Towards Model-Driven Engineering of Reliable Systems

Developing Fault-Tolerant Systems using Scalable Verification

Doctoral thesis
for the degree of Philosophiae Doctor

Trondheim, September 2014

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Telematics
Abstract

As many of us already depend on computer systems to lead our lives to a standard we find acceptable, we put a correspondingly strong emphasis on ensuring that these systems are indeed dependable. Of the various attributes that make up the system dependability, this work focuses on reliability, the ability of a system to provide continuous service. One of the main barriers to increasing the reliability of a system is that it requires special competences that most developers do not possess. Hence, many developers are faced with either spending resources on additional training or hiring external experts, who may know little of the system in question and its problem domain. As a result, increasing the reliability of a computer system can increase the acquisition cost so much that the system is built without a cost-effective level of reliability, when considering costs over its total life span. Another barrier is that a naive introduction of fault-tolerance mechanisms may actually worsen the reliability of a system. Introducing fault-tolerance mechanisms is sure to add to the complexity of the system. This added complexity, if not properly managed, is likely to result in a system with more faults, possibly contributing to a net decrease in the system reliability. Due to such barriers, computer systems often fail to meet our expectations in terms of reliability, and we as a society frequently pay a higher than necessary price when suffering the consequences of system failures.

This motivates our decision to work towards the goal of reducing the development effort and competence required to create reliable distributed reactive systems. Model-driven software engineering aids developers in breaking down complex systems by abstracting away from the implementation details and often provides viewpoints capturing one concern at a time. We therefore build on the existing model-driven engineering method SPACE. The abstractions provided by model-driven engineering also facilitate automated model checking to detect software faults. We take advantage of this to provide highly automated fault-removal features.
Even a system completely free from software faults will be affected by external faults, such as power failures and the breakdown of hardware components, if the system is not designed to tolerate them. We therefore focus on adding fault-tolerance mechanisms to systems, so that they can be relied upon also in a less than ideal operational environment. To easily ensure such mechanisms are correctly built and integrated, we again look to model checking.

Going into more detail, the first contribution described is an automatic transformation of SPACE models into a formal system specification along with properties to be verified, complete with visualization of erroneous behaviour and, in some cases, automatic diagnoses and fixes. Further, the work herein contributes new capabilities to build services tolerating message losses or process crashes, as well as a two-phase development method allowing developers to start with a simpler system and deal with the full complexity of fault tolerance afterwards. Finally, we introduce additional concepts for interface contracts allowing to compositionally verify properties of fault-tolerant services, even if internally replicated across several components.
Preface

This thesis is submitted to the Norwegian University of Science and Technology (NTNU) in partial fulfilment of the requirements for the degree of Philosophiae Doctor (PhD). The work has been performed at the Department of Telematics, NTNU, Trondheim, with Prof. Peter Herrmann as main supervisor and Prof. Rolv Bræk as co-supervisor.

Acknowledgements

I would like to thank Peter, my main supervisor, and Frank, colleague and co-author on most papers, for all the countless hours you have spent working together with me from the very start. While the German communication style was a bit intimidating at first, I have found it to be both efficient and honest. Thanks for the enlightening discussions we have had, the advice you have given and for being patient when I had little motivation left for this thesis.

I also want to thank Rolv, my co-supervisor, for good advice at the start of the PhD and for further insightful comments on my work, the times I asked for it. I probably should have asked for it more often.

A big thanks also to all my colleagues at the department for interesting lunch conversations and fun at social activities. A special thanks to Randi, Mona and Pål, for the good help with the various tasks that are by-products of writing a PhD. Further thanks to my colleagues in the Networked Systems group, providing stimulating discussions and interesting presentations on the topic. A special thanks to Máté, my office mate for most of the time, for being an example of good working discipline, yet a good friend to discuss non-work topics with, and for suggesting the office plant competition (mine is still alive!). I would also like to thank Linda, who started shortly after
me, for being that someone in the same situation that I could share both joys and frustrations with.

Thanks also goes out to the dinner gang, Harald, Fritjof, Tord, Laurent and Frank (again). Without some good dinner company, allowing to put in some extra evening hours, this thesis would be even more delayed.

Most of all, I would like to thank Anne-Lise, my dear wife, for sticking with me through the ups and downs of the last six years. I realize that “thought about stuff, wrote some of it down” is not a sufficient answer to “what did you do at work today?”, in the long run. Thanks for trusting me anyway, or rather, having faith in me, as it was often belief without evidence. I also want to thank Marianne and Arvid, my parents. You are wonderfully kind people, who give me an ingrained sense of security in my life, best explained as the feeling that there is a rescue net below me at all times. An example is how you always said “we don’t care whether you finish the thesis”. While it may not have driven me faster towards completion, this helped relive stress when it threatened to cause me to abort altogether.
Included Publications

**Paper 1** Model-Driven Engineering of Reliable Fault-Tolerant Systems – A State-of-the-Art Survey

**Paper 2** Engineering Support for UML Activities by Automated Model-Checking — An Example

**Paper 3** Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services

**Paper 4** Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services

**Paper 5** Towards a Model-Driven Method for Reliable Applications: From Ideal To Realistic Transmission Semantics

**Paper 6** Contracts for Multi-instance UML Activities

**Paper 7** Towards Automatic Generation of Formal Specifications to Validate and Verify Reliable Distributed Systems: A Method Exemplified

v
by an Industrial Case Study
Vidar Slåtten, Frank Alexander Kraemer and Peter Herrmann. Proceedings of the
10th ACM International Conference on Generative Programming and Component
Engineering (GPCE’11), ACM, p. 147–156, 2011.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>i</td>
</tr>
<tr>
<td>Preface</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>iii</td>
</tr>
<tr>
<td>Included Publications</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xiii</td>
</tr>
<tr>
<td><strong>I  Thesis Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>3</td>
</tr>
<tr>
<td>1.1 Scope and Goal</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Research Context</td>
<td>6</td>
</tr>
<tr>
<td>1.3 Research Questions</td>
<td>6</td>
</tr>
<tr>
<td>1.4 Contributions</td>
<td>7</td>
</tr>
<tr>
<td>1.5 Outline</td>
<td>8</td>
</tr>
<tr>
<td>2 Paper 1</td>
<td>11</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>13</td>
</tr>
<tr>
<td>2.2 Background and Scope</td>
<td>16</td>
</tr>
<tr>
<td>2.2.1 Models for Software Development</td>
<td>16</td>
</tr>
<tr>
<td>2.2.2 Dependability Concepts</td>
<td>21</td>
</tr>
<tr>
<td>2.2.3 Fault Tolerance</td>
<td>23</td>
</tr>
<tr>
<td>2.2.4 Fault Removal</td>
<td>26</td>
</tr>
<tr>
<td>2.2.5 Models of System Semantics</td>
<td>30</td>
</tr>
<tr>
<td>2.3 Characteristics to Classify Approaches</td>
<td>33</td>
</tr>
<tr>
<td>2.4 Approaches with Verification of User-Designed/ Integrated Fault-Tolerance Mechanisms</td>
<td>37</td>
</tr>
<tr>
<td>2.4.1 Extending Charmy for Fault Tolerance</td>
<td>37</td>
</tr>
<tr>
<td>2.4.2 SPACE</td>
<td>39</td>
</tr>
<tr>
<td>2.4.3 Event B</td>
<td>41</td>
</tr>
<tr>
<td>2.4.4 Mechatronic UML – FUJABA Real-Time Tool Suite</td>
<td>43</td>
</tr>
</tbody>
</table>
2.4.5 iFTElement ................................................. 45
2.4.6 Motorola WEAVR ......................................... 49
2.4.7 MOCAS ...................................................... 53
2.4.8 rCOS ......................................................... 56
2.5 Approaches Tied to Specific Fault-Tolerance Mechanisms .......... 59
2.5.1 GeneRative Aspects for Fault Tolerance (GRAFT) .............. 59
2.5.2 Aurora Management Workbench (AMW) ......................... 62
2.6 Comparison Tables ........................................... 65
2.7 Discussion .................................................... 67
2.7.1 Survey Results and Direction of the Field ....................... 67
2.7.2 About our Survey ........................................ 78
2.8 Conclusion ................................................................ 79

