fig. 4.9 c) is as follows (i) failures (in this case) are caused by incorrect computations; (ii) computations in the calculator model are implemented as transition effects; (iii) fault injection in the fire single function results in incorrect calculation of states left therefore possibly result in invalid configuration but does not directly interfere with transition selection (hence the actual computation is not directly affected) and finally (iv) faults injected in the enabled function result in selection of transitions to be fired directly enforcing the calculator to perform an incorrect operation by the transition effect. This example clearly indicates that faults in some functions of the statechart engine result mainly in internal inconsistencies (e.g., fire single) while another ones quickly propagate to the output by causing incorrect calculation (enabled).

- Analysis of physical faults (Fig. 4.11) concluded that the tiny workload typically does not resulted in failures. However the data mining on this subset was less effective in discovering interesting phenomena, this result clearly indicates that in case of a very short trigger sequence most of the data segment is not processed and most of the text segment is not executed at all, therefore most of faults were not activated.

### 4.4.2 Effectiveness of the Watchdog

Our second analysis campaign aims at identifying those factors that affect the error detection potential of PSC-WD. In the first data mining setup (Fig. 4.11 a), the entire database was processed: the data miner was provided with the workload, the type of the faultload, the implementation pattern and the C compiler options used. As indicated in the callouts of the figure (i) in case of non-trivial workloads (small, medium and complex) the watchdog was found to be nearly 100% effective in finding implementation faults, (ii) otherwise its error detection capability was low (over 93% of faults has
remained undetected by the watchdog in all other cases) especially (ii) when using the tiny workload the watchdog is practically incapable of detecting any errors.

Focusing on refinement faults the watchdog detected no errors at all (the trivial decision tree consisting of a single leaf node is not shown in the figure).

The results of analysis focusing on implementation faults is shown in part b of Fig. 4.11 the tree contains a single decision node indicating that PSC-WD detected nearly all (97.9%) of implementation faults in case of non-trivial workloads but when using the tiny workload no errors were detected.

The detailed analysis of physical faults (Fig. 4.11 c) suffers from the same issue as above: the data miner did not exactly identified binaries (implementation pattern and the exact compiler option) therefore the byte intervals selected are practically meaningless. The phenomena observed above can be explained as follows:

- The analysis of the entire database (Fig. 4.11 a) exactly highlighted that the watchdog was very effective for detecting implementation faults in case of non-trivial workloads but performs weak for other fault types or the tiny workload. There is no need for detailed explanation, the watchdog did what it was expected to do: detected the implementation faults if they were activated by the workload. It is important to highlight that this is a very nice result of intelligent data processing since the data miner unambiguously identified the most beneficial application area of the watchdog by the fully automatic analysis of a database of over 180000 records without relying on any a priori intuition.

- The role of the workload even in case of implementation faults (Fig. 4.11 b) is important since obviously faults can not be detected only after being activated and resulting in errors: this is the reason for not having detected practically anything in case of the trivial tiny workload.

- The relatively poor results of data mining in case of physical faults (Fig. 4.11 c) indicate again that effects of physical faults (including being detected or remaining hidden) can not be predicted barely by knowing the implementation pattern, compiler options and exact address of the fault – probably more information is needed for usable results e.g., the instruction or data affected by the fault injection etc.

### 4.4.3 Effectiveness of Low-Level Mechanisms

Our third analysis campaign aims at identifying those factors that affect the error detection potential of low-level mechanisms. Below we will discuss the effectiveness of assertions (i.e., basic sanity checks in the application written by the programmer) and HW/OS signals (i.e., exceptions of the CPU or the memory management unit (MMU) mapped to UNIX signals) separately.
4.4. ANALYSIS OF DATA FROM EVALUATION OF ERROR DETECTION TECHNIQUES

Figure 4.12: Data Mining Analyses Focusing on Low-Level Mechanisms

As assertions indicated errors in very few cases, the data mining analysis of the entire database resulted in the trivial tree containing a single node indicating that in over 95.5% of experiments there was no assertion failure. Similarly when having focused to refinement faults the data miner indicated that there were no assertion failures in all those cases. The first non-trivial tree was achieved when having focused on implementation faults (Fig. 4.12 a). As indicated in the callout, assertions were found to be effective for detecting implementation faults of the fire_single function in case of the medium workload. Having analyzed the effects of physical faults the data miner built similar trees as mentioned above: binaries could not be exactly identified therefore the byte intervals selected by the miner were practically meaningless.

The operating system uses various signals for indicating specific exceptional situations to the application, therefore the leaves of the decisions trees built by the data miner contain the number of the most frequently occurring signal (e.g., 11 for segmentation violation signals etc.). The initial analysis of the entire database resulted in low homogeneity subsets in leaves of the tree therefore we continued the analysis by selecting smaller subsets similarly to previous analyses.

The decision tree built when having focused on implementation faults is presented in part b of Fig. 4.12. It is easy to see that the tree is nearly the same as the one built for assertions above the same database subset (Fig. 4.12 a): where no assertion failures were reported the 0 signal number was found (indicating no signal) while assertion failures seem to be associated to signal number 6. The explanation is as follows: assertions are implemented using the standard ANSI-C assert macro (defined in the standard assert.h header file); this macro is translated by the C preprocessor to an if instruction that evaluates the predicate in the body of the assertion and if it does not hold calls the abort function; the abort function is implemented on UNIX by sending a SIGABORT signal to the process and this is the signal 6 we are searching for. To put together: as assertion failures are indicated by the OS signal number 6 the two error detection mechanisms (user-level assertions) and signals can not be discussed in total isolation.

The phenomena observed above can be explained as follows:

- Probably the most challenging task related to the data mining analysis was the explanation of the phenomenon that faults of the fire_single function in case of the medium workload resulted in assertions (Fig. 4.12 a) and were also indicated by analysis of signals (Fig. 4.12 b): we had to investigate the actual output of the applications and the corresponding watchdog logs (describing the traversal in the state space). We found that in these cases the fault injection caused the fire_single function to return an invalid set of states entered, resulting in an invalid configuration finally enforcing the engine to fire two transitions in parallel multiple times finally achieving another invalid configuration where a “state was entered multiple times” and this serious inconsistency was detected by an assertion.

The reconstructed scenario is as follows: (i) the state machine received a num19Event trigger in the configuration calc, operand1, frac1; (ii) during the execution of the fire_single function the fault injection forced the function to return an invalid set of states entered i.e., instead of
re-entering state \text{frac1}, \text{fire-single} reported \text{zero2} to be entered resulting in the invalid configuration \text{calc}, \text{operand1}, \text{frac1}, \text{zero2}; (iii) the next trigger was a \text{num19Event} again and in this case two transitions (\text{frac1} \rightarrow \text{frac1} and \text{zero2} \rightarrow \text{int2}) were found to be fireable -- since the transitions were not in conflict the sanity checking was passed resulting in another invalid configuration \text{calc}, \text{operand1}, \text{frac1}, \text{int2}; (iv) the next event was a \text{num19Event} again and the state machine performed two loop transitions in parallel again (\text{frac1} \rightarrow \text{frac1} and \text{int2} \rightarrow \text{int2}) remaining this way in the same configuration; (v) next an \text{operEvent} was received by the state machine that triggered the \text{operand1} \rightarrow \text{opentered} transition resulting in another invalid configuration \text{calc}, \text{opentered}, \text{int2}; (vi) finally an \text{num19Event} was received again triggering two transitions: \text{opentered} \rightarrow \text{int2} and \text{int2} \rightarrow \text{int2} -- this was the point when a basic sanity check indicated that the state \text{int2} was entered two multiple times resulting in an assertion failure.

The most important aspect of this scenario for being especially remarkable is to realize that the fault (invalid set returned by \text{fire-single}) resulted at once in an error (invalid configuration) but remained undetected for four entire run-to-completion steps by the assertion mechanism and only due to a fortunate accidental event (entering the same state twice) resulted in an assertion failure. We should recall here that the watchdog detected all implementation faults including these ones at once after entering the first invalid configuration. The lesson learnt here is as follows: although the low-level error detection mechanisms (here: assertions) have some minor chance for detecting higher-level faults, this phenomena require the fortunate coincidence of the actual fault and workload; in contrast error detection mechanisms operating at higher abstraction level (i.e., the watchdog) did not need to be “lucky” for identifying this kind of erroneous behavior. The benefit of using data intelligence here was to identify this very infrequently occurring phenomenon that was not anticipated by the designer of the experiment by any \textit{a priori} assumption.

- When having focused on \textit{physical faults} (Fig. 4.12, \textit{c}) we identified that faults on the text segment are more likely to result in segmentation violation signals than faults of the data segment. It is important to understand that the segmentation violation in a UNIX system occurs when an application tries to access a memory region outside of its valid memory area -- this is typically caused by an \textit{invalid pointer operation}. The phenomenon can be explained by taking into consideration the organization of initialized data in the data segment: most of the space is occupied by compact state or transition masks that are practically byte arrays using few pointers, therefore bit inversions in the data segment are not likely to result in corrupted pointers therefore the memory management unit can not detect them (and probably result in harder to detect semantic inconsistencies e.g., corrupting the set of enabling states in a transition) -- in contrast the text segment contains the implementations of relatively short functions that contain many tight loops; these loops are implemented at the machine code level by conditional branch instructions and when the target address of such a branch instruction is corrupted by the fault injection the application is likely to perform a jump to an invalid address resulting in an MMU exception delivered to the process as a segmentation violation signal.

4.5 Conclusions and Future Work

This chapter has presented a novel approach for automatically identifying the key factors determining the behavior of systems in the presence of faults by intelligent data processing methods. After a short introduction to basic concepts of data mining we discussed how to exploit the benefits of automated tools enabling this way the application of most advanced data processing intelligence for analyzing experiment results by the dependability community. We have proven the viability of our approach by analyzing a data set obtained during a standard dependability benchmarking experiment (DBench-OLTP) and by analyzing the data collected during the experimental evaluation of our techniques. The key contributions of this chapter are the method for automatic identification of key factors
determining dependability attributes and the analyses presented as illustration. We did not aim at introducing or improving any data mining techniques: the dependability community may benefit from our achievements by realizing that data intelligence methods that were successfully applied in the business field can provide valuable support for automatic analysis of large sets of experimental data without actual in-depth knowledge of the data processing algorithms themselves. The achievements discussed in this chapter are summarized by the following thesis.

**Thesis 4 (Application of Data Mining in Experimental Evaluation of Dependability).** I presented a data mining based method for automatic identification of those key factors that significantly affect the behavior of complex HW/SW systems in the presence of faults:

(i) I elaborated a *method for automatic processing of data obtained from dependability evaluation experiments* by the application of the classification data mining technique. The method provides support for automatic identification of those infrastructural, workload and faultload factors that significantly affect the behavior of complex HW/SW systems built up of COTS elements considered to be black boxes.

(ii) I demonstrated the viability of the approach by the investigation of a data set obtained from fault injection experiments conducted on a complex data warehouse (DBench-OLTP experiments).

(iii) I carried out a *comprehensive experiment* campaign by the joint application of main achievements presented in this document: I performed fault injection experiments (involving model refinement, implementation and physical faults) on applications generated from UML 2.0 statechart models according to the implementation method discussed in Chp. 2; the execution was monitored by the runtime error detection techniques introduced in Chp. 3; finally using the method presented in this chapter I identified those factors that had a significant effect on the effectiveness on the error detection methods.

Our preliminary approaches for introducing data intelligence methods for automatic analysis of experimental data were published in [113] and [138]. Our most important paper in this subject [123] was published in the European Dependable Computing Conference – this chapter is primarily based on this paper; an extended version of the conference presentation was prepared as a technical report [122] including those detailed discussions that did not fit into the conference paper (about 130 pages).

As discussed above, the *application* of our method enables the automatic analysis of large experimental data sets to identify those infrastructural, workload and faultload factors that primarily affect the behavior of the system in the presence of faults.

Our future research activities may be focused on the joint application of On-Line Analytical Processing (OLAP) and data mining techniques in the analysis of experimental data. Another interesting subject is providing feedback to designers of various benchmarks about the representativeness of their techniques e.g., in case of a standard CPU benchmark by the investigation of a large amount of measurement results using a method similar to the one discussed here we could identify that component of the system that had a primary impact on the score determined by the benchmark: in this example if we would find that the memory had a more important impact that the CPU itself, the benchmark should be redesigned.
Chapter 5

Conclusions

This thesis has presented achievements of over three years of research focusing on various aspects of software dependability. This final chapter enumerates the achievements once again and highlights their applicability in the engineering practice.

Chp. 1 presented a novel formal operational semantics for statecharts of the UML 2.0 modeling language. First we separated the fundamental modeling concepts of statecharts from shorthand notations and decided to present a formal operational semantics for fundamental concepts and a denotational semantics for shorthand concepts (i.e., presenting a transformation of shorthand notations to composite structures built up of fundamental elements). The solid foundations of our formalization efforts were established by defining the metamodel of precise statecharts and a set of well-formedness rules, formal definition of refinement concepts (state containment, refining states to regions etc.), compound transition structures and compound activity structures. Our approach puts emphasis on modeling parallel execution by defining compound activity structures based on PERT graphs enabling this way the straightforward exploitation of parallel computing resources of modern platforms. The operational semantics was defined by a Kripke transition system and the semantics was finally presented by imperative algorithms, enabling this way the straightforward implementation in popular programming languages.

Chp. 2 presented a method for automatic source-code level implementation of behavior specified by UML 2.0 statecharts. According to our knowledge this solution is the most complete approach for automatic source-code level implementation of UML 2.0 statecharts even supporting resource constrained embedded systems. The remarkable features of our method are as follows: (i) the solution corresponds to the steps of the Model Driven Architecture i.e., conceptually different tasks (identification of target platform’s features, abstract mapping to the available resources and the implementation) are clearly separated; (ii) a significant effort was laid on proving the correctness of subsequent transformation steps; (iii) being based on the semantics presented in Chp. 1 our solution supports the most complete toolkit of the most recent version of the UML standard, (iv) the mapping to resource constrained platforms does not restrict the set of usable modeling concepts and (v) for sufficiently complex models the performance and size of the resulting applications is even more beneficial as compared to the ones built according to less sophisticated approaches supporting much less modeling features.

Chp. 3 presented two techniques for runtime detection of behavioral errors in UML statechart implementations. First we established the foundations for reasoning about dependability impairments and fault tolerance, associated various dependability-related activities to the corresponding phases of software development and identified two aspects of runtime error detection where open issues were found. These unsolved problems were targeted by our related research: we addressed (i) model refinement faults by introducing a propositional linear temporal logic language (PSC-PLTL) for defining dependability criteria in the context of statecharts and (ii) faults in the implementation of statechart models by a statechart-level watchdog solution (PSC-WD). After presenting a background on temporal logic languages we formally defined the PSC-PLTL language. Being the PLTL evaluation a definitely
non-trivial issue, we presented a novel high performance method for the effective evaluation of PLTL formulae over finite execution traces. The second main part of the chapter was dedicated to detecting implementation faults: after an overview on watchdog processors our statechart-level watchdog solution (PSC-WD) was formally synthesized from the formal semantics of precise statecharts. Finally the error detection capabilities of the two solutions were experimentally evaluated: the fault injection campaign concluded that PSC-WD detected the majority of errors caused by implementation faults and the PSC-PLTL checker successfully addressed model refinement faults that were not detected by any other solutions.

Finally Chp. 4 presented a novel approach for automatic identification of key factors determining the behavior of systems in the presence of faults by intelligent data processing methods. After a short introduction to basic concepts of data mining we discussed how to exploit the benefits of automated tools enabling this way the application of most advanced data processing intelligence for analyzing experiment results by the dependability community. We have proven the viability of our approach by analyzing a data set obtained during a standard dependability benchmarking experiment (DBench-OLTP) and by analyzing the data collected during the experimental evaluation of our techniques. The key contributions of this chapter are the method for automatic identification of key factors determining dependability attributes and the analyses presented as illustration. We did not aim at introducing or improving any data mining techniques: the dependability community may benefit from our achievements by realizing that data intelligence methods that were successfully applied in the business field can provide valuable support for automatic analysis of large sets of experimental data without actual in-depth knowledge of the data processing algorithms themselves.

Having presented set of pragmatic approaches, the possible application of our achievements in quite straightforward. The automatic implementation scheme discussed in Chp. 2 enables the efficient implementation of systems specified by statecharts on resource constrained embedded systems, promising this way increased productivity, reduced development time and enhanced software quality. Automatic implementation of behavioral models can also be beneficial for fast prototyping purposes. Our error detection solutions were aimed at the runtime detection of errors in dependable applications caused by model refinement and implementation faults; furthermore in case of resource constrained systems where sophisticated low-level HW/OS error detection mechanisms are not available our solutions can be used even for detecting errors caused by physical faults. These error detection facilities can be the fundamental building blocks for the composition of fault tolerant solutions that are capable of initiating fault confinement and error removal activities on the detection of various errors in order to provide continuous service even in the presence of faults. With respect to our method for PLTL evaluation we have to highlight that the solution is not bound to statecharts or the PSC-PLTL language by any means: this approach can be used wherever resource efficient evaluation of PLTL formulae is needed e.g., runtime verification approaches where executable assertions are expressed in temporal logic.

Obviously both the automatic implementation scheme and the runtime error detection approaches are built on the semantics introduced in Chp. 1: furthermore a direct application of the semantics is the statechart simulator whose prototype implementation was already mentioned in the thesis. Our work related to intelligent data processing is mostly an application itself: the method enables the automatic analysis of large experimental data sets to identify those infrastructural, workload and faultload factors that primarily affect the behavior of the system in the presence of faults.

The four main chapters of the thesis have established the foundations for a comprehensive toolchain supporting the model driven development of dependable embedded systems. We are currently working on providing additional features for modeling, simulation and testing behavioral structures, e.g., a model checking solution based on translating the statechart semantics to the input language of a model checker tool for proving various dependability constraints in the context of statechart models and an automatic test generator (also based on a special application of a model checker) for the synthesis of efficient test cases aiming at the achievement of state, transition, etc. coverage goals.
Appendices
Appendix A

Details on Formal Semantics

This appendix discusses some details about the formal operational semantics presented in Chp. I that did not make into the main parts due to space restrictions: Sec. A.1 collects some quotations from the UML standard that were used for underpinning our design decisions; the detailed well-formedness requirements about statecharts are presented in Sec. A.2; some lengthy definitions and proofs are presented in Sec. A.3 and Sec. A.5 respectively. The main part of this document uses a pseudocode notation for presenting most of algorithms for space efficiency; unfortunately a pseudocode is mainly aimed at human reading but has no unambiguous formal semantics, thus in Sec. A.4 we present the formal specification of all the algorithms mentioned in the main part using the Microsoft Abstract Statemachine Language (AsmL), the well-established executable specification language developed for the Microsoft .Net platform. As our precise statechart formalism does not include those UML statechart features that are considered to be shorthand notations (history mechanism, junction and choice pseudostates, embedding state machines etc.), Sec. A.6 outlines a set of transformations for replacing these elements by core constructs, enabling this way our simulation, code generation and runtime error detection techniques to support the entire modeling toolkit of UML 2.0 statecharts without having to introduce more constructs into the core language (due to space restrictions the formal definitions of transformations are not discussed here, for a detailed presentation see [132]; obviously our all implementations automatically perform these transformations enabling this way the modeler to use all UML 2.0 statechart elements in simulation, code generation or runtime error detection). The final two sections present illustrative examples: Sec. A.7 discusses the final steps of statechart operation scenario started in Fig. 1.19 and Fig. 1.20 while Sec. A.8 presents the same scenario as the output of our simulator implementing our formal operational semantics for statecharts.

A.1 Citations from the UML 2.0 Standard

All the UML standard related discussions in the document are based on the UML 2.0 Superstructure Specification released at October 8, 2004 by the OMG (document id: ptc/04-10-02). As at the beginning of our semantics-related research the UML 2.0 standard was yet finalized we decided to use this nearly finished version (called “Revised Final Adopted Specification” by OMG) as basis of discussions. At the time of preparing the final version of the thesis, the most recent version of the standard is UML 2.1; in this version some issues mentioned below are already fixed, but most of key observations still hold.

Various sections of the standard were used for underpinning some of our design considerations, syntactic and semantic decisions; although the document expects a fair knowledge on the key UML statechart concepts and informal semantics, below we quote some fragments of the standard that were cited in our discussions. (An example for references to quotations below: “final states can not have outgoing transitions as discussed in Cit. I’.)
Citation 1 (15.3.2 FinalState (from BehaviorStateMachines)). A special kind of state signifying that the enclosing region is completed. If the enclosing region is directly contained in a state machine and all other regions in the state machine also are completed, then it means that the entire state machine is completed. [...] Notation: A final state is shown as a circle surrounding a small solid filled circle. [...] Constraints: (i) a final state cannot have any outgoing transitions, (ii) a final state cannot have regions, (iii) a final state cannot reference a submachine, (iv) a final state has no entry behavior, (v) a final state has no exit behavior, (vi) a final state has no state (doActivity) behavior.

Citation 2 (15.3.8 Pseudostate (from BehaviorStateMachines)). A pseudostate is an abstraction that encompasses different types of transient vertices in the state machine graph.

Citation 3 (Initial Pseudostate). An initial pseudostate represents a default vertex that is the source for a single transition to the default state of a composite state. There can be at most one initial vertex in a region. The initial transition may have an action. [...] Notation: An initial pseudostate is shown as a small solid filled circle. [...] Constraints: An initial vertex can have at most one outgoing transition.

Citation 4 (Deep History Pseudostate). Deep history represents the most recent active configuration of the composite state that directly contains this pseudostate; e.g. the state configuration that was active when the composite state was last exited. A composite state can have at most one deep history vertex. At most one transition may originate from the history connector to the default deep history state. This transition is taken in case the composite state had never been active before. Entry actions of states entered on the path to the state represented by a deep history are performed. [...] Notation: A shallow history is indicated by a small circle containing an “H”. [...] Constraints: History vertices can have at most one outgoing transition.

Citation 5 (Shallow History Pseudostate). Shallow history represents the most recent active substate of its containing state (but not the substates of that substate). A composite state can have at most one shallow history vertex. A transition coming into the shallow history vertex is equivalent to a transition coming into the most recent active substate of a state. At most one transition may originate from the history connector to the default shallow history state. This transition is taken in case the composite state had never been active before. Entry actions of states entered on the path to the state represented by a shallow history are performed. [...] Notation: A deep history is indicated by a small circle containing an “H”’. [...] Constraints: History vertices can have at most one outgoing transition.

Citation 6 (Join Pseudostate). Join vertices serve to merge several transitions emanating from source vertices in different orthogonal regions. The transitions entering a join vertex cannot have guards or triggers. [...] Notation: The notation for a fork and join is a short heavy bar. [...] Constraints: (i) a join vertex must have at least two incoming transitions and exactly one outgoing transition, (ii) all transitions incoming a join vertex must originate in different regions of an orthogonal state.

Citation 7 (Fork Pseudostate). Fork vertices serve to split an incoming transition into two or more transitions terminating on orthogonal target vertices (i.e. vertices in different regions of a composite state). The segments outgoing from a fork vertex must not have guards or triggers. [...] Notation: The notation for a fork and join is a short heavy bar. [...] Constraints: (i) a fork vertex must have at least two outgoing transitions and exactly one incoming transition, (ii) transitions outgoing a fork vertex must target states in different regions of an orthogonal state.

Citation 8 (Junction Pseudostate). Junction vertices are semantic-free vertices that are used to chain together multiple transitions. They are used to construct compound transition paths between
states. For example, a junction can be used to converge multiple incoming transitions into a single outgoing transition representing a shared transition path (this is known as an merge). Conversely, they can be used to split an incoming transition into multiple outgoing transition segments with different guard conditions. This realizes a static conditional branch. (In the latter case, outgoing transitions whose guard conditions evaluate to false are disabled. A predefined guard denoted “else” may be defined for at most one outgoing transition. This transition is enabled if all the guards labeling the other transitions are false.) Static conditional branches are distinct from dynamic conditional branches that are realized by choice vertices (described below).  

| Citation 9 (Choice Pseudostate). | Choice vertices which, when reached, result in the dynamic evaluation of the guards of the triggers of its outgoing transitions. This realizes a dynamic conditional branch. It allows splitting of transitions into multiple outgoing paths such that the decision on which path to take may be a function of the results of prior actions performed in the same run-to-completion step. If more than one of the guards evaluates to true, an arbitrary one is selected. If none of the guards evaluates to true, then the model is considered ill-formed. (To avoid this, it is recommended to define one outgoing transition with the predefined “else” guard for every choice vertex.) Choice vertices should be distinguished from static branch points that are based on junction points (described above). |
| Citation 10 (Entry Point Pseudostate). | An entry point pseudostate is an entry point of a state machine or composite state. In each region of the state machine or composite state it has a single transition to a vertex within the same region. |
| Citation 11 (Exit Point Pseudostate). | An exit point pseudostate is an exit point of a state machine or composite state. Entering an exit point within any region of the composite state or state machine referenced by a submachine state implies the exit of this composite state or submachine state and the triggering of the transition that has this exit point as source in the state machine enclosing the submachine or composite state. |
| Citation 12 (Terminate Pseudostate). | Entering a terminate pseudostate implies that the execution of this state machine by means of its context object is terminated. The state machine does not exit any states nor does it perform any exit actions other than those associated with the transition leading to the terminate pseudostate. |
| Citation 13 (15.3.10 Region (from BehaviorStateMachines)). | A region is an orthogonal part of either a composite state or a state machine. It contains states and transitions. |
| Citation 14 (15.3.14 Transition (from BehaviorStateMachines)). | A transition is a directed relationship between a source vertex and a target vertex. It may be part of a compound transition, which takes the state machine from one state configuration to another, representing the complete response of the state machine to an occurrence of an event of a particular type. |
APPENDIX A. DETAILS ON FORMAL SEMANTICS

Constraints:
(i) a fork segment must not have guards or triggers, (ii) a join segment must not have guards or triggers, (iii) a fork segment must always target a state, (iv) a join segment must always originate from a state, (v) transitions outgoing pseudostates may not have a trigger and (vi) an initial transition at the topmost level (region of a state machine) either has no trigger or it has a trigger with the stereotype “create”.

Citation 15 (15.3.15 TransitionKind (from BehaviorStateMachines)). TransitionKind is an enumeration type of the following literal values: external, internal or local. [...] Semantics: (i) The kind = internal implies that the transition, if triggered, occur without exiting or entering the source state. Thus, it does not cause a state change. This means that the entry or exit condition of the source state will not be invoked. An internal transition can be taken even if the state machine is in one or more regions nested within this state. (ii) The kind = local implies that the transition, if triggered, will not exit the composite (source) state, but it will apply to any state within the composite stat, and these will be exited and entered. (iii) The kind = external implies that the transition, if triggered, will exit the composite (source) state.

Citation 16 (15.3.11 State (from BehaviorStateMachines, ProtocolStateMachines)). A state models a situation during which some (usually implicit) invariant condition holds. [...] Notation: A state is in general shown as a rectangle with rounded corners, with the state name shown inside the rectangle. [...] Constraints: (i) there have to be at least two regions in an orthogonal composite state, (ii) only submachine states can have connection point references, (iii) the connection point references used as destinations/sources of transitions associated with a submachine state must be defined as entry/exit points in the submachine state machine (iv) a state is not allowed to have both a submachine and regions (v) a simple state is a state without any regions, (vi) a composite state is a state with at least one region, (vii) an orthogonal state is a composite state with at least 2 regions, (viii) only submachine states can have a reference state machine, (ix) the redefinition context of a state is the nearest containing state machine, (x) only composite states can have entry or exit pseudostates defined and (xi) only entry or exit pseudostates can serve as connection points. (See the standard for the (faulty) OCL formulæ.)

Citation 17 (15.3.12 StateMachine (from BehaviorStateMachines)). State machines can be used to express the behavior of part of a system. Behavior is modeled as a traversal of a graph of state nodes interconnected by one or more joined transition arcs that are triggered by the dispatching of series of (event) occurrences. During this traversal, the state machine executes a series of activities associated with various elements of the state machine. [...] Constraints: (i) the classifier context of a state machine cannot be an interface, (ii) the context classifier of the method state machine of a behavioral feature must be the classifier that owns the behavioral feature, (iii) connection points of a state machine are pseudostates of kind entry point or exit point (iv) a state machine as the method for a behavioral feature cannot have entry/exit connection points.

Citation 18 (State Entry and Exit Activities). If the transition goes to a substate of the composite state, then that substate becomes active and its entry code is executed after the execution of the entry code of the composite state. This rule applies recursively if the transition terminates on a transitively nested substate. [...] When exiting from a composite state, the active substate is exited recursively. This means that the exit activities are executed in sequence starting with the innermost active state in the current state configuration. (This citation was constructed from fragments of section 15.3.11.)

Citation 19 (Run-to-Completion Behavior). The processing of a single event occurrence by a state machine is known as a run-to-completion step. Before commencing on a run-to-completion step, a state machine is in a stable state configuration with all entry/exit/internal activities (but not necessarily state (do) activities) completed. The same conditions apply after the run-to-completion
step is completed. Thus, an event occurrence will never be processed while the state machine is in some intermediate and inconsistent situation. The run-to-completion step is the passage between two state configurations of the state machine. The run-to-completion assumption simplifies the transition function of the state machine, since concurrency conflicts are avoided during the processing of event, allowing the state machine to safely complete its run-to-completion step. (This citation was constructed from fragments of section 15.3.12.)

Citation 20 (Enabled Transition). A transition is enabled if and only if: (i) All of its source states are in the active state configuration. (ii) One of the triggers of the transition is satisfied by the event (type) of the current occurrence. An event satisfies a trigger if it matches the event specified by the trigger. (iii) If there exists at least one full path from the source state configuration to either the target state configuration or to a dynamic choice point in which all guard conditions are true (transitions without guards are treated as if their guards are always true). Since more than one transition may be enabled by the same event, being enabled is a necessary but not sufficient condition for the firing of a transition. (This citation was taken from section 15.3.14.)

Citation 21 (Conflicting Transitions). It was already noted that it is possible for more than one transition to be enabled within a state machine. If that happens, then such transitions may be in conflict with each other. For example, consider the case of two transitions originating from the same state, triggered by the same event, but with different guards. If that event occurs and both guard conditions are true, then only one transition will fire. In other words, in case of conflicting transitions, only one of them will fire in a single run-to-completion step. Two transitions are said to conflict if they both exit the same state, or, more precisely, that the intersection of the set of states they exit is non-empty. Only transitions that occur in mutually orthogonal regions may be fired simultaneously. This constraint guarantees that the new active state configuration resulting from executing the set of transitions is well formed. An internal transition in a state conflicts only with transitions that cause an exit from that state. (This citation was taken from section 15.3.12.)

Citation 22 (Transition Firing Priorities). In situations where there are conflicting transitions, the selection of which transitions will fire is based in part on an implicit priority. These priorities resolve some transition conflicts, but not all of them. The priorities of conflicting transitions are based on their relative position in the state hierarchy. By definition, a transition originating from a substate has higher priority than a conflicting transition originating from any of its containing states. The priority of a transition is defined based on its source state. The priority of joined transitions is based on the priority of the transition with the most transitively nested source state. In general, if t1 is a transition whose source state is s1, and t2 has source s2, then: (i) if s1 is a direct or transitively nested substate of s2, then t1 has higher priority than t2; (ii) if s1 and s2 are not in the same state configuration, then there is no priority difference between t1 and t2. (This citation was taken from section 15.3.12.)

Citation 23 (Transition Selection Algorithm). The set of transitions that will fire is a maximal set of transitions that satisfies the following conditions: (i) All transitions in the set are enabled. (ii) There are no conflicting transitions within the set. (iii) There is no transition outside the set that has higher priority than a transition in the set (that is, enabled transitions with highest priorities are in the set while conflicting transitions with lower priorities are left out). (This citation was taken from section 15.3.12.)

Citation 24 (Firing Compound Transitions). A compound transition is a derived semantic concept, represents a semantically complete path made of one or more transitions, originating from a set of states (as opposed to pseudo-state) and targeting a set of states. [...] In general, a compound transition is an acyclical unbroken chain of transitions joined via join, junction, choice, or fork pseudostates that define path from a set of source states (possibly a singleton) to a set of destination states, (possibly a singleton). [...] Behaviors are executed in sequence following their linear order along the
segments of the transition: The closer the behavior to the source state, the earlier it is executed. (This
citation was constructed from fragments of section 15.3.14.)

A.2 Well-Formedness of Precise Statecharts

The UML 2.0 standard uses the Object Constraint Language (OCL) for specifying most of the well-
formedness rules. Unfortunately in multiple cases the OCL expressions do not exactly represent the
textually introduced rules, some of them are simply missing from the document and there are some
syntactically incorrect statements as well. Another issue with OCL is that in non-trivial expressions the
semantic meaning of the OCL statement is questionable or rather hard to read. In our approach we use
first order logic for the specification of well-formedness rules and implement them in the semantically
well-founded Abstract State Machine Language (AsmL). Note again that the discussion below refers to
UML Superstructure Specification 2.0 released at October 8, 2004 (document id: ptc/04-10-02); some
of minor issues enumerated below are fixed in the UML 2.1 standard (e.g., ones related to final states)
but most of serious ones has still remained in the standard necessitating this way our formalization
efforts, (e.g., fork and join pseudostates, etc.).

A.2.1 Final States

The UML standard (15.3.2) specifies that a final state (i) cannot have any outgoing transitions, (ii)
cannot have regions, (iii) cannot reference a submachine, (iv) has no entry behavior, and (v) has no
exit behavior (Cit. [4]). We do not add any further restrictions but as the standard specifies only the
first restriction formally, the formal specification of the entire constraint set is our contribution.

\[ \forall f \in \text{FinalState} : (f.\text{outgoing} = \emptyset) \land (f.\text{region} = \emptyset) \land (f.\text{submachine} = \emptyset) \land (f.\text{entry} = \emptyset) \land (f.\text{exit} = \emptyset) \]  
(A.1)

A.2.2 Initial Pseudostates

The UML standard (15.3.8) specifies only that an initial vertex can have at most one outgoing transition
(Cit. [5]). We find this restriction nearly useless since it enables the modeler to put an initial pseudostate
without any outgoing transitions in a compound state and the meaning of this situation is not even
mentioned in the standard. On the other hand the standard enables the usage of compound states
without initial states that may result in ambiguous models if the compound state is entered through
the default entry (i.e., a transition targets the state itself not one of its substates). Another issue is
that the standard does not restrict that the transition originating in the initial vertex should actually
target a state in the same region.

Having considered these issues we introduce the following well-formedness constraints for precise
statecharts: (i) all regions should have exactly one initial pseudostate subvertex (Eq. [A.2]), (ii) each
initial pseudostates should have exactly one outgoing transition that (iii) targets a state (i.e., not a
pseudostate) (iv) in the same region and (v) this transition has neither trigger (vi), nor guard condition
or (vii) effect and (viii) an initial pseudostate may not be the target of any transitions (Eq. [A.3]):

\[ \forall r \in \text{Region} : \{i \mid i \in r.\text{subvertex} \land i \in \text{Pseudostate}^\text{In}\} = 1 \]  
(A.2)
\[ \forall i \in \text{Pseudostate}^\text{In} : (i.\text{outgoing} = 1) \land (i.\text{outgoing}.\text{target} \in \text{State}) \land (i.\text{outgoing}\.\text{container} = i.\text{container}) \land (i.\text{outgoing}\.\text{trigger} = \emptyset) \land (i.\text{outgoing}\.\text{guard} = \emptyset) \land (i.\text{outgoing}\.\text{effect} = \emptyset) \land (i.\text{incoming} = \emptyset) \]  
(A.3)

(Note that in (ii) of Eq. [A.3] we restricted that the set i.\text{outgoing} contains a single element, so in
(iii) – (vii) we could simply navigate through it.)
A.2. WELL-FORMEDNESS OF PRECISE STATECHARTS

A.2.3 Join Pseudostates

The UML standard (15.3.8) specifies that (i) a join vertex must have at least two incoming transitions and exactly one outgoing transition and (ii) all transitions incoming a join vertex must originate in different regions of an orthogonal state (Cit. 6).

Although the standard requires that transitions incoming a join vertex must originate in different concurrent regions, the corresponding OCL constraint specifies only that the least common ancestor (LCA) state of the source state is orthogonal allowing the modeler to build state machines where two non-concurrent substates (i.e., ones in the same region) of an orthogonal state are the sources of transitions targeting a join pseudostate resulting in a semantically meaningless construct. Another weakness of the standard that it actually lacks the formal definition of the LCA. These issues have been fixed in our formal specification below.

Furthermore with respect to transitions the standard (15.3.14) specifies that (i) a join segment (i.e., a transition targeting a join pseudostate) must not have guards or triggers and (ii) a join segment must always originate from a state (Cit. 14).

Below we collect all well-formedness constraints corresponding to join pseudostates (either introduced by the standard directly (Cit. 6) or in the context of transitions in (Cit. 14)) into a consistent set and present them formally: (i) a join vertex must have at least two incoming transitions, (ii) exactly one outgoing transition (Eq. A.4), (iii) all transitions targeting a join vertex should originate in different regions of an orthogonal state (Eq. A.5), (iv) the sources of these transitions are states, (v) these transitions may not have guards or (vi) triggers (Eq. A.6) and (vii) finally we require that the single transition originating in the join vertex should target a fork pseudostate or a state (Eq. A.7):

\[
\forall j \in \text{Pseudostate}^{jo} : (|j.\text{incoming}| \geq 2) \land (|j.\text{outgoing}| = 1) \tag{A.4}
\]

\[
\forall j \in \text{Pseudostate}^{jo} : \forall t, t_j \in j.\text{incoming} : \exists s \in (\text{State} \cup \{s_{sm0}\}) : \exists r_i, r_j \in \text{AllRef}^D(s) : r_i \neq r_j \land t_j.\text{source} \in \text{AllCont}^T(r_j) \land t.\text{source} \in \text{AllCont}^T(r_i) \tag{A.5}
\]

\[
\forall j \in \text{Pseudostate}^{jo} : \forall t \in j.\text{incoming} : (t.\text{source} \in \text{State}) \land (t.\text{guard} = \emptyset) \land (t.\text{trigger} = \emptyset) \tag{A.6}
\]

\[
\forall j \in \text{Pseudostate}^{jo} : j.\text{outgoing}.\text{target} \in (\text{Pseudostate}^{fo} \cup \text{State}) \tag{A.7}
\]

(Note that in Eq. A.4 we restricted that the set j.outgoing contains a single element, so in Eq. A.7 we could simply navigate through it. Symbols AllRef^D and AllCont^T used in Eq. A.5 represent all regions directly refining s and all states transitively contained by r respectively – see Def. 1 and Def. 6 for formal definitions.)

A.2.4 Fork Pseudostates

The UML standard (15.3.8) specifies that (i) a fork vertex must have at least two outgoing transitions and exactly one incoming transition and (ii) all transitions outgoing a fork vertex must target states in different regions of an orthogonal state (Cit. 7).

Although the standard requires that all transitions outgoing a fork vertex must target states in different concurrent regions, the corresponding OCL constraint specifies only that the LCA state of the target states is orthogonal allowing the modeler to build state machines where two non-concurrent substates (i.e., ones in the same region) of an orthogonal state are targeted by transitions originating in a fork pseudostate resulting in a semantically meaningless construct. This issue is fixed in our formal specification.

Furthermore with respect to transitions the standard (15.3.14) specifies that (i) a fork segment (i.e., a transition originating in a fork pseudostate) must not have guards or triggers and (ii) a fork segment must always target a state (Cit. 14).

Below we collect all well-formedness constraints corresponding to fork pseudostates (either introduced by the standard directly in (15.3.8) or in the context of transitions in (15.3.14)) into a consistent...
set and present them formally: (i) a fork vertex must have at least two outgoing transitions, (ii) exactly one incoming transition (Eq. A.8), (iii) all transitions originating in a fork vertex should target different regions of an orthogonal state (Eq. A.9), (iv) the targets of these transitions are states, (v) these transitions may not have guards or (vi) triggers (Eq. A.10) and finally we require that (vii) the single transition targeting the fork vertex should originate in a join, junction, shallow history or deep history pseudostate or a state (Eq. A.11).

\[ \forall f \in \text{Pseudostate}^{\text{Fo}}: (|f.\text{outgoing}| \geq 2) \land (|f.\text{incoming}| = 1) \]  
\[ (A.8) \]
\[ \forall f \in \text{Pseudostate}^{\text{Fo}}: \forall t_i, t_j, s_i, s_j \in f.\text{outgoing}: \exists s \in (\text{State} \cup \{s_{0}\}): \exists r_i, r_j | r_i \neq r_j \in \text{AllRef}^{\text{D}}(s) : t_i.\text{target} \in \text{AllCont}^{\text{T}}(r_i) \land t_j.\text{target} \in \text{AllCont}^{\text{T}}(r_j) \]  
\[ (A.9) \]
\[ \forall f \in \text{Pseudostate}^{\text{Fo}}: \forall t \in f.\text{outgoing}: (t.\text{target} \in \text{State}) \land (t.\text{guard} = \emptyset) \land (t.\text{trigger} = \emptyset) \]  
\[ (A.10) \]
\[ \forall f \in \text{Pseudostate}^{\text{Fo}}: f.\text{incoming}.\text{source} \in (\text{Pseudostate}^{\text{Fo}} \cup \text{State}) \]  
\[ (A.11) \]

(Note that in Eq. A.8 we restricted that the set $f.\text{incoming}$ contains a single element, so in Eq. A.11 we could simply navigate through it. Symbols $\text{AllRef}^{\text{D}}$ and $\text{AllCont}^{\text{T}}$ used in Eq. A.9 represent all regions directly refining $s$ and all states transitively contained by $r$ respectively – see Def. 1 and Def. 6 for formal definitions.)

**A.2.5 Termination States**

Termination states are newly introduced metaclasses in the precise statechart metamodel representing the UML terminate pseudostate concept in a syntactically modified form. (The syntactic modification is needed for eliminating the conceptual contradiction between the “transient vertex” nature of pseudostates and the notion of terminated status and for more straightforward handling of transition semantics as discussed in later parts of this chapter.)

The standard (15.3.8) mentions that “entering a terminate pseudostate implies that the execution of this state machine by means of its context object is terminated, the state machine does not exit any states nor does it perform entry and exit actions other that those associated with the transition leading to the terminate pseudostate” (Cit. 12). Although this specification implies that a terminate pseudostate may not be the source of any outgoing transitions, the standard does not express this formally. We could introduce this well-formedness rule formally for termination states, but since termination states are descendants of final states in our model and final states may not be sources of any transitions (see (i) in Eq. A.1), this restriction is already inherited, the further restrictions corresponding to final states (no refining regions, no submachines, no entry actions, no exit actions and no do activities) apply seamlessly to terminate pseudostates.

**A.2.6 Regions**

The UML standard (15.3.10) specifies that a region can have at most one initial vertex (Cit. 13). We have already incorporated a stronger (and semantically better established) version of this requirement into the set of requirements at initial pseudostates (Eq. A.2).

**A.2.7 States**

The UML standard (15.3.11) specifies that a state is not allowed to have both a submachine and regions (Cit. 16). Unfortunately the OCL expression in the standard is incomplete since it specifies only that composite states may not have submachines, the opposite logical direction is not included in the formula (on the other hand the definition of composite states is syntactically incorrect itself). This issue is fixed in our specification.

\[ \forall s \in \text{State}: (s.\text{region} \neq \emptyset \rightarrow s.\text{submachine} = \emptyset) \land (s.\text{submachine} \neq \emptyset \rightarrow s.\text{region} = \emptyset) \]  
\[ (A.12) \]
A.2. WELL-FORMEDNESS OF PRECISE STATECHARTS

A.2.8 State Machines

The UML standard (15.3.12) specifies that (i) connection points of a state machine are pseudostates of kind entry point or exit point (Cit. [17]).