References ........................................................................ 80

3 Context and Research Design ........................................ 97
3.1 Starting Point of the Work ........................................... 97
3.2 Research Questions Revisited ...................................... 99

4 Summary of Papers .................................................... 103

5 Discussion ............................................................ 109
5.1 Answers to Research Questions ................................... 109
5.2 Status of Tool Support ............................................. 112
5.3 Further work ....................................................... 113
5.3.1 Further Practical Validation of Method and Tools ............ 113
5.3.2 Deployment Model ............................................. 113
5.3.3 Model-Based Testing ......................................... 113
5.3.4 Implement a Consensus Algorithm in SPACE ................... 114
5.3.5 Building Blocks for Middleware ................................. 114
5.3.6 Model-Based Specification of Application-Specific Properties .................................................. 114
5.4 Summary .................................................................. 114

Bibliography ..................................................................... 117

II Included Papers ......................................................... 121

6 Paper 2 .................................................................. 125
6.1 Introduction .......................................................... 127
6.2 Related Work ....................................................... 129
6.3 The Temporal Logic of Actions ................................... 130
6.4 UML 2.0 Activities in the SPACE Approach ..................... 132
6.4.1 Informal Explanation of Activities ................................. 133
6.4.2 Semantics of UML 2.0 Activities in Temporal Logic ............ 134
6.4.3 Theorems for Well-Formed Activities ............................ 135
6.5 Developing and Model Checking the Example
   6.5.1 Solution 1: A Naive Start ........................................ 136
   6.5.2 Solution 2: Improved Version with a Sequencer .............. 137
   6.5.3 Solution 3: A Building Block for Mixed Initiatives ........ 139
6.6 Concluding Remarks .................................................. 140
References ............................................................................ 141

7 Paper 3 .............................................................................. 145
  7.1 Introduction ..................................................................... 147
  7.2 Composing Services from Building Blocks ....................... 149
     7.2.1 Elementary Building Blocks ..................................... 151
     7.2.2 Composing Building Blocks .................................... 153
  7.3 Automated Model Checking and Analysis ......................... 153
     7.3.1 Semantics in Temporal Logic ................................. 155
     7.3.2 Theorems for Correct Building Blocks .................... 157
     7.3.3 Error Trace Animation in Arctis ............................ 157
     7.3.4 Automatic Diagnose and Fixes ............................... 159
     7.3.5 A Building Block to Handle Mixed Initiatives ........... 160
     7.3.6 The Complete Treasure Hunt System ....................... 162
     7.3.7 Scalability of the Analysis .................................... 163
  7.4 Automated Transformation ............................................ 164
     7.4.1 Scalability of the Transformation ............................ 165
     7.4.2 State Machine for the Game Server ....................... 165
     7.4.3 Correctness of the Transformation .......................... 166
  7.5 Code Generation from State Machines ............................. 167
  7.6 Related Approaches ..................................................... 167
  7.7 Concluding Remarks .................................................... 169
References ............................................................................ 170

8 Paper 4 .............................................................................. 177
  8.1 Introduction ................................................................. 179
     8.1.1 Embedded Java on Sun SPOTs ............................... 180
     8.1.2 Runtime Support System ....................................... 180
     8.1.3 The SPACE Engineering Method ............................ 181
  8.2 A Sensor Network for Remote Home Monitoring ............... 182
  8.3 Building Blocks Specific for Sun SPOTs ......................... 184
     8.3.1 Building Block for Sensors .................................. 185
     8.3.2 SPOT Discovery ................................................... 186
  8.4 Collaborative Building Blocks for Leader Election .......... 187
     8.4.1 Infinitely Often Accurate Detector (IOD) ................. 188
     8.4.2 Composed Building Block for the Leader Election ... 188
  8.5 Automated Analysis ...................................................... 190
  8.6 Automated Implementation ........................................... 191
     8.6.1 Transformation to Executable State Machines .......... 191
     8.6.2 Code Generation for Sun SPOTs ........................... 193
  8.7 Estimation of Reuse Proportions ................................... 194
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.8</td>
<td>Related Work</td>
<td>195</td>
</tr>
<tr>
<td>8.9</td>
<td>Concluding Remarks</td>
<td>195</td>
</tr>
<tr>
<td>References</td>
<td>196</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Paper 5</td>
<td>199</td>
</tr>
<tr>
<td>9.1</td>
<td>Introduction</td>
<td>201</td>
</tr>
<tr>
<td>9.2</td>
<td>Idealized Application Models</td>
<td>203</td>
</tr>
<tr>
<td>9.3</td>
<td>Transmission Semantics</td>
<td>206</td>
</tr>
<tr>
<td>9.4</td>
<td>Reliable Application Models</td>
<td>207</td>
</tr>
<tr>
<td>9.4.1</td>
<td>Reliable Notify</td>
<td>207</td>
</tr>
<tr>
<td>9.4.2</td>
<td>Reliable Authenticate</td>
<td>208</td>
</tr>
<tr>
<td>9.4.3</td>
<td>Reliable Inquiry</td>
<td>210</td>
</tr>
<tr>
<td>9.4.4</td>
<td>Reliable Access Control System</td>
<td>211</td>
</tr>
<tr>
<td>9.5</td>
<td>Related Work</td>
<td>212</td>
</tr>
<tr>
<td>9.6</td>
<td>Discussion</td>
<td>215</td>
</tr>
<tr>
<td>9.7</td>
<td>Concluding Remarks</td>
<td>218</td>
</tr>
<tr>
<td>9.8</td>
<td>Acknowledgements</td>
<td>220</td>
</tr>
<tr>
<td>References</td>
<td>220</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Paper 6</td>
<td>225</td>
</tr>
<tr>
<td>10.1</td>
<td>Introduction</td>
<td>227</td>
</tr>
<tr>
<td>10.2</td>
<td>Related Work</td>
<td>228</td>
</tr>
<tr>
<td>10.3</td>
<td>A Load Balancing Client – Server Example</td>
<td>229</td>
</tr>
<tr>
<td>10.4</td>
<td>Contracts for Multi-instance Activities</td>
<td>232</td>
</tr>
<tr>
<td>10.5</td>
<td>Other Types of Multiplicity</td>
<td>237</td>
</tr>
<tr>
<td>10.6</td>
<td>Concluding Remarks</td>
<td>239</td>
</tr>
<tr>
<td>References</td>
<td>241</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Paper 7</td>
<td>243</td>
</tr>
<tr>
<td>11.1</td>
<td>Introduction</td>
<td>245</td>
</tr>
<tr>
<td>11.2</td>
<td>Texi Case Study: Lockers that Read RFID</td>
<td>247</td>
</tr>
<tr>
<td>11.2.1</td>
<td>System Requirements</td>
<td>248</td>
</tr>
<tr>
<td>11.3</td>
<td>The Method for Reliable Systems</td>
<td>248</td>
</tr>
<tr>
<td>11.3.1</td>
<td>System Design in SPACE</td>
<td>251</td>
</tr>
<tr>
<td>11.3.2</td>
<td>Analysis A2 of Timed Mutex Local</td>
<td>255</td>
</tr>
<tr>
<td>11.4</td>
<td>Modelling Realistic Semantics</td>
<td>259</td>
</tr>
<tr>
<td>11.5</td>
<td>Discussion</td>
<td>262</td>
</tr>
<tr>
<td>11.6</td>
<td>Related Work</td>
<td>265</td>
</tr>
<tr>
<td>11.7</td>
<td>Concluding Remarks</td>
<td>266</td>
</tr>
<tr>
<td>References</td>
<td>267</td>
<td></td>
</tr>
<tr>
<td>III</td>
<td>Appendices</td>
<td>271</td>
</tr>
<tr>
<td>A</td>
<td>Secondary Papers</td>
<td>273</td>
</tr>
<tr>
<td>A.1</td>
<td>Paper 8</td>
<td>274</td>
</tr>
</tbody>
</table>
A.2 Paper 9 .................................................. 275
List of Figures