The standard does not restrict the number of regions refining the root state machine conceptually enabling the modeler to create transitions between states residing in different regions of the root state machine: the behavior to be exposed by the state machine when taking this transition is unclear since the transition has no LCA state (the root state machine is not nested by a submachine state) therefore there is no state to be exited or entered. (Anyway the problem is of minor importance since according to our knowledge no UML modeling environments support drawing state machines with multiple regions.) For simplicity reasons we restrict below the root state machine (ii) to have exactly a single region:

\[
\forall sm \in \text{StateMachine} : \forall c \in sm\text{.connectionPoint} : c \in (\text{Pseudostate}_{\text{En}} \cup \text{Pseudostate}_{\text{Ex}})
\]

\[|sm_0\text{.region}| = 1\]

A.2.9 Transitions

The UML standard (15.3.14) specifies that (i) fork segments must not have guards or triggers, (ii) join segments must not have guards or triggers, (iii) a fork segment must target a state, (iv) a join segment must originate in a state, (v) transitions outgoing pseudostates (except for the junction, join and initial pseudostates) may not have a trigger and (vi) an initial transition at the topmost region of the state machine either has no trigger or it has a trigger with the stereotype “create” (Cit. [13]).

Requirements (i) - (iv) have been incorporated into our requirements specified in the context of fork and join pseudostates. Requirements (v) and (vi) are compatible with our previously specified constraints, the only difference is that we find the semantic meaning of triggers associated to initial transitions weak, our restrictions related to initial pseudostates completely disallow the usage of triggers in case of initial transitions.

Another aspect of transition well-formedness is related to the value of the kind attribute. The standard (15.3.15) has somewhat modified the representation of internal transitions in the metamodel and introduced the notion of local transitions since the previous UML version (1.5). The informal semantics of various transition kinds is introduced as:

- Kind internal implies that the transition, if triggered, occur without exiting or entering the source state. Thus, it does not cause a state change. This means that the entry or exit condition of the source state will not be invoked. An internal transition can be taken even if the state machine is in one or more regions nested within this state.

- Kind local implies that the transition, if triggered, will not exit the composite (source) state, but it will apply to any state within the composite state, and these will be exited and entered.

- Kind external implies that the transition, if triggered, will exit the composite (source) state.

However the discussion above clarifies somewhat the meaning of transition kinds but no formal well-formedness rules are specified in the standard. (To be more precise the standard says (informally) that (i) the source state of a transition with transition kind local must be a composite state and (ii) the source state of a transition with transition kind external must be a composite state. The first requirement is incorporated in our constraints, but the second one is definitely faulty since this implies that simple states may only be sources of internal transitions).
APPENDIX A. DETAILS ON FORMAL SEMANTICS

Listing A.1: Pseudocode of the EnteringHierarchy Method

Below we formally define the well-founded requirements based on the informal semantics. The key observation is that internal and local transitions are event processing activities that specify the behavior to be exposed while remaining in the original state: an internal transition represents an event processing without any state change while a local transition behaves as a “reset signal” to the state: the active substates are exited and the composite state (the source of the local transition) goes to its initial configuration. This semantics implies that internal and local transitions are always loop transitions (Eq. A.15) i.e., their source and target state is the same. On the other hand a local transition in a simple state is equivalent to an internal transition since there are no substates to be exited.

\[ \forall t \in \text{Transition} : (t.\text{kind} = \text{internal} \lor t.\text{kind} = \text{local}) \rightarrow (t.\text{source} = t.\text{target} \land t.\text{source} \in \text{State}) \] (A.15)

A.3 Detailed Definitions

A.3.1 Entering and Leaving State Hierarchies

Definition 68 (Entering a State Hierarchy). The entering a state hierarchy function (Listing A.1) when called with the state hierarchy node shn returns a \((p, S)\) pair, where \(p\) is a PERT graph describing the entry activity structure to be performed when entering the state hierarchy shn, and \(S\) is the set of states actually entered (i.e., states of shn in a plain set).

The algorithm is recursive: first it calls itself for all substates of the top state in the hierarchy (3) collecting the set of \((p, S)\) tuples into enteringSubstateTupleSet. The PERT graphs are extracted from enteringSubstateTupleSet and unified into a single PERT graph (4); state sets are also extracted and unified into a single set (5). Then a new PERT node is constructed containing the entry activity of state shn.state (8) and a PERT graph is constructed containing this single node (9). Since the entry activity of the parent state (shn.state) is to be performed before entry activities of substates the graph to be returned is constructed by appending the graph representing the entry activities of substates after the graph containing the entry activities of shn.state (12); the state set returned contains the substates entered and shn.state (13).

Definition 69 (Entering a Set of State Hierarchies). In Listing A.2 we extend the function EnteringHierarchy to Shn sets of state hierarchies for constructing the compound activity structure to be performed when entering multiple state hierarchies in parallel e.g., when entering concurrent regions in parallel.

Definition 70 (Leaving a State Hierarchy). The leaving a state hierarchy function (Listing A.3) when called with a configuration \(C\) (set of active states) and a state hierarchy node shn returns a \((p, S)\) pair, where \(p\) is a PERT graph describing the exit activity structure to be performed when recursively
### A.3. DETAILED DEFINITIONS

#### A.3.1 Entering Hierarchy Set

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\text{EnteringHierarchySet}(\text{Shn})</td>
</tr>
<tr>
<td>2</td>
<td>\text{enteringHierarchyTupleSet} = {\text{EnteringHierarchy}(\text{shn}) \mid \text{shn} \in \text{Shn}}</td>
</tr>
<tr>
<td>3</td>
<td>\text{p}_{\text{ret}} = \text{PERTUnion}({p \mid (p, S) \in \text{enteringHierarchyTupleSet}})</td>
</tr>
<tr>
<td>4</td>
<td>\text{S}_{\text{ret}} = \bigcup{S \mid (p, S) \in \text{enteringHierarchyTupleSet}}</td>
</tr>
<tr>
<td>5</td>
<td>\text{return } (p_{\text{ret}}, S_{\text{ret}})</td>
</tr>
</tbody>
</table>

Listing A.2: Pseudocode of the \text{EnteringHierarchySet} Method

#### A.3.2 Leaving Hierarchy

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\text{LeavingHierarchy}(C, \text{shn})</td>
</tr>
<tr>
<td>2</td>
<td>\text{require } \text{shn}.\text{state} \in C</td>
</tr>
<tr>
<td>3</td>
<td>// Obtain the leaving specification of currently active substates</td>
</tr>
<tr>
<td>4</td>
<td>\text{leavingSubstateTupleSet} = {\text{LeavingHierarchy}(C, \text{actSubShn}) \mid \text{actSubShn} \in {\text{subShn} \mid \text{subShn} \in \text{shn}.\text{substates} \land \text{subShn}.\text{state} \in C}}</td>
</tr>
<tr>
<td>5</td>
<td>\text{leavingSubstatePERTGraph} = \text{PERTUnion}({p \mid (p, S) \in \text{leavingSubstateTupleSet}})</td>
</tr>
<tr>
<td>6</td>
<td>\text{leavingSubstateStateSet} = \bigcup{S \mid (p, S) \in \text{leavingSubstateTupleSet}}</td>
</tr>
<tr>
<td>7</td>
<td>// Construct the new last node and the new PERT graph</td>
</tr>
<tr>
<td>8</td>
<td>\text{newLastNode} = \text{PERTNode}(\left{\begin{array}{ll} \text{shn}.\text{state}.\text{exit} &amp; \text{if } \text{shn}.\text{state}.\text{exit} = 1 \ \emptyset &amp; \text{otherwise} \end{array}\right.)</td>
</tr>
<tr>
<td>9</td>
<td>\text{newPERTGraph} = \text{PERTGraph}({\text{newLastNode}}, \emptyset)</td>
</tr>
<tr>
<td>10</td>
<td>// Construct the tuple to be returned</td>
</tr>
<tr>
<td>11</td>
<td>\text{p}_{\text{ret}} = \text{PERTAppend}(\text{leavingSubstatePERTGraph}, \text{newPERTGraph})</td>
</tr>
<tr>
<td>12</td>
<td>\text{S}_{\text{ret}} = {s} \cup \text{leavingSubstateStateSet}</td>
</tr>
<tr>
<td>13</td>
<td>\text{return } (\text{p}<em>{\text{ret}}, \text{S}</em>{\text{ret}})</td>
</tr>
</tbody>
</table>

Listing A.3: Pseudocode of the \text{LeavingHierarchy} Method

The algorithm is recursive: first it calls itself for all active direct substates of the top state of \text{shn} (5–7) collecting the set of \((p, S)\) tuples into \text{leavingSubstateTupleSet}. The PERT graphs are extracted from \text{leavingSubstateTupleSet} and unified into a single PERT graph (8); state sets are also extracted and unified into a single set (9). Then a new PERT node is constructed containing the exit activity of the top state of \text{shn} (12) and a PERT graph continuing this single node (13). Since the exit activity of the top state of \text{shn} is to be performed after exit activities of substates, the graph to be returned is constructed by appending the newly constructed graph containing the exit activity of the top state of \text{shn} after the graph representing exit activities of substates (16); the state set returned contains the substates left and the top state of \text{shn} (17).

**Definition 71 (Leaving a Set of State Hierarchies).** In Listing \[A.4\] we extend the function \text{LeavingHierarchy} to an \text{Shn} set of state hierarchies for constructing the compound activity structure to be performed when leaving multiple hierarchies in parallel.

#### A.3.2 Effects of Transition Conglomerates

**Definition 72 (Effects of Transition Conglomerates of Classes A, B and C).** As shown below (Eq. A.16) the PERT graph describing the effect of a transition conglomerate of class A, B and C consists of a single node referencing the effect of \(t\), the single transition involved:

\[
\text{TCEffect}[\text{tc} | \text{tc} \in (\text{TC}_a \cup \text{TC}_b)_\text{in} = (s_{\text{src}}, t, s_{\text{trg}}) \in \text{TC}_c] =
\text{PERTGraph}(\{\text{PERTNode}(\left\{\begin{array}{ll} \text{tc}.\text{effect} & \text{if } |\text{tc}.\text{effect}| = 1 \\ \emptyset & \text{otherwise} \end{array}\right.)\})
\] (A.16)

**Definition 73 (Effects of Transition Conglomerates of Class D).** According to the structure of transition conglomerates of class D (Listing \[A.5\]) the PERT graph describing the effect of \(tc = (S_{\text{src}}, t_{\text{in}}, J, t, s_{\text{trg}}) \in \text{TC}_d\) consists of two parts: (i) first the effects of \(t_{\text{in}} \in T_{\text{in}}\) join segments are to
Listing A.4: Pseudocode of the LeavingHierarchySet Method

1. TCEffect(tce = (src, tin, j, t, trg) ∈ TC)
2. TinNodeSet = \{PERTNode(tin, effect) \mid tin ∈ TinNodeSet\}
3. tNode = PERTNode\(\begin{cases} \text{effect} & \text{if } [t.in, effect] = 1 \\ \emptyset & \text{otherwise} \end{cases}\)
4. precedenceSet = \{(PERTPrecedence(tNode,tinNode) \mid tinNode ∈ TinNodeSet)\}
5. return \(\langle p^{set}, S^{set} \rangle\)

Listing A.5: Effects of Transition Conglomerates (Class D)

be performed in a non-specified order (line 2), (ii) then the effect of \(t\) (line 3) is to be performed. The precedence relations are constructed in line 4, finally the PERT graph is returned.

Definition 74 (Effects of Transition Conglomerates of Class E). According to the structure of transition conglomerates of class E (Listing A.6), the PERT graph describing the effect of \(tc = (s_{src}, t, f, t_{out}, s_{trg}) \in TC\) consists of two parts: (i) first the effect of \(t\) is to be performed (line 2), (ii) then the effects of \(t_{out} \in T_{out}\) fork segments are to be performed in a non-specified order (line 3). The precedence relations are constructed in line 4, finally the PERT graph is returned.

Definition 75 (Effects of Transition Conglomerates of class F). According to the structure of transition conglomerates of class F (Listing A.7), the PERT graph describing the effect of \(tc = (s_{src}, t, s_{trg}, f, t_{out}, s_{trg}) \in TC\) consists of three parts: (i) first the effects of \(t_{in} \in T_{in}\) join segments are to be performed in a non-specified order (line 2), (ii) then the effect of \(t\) (line 3), (iii) finally the effects of \(t_{out} \in T_{out}\) fork segments are to be performed in a non-specified order (line 4). The precedence relations are constructed in line 5, finally the PERT graph is returned.

A.3.3 Derived Transition Conglomerate Classes in the Metamodel

Transition conglomerate classes TCa, TCb, etc. are represented by metaclasses TCa, TCb, etc. derived from the abstract base TC as shown in Fig. A.1. Members of tuples defining a transition conglomerate instance are represented by association roles \(t, s_{src}, s_{trg}, s_{SrcSet}, s_{TrySet}, \in\) NSet, tOutSet, j and f, e.g., for a \((s_{src}, t, s_{trg}) \in TC\), \(s_{src}\) is in \(s_{src}\), \(t\) is in \(t\) and \(s_{trg}\) is in \(s_{trg}\), for \((s_{src}, T_{in}, j, t, f, t_{out}, s_{trg}) \in TC\), \(s_{src}\) is in \(s_{src}\), \(T_{in}\) in \(t_{in}\) set, \(j\) is in \(j\), \(t\) is in \(t\), \(f\) is in \(f\), \(T_{out}\) in \(t_{out}\) set and \(s_{trg}\) is in \(s_{trg}\).

A.4 AsmL Specification of the Platform-Independent Semantics

This chapter presents the specifications of algorithms and some related data structures introduced by the operational semantics. We will use the Microsoft AsmL executable specification language for easy to understand and unambiguous notation.

A.4.1 Base Data Types

The phase of operation of the KTS is implemented by the \{\text{Phs}_a, \text{Phs}_b, \text{Phs}_g, \text{Phs}_w\} set. This set is mapped to the AsmL enumeration KTSPhase (Listing A.8).

States of the Kripke transition system are unstructured entities and labels are assigned to them by the \(L^S\) composite labeling function (decomposed to \(L^S_{Chf}, L^S_{Eval}\) and \(L^S_{Phs}\)). In the spirit of object-oriented software design, the states of the Kripke transition system are represented by the class
Below we present the AsmL specification of functions introduced in a pseudocode notation in the main part of the document coupled with the analysis of their algorithmic complexity. All functions presented below are member methods of the KTS class: Enabled (Listing A.12), NonOverpowered (Listing A.13), ConflictFree (Listing A.14), Fireable (Listing A.15), LeavingHierarchy (Listing A.16), LeavingHierarchySet (Listing A.17), EnteringHierarchy (Listing A.18), EnteringHierarchySet (Listing A.19), LeavingSource (Listing A.20), PerformingEffect (Listing A.21), EnteringTarget (Listing A.22), FireSingle (Listing A.23), Fire (Listing A.24), Init (Listing A.25), InitializationStep (Listing A.26), TriggerProcessingStep (Listing A.27) and Operation (Listing A.28).

### A.4.2 Algorithms

The analysis of algorithmic complexity of function Enabled (Listing A.12) is as follows: (i) the calculation of TC\text{Cact} (17) requires the traversal of all transition conglomerates ($O(N_{TC})$) and checking
APPENDIX A. DETAILS ON FORMAL SEMANTICS

Listing A.8: Phase of Operation (PIM)

```java
public enum KTSPhase
// Alpha (pre-initialization)
PhsAlpha
// Unstable (during a run-to-completion step)
PhsUnst
// Stable (idle between two run-to-completion steps)
PhsStbl
// Omega (post-termination)
PhsOmega
```

Listing A.9: State of the Kripke Transition System (PIM)

```java
public class KTSState
// A state of the Kripke transition system

// Configuration
public primitive var LConf as Set of State

// Evaluation of variables
public primitive var LEval as Evaluation

// Phase of operation
public primitive var LPhs as KTSPhase

// Constructors
public KTSState()
LConf = {} as Set of State
LEval = new Evaluation()
LPhs = PhsAlpha

public KTSState(aKTSState as KTSState)
LConf = {s | s in aKTSState.LConf}
LEval = new Evaluation(aKTSState.LEval)
LPhs = aKTSState.LPhs
```
Listing A.10: Evaluation of Variables (PIM)

```java
public class Evaluation
{

    public Evaluation()
    {
    }

    public Evaluation(aEvaluation as Evaluation)
    {
    }

    // Evaluation of a constraint
    // aConstraint: The constraint to be evaluated.
    // Interactively evaluates a constraint (using standard I/O) and returns the value.
    public Models(aConstraint as Constraint?) as Boolean
    {
        if aConstraint = null then
            WriteLine("eval |= c0")
            return true
        else
            WriteLine("eval |= (" + aConstraint.specification.body + ") [y/n]")
            return ("y" = ReadLine())
    }

    // Issue a compound activity structure specified by a PERT graph
    // aPERTGraph: The PERT graph of activities to be executed.
    public Issue(aPERTGraph as PERTGraph) as Evaluation
    {
        return me
    }
}
```

Listing A.11: Key Data Members of the KTS class (PIM)

```java
public class KTS
{

    private const actState as KTSState

    public KTS()
    {
        actState = new KTSState()
    }
```

whether the subset relation of enabling states and the actual configuration ($O(N^2_{state})$); (ii) the calculation of $TCt$ (20) requires the traversal of $TCact$ and checking the triggers ($O(N_{TC})$) and (iii) the calculation of $TCen$ requires the traversal of $TCt$ and evaluating the guard $[^1] (O(N_{TC}))$ therefore the algorithmic complexity of $Enabled$ is $O(N_{TC}N^2_{state})$ (Eq. A.17).

The algorithmic complexity of $TCAllHigherPriority$ is high but can be easily pre-calculated and stored in a straightforward format (e.g., our prototype implementation uses hash tables for pre-calculating return values of such functions), this way we will consider the runtime of $TCAllHigherPriority$ to be constant (i.e., $O(1)$), only the size of the set possibly returned by it will be taken into consideration ($O(N_{TC})$).

The analysis of algorithmic complexity of function $NonOverpowered$ (Listing A.13) is as follows: for any $tc$ transition conglomerates there may be no more than $N_{TC}$ transition conglomerates of higher priority and the number of transition conglomerates in $TCen$ may be not more that $N_{TC}$ the algorithmic complexity of $NonOverpowered$ is $O(N^3_{TC})$ (Eq. A.18).

Similarly to $TCAllHigherPriority$ above, the algorithmic complexity of $TCAllInConflict$ is high but can be easily pre-calculated and stored as discussed above, this way we will consider its the runtime to be constant (i.e., $O(1)$), only the size of the set possibly returned by it will be taken into consideration ($O(N_{TC})$).

Since the number of $(tc1 \; tc2)$ pairs is $O(N_{TC}N_{TC})$ and the size of $TCAllInConflict$ for any transition conglomerate is $O(N_{TC})$ the algorithmic complexity of $ConflictFree$ (Listing A.14) is $O(N^3_{TC})$ (Eq. A.19).

The algorithmic complexity of $Fireable$ (Listing A.15) is the sum of complexities of $Enabled$, $NonOverpowered$ and $ConflictFree$ (Eq. A.20).

$$
\begin{align*}
\text{Enabled} & : O(N_{TC}N^2_{state} + N_{TC} + N_{TC}) = O(N_{TC}N^2_{state}) & \text{(A.17)} \\
\text{NonOverpowered} & : O(N_{TC}N_{TC}N_{TC}) = O(N^3_{TC}) & \text{(A.18)} \\
\text{ConflictFree} & : O(N_{TC}N_{TC}N_{TC}) = O(N^3_{TC}) & \text{(A.19)} \\
\text{Fireable} & : O(N_{TC}N^2_{state} + N^3_{TC} + N^3_{TC}) = O(N_{TC}N^2_{state} + N^3_{TC}) & \text{(A.20)}
\end{align*}
$$

Firing Transition Conglomerates

When estimating the algorithmic complexity of the recursive $LeavingHierarchy$ (Listing A.16) function, the following observation should be taken into consideration: the most complex work to be

\[^1\] The evaluation of a guard is considered to be an operation of constant ($O(1)$) execution time.
1 // /////////////////////////////////////////////////////////
2 // Enabled transition conglomerates
3 //
4 // C: Configuration (i.e., set of active states).
5 // eval: Current evaluation of variables.
6 // t: Name of the trigger received (may be null representing the "empty trigger").
7 //
8 // Returns: enabled transition conglomerates, i.e., ones (i) their enabling states are
9 // active in the current configuration (C), (ii) are triggered by the actually received
10 // trigger (t, if any) and (iii) their guard predicates evaluate to true in the current
11 // evaluation of extended variables (eval).
12 // /////////////////////////////////////////////////////////
13 private Enabled(C as Set of State, eval as Evaluation, t as String?) as Set of TC
14 // Active transition conglomerates (i.e., ones whose enabling states are active)
15 TCact as Set of TC = { tc | tc in TCSet where TCEnabling(tc) <= C}
16 // Triggered transition conglomerates (i.e., ones in TCact whose are triggered by t)
17 TCt as Set of TC = { tc | tc in TCact where TCTrigger(tc) = t}
18 // Enabled transition conglomerates (i.e., ones in TCt whose guard evaluates to true)
19 TCen as Set of TC = { tc | tc in TCt where eval.Models(TCGuard(tc))}
20 return TCen

Listing A.12: AsmL Specification of the Enabled Function (PIM)

1 // /////////////////////////////////////////////////////////
2 // Non-overpowered transition conglomerates
3 //
4 // TCen: Enabled transition conglomerates.
5 //
6 // Returns: set of transition conglomerates in TCen that are not not overpowered by
7 // any transition conglomerates of higher priority.
8 // /////////////////////////////////////////////////////////
9 private NonOverpowered(TCen as Set of TC) as Set of TC
10 // Non-overpowered transition conglomerates
11 TCnop as Set of TC =
12 { tc | tc in TCen where not (exists tch in TCen where tch in TCAllHigherPriority(tc))}
13 return TCnop

Listing A.13: AsmL Specification of the NonOverpowered Function (PIM)
APPENDIX A. DETAILS ON FORMAL SEMANTICS

Listing A.14: AsmL Specification of the ConflictFree Function (PIM)

Listing A.15: AsmL Specification of the Fireable Function (PIM)
Private LeavingHierarchy(C as Set of State, shn as StateHierarchyNode) as

(PERTGraph, Set of State)

Require shn.state in C

// Obtain the leaving specification of currently active substates
leavingSubstateTupleSet as Set of (PERTGraph, Set of State) =

(LeavingHierarchy(C, actSubShn) |
actSubShn in {subShn |
subShn in shn.substates where subShn.state in C})

leavingSubstatePERTGraph as PERTGraph =

PERTUnion({p | (p, S) in leavingSubstateTupleSet})

leavingSubstateStateSet as Set of State =

BigUnion({S | (p, S) in leavingSubstateTupleSet})

// Construct the new PERT graph node
comment as String = "Exit activity of " + shn.state.name
activity as Activity? =

if shn.state.exit.size = 0 then null else (any a | a in shn.state.exit)

newLastNode as PERTNode = new PERTNode(comment, activity)

// Construct the new PERT graph
newPERTGraph as PERTGraph = new PERTGraph({newLastNode}, ()

// Construct the tuple to be returned
pRet as PERTGraph = PERTAppend(leavingSubstatePERTGraph, newPERTGraph)

SLRet as Set of State = {shn.state} union leavingSubstateStateSet

return (pRet, SLRet)
performed by the algorithm occurs when the entire state hierarchy is to be left and the algorithm never traverses a state in the hierarchy more than once; therefore (i) not more than \( N_{\text{State}} \) PERT graphs are to be unified at \((22-23)\) and these PERT graphs have no more than \( N_{\text{State}} \) nodes and not more than \( N_{\text{State}}^2 \) edges altogether, (ii) we do not leave more than \( N_{\text{State}} \) states \((24-25)\); (iii) obviously the finally constructed PERT graph \( pRef \) does not contain more than \( N_{\text{State}} \) nodes and not more than \( N_{\text{State}}^2 \) edges and (iv) the finally constructed set \( SLRet \) does not contain more states that \( N_{\text{State}} \) therefore the algorithmic complexity of the entire recursive traversal for the entire state hierarchy is \( O(N_{\text{State}}^2) \) \((\text{Eq. A.21})\).

Obviously for general cases the algorithmic complexity of \( \text{LeavingHierarchySet} \) (Listing A.17) is \( |Shn| O(N_{\text{State}}^2) + O(N_{\text{State}}^2 + N_{\text{State}}) = |Shn| O(N_{\text{State}}^2) \) but we know that this algorithm will only be called for non-overlapping state hierarchies thus the entire operation is not more complex than calling \( \text{LeavingHierarchy} \) for the entire state hierarchy of the statechart \((\text{Eq. A.22})\).

The observations to be considered in case of \( \text{EnteringHierarchy} \) (Listing A.18) are similar to the ones at \( \text{LeavingHierarchy} \): the most complex work to be performed by the algorithm occurs when the entire state hierarchy is to be entered and the algorithm never traverses a state in the hierarchy more than once; therefore (i) not more than \( N_{\text{State}} \) PERT graphs are to be unified at \((17-18)\) and these PERT graphs have no more than \( N_{\text{State}} \) nodes and not more than \( N_{\text{State}}^2 \) edges altogether, (ii) we do not enter more than \( N_{\text{State}} \) states \((19-20)\); (iii) obviously the finally constructed PERT graph \( pRef \) does not contain more than \( N_{\text{State}} \) nodes and not more than \( N_{\text{State}}^2 \) edges and (iv) the finally constructed set \( SLRet \) does not contain more states that \( N_{\text{State}} \) therefore the algorithmic complexity of the entire recursive traversal for the entire state hierarchy is \( O(N_{\text{State}}^2) \) \((\text{Eq. A.23})\).

The observations to be considered in case of \( \text{EnteringHierarchySet} \) (Listing A.19) are similar to the ones at \( \text{LeavingHierarchySet} \): for general cases the algorithmic complexity of \( \text{EnteringHierarchySet} \) is \( |Shn| O(N_{\text{State}}^2) + O(N_{\text{State}}^2 + N_{\text{State}}) = |Shn| O(N_{\text{State}}^2) \) but we know that this algorithm will only be called for non-overlapping state hierarchies thus the entire operation is not more complex than calling \( \text{EnteringHierarchy} \) for the entire state hierarchy of the statechart \((\text{Eq. A.24})\).

Firing a transition conglomerate consists of leaving its source states, performing its effect and entering the target states; these steps were presented as a single function in the main part of the document, below we will divide the operation to subroutines \( \text{LeavingSource} \), \( \text{PerformingEffect} \) and \( \text{EnteringTarget} \) for easier analysis.

Since the calculation of \( \text{TCPossiblyLeftHierarchySet} \) is not more complex than the traversal of the entire state hierarchy, its algorithmic complexity will estimated as \( O(N_{\text{State}}) \) therefore the
private EnteringHierarchy(shn as StateHierarchyNode) as (PERTGraph, Set of State)

enteringSubstateTupleSet as Set of (PERTGraph, Set of State) =
(EnteringHierarchy(s) | s in shn.substates)
enteringSubstatePERTGraph as PERTGraph =
PERTUnion({p | (p, S) in enteringSubstateTupleSet})
enteringSubstateStateSet as Set of State =
BigUnion({S | (p, S) in enteringSubstateTupleSet})

newFirstNode as PERTNode = new PERTNode(comment, activity)

newPERTGraph as PERTGraph =
PERTAppend(newPERTGraph, enteringSubstatePERTGraph)
SERet as Set of State =
BigUnion({S | (p, S) in enteringSubstateTupleSet})

return (pRet, SERet)

private EnteringHierarchySet(Shn as Set of StateHierarchyNode) as (PERTGraph, Set of State)

enteringHierarchyTupleSet as Set of (PERTGraph, Set of State) =
(EnteringHierarchy(shn) | shn in Shn)
pRet as PERTGraph =
PERTUnion({p | (p, S) in enteringHierarchyTupleSet})
SERet as Set of State =
BigUnion({S | (p, S) in enteringHierarchyTupleSet})

return (pRet, SERet)
Leaving the source of a transition conglomerate

C: Configuration (i.e., set of active states).
tc: The transition conglomerate whose source is to be left.

Returns: a pair (pertGraph, leftStates) where pertGraph is a PERT graph representing
the state exit activities (and their precedence relations) to be performed while
recursively leaving the source of the transition conglomerate and leftStates is the
set of states left.

private LeavingSource(C as Set of State, tc as TC) as (PERTGraph, Set of State)
return LeavingHierarchySet(C, TCFossiblyLeftStateHierarchySet(tc))

Effect associated to a transition conglomerate

tc: The transition conglomerate.

Returns: the PERT graph that represents the effect associated to the transition
conglomerate.

private PerformingEffect(tc as TC) as PERTGraph
return TCEffect(tc)

complexity of LeavingSource (Listing A.20) is dominated by LeavingHierarchySet (Eq. A.25).
The function PerformingEffect (Listing A.21) barely forwards the PERT graph returned by
TCEffect and we will consider this operation to be of constant time (if the calculation time of the
effect PERT graphs is non-negligible, they can be simply pre-calculated as discussed above with respect
to TCAllInConflict and TCAllHigherPriority) as shown in Eq. A.26.

Since the calculation of TCEnteredHierarchySet is not more complex than the traversal of the
entire state hierarchy, its algorithmic complexity will estimated as \(O(N_{\text{State}})\) therefore the complexity
of EnteringTarget (Listing A.22) is dominated by EnteringHierarchySet (Eq. A.27).

The algorithmic complexity of FireSingle (Listing A.23) is the sum of complexities of
LeavingSource, PerformingEffect, EnteringTarget and appending the PERT graphs (Eq. A.28).

The function Fire (Listing A.24) (i) calls FireSingle for at most \(N_{\text{TC}}\) transition conglomerates,
(ii) unifies PERT graphs of aggregated size not more than \(N_{\text{State}}\) nodes and \(N_{\text{State}}^2\) edges, and (iii, iv)
unifies not greater than \(N_{\text{State}}\) sets, therefore its algorithmic complexity is \(O(N_{\text{TC}} N_{\text{State}}^2)\) (Eq. A.29).

LeavingHierarchy : \(O(N_{\text{State}} N_{\text{State}}^2 + N_{\text{State}} + N_{\text{State}}^2 + N_{\text{State}}) = O(N_{\text{State}}^2)\) (A.21)
LeavingHierarchySet : \(O(N_{\text{State}}^2)\) (A.22)
EnteringHierarchy : \(O(N_{\text{State}} + N_{\text{State}} + N_{\text{State}}^2) = O(N_{\text{State}}^2)\) (A.23)
EnteringHierarchySet : \(O(N_{\text{State}}^2)\) (A.24)
LeavingSource : \(O(N_{\text{State}}^2)\) (A.25)
PerformingEffect : \(O(1)\) (A.26)
EnteringTarget : \(O(N_{\text{State}}^2)\) (A.27)
FireSingle : \(O(N_{\text{State}}^2 + 1 + N_{\text{State}}^2 + N_{\text{State}}^2 + N_{\text{State}}^2) = O(N_{\text{State}}^2)\) (A.28)
Fire : \(O(N_{\text{TC}} N_{\text{State}}^2 + N_{\text{State}}^2 + N_{\text{State}} + N_{\text{State}}) = O(N_{\text{TC}} N_{\text{State}})\) (A.29)
A.4. ASML SPECIFICATION OF THE PLATFORM-INDEPENDENT SEMANTICS

Listing A.22: AsmL Specification of the *EnteringTarget* Function (PIM)

```plaintext
private EnteringTarget(tc as TC) as (PERTGraph, Set of State)
return EnteringHierarchySet(TCEnteredStateHierarchySet(tc))
```

Listing A.23: AsmL Specification of the *FireSingle* Function (PIM)

```plaintext
private FireSingle(C as Set of State, tc as TC) as
(PERTGraph, Set of State, Set of State)
(pLeave as PERTGraph, SLeave as Set of State) = LeavingSource(C, tc)
pEffect as PERTGraph = PerformingEffect(tc)
(pEnter as PERTGraph, SEnter as Set of State) = EnteringTarget(tc)
pFull as PERTGraph = PERTAppend(PERTAppend(pLeave, pEffect), pEnter)
return (pFull, SLeave, SEnter)
```
APPENDIX A. DETAILS ON FORMAL SEMANTICS

Listing A.24: AsmL Specification of the Fire Function (PIM)

Listing A.25: AsmL Specification of the Init Function (PIM)

Initialization

Since the calculation of InitialHierarchy() is not more complex than the traversal of the entire state hierarchy, its algorithmic complexity will estimated as $O(N_{State})$ therefore the complexity of Init (Listing A.25) is dominated by EnteringHierarchy (Eq. A.30).

$$\text{Init} : O(N_{State}^2) \quad (A.30)$$

High-Level Algorithms

For the analysis of algorithmic complexity with respect to high-level functions InitializationStep (Listing A.26) and TriggerProcessingStep (Listing A.27) we have to discuss the complexities of $\text{init}$, $\text{open}$ etc. steps. Issuing a PERT graph will be considered to be an operation of constant complexity ($O(1)$). The code sections implementing the six step kinds are marked by comments above: (i) $\text{init}$ (lines 10–14 in Listing A.26) involves a call for Init, two assignments and issuing the activity PERT graph (Eq. A.31); (ii) $\text{open}$ (lines 16–20 in Listing A.27) involves a call to Fire, a set minus and a set union operations, issuing the PERT graph, and an assignment (Eq. A.32); (iii) $\text{inter}$ (lines 28–31 in Listing A.26 and lines 34–37 in Listing A.27) involves a call to Fire, a set minus and a set union operations and issuing the PERT graph (Eq. A.33); (iv) $\text{close}$ (lines 25–26 in Listing A.26 and lines 31–32
private InitializationStep()
  require actState.LPhs = PhsAlpha

  // = [init] = >
  (pInit as PERTGraph, SInitL as Set of State) = Init()
  actState.LConf := SInitL
  actState.LEval := actState.LEval.Issue(pInit)
  actState.LPhs := PhsUnst

  step while actState.LPhs = PhsUnst
    if (exists st in TerminationStateSet where st in actState.LConf) then
      // = [term] = >
      actState.LPhs := PhsOmega
    else
      TCf as Set of TC = Fireable(actState.LConf, actState.LEval, null)
      if TCf.Size = 0 then
        // = [close] = >
        actState.LPhs := PhsStbl
      else
        // = [inter] = >
        (pInter, SInterL, SInterE) = Fire(actState.LConf, TCf)
        actState.LConf := (actState.LConf - SInterL) union SInterE
        actState.LEval := actState.LEval.Issue(pInter)

Listing A.26: AsmL Specification of the InitializationStep Function (PIM)

in Listing A.27 involves a single assignment (Eq. A.34); (v) term (lines 18–19 in Listing A.26) and lines 24–25 in Listing A.27 involves a single assignment (Eq. A.35) and (vi) drop (line 14 in Listing A.27) is an empty operation (Eq. A.36).

\[
\begin{align*}
\text{init} & : \mathcal{O}(N_{\text{State}}^2 + 1 + 1 + 1) = \mathcal{O}(N_{\text{State}}^2) \\
\text{open} & : \mathcal{O}(N_{\text{TC}} N_{\text{State}}^2 + 2 N_{\text{State}}^2 + 1 + 1) = \mathcal{O}(N_{\text{TC}} N_{\text{State}}^2) \\
\text{inter} & : \mathcal{O}(N_{\text{TC}} N_{\text{State}}^2 + 2 N_{\text{State}}^2 + 1 + 1) = \mathcal{O}(N_{\text{TC}} N_{\text{State}}^2) \\
\text{close} & : \mathcal{O}(1) \\
\text{term} & : \mathcal{O}(1) \\
\text{drop} & : \mathcal{O}(1)
\end{align*}
\]


The discussion above does not take into consideration the guard predicates shown in the high-level statechart view of the Kripke transition system (Fig. 1.21). Below we will present the algorithmic complexities of calculations related to transitions between various phases of the Kripke transition system in the form Phs_{src} \xrightarrow{[\text{Guard}]/\text{Step}} Phs_{tgr}:

- The transition between Phs_{src} and Phs_{U} (initialization step by \text{init} as defined in Eq. 1.119) has no guard, therefore the algorithmic complexity of the transition is determined by \text{init} (Eq. A.37).

- The transition between Phs_{U} and Phs_{U} (internal step by \text{inter} as defined in Eq. 1.121) is guarded by ((\text{TerminationState} \cap L_{\text{Crit}}(s_u)) = \emptyset) \wedge (\text{Fireable}(L_{\text{Crit}}(s_u), L_{\text{Eval}}(s_u), t_0) \neq \emptyset) therefore the algorithmic complexity of the transition is the sum of complexities of (i) the set intersection operation, (ii) the \text{Fireable} function and (iii) the \text{inter} step (Eq. A.38).

- The transition between Phs_{U} and Phs_{S} (closing an RTC step by \text{close} as defined in Eq. 1.122) is
private TriggerProcessingStep(t as String)

require (actState.LPhs = PhsStbl) or (actState.LPhs = PhsOmega)
if actState.LPhs = PhsStbl then
   if T Cf . Size = 0 then
      // = [drop] =>
   else
      // = [open] =>
      (pOpen, SOpenL, SOpenE) = Fire(actState.LConf, TCf)
      actState.LConf := (actState.LConf - SOpenL) union SOpenE
      actState.LEval := actState.LEval.Issue(pOpen)
      actState.LPhs := PhsUnst
   end
step while actState.LPhs = PhsUnst
   if (exists st in TerminationStateSet where st in actState.LConf) then
      // = [term] =>
      actState.LPhs := PhsOmega
   else
      TCf := Fireable(actState.LConf, actState.LEval, null)
      if TCf.Size = 0 then
         // = [close] =>
      else
         // = [inter] =>
         (pInter, SInterL, SInterE) = Fire(actState.LConf, TCf)
         actState.LConf := (actState.LConf - SInterL) union SInterE
         actState.LEval := actState.LEval.Issue(pInter)
guarded by \((\text{TerminationState} \cap L_{\text{Cnf}}^S(s_u)) = \emptyset) \land (\emptyset = \text{Fireable}(L_{\text{Cnf}}^S(s_u), L_{\text{Eval}}^S(s_u), t_\emptyset))\) therefore the algorithmic complexity of the transition is the sum of complexities of (i) the set intersection operation, (ii) the \text{Fireable} function and (iii) the close step (Eq. A.39).

- The transition between \text{Phs}_{\text{S}} and \text{Phs}_{\text{S}} (dropping a trigger by \text{drop} as defined in Eq. A.12) is guarded by \((\emptyset = \text{Fireable}(L_{\text{Cnf}}^S(s_u), L_{\text{Eval}}^S(s_u), t))\) therefore the algorithmic complexity of the transition is the sum of complexities of (i) the \text{Fireable} function and (ii) the \text{drop} step (Eq. A.40).

- The transition between \text{Phs}_{\text{U}} and \text{Phs}_{\omega} (termination step by \text{term} as defined in Eq. A.13) is guarded by \(((\text{TerminationState} \cap L_{\text{Cnf}}^S(s_u)) \neq \emptyset)\) therefore the algorithmic complexity of the transition is the sum of complexities of (i) the set intersection operation and (ii) the \text{term} step (Eq. A.41).

- The transition between \text{Phs}_{\text{S}} and \text{Phs}_{\text{U}} (opening an RTC step by \text{open} as defined in Eq. A.14) is guarded by \((\emptyset \neq \text{Fireable}(L_{\text{Cnf}}^S(s_u), L_{\text{Eval}}^S(s_u), t))\) therefore the algorithmic complexity of the transition is the sum of complexities of (i) the \text{Fireable} function and (ii) the \text{open} step (Eq. A.42).

\[
\begin{align*}
\text{Phs}_{\text{S}} \xrightarrow{\text{int}} \text{Phs}_{\text{U}} & : O(N_{\text{State}}^2) \quad \text{(A.37)} \\
\text{Phs}_{\text{U}} \xrightarrow{\left(\text{TerminationState} \cap L_{\text{Cnf}}^S(s_u) = \emptyset \right) \land \left(\text{Fireable}(L_{\text{Cnf}}^S(s_u), L_{\text{Eval}}^S(s_u), t_\emptyset) \neq \emptyset\right)} \xrightarrow{\text{inter}} \text{Phs}_{\text{S}} & : O(N_{\text{State}}^2 + (N_{\text{TC}} N_{\text{State}}^2 + N_{\text{TC}}^3) + N_{\text{TC}} N_{\text{State}}^3) = O(N_{\text{TC}} N_{\text{State}}^2 + N_{\text{TC}}^3) \quad \text{(A.38)} \\
\text{Phs}_{\text{U}} \xrightarrow{\left(\text{TerminationState} \cap L_{\text{Cnf}}^S(s_u) = \emptyset \right) \land \left(\emptyset = \text{Fireable}(L_{\text{Cnf}}^S(s_u), L_{\text{Eval}}^S(s_u), t_\emptyset)\right)} \xrightarrow{\text{open}} \text{Phs}_{\text{S}} & : O(N_{\text{State}}^2 + (N_{\text{TC}} N_{\text{State}}^2 + N_{\text{TC}}^3) + 1) = O(N_{\text{TC}} N_{\text{State}}^2 + N_{\text{TC}}^3) \quad \text{(A.39)} \\
\text{Phs}_{\text{S}} \xrightarrow{\left(\emptyset = \text{Fireable}(L_{\text{Cnf}}^S(s_u), L_{\text{Eval}}^S(s_u), t)\right)} \xrightarrow{\text{drop}} \text{Phs}_{\text{S}} & : O((N_{\text{TC}} N_{\text{State}}^2 + N_{\text{TC}}^3) + 1) = O(N_{\text{TC}} N_{\text{State}}^2 + N_{\text{TC}}^3) \quad \text{(A.40)} \\
\text{Phs}_{\text{U}} \xrightarrow{\left(\emptyset \neq \text{Fireable}(L_{\text{Cnf}}^S(s_u), L_{\text{Eval}}^S(s_u), t)\right)} \xrightarrow{\text{open}} \text{Phs}_{\text{U}} & : O(N_{\text{State}}^2 + N_{\text{TC}}) + N_{\text{TC}} N_{\text{State}}^2) = O(N_{\text{TC}} N_{\text{State}}^2 + N_{\text{TC}}^3) \quad \text{(A.41)} \\
\end{align*}
\]

Finally we can present the algorithmic complexity of high-level functions \text{InitializationStep} (Listing A.26) and \text{TriggerProcessingStep} (Listing A.27). Since both functions may involve any number of internal steps (\text{inter}) we have to assume that there is a \(k\) constant that there may be no more than \(k\) internal steps during initialization and processing of triggers (this is a rational assumption if the statechart may not fall into infinite loops). This way (i) the algorithmic complexity of \text{InitializationStep} is the sum of complexities of the initialization, at most \(k\) internal steps, closing the RTC step and (possibly) a termination step (Eq. A.43): (ii) the algorithmic complexity of \text{TriggerProcessingStep} is the sum of complexities of (possibly) dropping the trigger, (or otherwise) opening an RTC step, at most \(k\) internal steps, closing the RTC step and (possibly) a termination step (Eq. A.44).

\[
\begin{align*}
\text{InitializationStep} & : O(N_{\text{State}}^2 + k (N_{\text{TC}} N_{\text{State}}^2 + N_{\text{TC}}^3) + N_{\text{TC}} N_{\text{State}}^2 + N_{\text{TC}}^3 + N_{\text{State}}^2) = \\
& = O(N_{\text{State}}^2 + N_{\text{TC}} N_{\text{State}}^2 + N_{\text{TC}}^3) \quad \text{(A.43)} \\
\text{TriggerProcessingStep} & : O(N_{\text{State}}^2 + N_{\text{TC}} + N_{\text{TC}} N_{\text{State}}^2 + N_{\text{TC}}^3 + k (N_{\text{TC}} N_{\text{State}}^2 + N_{\text{TC}}^3) + \\
& = O(N_{\text{State}}^2 + N_{\text{TC}} N_{\text{State}}^2 + N_{\text{TC}}^3) \quad \text{(A.44)}
\end{align*}
\]
### Appendix A. Details on Formal Semantics

Finally the AsmL specification of the function encapsulating the entire operation is presented in Listing A.28.