1.1 Focus areas of each paper ............................................. 8
2.1 Example illustrating four viewpoints using different representations ........................................................................ 18
6.1 Tool Support for the SPACE engineering approach .................................................. 128
6.2 TLA Module .................................................................................. 131
6.3 Reception Panel .............................................................................. 132
6.4 Activity for the entire system ............................................................. 132
6.5 External state machines ...................................................................... 133
6.6 Solution 1 ....................................................................................... 134
6.7 Error trace of solution 1 ..................................................................... 137
6.8 Solution 2 ....................................................................................... 138
6.9 Error trace of solution 2 ..................................................................... 138
6.10 Correct solution with a building block to handle mixed initiatives .................................................................... 139
6.11 ESM for Mixed Initiative Secondary Starter .............................................. 140
7.1 The SPACE Engineering Method ......................................................... 148
7.2 Overview of the tool support ................................................................. 149
7.3 Eclipse workbench with the Arctis library browser and editor ................... 150
7.4 Building blocks provided by a network operator ......................................... 151
7.5 First solution of the proximity alert ......................................................... 154
7.6 TLA+ module for the semantics of ProximityAlert ......................................... 155
7.7 Simplified Proximity Alert without data .................................................. 156
7.8 Visualization of TLC’s error trace in Arctis .............................................. 158
7.9 Suggested improvement by Arctis with sequencing .................................. 160
7.10 New error situation in the altered design ................................................ 161
7.11 Building block to handle mixed initiatives with local ESMs for both ......................................................... 161
7.12 Correct Proximity Alert with the Mixed Initiative Building Block ................. 162
7.13 Complete system specification of the treasure hunt ................................ 163
7.14 Executable state machine for the game server generated by Arctis ........... 166
8.1 Sun SPOT and Runtime Support System .................................................... 180
8.2 The SPACE engineering method .......................................................... 181
<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.3</td>
<td>Sensor network with the elected leader</td>
<td>182</td>
</tr>
<tr>
<td>8.4</td>
<td>Activity describing the composition of SPOT sensors from building</td>
<td>183</td>
</tr>
<tr>
<td>8.5</td>
<td>Overview of reused block from libraries and application-specific blocks</td>
<td>185</td>
</tr>
<tr>
<td>8.6</td>
<td>Building block for the motion sensor</td>
<td>185</td>
</tr>
<tr>
<td>8.7</td>
<td>Building block for the service discovery</td>
<td>186</td>
</tr>
<tr>
<td>8.8</td>
<td>Collaboration for the leader election</td>
<td>186</td>
</tr>
<tr>
<td>8.9</td>
<td>Building block for the Infinitely Often Accurate Detector</td>
<td>187</td>
</tr>
<tr>
<td>8.10</td>
<td>Building block for leader election</td>
<td>189</td>
</tr>
<tr>
<td>8.11</td>
<td>Bird’s eye view of the synthesized state machine for the Spot Sensor</td>
<td>192</td>
</tr>
<tr>
<td>8.12</td>
<td>The synthesized state machines for the IO detector</td>
<td>193</td>
</tr>
<tr>
<td>9.1</td>
<td>Development Method</td>
<td>202</td>
</tr>
<tr>
<td>9.2</td>
<td>Access Control System</td>
<td>204</td>
</tr>
<tr>
<td>9.3</td>
<td>Behaviour of the Authenticate building block</td>
<td>205</td>
</tr>
<tr>
<td>9.4</td>
<td>External descriptions of the Panel Control and Door Control building blocks</td>
<td>205</td>
</tr>
<tr>
<td>9.5</td>
<td>Implicit waiting place for transmissions</td>
<td>206</td>
</tr>
<tr>
<td>9.6</td>
<td>Ideal transmission semantics</td>
<td>206</td>
</tr>
<tr>
<td>9.7</td>
<td>Realistic transmission semantics</td>
<td>207</td>
</tr>
<tr>
<td>9.8</td>
<td>Behaviour of the Notify R building block</td>
<td>208</td>
</tr>
<tr>
<td>9.9</td>
<td>ESM of the Notify R building block</td>
<td>209</td>
</tr>
<tr>
<td>9.10</td>
<td>The Authenticate building block after syntactic substitution of top-most flow</td>
<td>209</td>
</tr>
<tr>
<td>9.11</td>
<td>The reliable Authenticate building block completed</td>
<td>210</td>
</tr>
<tr>
<td>9.12</td>
<td>Behaviour of the Inquiry R building block</td>
<td>210</td>
</tr>
<tr>
<td>9.13</td>
<td>ESM of the Inquiry R building block</td>
<td>211</td>
</tr>
<tr>
<td>9.14</td>
<td>Reliable Access Control System</td>
<td>212</td>
</tr>
<tr>
<td>9.15</td>
<td>Idealized executable state machine for central station component</td>
<td>216</td>
</tr>
<tr>
<td>9.16</td>
<td>Reliable executable state machine for central station component</td>
<td>217</td>
</tr>
<tr>
<td>9.17</td>
<td>An alternative reliable Authenticate building block using middleware exceptions</td>
<td>218</td>
</tr>
<tr>
<td>10.1</td>
<td>System example: A load balancing client – server system</td>
<td>230</td>
</tr>
<tr>
<td>10.2</td>
<td>Contracts for the Make Response block</td>
<td>232</td>
</tr>
<tr>
<td>10.3</td>
<td>TLA⁺ module for the EESM of Router m-n</td>
<td>234</td>
</tr>
<tr>
<td>10.4</td>
<td>UML notation for the EESM of Router m-n</td>
<td>235</td>
</tr>
<tr>
<td>10.5</td>
<td>TLA⁺ module for the system activity</td>
<td>236</td>
</tr>
<tr>
<td>10.6</td>
<td>The two other variants of the router activity, with respect to select statements</td>
<td>238</td>
</tr>
<tr>
<td>10.7</td>
<td>UML notation for the EESM of Router 1-1</td>
<td>238</td>
</tr>
<tr>
<td>10.8</td>
<td>UML notation for the ESM of Router 1-n</td>
<td>239</td>
</tr>
<tr>
<td>11.1</td>
<td>The development method</td>
<td>250</td>
</tr>
<tr>
<td>11.2</td>
<td>UML activity showing the mutual exclusion protocol</td>
<td>252</td>
</tr>
<tr>
<td>11.3</td>
<td>ESM of the Timer block</td>
<td>252</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>11.4</td>
<td>EESM of the Timed Mutex Local block</td>
<td>253</td>
</tr>
<tr>
<td>11.5</td>
<td>EESM of the Timed Mutex Distributed block</td>
<td>254</td>
</tr>
<tr>
<td>11.6</td>
<td>UML activity showing the part of the mutual exclusion protocol that is local to the controller</td>
<td>255</td>
</tr>
<tr>
<td>11.7</td>
<td>TLA+ excerpt for the proof of the Timed Mutex Local block</td>
<td>257</td>
</tr>
<tr>
<td>11.8</td>
<td>The new and improved activity of Timed Mutex Local</td>
<td>259</td>
</tr>
<tr>
<td>11.9</td>
<td>ERC of the Timed Mutex Local block</td>
<td>260</td>
</tr>
</tbody>
</table>
Part I

Thesis Introduction
Chapter 1

Introduction

Since computer systems, or more precisely software and software-intensive systems, provide services that most of us depend on in our everyday lives, we think it important to ensure that such systems are indeed dependable. In [Avi+04], one of the definitions of the dependability of a system is “the ability to avoid service failures that are more frequent and more severe than is acceptable”, a service failure being a deviation in the system behaviour that is visible to its users.

Service failures can range from being just annoying, like your web browser crashing, to catastrophic failures with loss of lives and/or property as a consequence, like in the case of an incorrect railway signal causing one train to crash into another. While the latter example is certainly a dependability failure [Avi+04] by our definition of dependability, the web browser crashing is probably not, especially if it saved all your open tabs.

Depending on the likelihood of faults and the consequences of service failures, some share of the development resources should go towards ensuring the dependability of any system being developed. However, increasing the dependability of a system always comes at a cost, in terms of the effort and competence required of the developers. [Hel09] states that “In high dependability systems, the additional investment in extra hardware and in development of fault handling, maintenance support etc., may be larger than the cost [of] the development of a system with “ordinary” dependability”. He then adds that for a balanced application of dependability means, this acquisition cost may be more than offset by lower operation costs over the system life cycle. The same argument is put forward by Boehm and Basili in [BB01].

A typical way of improving the dependability of a system is to add fault-tolerance mechanisms so as to avoid service failures even in the presence of active faults. Knowledge in both the application and the dependability domain is required of the

---

1An active fault is a fault that produces an error [Avi+04]. See also Sect. 2.2.2
developers to correctly implement this. And in so doing, the system is likely to become more complex, increasing the chance of software faults being introduced during development or later maintenance \cite{FB08}. This is especially true when fault tolerance is an afterthought to an already implemented system. As a result, adding fault tolerance to improve dependability can, in fact, reduce it. Romanovsky \cite{Rom07} gives some examples of this, for example how a big electrical power outage in 2003, in the US and Canada, was caused primarily by poorly designed fault-tolerance mechanisms. Romanovsky believes we are heading for a crisis within the field of fault-tolerant software and describes the current state of affairs as follows:

“At present, fault tolerance is not trustworthy as it is the least understood, documented and tested part of the system, is frequently misused or poorly designed, regularly left until too late in the development process, not typically introduced in a systematic, disciplined or rigorous way, and often not suitable for the specific situations in which it is applied.” \cite{Rom07}

In \cite{Pel07}, Pelliccione et al. argue that fault tolerance needs to be given a more prominent place in the field of software engineering and be included in all steps of development, including the early phases, so as to explicitly model faults and fault-tolerance mechanisms.

All in all, we find that it is currently very hard to create dependable systems. The question at hand is then: \textit{How can we create more dependable systems without paying too high a price in terms of effort and competence required?}

\section{1.1 Scope and Goal}

As we will later see, the term dependability encompasses a range of attributes. Of these, the ability to provide continuous service, reliability, is the main target of this work.

Further, we concern ourselves with distributed reactive systems. Reactive systems are systems that are maintaining an ongoing interaction with their environment \cite{Pnu86}. This as opposed to a transformational system whose task is to produce an output from an input given at initialization and then terminate. We emphasize the focus on distributed systems, both because we see single-node systems as just a special case of distributed systems, and because distribution is often necessary to achieve the level of redundancy needed to create truly reliable systems.

As is described in \textit{Paper 1}, dependability can be improved by the means of fault prevention, fault removal, fault tolerance and fault forecasting. This work focuses on removing as many faults as possible at design time and enabling the system to tolerate faults that occur when it is operational. Further, we narrow our scope
to fault-tolerance mechanisms that can be implemented in software, using only off-the-shelf hardware. Our scope can also be limited by looking at what kind of faults we expect to deal with. [Avi+04] classifies faults by eight viewpoints. One of these is the objective, whether the fault is malicious or not. Protecting against a malicious attacker is outside the scope of this work.

In most of the following we will use the term reliable systems to mean systems both with few or no design faults, as well as a capacity to tolerate operational, non-malicious faults. We acknowledge the need to also focus on fault prevention and fault forecasting, but see it as much too wide a scope to also attempt to make contributions to those areas.

To sum up, we would say that the goal of this work is to reduce the development effort and competence required to create reliable, distributed, reactive systems.

Having reduced the scope of the problem, we now turn to the scope of the solution. For distributed reactive systems, many faults are the kind that stem from mixed initiatives [BH93 Floo3], i.e., multiple concurrent initiatives in the system leading to situations where the behaviour changes depending on the precise timing of competing events in relation to each other. Finding all such faults is difficult using verification techniques such as traditional testing of the executable system, because the corner cases that reveal faults can be very difficult to reproduce [UK99 Mus+08]. Testing also requires the systems to be implemented first, likely greatly increasing the cost of fixing any problems found [BB01]. Formal verification techniques like theorem proving and model checking, on the other hand, are well-suited to find such faults, as they cover the entire set of possible behaviours. Model-driven engineering is a development paradigm especially suitable to combine with formal verification techniques, as it naturally leads to the creation of models at one or more levels of abstraction up from the executable system (see Sect. 2.2.1). Such abstractions are usually a necessity, as the formal verification of a full executable system is unfeasible for all but the smallest and most critical systems or system parts.