#### A.5 Detailed Proofs

##### A.5.1 Refinement Formalism

**Theorem 4.** If regions \( r_1, r_2 \mid r_1 \neq r_2 \in \text{Region} \) are not in a refinement relation \( (r_1 \notin \text{AllSubreg}^T(r_2) \land r_2 \notin \text{AllSubreg}^T(r_1)) \) the set of states directly or transitively contained by them is disjoint: \( \text{AllCont}^T(r_1) \cap \text{AllCont}^T(r_2) = \emptyset \).

**Proof.** Let us assume that there exist \( r_1, r_2 \mid r_1 \neq r_2 \in \text{Region} \) that are not in a refinement relation but the set of states contained them is not disjoint:

\[
\exists r_1, r_2 \mid r_1 \neq r_2 \in \text{Region}, s \in \text{State} : \quad r_1 \notin \text{AllSubreg}^T(r_2) \land r_2 \notin \text{AllSubreg}^T(r_1) \land s \in \text{AllCont}^T(r_1) \cap \text{AllCont}^T(r_2) \quad (A.45)
\]

This way (i) there are two regions \( r_1' \in \{ r_1 \} \cup \text{AllSubreg}^T(r_1) \) and \( r_2' \in \{ r_2 \} \cup \text{AllSubreg}^T(r_2) \) that \( \exists s \in \text{State} : s \in \text{AllCont}^D(r_1') \land s \in \text{AllCont}^D(r_2') \) (Fig. A.2 left side) or (ii) there are two states \( s_1 \in \text{AllSubst}^T(r_1) \) and \( s_2 \in \text{AllSubst}^T(r_2) \) that \( \exists r : (s_1, r) \in \text{Ref}^D \land (s_2, r) \in \text{Ref}^D \) this way all states contained by \( r \) are (directly or transitively) contained by both \( r_1 \) and \( r_2 \) (Fig. A.2 right side).

The first case may not happen because any state is contained by exactly one region (Eq. 1.65) the second may not be true since any region refines exactly one state (Eq. 1.66).

**Theorem 5.** If states \( s_1, s_2 \mid s_1 \neq s_2 \in \text{State} \) are not in a refinement relation, \( (s_1 \notin \text{AllSubst}^T(s_2) \land s_2 \notin \text{AllSubst}^T(s_1)) \) the set of their substates is disjoint: \( \text{AllSubst}^T(s_1) \cap \text{AllSubst}^T(s_2) = \emptyset \).

**Proof.** Let \( R_1, R_2 \) be the set of regions directly refining \( s_1 \) and \( s_2 \) respectively: \( R_1 = \text{AllRef}^D(s_1), R_2 = \text{AllRef}^D(s_2) \). Since \( s_1 \) and \( s_2 \) are not in refinement relation, any \( r_1 \in R_1, r_2 \in R_2 \) pairs of regions
are not in a refinement relation either. This way for any such \(r_1, r_2\) pair the set of states contained by them is disjoint (Thm. 4).

Since \(\text{AllSubst}_T(s_1) = \bigcup_{r_1 \in \text{AllRef}_D(s_1)} \text{AllCont}_T(r_1)\) and \(\text{AllSubst}_T(s_2) = \bigcup_{r_2 \in \text{AllRef}_D(s_2)} \text{AllCont}_T(r_2)\) it is obvious that \(\text{AllSubst}_T(s_1) \cap \text{AllSubst}_T(s_2) = \emptyset\).

**Theorem 7.**

**Proof.** Since states \(s_1, s_2\) are different and are not in a refinement relation, taking into consideration Thm. 5 the statement is obviously true.

### A.5.2 Compound Transition Structures

**Theorem 6.** For any \(S \subseteq \text{State}\) non-empty set of states \((S \neq \emptyset)\) the set returned by \(\text{BC}_S\) contains exactly one region: \(\forall S \subseteq \text{State}|S| : |\text{BC}_S(S)| = 1\). Below we prove the theorem in two steps: (i) first we prove that \(\text{BC}_S\) does not return an empty set \((\text{AS} \subseteq \text{State}|S| : |\text{BC}_S(S)| = \emptyset)\), (ii) then we show that \(\text{BC}_S\) can not return more than one elements \((\text{BC} \subseteq \text{State}|S| : |\text{BC}_S(S)| > 1)\).

**Proof.** In the first case \((\text{AS} \subseteq \text{State}|S| : |\text{BC}_S(S)| = \emptyset)\) let us indirectly assume that:

\[
\exists S \subseteq \text{State}|S| \ni \{r \in \text{Region}|S \subseteq \text{AllCont}_T(r) \land (\exists r' \in \text{AllSubreg}_T(r) : S \subseteq \text{AllCont}_T(r'))\} = 0 \quad (A.46)
\]

Let \(R' \subseteq \text{Region}\) be a set of regions that: \(R' = \{r \in \text{Region}|S \subseteq \text{AllCont}_T(r)\}\) (it is obvious that such a non-empty \(R'\) set exists since \(\text{sm}_{0, \text{region}}\) is such a region (Eq. A.13)). Furthermore it is easy to see that \(\text{BC}_S\) can return elements from \(R'\) only, this way we can re-write Eq. (A.46) as:

\[
\text{BC}_S(S) = \{r \in R'|(\exists r' \in \text{AllSubreg}_T(r) : S \subseteq \text{AllCont}_T(r'))\}
\]

If our original assumption was correct this means that:

\[
\exists S \subseteq \text{State}|S| \ni \{r \in R'|(\exists r' \in \text{AllSubreg}_T(r) : S \subseteq \text{AllCont}_T(r'))\} = 0 \quad (A.48)
\]

\[
\exists S \subseteq \text{State}|S| \ni \forall r \in R' : \exists r' \in \text{AllSubreg}_T(r) : S \subseteq \text{AllCont}_T(r') \quad (A.49)
\]

But this may not be true, since \(R'\) is a finite set and there are no cycles in the region refinement hierarchy (Eq. i.06).

In the second case \((\text{BC} \subseteq \text{State}|S| : |\text{BC}_S(S)| > 1)\) let us indirectly assume that:

\[
\exists S \subseteq \text{State}|S| \ni \forall r_1, r_2 : \{r_1, r_2\} \subseteq \{r \in \text{Region}|S \subseteq \text{AllCont}_T(r)\} \land \{\exists r' \in \text{AllSubreg}_T(r) : S \subseteq \text{AllCont}_T(r')\}
\]

Since \(r_1\) and \(r_2\) may not be in refinement relation the sets of states contained by them are disjoint (Thm. 4), this way \(S\) is an empty set, but \(\text{BC}_S\) is not applicable to empty sets.

**Theorem 8.** The transition conglomerate priority relation is transitive i.e., if for \(\{tc_1, tc_m, tc_h\} \subseteq \text{TC}\) holds that \(tc_1 < tc_m^h\) and \(tc_m^h < tc_h\) then \(tc_1 < tc_h\) also holds:

\[
\forall \{tc_1, tc_m, tc_h\} \subseteq \text{TC} : ((tc_1 < tc_m) \land (tc_m < tc_h)) \rightarrow (tc_1 < tc_h)
\]
APPENDIX A. DETAILS ON FORMAL SEMANTICS

Figure A.3: Transitive theness of the Transition Conglomerate Priority Relation

Proof. Let the enabling states of $tc^j$, $tc^m$, and $tc^h$ be $E^j$, $E^m$, and $E^h$ respectively ($TC_{Enabling}(tc^j) = E^j$, $TC_{Enabling}(tc^m) = E^m$ and $TC_{Enabling}(tc^h) = E^h$), let $s'$ and $s''$ be the state $s^h$ occurring in the second part of the definition (Def. 30) in case of $tc^j < tc^m$ (Eq. A.52) and $tc^m < tc^h$ (Eq. A.53) respectively:

$$\forall s^m \in E^m : \exists s' \in E^j : s^m \in (\{s^j\} \cup AllSubst^T(s')) \wedge \exists s' \in E^m : \exists s' \in E^j : s' \in AllSubst^T(s') \tag{A.52}$$

$$\forall s' \in E^h : \exists s^m \in E^m : s^h \in (\{s^m\} \cup AllSubst^T(s^h)) \wedge \exists s'' \in E^h : \exists s^m \in E^m : s'' \in AllSubst^T(s^m) \tag{A.53}$$

With respect to the relation of $tc^j$ and $tc^h$ it is easy to see that $s'$ or a substate of $s'$ is an enabling state of $tc^h$ thus $tc^j < tc^h$ holds. An illustrative example is shown in Fig. A.3.

Theorem 9. For any $tc \in TC$ transition conglomerate $TC_{TopSource}$ always returns a single state: \( \forall tc \in TC : |TC_{TopSource}(tc)| = 1. \)

Proof. If $tc$ belongs to the A, B, C or E transition conglomerate classes $tc \in TC_a \cup TC_b \cup TC_c \cup TC_e$ the statement is trivial since $TC_{TopSource}$ returns a single state (ancestor state or the state itself).

If $tc$ belongs to D or F classes $tc \in TC_d \cup TC_f$ we have to first show that (i) $TC_{TopSource}$ may not return more than one states and (ii) $TC_{TopSource}$ may not return the empty set.

On one hand it is easy to see that $TC_{TopSource}$ may not return more than one states of the bottom container region since both of the states should have all states in $TC_{Enabling}(tc)$ as substates (or be one of them themselves) but this is can not happen because the states in a region are not in a refinement relation, therefore the sets of their substates are disjoint (Thm. D).

On the other hand it can be shown that $TC_{TopSource}$ may not return the empty set, so let us assume that the statement is false i.e.,

$$\exists tc \in TC : |TC_{TopSource}(tc)| = 0 \tag{A.54}$$

The top container region by definition contains all the enabling states directly or transitively (Def. 23). This implies that there are some directly contained substates $S = \{s_1, s_2, \ldots, s_n\} \subseteq AllCont^D(BC^{TC}(tc))$ in the bottom container region that:

$$\bigcup_{s_i \in \{s_1, s_2, \ldots, s_n\}} (\{s_i\} \cup AllSubst^T(s_i)) \subseteq TC_{Enabling}(tc) \tag{A.55}$$

Actually we try to prove that $S$ contains a single state $s$ and $\forall s' \in TC_{Enabling}(tc) : s' \in (\{s\} \cup AllSubst^T(s))$. Since our assumption says that there exists a $tc$ that $TC_{TopSource}$ returned the empty set, this means that (i) none of the states in $s_i \in S$ meets the criterion that $\forall s' \in TC_{Enabling}(tc) : s' \in (\{s\} \cup AllSubst^T(s))$ and (ii) $S$ contains at least two states $s_i$ and $s_j$ (Eq. A.55). Obviously $s_i$ and $s_j$ are one of the states in $TC_{Enabling}(tc)$ or ancestors of a state in $TC_{Enabling}(tc)$. Since $tc$ belongs to $TC_d$ or $TC_e$ classes the enabling states are connected to a join pseudostate. Let two of the source states be $s_i' \in \{s_i\} \cup AllSubst^T(s_i)$ and $s_j' \in \{s_j\} \cup AllSubst^T(s_j)$.

The well-formedness of join pseudostates require that all transitions targeting a join vertex should
originated in states in different regions of an orthogonal state (Eq. A.5). In case of \( s_i \) and \( s_j \) this means that:

\[
\exists s \in \text{State} : \exists r_i, r_j \in \text{Region} : (s, r_i) \in \text{Ref}^D \land (s, r_j) \in \text{Ref}^D \land s'_i \in \text{AllCont}^T (r_i) \land s'_j \in \text{AllCont}^T (r_j) \quad (A.56)
\]

But this may not happen, since there exist \( s_i \) and \( s_j \) that \( s'_i \in \text{AllSubst}^T (s_i) \) and \( s'_j \in \text{AllSubst}^T (s_j) \) in the same bottom container region, the ancestor states of \( s_i \) and \( s_j \) are the same, and their descendant states are different: there may not such an \( s \) exist that meets Eq. A.56.

**Theorem 10.** For any \( tc \in \text{TC} \) transition conglomerate \( \text{TCTopTarget} \) always returns a single state: 
\[
\forall tc \in \text{TC} : |\text{TCTopTarget}(tc)| = 1.
\]

**Proof.** The proof of the statement is similar to Thm. 9 if \( tc \) belongs to the A, B, C or D transition conglomerate classes \( \text{TCTopTarget} \) trivially returns a single state. If \( tc \) belongs to classes E or F the discussion is similar to the proof above but in this case we have to take into consideration that the states determined by \( tc \) are connected by transitions to a fork pseudostate and a corresponding well-formedness rule requires that all transitions originating in a fork vertex should target different regions of an orthogonal state (Eq. A.9).

**Theorem 11.** Let \( r \in \text{Region} \) be a region and \( tc \in \text{TC} \) a transition conglomerate. The set \( S \) defined below is empty or contains a single element:

\[
S = \{ s \in \text{AllCont}^D (r) | s \in \text{TCDetermined}(tc) \lor \exists s' \in \text{AllSubst}^T (s) : s' \in \text{TCDetermined}(tc) \} \quad (A.57)
\]

**Proof.** The definition of \( S \) can be rewritten as:

\[
S = \{ s \in \text{AllCont}^D (r) | (\{s\} \cup \text{AllSubst}^T (s)) \cap \text{TCDetermined}(tc) \neq \emptyset \} \quad (A.58)
\]

The sub-expression \( (\{s\} \cup \text{AllSubst}^T (s)) \) contains all states of the hierarchy originating in \( s \). Based on Thm. 9 for any \( s \in \text{AllCont}^D (r) \) the sets \( (\{s\} \cup \text{AllSubst}^T (s)) \) are disjoint:

\[
\forall \{s_1, s_2\} \subseteq \text{AllCont}^D (r) : ((\{s_1\} \cup \text{AllSubst}^T (s_1)) \cap (\{s_2\} \cup \text{AllSubst}^T (s_2))) = \emptyset \quad (A.59)
\]

If \( tc \) belongs to classes C or D, \( |\text{TCDetermined}(tc)| = 1 \), therefore since \( (\{s\} \cup \text{AllSubst}^T (s)) \) sets are disjoint, the single state determined by \( tc \) may be in the state hierarchy originating in at most one of \( s \in \text{AllCont}^D (r) \) states.

If \( tc \) belongs to classes E or F, let us indirectly assume that \( |S| > 1 \). Let \( \{s_1, s_2\} \subseteq S \). This means that there are \( s'_1 \) and \( s'_2 \) that \( s'_i \in (\{s_1\} \cup \text{AllSubst}^T (s_1)) \) and \( s'_2 \in (\{s_2\} \cup \text{AllSubst}^T (s_2)) \). In this case (classes E or F) \( s'_1 \) and \( s'_2 \) are targeted by fork segments. But since all transitions originating in a fork vertex should target different regions of an orthogonal state (Eq. A.9) \( s_1 \) and \( s_2 \) may not be in the same region therefore our assumption was false.

**A.5.3 Correctness of Operation**

**Definition 76 (Consistent Configuration).** A configuration of a statechart is said to be consistent if it is for all active states (or root state machine) \( s \) holds that for all regions \( r \) directly refining \( s \) there is exactly one active state directly contained by \( r \). Below we recursively define the function \( \text{Consistent} : 2^{\text{State}} \times ((\{s_{\text{sm0}}\} \cup \text{State}) \rightarrow \{\top, \bot\}, \text{that for a configuration (set of active states) } C \) and a state (or root state machine) \( s \) checks whether the state hierarchy rooting in \( s \) is consistent. Obviously a configuration \( C \) is consistent if \( \text{Consistent}(C, s_{\text{sm0}}) \).

\[
\text{Consistent}(C, s \in \{s_{\text{sm0}}\} \cup \text{State}) = \forall r \in \text{AllRef}^D (s) : \exists s' \in \text{AllCont}^D (r) : s' \in C \land \text{Consistent}(C, s') \quad (A.60)
\]
Theorem 12. The initial configuration of the statechart (i.e., the states in the hierarchy returned by InitialHierarchy) is consistent.

Proof. Since the traversal of the state refinement hierarchy implemented in InitialHierarchy is rooted in the single region refining \( sm_0 \) and for all regions there is exactly a single initial state the validity of the statement is trivial.

Theorem 13. Firing a transition conglomerate in a consistent configuration will result in another consistent configuration.

Proof. Transition conglomerates of class B (internal transitions) do not modify the configuration thus the statement obviously holds. In case of transition conglomerates of class A (local transitions) all regions refining the source state (that is obviously equivalent to the target state) are reset to their initial state hierarchy thus the statement is obviously true again. For the remaining transition conglomerate classes it is easy to see that firing affects only the bottom container region \( r \): first all states in the region are left then the target configuration (rooting in a state directly contained by \( r \)) is entered. The observations to be highlighted here are as follows: (i) the hierarchy \( h \) entered is calculated by the function EnteredHierarchy\(^S\), (ii) the resulting configuration \( C' \) is calculated by first removing all states directly or transitively contained by \( r \) from \( C \) (see the definitions of FireSingle and LeavingHierarchySet) then adding all states in \( h \) and (iii) based on the definition of EnteredHierarchy\(^S\) it is easy to see that Consistent\((C', h, \text{state})\) thus (iv) Consistent\((C', sm_0)\).

Theorem 14. Firing a set of fireable transition conglomerates by Fire in a consistent configuration results in another consistent configuration.

Proof. Since there may be no conflicts amongst fireable transitions the transition conglomerates fired affect non-overlapping regions (i.e., non-overlapping subtrees in the state refinement hierarchy) therefore the observations of Thm. 13 hold even for sets of fireable transition conglomerates.

Theorem 15. The execution specified by the Kripke transition system in Sec. 1.8 establishes a consistent configuration by \( \text{init} \) and maintains the consistency of the configuration throughout the entire execution.

Proof. The statement is implied by the definition of the Kripke transition system and Thm. 12 and Thm. 14.

A.6 Implementation of Shorthand Constructs

The metamodel presented in Sec. 1.3 does not include various modeling constructs introduced by the UML 2.0 standard like join and choice pseudostates, history vertices and facilities for embedding entire state machines (i.e., connection point references and entry and exit pseudostates). In our approach these artifacts are considered to be shorthand notations that can be substituted by core modeling concepts used in the metamodel and the formal operational semantics. The set of transformation rules that convert shorthand constructs to core constructs forms a denotational semantics for these UML
2.0 constructs. Due to space restrictions we present an illustrative overview of the transformations by substituting all shorthand constructs in the statechart shown in Fig. A.4. The detailed formal (AsmL) specification can be found in [132].

Fig. A.4 presents a statechart that contains multiple non core concepts: (i) a shallow history pseudostate in the concurrent state c; (ii) a state machine named BSM is embedded in the submachine state b; (iii) there is a choice pseudostate that divides the transition originating in c into two segments and (iv) there is a terminate pseudostate that is to be replaced by a termination state as discussed in Sec. 1.3. Below we will substitute these constructs by core concepts step by step, the construct to be substituted in the subsequent step is highlighted by blue in the figures.

In the first step we will substitute the history pseudostate by multiple transitions. By definition the shallow history vertex represents the most recent active direct substate of the containing state. In case of concurrent states like c there are multiple most recent active direct substates (one from each region directly refining c). The single transition originating in the history vertex should target the default configuration to be entered if the containing state has never been active before. In case of concurrent states the default configuration contains multiple direct substates (one from each region directly refining the containing state), this way the single transition should be divided to multiple segments by a fork vertex.

It is easy to see that a transition targeting a shallow history pseudostate can be substituted by multiple transitions targeting all possible shallow configurations guarded by mutually exclusive guard predicates. Furthermore we have to introduce the most recent full configuration of a state function \( \text{MRConfFull} : \text{State} \rightarrow \text{2}^{\text{State}} \) that for a state \( s \) returns its most recent configuration (i.e., all of its substates that were active in the most recent case when \( s \) was active), or the empty set if \( s \) has never been active before. Using this function the guard predicates are easy to define: in case of a newly introduced transition representing the shallow configuration \( C_S \) the predicate is \( S_C \subseteq \text{MRConfFull}(s) \); the transition corresponding to the default configuration should by guarded by \( \emptyset = \text{MRConfFull}(s) \).
The history vertex itself is to be substituted by a junction pseudostate.

In case of the history vertex shown in the figure there are four possible shallow configurations: \{c_{11}, c_{21}\}, \{c_{11}, c_{22}\}, \{c_{12}, c_{21}\} and \{c_{12}, c_{22}\}, therefore there are four transitions to be introduced guarded by \{c_{11}, c_{21}\} \subseteq \text{MRConf}_{\text{Full}}(c), \{c_{11}, c_{22}\} \subseteq \text{MRConf}_{\text{Full}}(c), \{c_{12}, c_{21}\} \subseteq \text{MRConf}_{\text{Full}}(c) and \{c_{12}, c_{22}\} \subseteq \text{MRConf}_{\text{Full}}(c) respectively; the transition corresponding to the default configuration is guarded by \emptyset = \text{MRConf}_{\text{Full}}(c). The resulting statechart is shown in Fig. A.7.