In order to get practical results within the time frame of a PhD study, we choose to extend the existing model-driven engineering method SPACE [Kra08], which already has a tool suite, Arctis, for developing reactive systems. We think SPACE provides a good starting point for what we are trying to achieve (see Sect. 3.1), but we did not conduct a full survey to compare it to other model-driven methods before limiting the scope to SPACE.\(^2\)

\(^2\) Instead, we conducted an extensive survey at the end of the PhD period, both to summarize the state of the art in the field of model-driven engineering of reliable fault-tolerant systems, but also to evaluate our work in light of comparative approaches (see Paper 1).
1.2 Research Context

This work has been carried out at the Department of Telematics, NTNU. While I have been funded directly by the department, parts of the work has been carried out in collaboration with the ISIS (Infrastructure for Integrated Services) project [ISI11; AL11; Su+11], a joint research project by NTNU, Universitetet i Agder, Telenor, Ericsson, TellU and Norsk Automatisering AS that was funded by the Research Council of Norway (project #180122). This means, I have not had any particular limitations placed on the direction of my research, but I have chosen to collaborate with the ISIS project where suitable. The same is the case for the Arctis FORNY project, also funded by the Research Council of Norway (project #199644).

I did an integrated PhD [IME13; ITE12], meaning I started it at the same time as the last year of my master’s degree. This way, the final year of the master’s degree and the first year of the PhD were carried out in parallel over two years, so that the project and master’s thesis could be integrated into the PhD research.

1.3 Research Questions

When doing the work in this thesis, the focus has been on the three main research questions, which we elaborate on in Sect. 3.2. The research questions stem from our goal to reduce the development effort and competence required to create reliable, distributed, reactive systems. To denote a low cost in terms of both effort and competence, we say that we want to do something easily.

First and foremost, we see it as important that a system does not violate any of its properties due to some parts of the system being specified in a way that is inconsistent with other parts. This poses the question:

RQ1: How can we enable developers to easily remove faults in the functional design of distributed reactive systems?

Once a system can be said to be free from self-inflicted problems, our next goal is to enable it to tolerate faults in its environment, including its subsystems.

RQ2: How can we enable developers to easily augment distributed reactive systems with fault-tolerance mechanisms?

In this thesis, the plural we is used to refer to myself, signifying the fact that most ideas and decisions are to some degree influenced by others. The exception to this is when the personal pronoun can only refer to my physical person or there are other reasons to make an explicit separation between myself and others. An example is when specifying individual contributions.
As already mentioned, adding fault-tolerance mechanisms to a system design makes it larger and more complex than a non-fault-tolerant version would be, making it harder to spot mistakes. In fact, it is not unusual that design faults are introduced precisely in the fault-tolerance mechanisms themselves [Rom07]. We hence see it as even more important to be able to detect and remove any faults in the fault-tolerant system design. We ask:

RQ3: How can we enable developers to easily remove design faults in fault-tolerance-augmented systems? 4

1.4 Contributions

The contributions of the work presented in this thesis can be summarized as follows, grouped by the research questions they address:

- Verification of existing SPACE activities (addresses RQ1)
  - Automatic generation of TLA+ from SPACE models [Paper 2 and 3]
  - Automatic generation of theorems from SPACE models [Paper 2 and 3]
  - Visualization of error traces [Paper 3]
  - Proof of concept of automatic diagnoses and fixes [Paper 3]

- Extending the SPACE method to build fault-tolerant systems (addresses RQ2)
  - Example blocks for handling mixed initiatives [Paper 2 and 3], message reordering [Paper 2], message loss [Paper 4, 5 and 7] and process crashes [Paper 4 and 7]
  - A concept of multi-instance activities suitable to express replicated services [Paper 4, 6 and 7]
  - A method for building reliable fault-tolerant systems [Paper 5 and 7]

- Verification of fault-tolerant SPACE systems (addresses RQ3)
  - The concept of EESMs. This is a graphical notation with semantics defined in temporal logic, and it is used to accurately express contracts of multi-instance activities and allows their compositional verification [Paper 6]
  - The concept of ERCs. This is a graphical aspect-oriented notation with semantics defined in temporal logic. It is used to express contracts of building blocks that can exhibit failure behaviour and allows their compositional verification [Paper 7]

4To be clear, this research question is really just the consequence of the other two, but we have chosen to make it explicit, as it reflects the way the work was carried out.
– Empirical data demonstrating the scalability and benefits of compositional model checking compared to monolithic model checking (Paper 3, 6, and 7)

- Overview of the field of model-driven engineering of reliable fault-tolerant systems
  - A set of characteristics to characterize and compare approaches in the field (Paper 1)
  - A survey of state-of-the-art approaches in the field (Paper 1)

In Figure 1.1, we show a slightly different mapping, showing which areas the various papers focus on. Here, the verification field credits a general focus on verification and not just the contributions listed under RQ1 and RQ3. Also, message interleaving here covers both message reordering and buffering of messages leading to things like mixed-initiative problems.

<table>
<thead>
<tr>
<th>Paper</th>
<th>Verification</th>
<th>Verification scalability</th>
<th>Message interleaving</th>
<th>Message loss</th>
<th>Process crash</th>
<th>Multi-instance activities</th>
<th>Method for FT systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paper 2</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Paper 3</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Paper 4</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Paper 5</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Paper 6</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Paper 7</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

In Figure 1.1, we show a slightly different mapping, showing which areas the various papers focus on. Here, the verification field credits a general focus on verification and not just the contributions listed under RQ1 and RQ3. Also, message interleaving here covers both message reordering and buffering of messages leading to things like mixed-initiative problems.

Figure 1.1: Focus areas of each paper, denoted by black cells. Grey cells denote some focus.

1.5 Outline

The rest of this thesis is structured as follows. Rather than having a traditional chapter for background and related work, we include Paper 1 as Chapter 2 which
1.5. Outline

gives an introduction to the field of model-driven engineering of reliable fault-tolerant systems and the relevant parts of theory it builds on. The paper also surveys 10 approaches in the same field, including this one, constituting an in-depth presentation of related work. Chapter 3 contains a description of the starting point of the work and detailed formulations of the research questions. The papers that are included in the thesis are summarized in Chapter 4 in addition to two secondary papers that are not considered relevant enough to be part of the thesis itself. Chapter 5 discusses the results, answering the research questions and presenting the status of tool support. This chapter also describes some limitations and ideas for further work, before concluding with a summary of the thesis, also concluding Part I. Part II contains the verbatim text of the included publications, while Part III contains an appendix with the abstracts of the two secondary papers.
Chapter 2

Paper 1

Model-Driven Engineering of Reliable Fault-Tolerant Systems – A State-of-the-Art Survey

By Vidar Slåtten, Peter Herrmann and Frank Alexander Kraemer.


The original publication is available at www.sciencedirect.com via http://dx.doi.org/10.1016/B978-0-12-408089-8.00004-5
Model-Driven Engineering of Reliable Fault-Tolerant Systems – A State-of-the-Art Survey

Abstract. To improve the reliability of a system, we can add fault-tolerance mechanisms, in order to tolerate faults that cannot be removed at design-time. This, however, leads to a rise of complexity that increases the probability of software faults being introduced. Hence, unless the process is handled carefully, adding fault tolerance may even lead to a less reliable system. As a way to deal with the inherently high level of complexity of fault-tolerant systems, some research groups have turned to the paradigm of model-driven engineering. This results in a research field that cross-cuts the established fields of software engineering, system verification, fault-tolerant systems and distributed systems. Many works are presented in the context of one of these traditional fields, making it difficult to get a good overview of what is presently offered. We survey 10 approaches for model-driven engineering of reliable fault-tolerant systems and present 13 characteristics classifying the approaches in a manner useful for both users and developers of such approaches. We further discuss the state of the field and what the future may bring.

2.1 Introduction

We increasingly depend on software-based systems, and we expect them to guarantee reliable operation. In consequence, more and more systems rely on fault-tolerance technology. The downside of adding fault-tolerance capabilities to a system is that it often makes the systems more complex and error-prone. Thus, the goal to achieve more reliable systems is effectively thwarted by the very means applied to achieve it. In [Rom07], Romanovsky names some distressing examples of this fact, e.g., an interim report [Can03] on the causes for the major electrical power outage in the US and Canada in 2003, which was mainly caused by badly designed fault-tolerance mechanisms. Romanovsky goes to such lengths as to forecast a fault tolerance software crisis and provides the following disillusioning description about the state of the art in designing fault-tolerant systems:

“At present, fault tolerance is not trustworthy as it is the least understood, documented and tested part of the system, is frequently misused or poorly designed, regularly left until too late in the development process, not typically introduced in a systematic, disciplined or rigorous way, and often not suitable for the specific situations in which it is applied.” [Rom07]

As a solution Pelliccione et al. suggest that “[...] fault tolerance needs to be explicitly included into the traditional software engineering theories and practices, and it should become a part of all steps of software development.” [Pel07]. They suggest that fault tolerance should be integrated already in the early phases of
the software development process including the explicit modelling of faults, the measures to alleviate them, as well as the necessary adaptation of the software architecture. The correctness of the fault-tolerance means should further be verifiable and be guaranteed in the model transformation steps. Finally, dedicated tools to model fault tolerance are considered necessary, and it is argued for the provision of domain-specific fault-tolerance mechanisms at the application level [Pel+07]. Rodrigues [Rod08] recommends to extend the Model Driven Architecture (MDA) to offer a better support of designing reliable systems. Also her approach “[…] aims to systematically address dependability concerns from the early to the late stages of software development.” [Rod04]. In general, the application of model-driven approaches may help to effectively introduce fault-tolerance mechanisms into applications. First, model-driven approaches are interesting for fault-tolerance mechanisms themselves:

- Fault-tolerance mechanisms naturally make use of abstractions (for instance, some mathematical models or other logic), which would easily get obscured if implementation details were added.

- Fault-tolerance mechanisms need to be checked for their correctness. Since most mechanisms employ more or less complex behaviour and may require interaction between several components, abstractions, as provided by model-driven approaches, may be essential to make automated analysis (for instance by model checking) feasible.

- A fundamental concept of model-driven approaches are automatic transformations. Such transformations can help to ensure that an abstract model is indeed implemented by a more detailed one (for instance, programming code). Therefore, model-driven approaches can aid in ensuring that fault-tolerance mechanisms are implemented properly.

Second, model-driven approaches can help to apply a given fault-tolerance mechanism consistently to application-specific logic:

- Fault-tolerance mechanisms can be integrated with an application on a more abstract level than if they were formulated in code. This should facilitate that also non-fault-tolerance experts may integrate them.

- As with the analysis of fault-tolerance mechanisms mentioned above, model-driven approaches can facilitate an automated analysis that can ensure that fault-tolerance mechanisms are integrated consistently with the application logic.

This poses the questions of what work has been effectively undertaken in the cross-cutting field of model-driven engineering of reliable fault-tolerant systems, and how one might effectively characterize it to enable a comparison. With these questions in mind, this book chapter provides a comprehensive survey of the state-of-the-art approaches within this field. In particular, we will provide a comparative review
of 10 approaches. For that, we have identified 13 key characteristics that are useful to make a reasonable comparison, and we summarize our findings in tables enabling easy look-up of relevant approaches. This work is complemented by an introduction to model-driven engineering of reliable fault-tolerant systems including basic concepts that are prerequisite for understanding it, as well as a discussion about the current state of the field and where it might be heading to.

Since we are especially interested in the potential of model-driven approaches to facilitate the integration of fault-tolerance mechanisms in reliable systems, we include only approaches that we consider as fully model-driven. This entails the following:

- They must use a modelling language that abstracts away from the implementation platform more than traditional programming languages do.
- They must offer some form of tool support for model creation and management.
- They must provide tool support for automatically ensuring consistency between the models and the resulting implementation, so as to minimize the impact of faults resulting from manual coding. This can be in the form of automatic transformation to code or generation of executable\textsuperscript{1} test cases from the models.

Further, for an approach to be considered aimed at reliable fault-tolerant systems, it must include techniques enabling to verify and/or validate a system model under faulty conditions. Alternatively, it may comprise pre-made fault-tolerance mechanisms that are guaranteed to work in the model of system semantics targeted. Moreover, an approach should aid in separating the concerns of functional requirements and reliability.

There are several previous surveys that cover similar topics as this one, but none that address exactly the field of model-driven engineering of reliable fault-tolerant systems. With regards to model-driven engineering, [GH06] survey model-driven engineering approaches for building real-time and hybrid systems, whereas [AR08] looks at development methodologies following the MDA [MDA11] standard. There is also [LRS11] that surveys various UML models for describing behaviour and how their formal semantics compare. From the field of dependability and fault tolerance, there are surveys like [MK05] that compare general software development approaches that explicitly address dependability. Further, a wide range of contributors have cooperated on a survey and research roadmap for self-adaptive systems, where resilience is one sub-dimension considered [Che+09b]. The authors of [CP09] review the state of the art in the treatment of non-functional properties, including reliability, in software engineering, but focus solely on the requirements phase, rather than the full process. A technical report [MR07] that surveys approaches for

\textsuperscript{1}Test cases need not be directly executable in the form of a binary, but they should not require further manual interpretation before being runnable.
creating architectural descriptions of fault-tolerant systems comes closest to what
we do in this survey. Here, Muccini and Romanovsky classify a large set of ap-
proaches according to both a software-architecture and a fault-tolerance viewpoint.
The main difference from our work is a broader scope, not requiring approaches
to include verification/validation or be model-driven, and naturally that the other
survey includes approaches published in 2006 and earlier. However, there is also
a difference in that we focus more on the practical aspects of each approach, and
less on some of the academic details that are included in [MR07]. This mainly
manifests itself in that we provide more details about available tools and case stud-
ies, and we consider things like platform constraints and what system domain is
targeted. We also go in more detail about the approaches we cover, rather than
listing approaches that fulfill each classification criterion.