In the second step we will substitute the submachine state b by a compound state containing the entire referenced state machine. The original (referenced) substate machine BSM is shown in a callout in Fig. A.5. The embedded state machine has an entry point pseudostate \textit{bentry} and an exit point pseudostate \textit{bexit}; these vertices are referenced by the two connection point references attached to state b called \textit{bentryref} and \textit{bexitref} respectively.

The substitution of the state machine embedding is easy: the entire state hierarchy of the referenced state machine is to be moved into the referenced state as substates, and for each connection point vertex with the corresponding connection point reference is to be replaced by a single junction pseudostate and all transition targeting the original vertices or targeting them are to be connected to the junction vertex.

In case of the state machine embedding shown in the figure the entire content of the referenced state machine BSM is moved into the compound state b, the entry point pseudostate \textit{bentry} with the referring \textit{bentryref} connection point reference was replaced by a single junction pseudostate (also called \textit{bentry} for simplicity), the exit point pseudostate \textit{bexit} with the referring \textit{bexitref} connection point reference was replaced by a single junction pseudostate (also called \textit{bexit} for simplicity). The transition originally targeting \textit{bentryref} was re-targeted to the newly introduced \textit{bentry} junction point and the transition originating in \textit{bexitref} was connected to the newly introduced junction pseudostate \textit{bexit}. The resulting statechart is shown in Fig. A.6.

In the third step we will substitute the choice pseudostate and the junction vertices introduced by
us during the previous transformation steps. Although the semantics of junction and choice vertices is slightly different according to the informal specification of the standard, we will not distinguish them (see [132] for further discussion). The substitution of junction and choice vertices is an iterative process that traverses all junction and choice vertices and introduces multiple new transitions: let \( v \) be a junction or choice pseudostate and the \( T_{in} \) be the set of transitions targeting \( v \) and \( T_{out} \) be the set of transition originating in \( v \); for each \((t_{in} \in T_{in}, t_{out} \in T_{out})\) pairs we introduce a new transition \( t \) whose source vertex is the source of \( t_{in} \) and whose target vertex is the target of \( t_{out} \). Let the guard of \( t_{in} \) be \( g_{in} \) and the guard of \( t_{out} \) be \( g_{out} \); the guard of the new transition is \( g_{out} \wedge g_{out} \). Only one of the original transitions \( t_{in} \) and \( t_{out} \) may have a trigger, the trigger of the new transition is this single trigger (or no trigger at all). After substituting all possible pairs the vertex is to be removed with all incoming and outgoing transitions.

In case of our example (Fig. A.6), three junction pseudostates and a choice vertex is to be eliminated: the elimination of \( bentry \) was the simplest case, the incoming and the outgoing transitions were replaced by a single transition, the two other junction vertices were replaced by multiple transitions originating in \( b_2 \) and targeting the corresponding fork vertices (note that neither the \( e \) trigger, nor the \( \{ c_{11}, c_{21} \} \subseteq MRC\text{ConfFull}(e) \) etc. guards were lost) and the choice pseudostate was handled similarly. The resulting statechart is shown in Fig. A.7.

The final step is the substitution of the terminate pseudostate with a termination state, the resulting statechart is shown in Fig. A.8.

Having reached this point of the discussion, all shorthand constructs were substituted by core concepts, therefore the formal operational semantics is applicable for the resulting statechart. Obviously the example above is just an illustration, the fully detailed discussion, specification of the transformation algorithms, well-formedness rules etc. can be found in [132].

A.7 Detailed Examples

Example 36 (Dropping a Trigger and Termination). The example scenario shown in Fig. 1.19 and Fig. 1.20 continues in Fig. A.9 by dropping a trigger and termination. The reception of \( t_1 \) in

Figure A.9: Illustration of Operation – Dropping a Trigger and Termination
the configuration \( L_{conf}(s_g) = \{s_6, s_8\} \) does not trigger any transition conglomerates thus the trigger is dropped making \( B^{KTS} \) remain in the \( s_g \) state.

The reception of \( t_2 \) in \( s_g \) takes \( B^{KTS} \) to \( s_h \) by opening a run-to-completion step. The step involves the parallel execution of exit activities of states \( s_6 \) and \( s_8 \), then the exit activity of \( s_4 \), the effect of \( tc_5 \) and the entry activities of \( s_9 \) and \( s_{10} \) in the correct order.

As \( s_h \) is still unstable and there is a fireable transition conglomerate without trigger (\( tc_6 \)), an internal step is performed resulting in \( s_i \). (Note that \( s_{11} \) is a termination state therefore it has no entry activity, this is the reason for not being indicated in the PERT graph.)

In the unstable state \( s_i \) there is a termination state active (\( s_{11} \)), the only possible step is a termination step taking \( B^{KTS} \) to \( s_j \). As \( s_j \) is in the terminated phase, the entire execution of the state machine is terminated.

A.8 Example for Simulation

This section presents an example run of our simulator developed for illustrating the platform-independent formal operational semantics discussed in Chp. 4. The example statechart and the scenario as the same as shown in figures Fig. A.9, Fig. A.10 and Fig. A.11. Listing A.30 presents the initialization, Listing A.31 presents processing a trigger and Listing A.32 presents dropping a trigger and termination.

The example statechart was drawn in a modeling tool (Poseidon for UML as shown in Fig. A.10).
A.8. EXAMPLE FOR SIMULATION

Listing A.29: Call Syntax of the Simulator

```
C:\precise-statechart>bin\simulator2
simulator MODEL-PATH CLASS-NAME SM0-NAME TRIGGER...
```

MODEL-PATH Path of the UML model.
CLASS-NAME Name of the class containing the state machine.
SM0-NAME Name of the root state machine.
TRIGGER Name of a trigger event.

Note that any number of triggers can be specified in a sequence delimited by spaces.

The simulator is a command line tool with a straightforward syntax (Listing A.29) where the user has to specify the model (an XMI file or a Poseidon .zuml archive), the name of the class and the root state machine finally a sequence of trigger names. In our example (first line of Listing A.30) the triggers sequence was t1 t1 t2 similarly to the figures.

The first step of initialization (Listing A.30) is an \texttt{init} step (3–13) taking the statechart from the uninitialized state (5) to the unstable configuration \{s1, s2\} (13) after performing the entry activity s1 then the entry activity of s2 (note the indication of nodes and precedences in textual representation of PERT graphs at lines 7–12); the step was not triggered by any triggers indicated by t0 (6). The next step is an \texttt{inter} (15–29) because the empty constraint \(c_0\) evaluates to true (15) enabling this way the transition conglomerate \(tc_1\). The resulting (unstable) configuration is \{s1, s3\} (29); during the step s2 was left, the (empty) effect of \(tc_1\) was performed, finally s3 was entered (21–28); the step was triggered by t0 again (20). The last step of initialization is \texttt{close} (32–36) making the configuration \{s1, s3\} stable (36). This scenario is the same as shown in Fig. A.19.

Processing the trigger \(t1\) (Listing A.31) consists of an \texttt{open} (37–62), two \texttt{inter} (64–80 and 82–97) and finally a \texttt{close} (100–104). The syntax of the output is the same as discussed above, the only thing to be highlighted is the evaluation of non-empty guards: as the simulator does not interpret the language of either activities or guard constraints, the maintenance of extended variables and evaluation of constraints referring to variables is the responsibility of the user, e.g., the guard predicate of \(tc_4\) \((v_1 > 0)\) is “evaluated” by asking a yes/no question from the standard input (lines 65-66 and 82–83). This scenario is the same as shown in Fig. A.20.

Finally Listing A.32 presents dropping a trigger \(t1\) and processing the trigger \(t2\) resulting in termination. Dropping the trigger \(t1\) is a single \texttt{drop} step (105–109). The rest of execution is initiated by processing the trigger \(t2\) by first opening an RTC step \texttt{open} at lines 112–132), performing an internal step \texttt{inter} at lines 134–150) finally terminating in the configuration \{s11\} \texttt{term} at lines 153–157). This scenario is the same as shown in Fig. A.9.
Listing A.30: Initialization in the Simulator
A.8. EXAMPLE FOR SIMULATION

Listing A.31: Processing a Trigger in the Simulator
|=\text{drop}=>|

s: (PhsStbl, \{s_6, s_8, s_4\})

t: t_1

s': (PhsStbl, \{s_6, s_8, s_4\})

eval |= c_0

|=\text{open}=>|

s: (PhsStbl, \{s_6, s_8, s_4\})

t: t_2

=\text{open}=>|

s: (PhsStbl, \{s_6, s_8, s_4\})

t: t_2

Nodes:

3: Entry activity of s_{10}

2: Entry activity of s_9

1: Exit activity of s_8

0: Exit activity of s_6

5: Effect of tc_5

4: Exit activity of s_4

Precedences:

0->4

5->2

4->5

2->3

1->4

s': (PhsUnst, \{s_{10}, s_9\})

eval |= c_0

|=\text{inter}=>|

s: (PhsUnst, \{s_{10}, s_9\})

t: t_0

p:

Nodes:

0: Effect of tc_6

3: Exit activity of s_9

2: Entry activity of s_{11}

1: Exit activity of s_{10}

Precedences:

0->2

3->0

1->3

s': (PhsUnst, \{s_{11}\})

|=\text{term}=>|

s: (PhsUnst, \{s_{11}\})

t: t_0

s': (PhsOmega, \{s_{11}\})

Listing A.32: Dropping a Trigger and Termination in the Simulator
Appendix B

Details on Automatic Implementation

This appendix presents some detailed discussions on implementation of statecharts that were not included in the main part of the document due to space restrictions: in Sec. B.1 we present some examples for statechart implementation approaches published previously in the literature (nested switch and state table organization, the State design pattern and the QHsm pattern); the AsmL specification of algorithms specifying the platform-specific semantics outlined in Chp. 2 is presented in Sec. B.2; finally Sec. B.3 presents a practical overview on the process of automatic code generation for an embedded platform with source code fragments and photos of the device in operation.

B.1 Detailed Examples for Implementation Patterns

Below we will illustrate the implementation techniques mentioned in Sec. 2.1 by the implementation of an ANSI-C preprocessor that removes /* ... */ comments from the source code and prints the remaining contents (the example was taken from [144]). The statechart of the preprocessor and an example input–output pair is shown in Fig. B.1.

Listing B.1 presents the definition of these identifiers as ANSI-C enum constructs.

Both the nested switch and the state table approaches use identifiers for referring to states and triggers; Listing B.1 presents the definition of these identifiers as ANSI-C enum constructs.

Listing B.2 presents a code fragment from the implementation of the preprocessor according to the nested switch pattern: the actual state is stored in the variable state (5) initialized to the code state, the outer switch branches according to the actual state (10); outer case blocks contain another switch statements branching according to the actually received trigger; the case blocks of triggers contain the implementations of activities to be carried out and transitions to be performed (by assignments to the variable state), e.g., in the state code (11–22) upon the reception of the triggers star and char (13–16) the state does not change and the character actually received is printed (15); in the state slash (24–39) upon the reception of the trigger star (26–28) a transition to the state comment is to be performed (27) without any activities to be carried out. (The example function processes a string received as parameter input containing the source code; the translation of the actual input character to an abstract trigger (i.e., a value of trigger_t) is done by the function translate_trigger.)

The implementation of the preprocessor according to the state table pattern is shown in Listing B.3. The code fragment first defines a function pointer type for activity implementations (1–2) and cells of the state table (4–8) containing the identifier of the state to be stepped to (6) and the pointer to the function implementing the activity to be carried out (7). The table is an initialized static constant two dimensional array of cells (16–25); cells contain the same information as the bodies of internal case blocks of the nested switch solution, e.g., in the state code (17–18) upon the reception of the trigger star (first cell in the row) the state does not change and the character actually received is printed (implemented in the function act_out_chr); in the state slash (19–20) upon the reception of the trigger star (first cell in the row) a transition to the state comment is to be performed without any activities to be carried out (indicated by NULL). The input is processed according to the table in the
APPENDIX B. DETAILS ON AUTOMATIC IMPLEMENTATION

Figure B.1: Statechart of the C Preprocessor and an Example Input–Output Pair

```c
/* * Enumerated type for trigger identifiers. */
typedef enum {
    T_STAR, /* A star '*'. */
    T_SLASH, /* A slash '/'. */
    T_CHAR, /* Another character. */
    NUM_TRIGGER /* Number of triggers. */
} trigger_t;

/* * Enumerated type for state identifiers. */
typedef enum {
    S_CODE, /* In a code region. */
    S_SLASH, /* After reading a slash. */
    S_COMMENT, /* In a comment region. */
    S_STAR, /* After reading a star. */
    NUM_STATE /* Number of states. */
} state_t;
```

Listing B.1: Enumeration of States and Triggers

loop 27–36: after translating the input character to an abstract trigger identifier (29) the appropriate cell of the table is accessed corresponding to the actual state and the trigger (31); if the activity field of the cell is not NULL the function is called (32–33), finally the state variable is updated (35).

The class diagram of the State design pattern applied to the C preprocessor example with C++ notation in comments is shown in Fig. B.2: states are derived from the abstract PreprocessorState state that provides pure virtual function corresponding to triggers (e.g., onStar for processing a star received from the input), these functions are implemented in singleton classes corresponding to actual states (e.g., CodeState corresponding to the code state). The input is processed by the class Preprocessor that contains the actual state through the role state; Preprocessor has similarly named functions as the state classes, these functions propagate the call to the actual state as shown in the comment of Preprocessor::onChar. The functions of state classes implement the state transitions and the activities to be performed on the reception of the corresponding trigger (i.e., the bodies of internal case blocks in the nested switch approach, the functions referenced in the table of the state table solution and the functions of state classes in the State design pattern play the same role), e.g., see the comments illustrating the activity of writing the character received from the input in the code state and the state transition from slash to comment on the reception of a slash.

B.2 AsmL Specification of the Platform-Specific Semantics

The AsmL listings below formally present the platform-specific mappings of algorithms in the operational semantics for statecharts. The discussion identifies the differences between the platform-independent and the platform-specific semantics and proves the equivalence of the two representations. Due to space restrictions some details were not shown even here, for more details see [134].
void nested_switch_implementation(const char* input) {
    const char *input_ptr;
    state_t state = S_CODE;
    for (input_ptr = input; *input_ptr != '\0'; input_ptr += 1) {
        trigger_t trigger = translate_trigger(*input_ptr);
        switch (state) {
            case S_CODE:
                switch (trigger) {
                    case T_STAR:
                        /* Output the character. */
                        printf("%c", *input_ptr);
                        break;
                    case T_SLASH:
                        /* Transition to state 'SLASH'. */
                        state = S_SLASH;
                        break;
                    }
                    break;
            case S_SLASH:
                switch (trigger) {
                    case T_STAR:
                        /* Transition to state 'COMMENT'. */
                        state = S_COMMENT;
                        break;
                    case T_SLASH:
                        /* Output a slash character. */
                        printf("/");
                        break;
                    case T_CHAR:
                        /* Output and transition to state 'CODE'. */
                        printf("/%c", *input_ptr);
                        state = S_CODE;
                        break;
                }
                break;
        }
    }
    [...]}

Listing B.2: Implementation by the Nested Switch Pattern

B.2.1 Base Data Types

The phase of operation of the Kripke transition system is implemented by the AsmL enumeration KTSPhase (Listing A.8). It is easy to see that Listing A.8 (PIM) and Listing B.4 (PSM) are exactly the same.

The state of the Kripke transition system is implemented by the AsmL class KTSState. It is easy to see that Listing A.9 (PIM) and Listing B.5 (PSM) are nearly the same: the only difference is that the actual configuration is represented by a fat state mask in the platform-specific model as discussed previously.

The evaluation of variables is implemented by the AsmL class Evaluation. It is easy to see that Listing A.10 (PIM) and Listing B.6 (PSM) are exactly the same.

The Kripke transition system is implemented by the AsmL class KTS. The only difference between Listing A.11 (PIM) and Listing B.7 (PSM) is that while the platform-independent specification focuses to exactly one precise statechart (therefore the precise statechart need not be exactly specified in algorithms), the platform-specific solution is closer to a real implementation where a generic interpreter should handle any number of precise statecharts, that is the reason for explicitly storing the context precise statechart in a member variable on the class KTS.
APPENDIX B. DETAILS ON AUTOMATIC IMPLEMENTATION

```c
/* Pointer to activity function type. */
typedef void (*activity_t)(const char input_char);

/* Type of a cell in a state table. */
typedef struct {
    state_t next_state;  /* The next state to be stepped to. */
    activity_t activity;  /* The activity to be performed. */
} cell_t;

void state_table_implementation(const char* input) {
    const char* input_ptr;
    state_t state = S_CODE;

    const cell_t table[NUM_STATE][NUM_TRIGGER] = {
        /* State 'CODE' (triggers: star, slash, char) */
        {{S_CODE, act_out_chr}, {S_SLASH, NULL}, {S_CODE, act_out_chr}},
        /* State 'SLASH' (triggers: star, slash, char) */
        {{S_COMMENT, NULL}, {S_SLASH, act_out_slash}, {S_CODE, act_out_slash_chr}},
        /* State 'COMMENT' (triggers: star, slash, char) */
        {{S_STAR, NULL}, {S_COMMENT, NULL}, {S_COMMENT, NULL}},
        /* State 'STAR' (triggers: star, slash, char) */
        {{S_STAR, NULL}, {S_CODE, NULL}, {S_COMMENT, NULL}}
    };

    for (input_ptr = input; *input_ptr != '\0'; input_ptr += 1) {
        const cell_t* cell;
        trigger_t trigger = translate_trigger(*input_ptr);
        cell = &(table[state][trigger]);
        if (NULL != cell -> activity)
            (cell -> activity)(*input_ptr);
        state = cell -> next_state;
    }
}
```

Listing B.3: Implementation by the State Table Pattern

B.2.2 Operation Algorithms

The platform-specific mapping of Enabled is shown in Listing B.8. We have to prove that the algorithm specified in Listing A.12 (PIM) is correctly implemented by Listing B.8 (PSM):

**Function signature:** The PSM mapping uses the output argument `retFatTCMask` instead of dynamically allocating and returning a set. The actual configuration needs not be specified in the argument list of the PSM function since it is stored in the labeling of the actual state.

**Block 16–17 (Listing A.12) compared to 16–21 (Listing B.8):** (i) it iterates over the entire transition conglomerate identifier sequence (ii) the configuration is represented by `actState.LConf`, (iii) enabling states are accessed through the .en role and (iv) the function `IsSupersetEqOfMask` implements the \( \subseteq \) set operation.

**Block 19–20 (Listing A.12) compared to 23–31 (Listing B.8):** it is easy to see that after executing the block `retFatTCMask.GetValue` will return True only for those \( i \) indexes, where (i) `retFatTCMask.GetValue(i)` was initially true (i.e., \( tc \) in TCact) and (ii) triggers match.

**Block 22–23 (Listing A.12) compared to 33–44 (Listing B.8):** it is easy to see that after executing the block `retFatTCMask.GetValue` will return True only for those \( i \) indexes, where (i) `retFatTCMask.GetValue(i)` was initially true \( tc \) in TCt and (ii) the transition conglomerate has no guard (representing the \( c_0 \) empty constraint) or the guard evaluates to true.

The platform-specific mapping of NonOverpowered is shown in Listing B.9. We have to prove that the algorithm specified in Listing A.13 (PIM) is correctly implemented by Listing B.9 (PSM):
B.2. ASML SPECIFICATION OF THE PLATFORM-SPECIFIC SEMANTICS

Listing B.4: Phase of Operation (PSM)

```java
public enum KTSPhase {
    // Alpha (pre-initialization)
    PhsAlpha,
    // Unstable (during a run-to-completion step)
    PhsUnst,
    // Stable (idle between two run-to-completion steps)
    PhsStbl,
    // Omega (post-termination)
    PhsOmega
}
```

Listing B.5: State of the Kripke Transition System (PSM)

```java
public class KTSState {
    public primitive var LConf as FatStateMask
    public primitive var LEval as Evaluation
    public primitive var LPhs as KTSPhase

    public KTSState(LConf as FatStateMask, LEval as Evaluation, LPhs as KTSPhase) {
        LConf = aLConf
        LEval = aLEval
        LPhs = aLPhs
    }
    public KTSState(aKTSState as KTSState) {
        LConf = new FatStateMask(aKTSState.LConf.Length)
        step foreach i in [0..(aKTSState.LConf.Length - 1)]
            LConf.SetValue(i, (aKTSState.LConf.GetValue(i)))
        LEval = new Evaluation(aKTSState.LEval)
        LPhs = aKTSState.LPhs
    }
```

Listing B.6: Evaluation of Variables (PSM)

Listing B.7: Key Data Members of the KTS Class (PSM)
B.2. ASML SPECIFICATION OF THE PLATFORM-SPECIFIC SEMANTICS

Listing B.8: AsmL Specification of the Enabled Function (PSM)

---

```asm
private Enabled(t as Integer, retFatTCMask as FatTCMask)

  require retFatTCMask.Length = psc.NumTCs

  // Transition conglomerates with active enabling states
  // This implementation directly stores enabled TCs in retFatTCMask
  step foreach i in psc.TCIndexSeq
    val as Boolean =
      actState.LConf.IsSupersetEqOfMask(psc.transitionConglomerateSeq(i).en)
      retFatTCMask.SetValue(i, val)

  // Active transition conglomerates that are triggered by the trigger
  // This implementation directly stores triggered TCs in retFatTCMask
  step foreach i in psc.TCIndexSeq
    val as Boolean =
      if retFatTCMask.GetValue(i)
      then (psc.transitionConglomerateSeq(i).trigger = t)
      else false
      if psc.transitionConglomerateSeq(i).guard = null then
        actState.LEval.Models(psc.transitionConglomerateSeq(i).guard)
      else false
      retFatTCMask.SetValue(i, val)
```
APPENDIX B. DETAILS ON AUTOMATIC IMPLEMENTATION

Figure B.2: The State Design Pattern and Its Application to the C Preprocessor Example

Listing B.9: AsmL Specification of the NonOverpowered Function (PSM)

Function signature: The PSM mapping uses the output argument retFatTCMask instead of dynamically allocating and returning a set.