The chapter continues with a comprehensive look at the background and scope
of the field surveyed, followed by an introduction of the 13 key characteristics
used to classify and compare the approaches in Sect. 2.3 Section 2.4 describes
the approaches comprising verification of user-designed or user-integrated fault-
tolerance mechanisms, while those tied to a fixed set of mechanisms are discussed
in Sect. 2.5 Thereafter, Sect. 2.6 presents tables allowing for an easy comparison
of the approaches. We next discuss the current state of the field and present our
expectations about its future direction, in Sect. 2.7 Finally, Sect. 2.8 summarizes
the survey.

2.2 Background and Scope

The field of model-driven engineering of reliable fault-tolerant systems, requires
knowledge from the areas of software engineering, system verification, fault-tolerant
systems and distributed systems. This section introduces the necessary background
concepts for readers to follow the discussion later on, as well as limits the scope of
the survey.

2.2.1 Models for Software Development

A fundamental challenge when describing software is that concepts found in an
application domain do not match the concepts that are necessary to address when
one creates an implementation to execute on a specific machine. To describe a
flight booking application, for instance, one needs to specify an algorithm that finds
sensible flight connections, a task that needs to take concepts from the domain of
air travel into account. In the end, however, one has to provide an implementation
in terms of operations that can be executed by a computer. Over time, the gap
between domain concepts and machine instructions has gotten smaller with the
evolution of programming languages. The first generation of a machine language
was only executable on exactly the processor architecture it was written for, and
programs had to know specific register addresses and instructions. In comparison, a program developed in a 3rd generation language may run on several machines and programmers may not have to address memory management, for instance. However, we see that these languages are still on an inherently technical level that requires the programmer to think in terms of the machine in order to create efficient implementations.

The problem of this mismatch between application domain and implementation is that programming languages often obscure the solution of the problem with their technicalities. This prolongs the time needed to write a program in the first place, but also makes maintenance and evolution of software much more difficult. When an application should be adapted to a change in the domain, a developer has to regain an understanding of the domain problem from the code to ensure that changes are correct. Due to this, most software developers are experts on programming, rather than experts on the application domain. Another problem is the constant evolution of execution platforms, which require that applications are re-written to match the specifics of a new operating system, middleware or application framework.

To address these challenges, model-driven engineering (MDE) approaches introduce models that abstract away from the implementation details of a system. Such models may describe a system on different levels of abstraction and from different viewpoints. For example, system structure and behaviour are often separated, and non-functional properties like security and dependability may have different models or model views that help emphasize parts of a model relevant to them while hiding others.

Model-Driven Architecture (MDA) and UML

While the principles of software modelling are not new, the term Model-Driven has gained considerable attention during the last decade. One specific effort is that of the Object Management Group (OMG) with the framework of Model-Driven Architecture (MDA, [MDA11]) that tries to consolidate and align numerous standards, techniques and tools. One cornerstone of MDA is the Unified Modeling Language (UML, [Obj11d]). This is an attempt to harmonize previous versions of UML, the Specification and Description Language (SDL, [ITU07]), Message Sequence Charts (MSC, [ITU11]) and Real-time Object-Oriented Modeling (ROOM, [SGW94]) together with concepts found in object-oriented programming and component-oriented software approaches such as J2EE.

An important element of MDE is model transformations, automated translations between different forms of descriptions. Code generators can be seen as special forms of model transformations that can derive complete or partial code for an implementation based on a more abstract description. In particular, MDA categorizes models into different layers of abstraction, with computation independent models (CIM) as the most abstract ones, followed by platform-independent models.
(PIM) and platform-specific models (PSM). These terms are of course relative to the definition of the term *platform*, which may vary.

To illustrate some of the modelling notations employed, we provide a simple example in Fig. 2.1. The example shows how a mobile device may check road conditions by data retrieved from weather and traffic databases. To keep the description concrete, we here explicitly refer to UML, but many of these concepts can also be found in (or go back to) other modelling languages.

- Fig. 2.1(a) shows a UML collaboration, a structural view on the system that depicts the participants in an application represented by collaboration roles $m$, $w$, and $t$.

- Fig. 2.1(b) is a UML sequence diagram. Each of the participants in the system is represented by a vertical lifeline, with messages passed between the lifelines. These messages may correspond to asynchronously sent signals or operation calls. In the example we see that the mobile $m$ passes messages to $w$ and $t$, who then respond. One particularity about sequence diagrams is the ordering of events, since lifelines are independent. Even though the
mobile sends a message first to \( w \) and then to \( t \), \( t \) may well receive its message before \( w \) does. The brackets on the lifeline of \( m \) specify that the messages of \( w \) and \( t \) may arrive in any order.

- Fig. 2.1(c) shows a state machine for the mobile unit. State machines describe behaviour in terms of control states and transitions. A control state (e.g., \emph{wait}) describes a certain status of the machine in which it waits for events to arrive at its input queue. Each control state can have multiple outgoing transitions starting with the reception of an event. These transitions describe the actions executed as a reaction to the arrival of their triggering event. When all actions of a transition are processed, the state machine changes into the control state targeted by the just executed transition and waits for new events. Transitions are executed in run-to-completion steps, carrying the machine from a source state into a target state in one atomic step. The example shows how the state machine explicitly handles the reception of messages from \( t \) and \( w \) in any order. Advanced features contain deferred triggers, composite states, and states with parallel regions executed in parallel.

- Fig. 2.1(d) shows a UML activity that denotes the same application. Different participants can be represented by separate activity partitions (formerly also known as \emph{swimlanes}). The activity is built by flows that connect actions. The fork node indicates that downstream flows may be executed concurrently, and the join node specifies that partition \emph{mobile} waits for the results of both \emph{prepareStatus} operations before it continues.

Depending on the type of application and desired focus, one or more of these modelling views may be used. Although sequence diagrams may show local operations and can use advanced control structures for elaborate behaviour, they are typically used to specify only partial behaviour (i.e., a specific scenario or use case) and how it evolves. In contrast, state machines usually define complete operational behaviour. For this reason, system are often described by several models from different viewpoints. In the telecommunication domain, for instance, systems are often specified by a combination of MSCs (similar to sequence diagrams) and SDL processes (similar to UML state machines). In this way, both the interactions between components and the complete local behaviour of each component can be covered.