Block 13–14 (Listing [A.13] compared to 16–21 (Listing [B.9]): it is easy to see that the where not... expression is correctly implemented by the AreDisjoint function.

The platform-specific mapping of ConflictFree is shown in Listing [B.10] We have to prove that the algorithm specified in Listing [A.14] (PIM) is correctly implemented by Listing [B.10] (PSM):

Collection of conflicting pairs: In order to prevent dynamic creation of sets, the PSM mapping provides somewhat less diagnostic information: the set of conflicting transition conglomerates is not collected, the function only traverses the set of transition conglomerate identifiers and for all identifiers corresponding to the set TCchk validates that TCchk and the set of transition conglomerates of higher priority than the one represented by i are disjoint.

The platform-specific mapping of the function Fireable is shown in Listing [B.11] We have to prove that the algorithm specified in Listing [A.15] (PIM) is correctly implemented by Listing [B.11] (PSM):

Function signature: The PSM mapping uses the output argument retFatTCMask as accumulator instead of dynamically allocating and returning a set and the actual configuration needs not be specified in the argument list of the PSM function since it is stored in the labeling of the actual state, otherwise the two functions are conceptually the same.

The platform-specific mapping of the function LeavingHierarchy is shown in Listing [B.12] We have to prove that the algorithm specified in Listing [A.16] (PIM) is correctly implemented by Listing [B.12] (PSM):

Function signature: The PSM mapping uses the output argument retFatTCMask as accumulator instead of dynamically allocating and returning a set and the actual configuration needs not be specified in the argument list of the PSM function since it is stored in the labeling of the actual state, otherwise the two functions are conceptually the same.
B.2. ASML SPECIFICATION OF THE PLATFORM-SPECIFIC SEMANTICS

Listing B.10: AsmL Specification of the `ConflictFree` Function (PSM)

Listing B.11: AsmL Specification of the `Fireable` Function (PSM)
APPENDIX B. DETAILS ON AUTOMATIC IMPLEMENTATION

Listing B.12: AsmL Specification of the LeavingHierarchy Function (PSM)

Function signature: The PSM mapping uses the output arguments retFatPERTGraph and retFatStateMask as accumulators instead of dynamically allocating and returning sets. The input argument shn is a state hierarchy node in the PIM specification while it is a state bit mask representing the same hierarchy in the PSM mapping (as discussed above). The actual configuration needs not be specified in the argument list of the PSM function since it is stored in the labeling of the actual state.

Non-recursive behavior: The PSM mapping is a non-recursive implementation of the original PIM specification: as discussed previously collecting the states actually left and building the corresponding sequential PERT graph of exit activities is simply a backward iteration through the state bit mask and checking whether the actual state is active in the current configuration.

The platform-specific mapping of the function LeavingHierarchySet is shown in Listing B.13. Since as discussed above state bit masks are capable of representing even sets of state hierarchies, the PSM function LeavingHierarchy fully implements LeavingHierarchySet, this way the PSM mapping of LeavingHierarchySet simply calls LeavingHierarchy with the same argument list.

The platform-specific mapping of the function EnteringHierarchy is shown in Listing B.14. We have to prove that the algorithm specified in Listing A.18 (PIM) is correctly implemented by Listing B.14 (PSM). The discussion is similar to the comparison of LeavingHierarchy functions:

Function signature: The PSM mapping uses the output arguments retFatPERTGraph and retEnteredFatStateMask as accumulators instead of dynamically allocating and returning sets. The input argument shn is a state hierarchy node in the PIM specification while it is a state bit mask representing the same hierarchy in the PSM mapping (as discussed above).

Non-recursive behavior: The PSM mapping is a non-recursive implementation of the original PIM specification: as discussed previously collecting the states entered and building the corresponding sequential PERT graph of entry activities is simply a forward iteration through the state bit mask.
B.2. ASML SPECIFICATION OF THE PLATFORM-SPECIFIC SEMANTICS

Listing B.13: AsmL Specification of the LeavingHierarchySet Function (PSM)

Listing B.14: AsmL Specification of the EnteringHierarchy Function (PSM)
APPENDIX B. DETAILS ON AUTOMATIC IMPLEMENTATION

Listed B.15: AsmL Specification of the \texttt{EnteringHierarchySet} Function (PSM)

```plaintext
private EnteringHierarchySet(Shn as CompactStateMask, retPERTGraph as PERTGraph,
retEnteredStateMask as FatStateMask)
EnteringHierarchy(Shn, retPERTGraph, retEnteredStateMask)
```

Listed B.16: AsmL Specification of the \texttt{LeavingSource} Function (PSM)

```plaintext
private LeavingSource(tcID as Integer, retPERTGraph as PERTGraph,
retLeftFatStateMask as FatStateMask)
LeavingHierarchySet(psc.transactionConglomerateSeq(tcID).possiblyLeftStates,
retPERTGraph, retLeftFatStateMask)
```

The platform-specific mapping of the function \texttt{EnteringHierarchySet} is shown in Listing [B.15]. Since as discussed above state bit masks are capable of representing even sets of state hierarchies, the PSM function \texttt{EnteringHierarchy} fully implements \texttt{EnteringHierarchySet}, this way the PSM mapping of \texttt{EnteringHierarchySet} simply calls \texttt{EnteringHierarchy} with the same argument list.

The platform-specific mapping of the function \texttt{LeavingSource} is shown in Listing [B.16]. The PIM (Listing [A.20]) and PSM (Listing [B.16]) functions differ mainly in signature: the PSM mapping does not need the actual state configuration since it is stored in the labeling of the actual state, the transition conglomerate is identified by its identifier and output arguments are used instead of dynamically allocated and returned sets.

The platform-specific mapping of the function \texttt{PerformingEffect} is shown in Listing [B.17]. The PIM (Listing [A.21]) and PSM (Listing [B.17]) functions differ mainly in signature: the transition conglomerate is identified by its identifier and an output argument is used instead of dynamically allocated and returned set (the effect of the transition conglomerate is simply copied into the output argument).

The platform-specific mapping of the function \texttt{EnteringTarget} is shown in Listing [B.18]. The PIM (Listing [A.22]) and PSM (Listing [B.18]) functions differ only in signature: the transition conglomerate is identified by its identifier and output arguments are used instead of dynamically allocated and returned sets.
B.2. ASML SPECIFICATION OF THE PLATFORM-SPECIFIC SEMANTICS

Listing B.17: AsmL Specification of the PerformingEffect Function (PSM)

Listing B.18: AsmL Specification of the EnteringTarget Function (PSM)
Appendix B. Details on Automatic Implementation

// ///////////////////////////////////////////////////////////////////////
// Firing a transition conglomerate
// // tcID: Identifier of the transition conglomerate.
// // retPERTGraph: PERT graph for storing the sequence of activities to be performed.
// // retLeftFatStateMask: State mask for storing the set of states left.
// // retEnteredFatStateMask: State mask for storing the set of states entered.
// ///////////////////////////////////////////////////////////////////////
private FireSingle (tcID as Integer, retPERTGraph as PERTGraph, retLeftFatStateMask as FatStateMask, retEnteredFatStateMask as FatStateMask)

leavePERTGraph as PERTGraph = new PERTGraph ([] as Seq of PERTNode)
effectPERTGraph as PERTGraph = new PERTGraph ([] as Seq of PERTNode)
enterPERTGraph as PERTGraph = new PERTGraph ([] as Seq of PERTNode)

LeavingSource (tcID, leavePERTGraph, retLeftFatStateMask)
PerformingEffect (tcID, effectPERTGraph)
EnteringTarget (tcID, enterPERTGraph, retEnteredFatStateMask)

retPERTGraph.nodes :=
leavePERTGraph.nodes + effectPERTGraph.nodes + enterPERTGraph.nodes

Listing B.19: AsmL Specification of the FireSingle Function (PSM)

The platform-specific mapping of the function FireSingle is shown in Listing B.19. We have to prove the equivalence of Listing A.23 (PIM) and Listing B.19 (PSM):

Function signature: The PSM mapping uses the output arguments retFatPERTGraph, retLeftFatStateMask and retEnteredFatStateMask as accumulators instead of dynamically allocating and returning sets. The transition conglomerate is identified by its identifier and the actual configuration needs not be specified in the argument list of the PSM function since it is stored in the labeling of the actual state.

Dynamic construction of PERT graphs as work-variables (22-24 in Listing B.19): for simplicity this function dynamically allocates the PERT graphs required by functions LeavingSource, PerformingEffect and EnteringTarget. This dynamic memory allocation (and all PERT graph operations) will be eliminated from the final ANSI-C level implementation.

Block 18-20 (Listing A.23) compared to 26–28 (Listing B.19): the two blocks are conceptually equivalent regardless of minor differences in function signatures.

Block 23 (Listing A.23) compared to 30–31 (Listing B.19): appending two or more sequential PERT graphs is simply the concatenation of node sequences.

The platform-specific mapping of the function Fire is shown in Listing B.20. We have to prove the equivalence of Listing A.24 (PIM) and Listing B.20 (PSM):

Function signature: The PSM mapping uses the output arguments retFatPERTGraph, retLeftFatStateMask and retEnteredFatStateMask as accumulators instead of dynamically allocating and returning sets, and two work variables workLeftFatStateMask and workEnteredFatStateMask. The set of transition conglomerates is specified by a fat transition conglomerate mask firedFatTCMask instead of a transition conglomerate set.
Preparation output arguments: The PSM mapping explicitly clears all of its state masks and the PERT graph used as work variables for ensuring that the sets and the graph are initially empty.

Iteration through the transition conglomerate set and calling FireSingle: The original PIM function simply collects the three tuples returned by the Fire function in a set expression (19–20 in Listing B.20). The iteration is explicitly implemented by letting the variable i take the identifier of all transition conglomerates in firedFatTCMask in the PSM mapping (34–35 in Listing B.20) and calls FireSingle for these transition conglomerates (36–37).

Constructing the entire PERT graph: The PIM function extracts the PERT graphs from the tuple set and uses the PERTAppend function for constructing the PERT graph that describes the entire operation structure to be performed when firing the transition conglomerates in parallel (22 in Listing A.24). The PSM mapping iteratively appends the PERT graphs describing the activity structure to be performed when firing a single transition conglomerate (workPERTGraph) to the accumulator PERT graph retPERTGraph. It is important to see that appending the sequential PERT graphs after each other is a restrictive but valid implementation of the union operation, since we only introduce new (non-required) dependencies but will not violate any existing ones – as we are mapping to a sequential scheduling anyway, we do not lose anything.

Unifying sets: The PSM function extracts and unifies the sets of states left and entered by individual transition conglomerates (lines 23 and 24 in Listing A.24 respectively). The PSM mapping simply iterates over the index set and adds the states to the accumulator variables one-by-one (blocks 40–43 and 45–48 in Listing A.24).

The platform-specific mapping of the function Init is shown in Listing B.21. We have to prove the equivalence of Listing A.25 (PIM) and Listing B.21 (PSM):

Function signature: The PSM mapping uses the output arguments retFatPERTGraph and retEnteredFatStateMask instead of dynamically constructing and returning them.

Calling EnteringHierarchy: There are two differences between the function bodies: the PSM mapping (i) uses the pre-calculated initial state hierarchy stored in the precise statechart instead of calling InitialHierarchy and (ii) obviously the signature of EnteringHierarchy is different.

The platform-specific mapping of the function InitializationStep is shown in Listing B.22. We have to prove the equivalence of Listing A.26 (PIM) and Listing B.22 (PSM):

Function signature: The PSM mapping uses work variables workFatTCMask, workFiredFatTCMask, workLeftFatStateMask, workEnteredFatStateMask, workRetLeftFatStateMask, and workRetEnteredFatStateMask instead of dynamically allocating them.

Block 10–14 (Listing A.26) compared to 32–37 (Listing B.22): The implementation of init is nearly the same, only the signature of Init is different and the PSM mapping implements the set assignment by adding the states one-by-one to the state mask (line 12 in Listing A.26 and block 34–35 in Listing B.22).

Checking termination: As termination states are represented by the terminationStates association of the precise statechart, the PSM mapping uses the corresponding state bit mask for checking whether a termination state is active (line 17 in Listing A.26 and block 40–41 in Listing B.22).

Collecting fireable transition conglomerates: The signature of the Fireable function is different (lines 22 in Listing A.26 and 47 in Listing B.22 respectively).
APPENDIX B. DETAILS ON AUTOMATIC IMPLEMENTATION

Listing B.20: AsmL Specification of the Fire Function (PSM)

Listing B.21: AsmL Specification of the Init Function (PSM)
B.3 An Example for Code Generation

This section presents an example for automatic source code synthesis by the prototype implementation of our code generator. The process of implementing statecharts is shown in Fig. B.3: modelers build the statechart in a modeling tool and programmers implement the basic activities (state entry and exit activities, transition effects etc.) and some glue code in the chosen programming language. Activity implementations can be included in the UML model itself but in order to keep the model independent of the final target language it is advisable to implement the activities in separated source files. The necessary glue code should implement those heavily target device-specific routines that can not be automatically generated, e.g., initialization of the device, interrupt handling routines, polling controls and the main loop. (The device shown in the figure and used as target platform in the example is a mitmót [158] microcontroller board based on an Atmel Mega-128 8-bit microcontroller running at 6 MHz equipped with 4 KByte RAM and 128 KByte flash ROM.)
APPENDIX B. DETAILS ON AUTOMATIC IMPLEMENTATION

Listing B.22: AsmL Specification of the InitializationStep Function (PSM)
B.3. AN EXAMPLE FOR CODE GENERATION

```java
// A trigger processing step of the precise statechart

// t: Identifier of the trigger.
// workFatTCMask: Transition conglomerate mask for storing temporary data.
// workFiredFatTCMask: Transition conglomerate mask for storing temporary data.
// workLeftFatStateMask: State mask for storing temporary data.
// workEnteredFatStateMask: State mask for storing temporary data.
// workRetLeftFatStateMask: State mask for storing temporary data.
// workRetEnteredFatStateMask: State mask for storing temporary data.

// Performs a trigger processing step of the precise statechart.

private TriggerProcessingStep(t as Integer,
workFatTCMask as FatTCMask,
workFiredFatTCMask as FatTCMask,
workLeftFatStateMask as FatStateMask,
workEnteredFatStateMask as FatStateMask,
workRetLeftFatStateMask as FatStateMask,
workRetEnteredFatStateMask as FatStateMask)

require (actState.LPhs = PhsStbl) or (actState.LPhs = PhsOmega)
primitive var workPERTGraph as PERTGraph = new PERTGraph([] as Seq of PERTNode)

if actState.LPhs = PhsStbl then
  Fireable(t, workFiredFatTCMask, workFatTCMask)
  if workFiredFatTCMask.IsSet = false then
    // = [drop] = >
  else
    // = [open] = >
    Fire(workFiredFatTCMask, workLeftFatStateMask, workEnteredFatStateMask, workPERTGraph, workRetLeftFatStateMask, workRetEnteredFatStateMask)
  step foreach i in psc.StateIndexSeq
    primitive var val as Boolean = actState.LConf.GetValue(i)
    if workRetLeftFatStateMask.GetValue(i) then
      val := false
    if workRetEnteredFatStateMask.GetValue(i) then
      val := true
      actState.LConf.SetValue(i, val)
      actState.LEval := actState.LEval.Issue(workPERTGraph)
    actState.LPhs := PhsUnst
  step while actState.LPhs = PhsUnst
  if (exists i in psc.StateIndexSet where
      (actState.LConf.GetValue(i) and psc.terminationStates.GetValue(i))) then
    // = [term] = >
    actState.LPhs := PhsOmega
  else
    Fireable(-1, workFiredFatTCMask, workFatTCMask)
    if workFiredFatTCMask.IsSet = false then
      // = [close] = >
      actState.LPhs := PhsStbl
    else
      // = [inter] = >
      Fire(workFiredFatTCMask, workLeftFatStateMask, workEnteredFatStateMask, workPERTGraph, workRetLeftFatStateMask, workRetEnteredFatStateMask)
    step foreach i in psc.StateIndexSeq
      primitive var val as Boolean = actState.LConf.GetValue(i)
      if workRetLeftFatStateMask.GetValue(i) then
        val := false
      if workRetEnteredFatStateMask.GetValue(i) then
        val := true
        actState.LConf.SetValue(i, val)
      actState.LEval := actState.LEval.Issue(workPERTGraph)
```

Listing B.23: AsmL Specification of the `TriggerProcessingStep` Function (PSM)
APPENDIX B. DETAILS ON AUTOMATIC IMPLEMENTATION

Listing B.24: AsmL Specification of the Operation Function (PSM)

Listing B.25: Simplified AsmL Specification of the Operation Function (PSM)
B.3. AN EXAMPLE FOR CODE GENERATION

Based on the statechart model the code generator constructs that part of the source that is responsible for implementing the event-driven behavior (i.e., algorithms discussed in Chp. 2 and an initialized data structure describing the statechart). The library built this way will maintain the configuration and call the user provided routines (implementations of atomic activities) according to the actual configuration and event received. If the activity implementations are integrated into the model (e.g., C code fragments were typed into the entry activity field of states) the code generator copies these fragments into appropriate sources for convenience.

The three sets of sources (activity implementations, glue code and generated files) are to be built (compiled, assembled and linked) using the usual tool-chain and the finally resulting binary is to be deployed to the target platform.

The example model will be the statechart of a traffic light shown in Fig. B.4.

The traffic light is placed in the crossing of a main road and a country road. The equipment at the main road consists of two sensors, a camera, a traffic light and the controller. For simplicity reasons only the light control of the main road is investigated here. The first sensor signals the arrival of a car to the crossing from the main road and the second one sends a signal to the controller if a car runs in the crossing from the main road. The controller system provides higher precedence to the main road i.e., it does not wait until the normal time of switching from red to red-yellow if more than two cars are waiting at the main road but switches immediately. Cars running illegally in the crossing during the red signal are detected by the second sensor and can be recorded by the camera connected to the controller. The camera can be switched on and off manually.
Figure B.5: UML Model in a Modeling Environment

The statechart diagram in Fig. B.4 shows the behavioral specification of the light control at the main road. The light can be operating (On state) or switched off (Off state). The transition between them is triggered by the SwitchEvt event.

Representation of light states (red, red-yellow, green and yellow) is obvious, transitions between these states are triggered by time events. State red is decomposed orthogonally into two regions, one for the camera and one for counting the cars waiting at the main road. The most recent state of the camera (operating or switched off) is restored when entering the red state through the history pseudostate. In order to prevent the shallow history mechanism from restoring the most recent number of cars waiting in the crossing the car counter mechanism is wrapped into a container state. The interlevel transition from waiting2 to red-yellow is triggered by the arrival of the third car from the main road. This way the crossing is provided to the traffic from the main road.

The model being drawn in the Poseidon for UML modeling tool is shown in Fig. B.5. Poseidon models are saved into the standard XMI model interchange format wrapped in a .zuml compressed archive. The XMI file can be extracted from the archive for being used directly but for convenience our XMI library supports Poseidon archives without having to unpack manually.

For visual demonstration purposes we used the standard display card of the mitmót device for representing input events (sensors) and activities performed by the traffic light (Fig. B.6). The display card contains a thermometer, a block of four two-state switches, four LEDs, three 7-segment displays and three push buttons. Obviously we will not use the thermometer in the example. The switch block
B.3. AN EXAMPLE FOR CODE GENERATION

Guard predicates are mapped to switches.

LEDs indicating the number of cars waiting in the crossing.

7-segment display indicating the state of the traffic light.

Events are mapped to buttons

Switch event
Timer event
Car arrival

Off (‘OFF’):
Red (‘P’):
Green (‘Z’):
Red-Yellow (‘PS’):
Yellow (‘S’):

Figure B.6: Mapping Events and Responses to the Display Card

is ideal for visualizing guard predicates (but in this example we will not use them either). We will use the LEDs for indicating the number of cars waiting at the main road (i.e., the entry and exit activities of waiting states and carCounter maintain the state of LEDs). The light actually emitted by the traffic light is indicated on the 7-segment displays using Hungarian abbreviations: P for red, PS for red-yellow, Z for green and S for yellow (since English abbreviations would be less readable on a 7-segment display). The events possibly received by the traffic light are mapped to push buttons from left to right as: switch button, timer event and arrival of a car respectively.

An example for accessing the devices on the display card is shown in Listing B.26 presenting the entry activity of state red-yellow: the entry activity consists of a single statement that prints P and S to the first two 7-segment displays and blanks the third.

Having designed the model in a UML environment the next step is the code generation. The prototype of our code generator is a command line utility with a straightforward usage syntax (Listing B.27) where the user has to specify the target language (currently C and AsmL are supported), the name prefix of the files to be generated, the path of the model file (XMI or .zuml), the name of the class whose source is to be generated and the name of the root state machine (the command line in case of our example is shown in Fig. B.28).

Finally some glue code is to be written (Fig. B.7) and the sources are to be built and the binary is to be deployed to the target platform. An example scenario using the finally established system is shown in Fig. B.8. The photos present the actual state of the display card, the configuration of the statechart corresponding to the subsequent steps is indicated in blue callouts and the events triggering the operation are indicated in orange callouts under the corresponding button.
APPENDIX B. DETAILS ON AUTOMATIC IMPLEMENTATION

Listing B.26: Entry Activity of State Red-Yellow

```c
void entry_Lamp_RedYellow(void *context, fat_state_mask_t configuration) {
    /* Copied from the model. */
    dpy_trm_s01__7seq_write_3digit(SEG7_P, SEG7_S, SEG7_EMPTY);
}
```

Listing B.27: Call Syntax of the Code Generator

```
C:\lamp> generator
3 generator TARGET-LANGUAGE TARGET-FILE-NAME MODEL-PATH
4 CLASS-NAME SM0-NAME [CLASS-NAME SM0-NAME]...
5 TARGET-LANGUAGE Language of the source to be generated, one of
6 "AsmL", "ANSI-C" or "ANSI-C-perf"
7 TARGET-FILE-NAME Name of the generated source(s) without extension.
8 MODEL-PATH Path of the UML model.
9 CLASS-NAME Name of the class containing the state machine.
10 SM0-NAME Name of the root state machine.
11
12 Multiple class name, root state machine name pairs can be specified at
13 the command line.
```

Listing B.28: Generation of Source Code for the Traffic Light

```
C:\lamp> generator ANSI-C lamp_generated lamp.zuml Lamp State_Machine_1
3 Processing the command line...
4 Target language: ANSI-C
5 Target file name: lamp_generated
6 Model path: lamp.zuml
7 Class name, root state machine name tuple sequence: [(Lamp, State_Machine_1)]
8
9 Initializing the logger module of the precise statechart library...
10 Log file path: libpsc.log
11
12 Loading model...
13 Running model transformation and preparing TC set collections...
14 PIM -> PSM transformation...
15 PSM -> ANSI-C code generation...
16
17 C:\lamp>dir
18 Volume in drive C has no label.
19 Volume Serial Number is D05E-BB1E
20
21 Directory of C:\lamp
22
23 11/14/2006 11:05 a <DIR> .
24 11/14/2006 11:05 a <DIR> ..
25 02/13/2006 05:46 p 19,217 lamp.zuml
26 11/14/2006 11:05 a 47,105 lamp_generated.c
27 11/14/2006 11:05 a 5,328 lamp_generated.h
28 11/14/2006 11:05 a 12,875 lamp_generated_functions.c
29 11/14/2006 11:05 a 5,712 libpsc.log
```
Figure B.7: Editing the Glue Code in Eclipse
Figure B.8: Example Scenario
Appendix C

Details on Error Detection

This appendix presents some detailed discussions on implementation of statecharts that were not included in the main part of the document due to space restrictions: Sec. C.1 presents the proof of correctness for our novel method for evaluating PLTL expressions on finite execution traces; the analysis of algorithmic complexity for these functions is outlined in Sec. C.2, finally Sec. C.3 presents the code generated for the PLTL evaluation example in the main part.

C.1 Proof of Correctness

Theorem 16. CheckTrace constructs and maintains valid nodes throughout its entire execution.

Proof. We prove the theorem through six intermediate observations:

(i) Based on the definition of validity (Def. 58) and the specification of the Head utility (Def. 59), it is trivial that Head constructs a valid node.

(ii) Based on the definition of validity (Def. 58) and the specification of the Next utility (Def. 60), it is trivial that when being called for a valid node as previous instance, Next constructs another valid node.

(iii) When called with a valid node EN_E^N_E and expression e ∈ E, Eval_3 (Def. 61) returns T or ? if II |= e and returns ⊥ or ? if II |≠ e. Based on this it is obvious that when calling IntProp (Def. 62) for a valid node it remains valid.

(iv) With respect to the external propagation utilities we can say that when calling ExtProp(EN_E^N_E, d, v) (Def. 63) so that EN_E^N_E is valid and v is valid (i.e., v = T iff II^i+1 |= d and v = ⊥ iff II^i+1 |≠ d) the modified node remains valid.

(v) Similarly, when calling TermTrace (Def. 64) with a valid node EN_E^N_E corresponding to the final step of the trace EN_E^{N-1} remains valid because of the semantics of the X operator.

(vi) Being based on utilities IntProp and ExtProp it is easy to see that when being called for a valid node, any modifications performed by RecProp will not compromise the validity.

Finally we can conclude that since CheckTrace calls the Head utility for constructing the head node (i.e., the head node is initially valid), constructs non-head nodes by calling Next on valid nodes, uses the RecProp routine for recursive propagation, and calls TermTrace only for the last node in the chain, all nodes ever constructed or modified are valid during the entire execution. □
This statement has proven that after the execution of CheckTrace the top interface of the head node will not contain inconsistent values. In App. C.2 we will prove that (i) the algorithm always terminates, (ii) evaluates all expressions on the top interface of the head node to a certain value and (iii) the execution time is linear in the length of the trace.

### C.2 Algorithmic Complexity

**Definition 77 (Propagated Node).** A valid EN\textsuperscript{TI} node is propagated if (i) all expressions on its top interface are evaluated to a certain value that can be evaluated to a certain value (i.e., not the \(?\) unknown value) according to the semantics of ternary logic operators (Eq. C.1) and (ii) if EN\textsuperscript{TI} is not the head instance, for expressions on its top interface that are evaluated to a certain value this value is copied to the bottom interface of the previous node (let the EN\textsuperscript{TI} :: P previous node by EN\textsuperscript{TI−1}) (Eq. C.2):

\[
\forall e \in E : ((\text{Eval}(\text{EN}\textsuperscript{TI}, e) = \top) \rightarrow (\text{EN}\textsuperscript{TI} : : \text{T}[e] = \top)) \land (\text{Eval}(\text{EN}\textsuperscript{TI−1}, e) = \bot) \rightarrow (\text{EN}\textsuperscript{TI−1} : : \text{T}[e] = \bot)) \quad (C.1)
\]

\[
\forall e \in E : (\text{EN}\textsuperscript{TI} : : \text{T}[e] \in \{\top, \bot\} \rightarrow (\forall \chi(f) \in \text{Collect\textsuperscript{CNF}}(f) : e \rightarrow (\text{EN}\textsuperscript{TI} : : \text{T}[e] = \text{EN}\textsuperscript{TI−1} : : \text{B}[\chi(f)]))) \quad (C.2)
\]

**Theorem 17.** When calling RecProp on a node EN\textsuperscript{TI} where all previous nodes in the chain are propagated, after the execution of RecProp EN\textsuperscript{TI} will be propagated and all previous nodes in the chain will remain propagated.

**Proof.** The subsequent internal and external propagation steps in RecProp ensure that the actually processed node becomes propagated and the prerequisite that all previous nodes in the chain were propagated ensures that (i) the algorithm terminates only if there are no values to be propagated and (ii) when having nothing to be propagated to the previous node it means that both the actual and the previous nodes are propagated.

**Theorem 18.** It is easy to see that CheckTrace calls RecProp for a node EN\textsuperscript{TI} so that all previous nodes in the chain are propagated: (i) the statement is true for the node returned by Head, (ii) true for all nodes newly constructed by Next and similarly (iii) true for the last node after the TermTrace operation.

**Theorem 19.** Let the left and bottom interface of EN\textsuperscript{TI} contain certain values; it is trivially true that all expressions on the top interface can be evaluated to certain values according to \text{Eval\textsuperscript{3}}.

**Theorem 20.** Based on Thm. 19 it is obvious that for a chain consisting of fully propagated nodes if the top interface of the head node contains an uncertain value, all nodes in the chain contain uncertain values on their left and bottom interfaces.

**Theorem 21.** The execution of CheckTrace always terminates and at the end of execution all expressions on the top interface of the head node will be evaluated to a certain value, i.e., the head node is ready.

**Proof.** There are two exit points of the algorithm, lines 7 and 12:

(i) The algorithm terminates at line 7 if RecProp returns \(\top\); based on the definition of RecProp it is easy to see that RecProp may only return \(\top\) if the head node is ready.

(ii) The algorithm terminates at line 12 after having built a chain of \(n\) nodes and having called TermTrace for the last node. As proven above (Thm. 13) CheckTrace calls RecProp on propagated nodes, hence all nodes in the chain were propagated before calling TermTrace. Since
the execution of CheckTrace reached this point it means that there is at least one expression on the top interface of the head node that was not evaluated to a certain value and this also implies that (Thm. 20) there are uncertain values on top and bottom interfaces of all nodes in the chain (before calling TermTrace). TermTrace fills the bottom interface of the last node in the chain with certain values ($\bot$ values), hence RecProp will be able to evaluate all expressions on the top interface of the last node, propagate these values to the bottom interface of the last but one node, filling it with certain values; evaluates all nodes on the top interface of the last but one node, propagates these values etc... It is easy to see that this call for RecProp will be able to walk through the entire chain and finally evaluate all expressions on the top interface of the head node.

**Theorem 22.** The execution time of CheckTrace is linear with respect to the length of the trace i.e., for a trace of $n$ steps the algorithmic complexity is $O(n)$.

*Proof.* It is easy to see that (i) CheckTrace calls RecProp at most $n$ times; (ii) RecProp starts a recursive call only if there are new values to be propagated i.e., an expression on the top interface was evaluated to a certain value; obviously the number of expressions on the top interface of a node is finite; let $m = \max\{|E| : E \in Drvs(R)\}$ be the maximal number of expressions on the top interface of node types used in the evaluation; this means that RecProp will not process a node more than $n \times m$ times and since all other operations of CheckTrace can be considered constant and $m$ is another constant, the execution time is $O(n)$.

## C.3 An Example for Code Generation

This section presents a discussion of the code generated for evaluating the expression $G(r \rightarrow (p \lor d))$ (the same as used in Chp. 3.5 and Fig. 3.9). The prototype implementation of our code generator supports C++ and Java languages, below we will present C++ source fragments as illustration. An annotated visual overview on the code generation with source fragments is shown in Fig. C.1.

### C.3.1 Evaluation Node Classes and Top Interfaces

As discussed in Chp. 3.5 the expressions on top interfaces of evaluation nodes are as follows:

\begin{equation}
Drvs(T_{\text{Set}}^{\text{CNF}}(\{G(r \rightarrow (p \lor d))\})) = \{T_{\text{Set}}^{\text{CNF}}(\{G(r \rightarrow (p \lor d))\}), T_{\text{Set}}^{\text{CNF}}(\{p \lor d, \top \lor \lnot(p \lor d)\})\}
\end{equation}

The set of expression sets returned by Drvs contains two expression sets: $T_{\text{Set}}^{\text{CNF}}(\{G(r \rightarrow (p \lor d))\})$ and $T_{\text{Set}}^{\text{CNF}}(\{p \lor d, \top \lor (p \land \lnot(p \lor d))\})$. These expression sets define two evaluation node types. The code generator assigns automatically generated unique numeric identifiers to all expressions, expression sets etc. e.g., the identifiers of CNF expressions in this example were as follows (numeric identifiers of top interface expressions are prefixed by TExpr to make them valid C++ identifiers):

\begin{align}
T_{\text{Set}}^{\text{CNF}}(p \lor d) &= \neg(\neg(d) \land \neg(p \land X(p \lor d))) & \text{TExpr6388} \quad (C.4) \\
T_{\text{Set}}^{\text{CNF}}(G(r \rightarrow (p \lor d))) &= (\neg(r) \land \neg(d) \land \neg(p \land X(p \lor d))) \land \neg(\top \land X(\top \lor (r \land \lnot(p \lor d)))) & \text{TExpr55002} \quad (C.5) \\
T_{\text{Set}}^{\text{CNF}}(\top \lor (r \land \lnot(p \lor d))) &= (\neg(r) \land (\neg(d) \land \neg(p \land X(p \lor d))) \land \neg(\top \land X(\top \lor (r \land \lnot(p \lor d)))) & \text{TExpr30491} \quad (C.6)
\end{align}

Identifiers of top interface expressions are implemented as an enumeration as shown in Listing C.1; the type defined by the enum construct is TExpr. The two top interface expression sets specify two evaluation node classes as shown in Fig. C.1; class names are also automatically generated: the top interface of EN47613 contains the single expression $T_{\text{Set}}^{\text{CNF}}(G(r \rightarrow (p \lor d))) =
APPENDIX C. DETAILS ON ERROR DETECTION

C.3.2 Bottom Interfaces and Evaluation Node Base Class

Bottom interfaces of evaluation node classes are constructed by collecting next-time subexpressions as discussed in Chp. 3.3 in our example the next-time subexpressions occurring on the bottom interface of any nodes and the corresponding automatically generated identifiers (prefix by BEXPR) are as follows:

\[
\text{Collect}^{\text{Set}}_{\text{Set}}(\text{To}\text{CNF}(\{ T U(\neg r \land \neg d \land \neg (p \land X(pU d))) \land \neg (T \land X(T U(r \land \neg (pU d)))) \})) = \{ BEXPR60447, BEXPR57000 \}.
\]  

Identifiers of bottom interface expressions are implemented as an enumeration as shown in Listing C.3, the type defined by the enum construct is BExpr. (The identifiers of various expressions are indicated in callouts of the corresponding color in Fig. C.1.) For simplicity reasons interfaces are implemented in the abstract base class EN as arrays of binary and ternary types (Listing C.4): top, left and bottom interfaces are represented by members t, l and b respectively; the reference to the previous node is implemented by the pointer p, functions Head, Next, IntProp, ExtProp, TermTrace and
C.3. AN EXAMPLE FOR CODE GENERATION

/* *
* Enumerated type specifying identifiers for top interface expressions.
*/
typedef enum {
    /* * Identifier of top interface expression : !(!d AND !(p AND X(p U d))). */
    TEXPRESS86,
    /* * Identifier of top interface expression : !((r AND (!d AND !(p AND X(p U d)))) AND !(true AND X(true U (r AND !(p U d))))) . */
    TEXPRESS0491,
    /* * Identifier of top interface expression : (!(r AND (!d AND !(p AND X(p U d)))) AND !(true AND X(true U (r AND !(p U d))))) . */
    TEXPRESS56002,
    /* * Number of top interface expressions. */
    NUM_TEXPR
} TExpr;

Listing C.1: Enumeration of Top Interface Expression Identifiers

IsReady are mapped to functions head, next, intProp, extProp, termTrace and isReady respectively. As the head node of the chain is always of the same type, head is a static function, the other ones are virtual.

C.3.3 Head and Next Nodes

The Head utility is implemented as a static member function of the EN abstract base class (Listing C.5). Obviously head returns a newly created instance of EN12544 (also see the corresponding callout in Fig. C.1). (The constructors of node classes have two parameters: (i) values of left interface expressions for initialization and (ii) the address of the previous node in the chain i.e., the value of p; for the first node this pointer is obviously null.)

As the type of the next node in the chain depends on the type of the actual node, the utility Next is implemented as virtual member functions in corresponding classes. Since in our case the subsequent node after both EN47613 and EN12544 are of type EN47613 both implementations return a newly constructed instance of EN47613 (Listing C.6). (Also see the corresponding callouts in Fig. C.1)

C.3.4 Evaluation of Expressions in Ternary Logic

The method for evaluation of expressions in ternary logic as defined in Chp. 3.5 can be easily mapped to some functions; unfortunately as the performance of expression evaluation is of primary importance in the implementation, the direct translation of the abstract semantics to C++ would introduce considerable overhead. In order to provide very high performance evaluation the evaluation of ternary expressions is divided into two parts: (i) we can easily check whether the expression is certainly true and (ii) we can check whether the expression is certainly false, in correspondence to the truth table of ternary operators shown in Tab. 3.3. The code generator synthesizes C preprocessor macros for these partial evaluations as shown for the expression ToCNF(\(\top \cup (r \land \neg(p \cup d))\)) = \(\neg(\neg(r \land \neg d \land (p \land X(p \cup d))) \land \neg(\neg(r \land \neg d \land (p \cup d)))\)) (identified by TEXPR0491) in Listing C.7 the implementation uses the get method for obtaining values of expressions on interfaces; bottom interface expressions are referenced by their BExpr identifiers, left interface expressions are considered to be identified by plain letters starting with a as used in the example, e.g., the call l.get('r' - 'a') obtains the value of r from the left interface.

The implementation of IntProp is based on using the “certainly true” and “certainly false” split approach mentioned above. The automatically generated intProp virtual method for the class EN47613 in shown in Listing C.8 It is easy to see that the body of the function tries to evaluate the two top interface expressions identified by TEXPR0388 and TEXPR0491 if any of them is still uncertain; if a
C.3.5  External Propagation

The implementation of ExtProp in case of the evaluation node class EN47613 is shown in Listing C.9: it is easy to see that evaluation nodes of the class EN47613 expect the values of top interface expressions TEXPR6388 and TEXPR30491 from the next node in the chain to be propagated to their bottom interface expressions BEXPR40447 and BEXPR57000 respectively (also see the corresponding callout in Fig. C.1).

The implementation of TermTrace in case of the evaluation node class EN47613 is shown in Listing C.10: the function sets the values of bottom interface expressions BEXPR40447 and BEXPR57000 to false (also see the corresponding callout in Fig. C.1).

The implementation of TermTrace in case of the evaluation node class EN47613 is shown in Listing C.10: the function returns true if both top interface expressions TEXPR6388 and TEXPR30491 are evaluated to a certain value (i.e., not the uncertain ternary value); Fig. C.1 presents another example for the case of the EN12544 class.
C.3. AN EXAMPLE FOR CODE GENERATION

```
/** *
 * Abstract base class for evaluation nodes.
 */

class EN {
    protected:
        /** Top interface */
        TernaryArray<NUM_TEXPR> t;
    /** Left interface */
        BinaryArray <NUM_LEXPR> l;
    /** Bottom interface */
        TernaryArray<NUM_BEXPR> b;
    /** Previous node */
        EN *p;

    static inline EN* head(BinaryArray<NUM_LEXPR> & l);
    virtual EN* next(BinaryArray<NUM_LEXPR> & l) = 0;
    virtual BinaryArray<NUM_TEXPR> intProp() = 0;
    virtual void extProp(TExpr tExpr, Ternary value) = 0;
    virtual void termTrace() = 0;

    // ...
};
```

Listing C.4: The Evaluation Node Base Class (Fragment)

```
EN *
EN::head(BinaryArray<NUM_LEXPR> & l) {
    return new EN12544(l, 0);
}
```

Listing C.5: Implementation of the Head Method

C.3.6 Top-Level Algorithms

The function RecProp is implemented in a non-recursive organization as a static member function of the EN abstract base class as shown in Listing C.12. CheckTrace is implemented as a stand-alone function as shown in Listing C.13. It is easy to see that the two source fragments are nearly the direct syntactic rewriting of pseudocodes (Listing 3.1 and Listing 3.2 respectively) to C++. 

Listing C.6: Implementation of the Next Method in Evaluation Node Classes EN47613 and EN12544

Listing C.7: Evaluation of Top Interface Expressions in Ternary Logic

Listing C.8: Implementation of the IntProp Method in Evaluation Node Class EN47613
C.3. AN EXAMPLE FOR CODE GENERATION

```cpp
1 class EN47613 : public EN {
2   // ...
3   /** Propagates the value of an expression from the top interface of the next node to the
4      bottom interface of the current node. */
5   virtual void
6   extProp(TExpr tExpr, Ternary value) {
7     switch (tExpr) {
8       case TEXPR6388:
9         b.set(BEXPR40447, value);
10        break;
11       case TEXPR30491:
12         b.set(BEXPR57000, value);
13        break;
14       default:
15         assert(0);
16       }
17     // ...
18 }
19 }
20
```

Listing C.9: Implementation of the `ExtProp` Method in Evaluation Node Class `EN47613`

```cpp
1 class EN47613 : public EN {
2   // ...
3   /** Sets the value of all expression on the bottom interface to TFalse. */
4   virtual void
5   termTrace () {
6     b.set(BEXPR57000, BFalse);
7     b.set(BEXPR40447, BFalse);
8   }
9   // ...
10 }
```

Listing C.10: Implementation of the `TermTrace` Method in Evaluation Node Class `EN47613`

```cpp
1 class EN47613 : public EN {
2   // ...
3   /** Decides whether all expressions in the top interface were evaluated. */
4   virtual Binary
5   isReady () {
6     if ((TUncertain != t.get(TEXPR6388)) && (TUncertain != t.get(TEXPR30491)) && true)
7        return BTrue;
8     else
9        return BFalse;
10   }
11   // ...
12 }
```

class EN {
    // ...
    inline static Binary recProp(EN *en) {
        while (true) {
            // Evaluate expressions on the top interface where possible.
            BinaryArray<NUM_TEXPR> updated = en -> intProp();

            // If this is the head instance in the chain...
            if (0 == en -> p) {
                // ...return the readiness of the head instance.
                return en -> isReady();
            } else {
                // ...otherwise...
                // ...if some expressions on the top interface were made certain...
                if (updated.toInt()) {
                    // ...propagate these new values to the previous instance and...
                    for (unsigned i = 0; i < NUM_TEXPR; i++)
                        if (updated.get(i))
                            en -> p -> extProp(static_cast<TExpr>(i), static_cast<Ternary>(en -> t.get(i)));
                    // ...recursively continue the operation on the previous instance.
                    en = en -> p;
                } else {
                    // If there is nothing to propagate, return false.
                    return BFalse;
                }
            }
        }
    }
    // ...
}

Listing C.12: Implementation of the RecProp Method
C.3. AN EXAMPLE FOR CODE GENERATION

```cpp
void checkTrace(vector<unsigned> trace) {
    BinaryArray[NUM_LEXPR] traceHead(trace[0]);

    // The head of the evaluation chain
    EN *enHead = EN::head(traceHead);

    // Variable pointing to the actually last element in the chain
    EN *enTail = enHead;

    // Trace suffix index
    unsigned i = 1;

    // Length of the trace
    unsigned n = trace.size();

    while (i < n)
        if (true == (EN::recProp(enTail))) {
            cout << "Decision after element number " << i << endl <<
                 enHead->toString() << endl;
            return;
        }
        else {
            BinaryArray[NUM_LEXPR] traceElement(trace[i]);
            enTail = enTail->next(traceElement);
            i += 1;
        }

    enTail->termTrace();
    assert(EN::recProp(enTail));
    cout << "Decision after processing the entire trace."
         << endl <<
         enHead->toString() << endl;
}
```

Listing C.13: Implementation of the `CheckTrace` Method
Bibliography


[47] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns Elements of Reusable Object-Oriented Software. Addison Wesley, 1994.


[54] Jiawei Han and Micheline Kamber. Data Mining: Concepts and Techniques. Morgan Kaufmann Publisher, 2001.


[84] Institut für Informatik Lehr- und Forschungseinheit für Programmierung und Softwaretechnik Ludwig-Maximilians-Universität, München. Hugo/RT. [http://www.pst.informatik.uni-muenchen.de/projekte/hugo/](http://www.pst.informatik.uni-muenchen.de/projekte/hugo/)