**Domain-Specific Modelling**

Plain UML uses concepts that are general, meaning concepts that do not presume a specific application domain. In contrast, domain-specific modelling (see for instance [KT08]) utilizes dedicated languages for specific domains, which explicitly refer to domain concepts. This should make communication with domain experts easier, since they are familiar with the concepts of the language. The scope of domain-specific languages is narrow. They are often not only specific for a domain,
but may only be used within a single company and for a dedicated set of hardware. For this reason, creating effective implementations from a domain specific language may be easier than for more generic languages. An important enabler of such an approach is the availability of tools and frameworks that facilitate the definition of domain-specific languages, as well as tool support to use them. This may lead to a high number of different languages, which in turn raises a concern about the compatibility and interoperability between them. Having many specific languages, may in the end also reduce the reuse of solutions across domains, due to the different languages they are defined in.

Aspect-Oriented Modelling

Aspect orientation seeks to modularize cross-cutting concerns that would otherwise end up scattered across the modules realizing the main functionality of a system. Examples of such concerns are persistence, logging, security and fault tolerance. Aspect-oriented languages can be at the level of programming languages or more abstract models. Following the terminology introduced with the AspectJ aspect-oriented programming language [Kic+01], the modules containing the main functionality are called base modules. Any aspect-oriented language contains a join point model that defines where in the execution of a base module, an aspect can be inserted. In the case of aspect-oriented modelling (AOM), examples of join points are the reception or sending of a message, the calling of an operation or the entering of a state. There are also approaches that consider everything in a model a potential join point [Whi+09]. An aspect is a module that consists of a pointcut and an advice. The pointcut specifies a set of matching constraints to find join points in the base model, at which the advice can be inserted, whereas the advice describes the behaviour that realizes the aspect concern. A pointcut may also select information about the context of the join point, like data values, and expose it to the advice.

The process of applying aspects to the base modules, is called weaving. Often this is a matter of searching for pointcut matches for each aspect and inserting the advice at the matching join points. Care must be taken, however, if more than one aspect matches the same join point. If so, the order in which the aspects are applied is significant. The application of one aspect may change the join point so it no longer matches the pointcut of an aspect to be applied later. Similarly, some aspects may have pointcuts that match the advice from aspects that should be applied before them. Therefore, many aspect-oriented approaches include ways of assigning a priority to aspects, so that the weaving order can be controlled. We refer to [Whi+09] for a further introduction to AOM.
Semantics and Formal Analysis

Formal techniques are used to precisely describe behaviour. With respect to modelling, there are three major uses for formal techniques. First, they may be used to define the semantics of a modelling language. The semantics of SDL, for instance, is described in terms of Abstract State Machines [BS03]. The UML 2.x standard defines UML semantics using English text. This motivated practitioners to define the semantics in terms of various formal techniques, to make it interpretable by machines. These formal definitions, however, usually only cover parts of the standard and interpret semantics from a less general viewpoint than UML intends. For a discussion on UML semantics, see for instance [OKe06] and [WES10]. Recently, the Foundational UML (fUML) standard has been released [Obj11c], which defines the semantics of a subset of UML concepts, using both a definition of virtual machines that execute specifications and first-order logic.

A second use of formal techniques is the actual analysis of the application itself (discussed in detail in Sect. 2.2.4). Formal specifications derived from models are often more suitable than executable code as input for formal verification tools. Models abstract away many of the details that would increase the complexity of the analysis without adding any more relevant information. For instance, when analysing the interactions among two distributed software components communicating via message passing, the transmission queue may be modelled as a simple queue with discrete elements representing the messages. To find out in which sequences messages may be sent, this may be the appropriate abstraction level, since details about how a particular channel serializes data, for example, does not change the overall communication pattern.

A third use of formal techniques is to ensure that model transformations preserve certain properties of the input model also in the output model. This is needed to ensure that, if an analysis of the more abstract model shows that a system adheres to certain properties (for instance, the absence of deadlocks or the correct ordering of certain messages), these properties are also obeyed by the implementation generated through the transformation. Such reasoning can be done for instance by theorem proving of the transformation algorithm itself, or by finding refinement mappings [AL91] between source and target models.

2.2.2 Dependability Concepts

The authors of [Avi+04] define the threats to dependability to be faults, errors and failures. Defining a service to be the behaviour of the system that is visible to its users, a failure is when the actual service provided by a system deviates from the intended one. Examples are a corrupted response to a request or no response at all. An error is then the part of the total state of a system that may lead to a failure, i.e., a deviation in the actual state of a system from the correct state, like an incorrect value stored in a program variable. Finally, a fault is the
(hypothesized) cause of an error, e.g., a bug in the program code. An error in one system component may propagate to its service interface where it will result in a failure that appears as a fault to other system components receiving this service. This may then trigger new errors and so the error is propagated onwards.

The concept of dependability is further defined to encompass the following attributes

- **Availability:** readiness for correct service.
- **Reliability:** continuity of a correct service.
- **Safety:** absence of catastrophic consequences on the user(s) and the environment.
- **Integrity:** absence of improper system alterations.
- **Maintainability:** ability to undergo modifications and repair.”

Note that both availability and integrity are also considered security attributes, along with confidentiality, the absence of unauthorized disclosure of information. This overlap of attributes between dependability and security is natural, as faulty behaviour of systems can be caused by coincidental natural phenomena, non-voluntary human mistakes, and voluntary malicious human intent. The large difference between these causes is that the methods to overcome the problem have to be different, since in the latter case, we have to face vicious human intelligence that will try to circumvent our precautions. This calls, however, for very complex measures as demonstrated in the huge area of information and network security. Covering these would exceed our scope by far, so we limit our scope to the two former causes of faulty behaviour.

While all of the dependability attributes can be worthy of attention when making a technical system, the focus of this survey is to compare approaches that focus on reliability and to some extent availability. We believe that a system should first and foremost be trusted to complete any interaction that has been started (i.e., to have high reliability), ideally completely masking any errors that occur. When this is not possible or cost-effective, the next step is to reduce the necessary time for automatic repair or switching to a degraded mode (i.e., improve availability).

There is a link between the reliability of software and the quality of its description. Even as early as 1975, Randell argued that the increasing complexity of software was causing software faults to become an increasingly prevalent cause of system failures. While new development and programming techniques have mitigated the problem to some extent, the increase in software complexity has arguably continued even faster, to the extent where many software projects now suffer so-called development failures, i.e., budget or schedule failures causing the project to be aborted before the system becomes operational.

In addition to defining dependability, its attributes and its threats, the authors of lay out the ways to achieve dependability:
2.2. Background and Scope

- “Fault prevention means to prevent the occurrence or introduction of faults.
- Fault tolerance means to avoid service failures in the presence of faults.
- Fault removal means to reduce the number and severity of faults.
- Fault forecasting means to estimate the present number, the future incidence, and the likely consequences of faults.” [Avi+04]

They further argue that none of these so-called dependability means are possible to get perfect, hence a balanced approach using all the means is the best way for achieving dependability.

Fault removal can be further divided into verification, diagnosis and correction [Avi+04]. To make fault removal easy for system developers, these processes should be as automated as possible. In this survey, we cover aspects of fault removal by the characteristics Fault removal and Property types and specification style, which are discussed in Sect. 2.3. In particular, we focus on approaches that employ fault removal at design time, i.e., before the system is deployed.

Fault prevention is closely related to the maintainability attribute, as the methods for fault prevention result in easily maintainable specifications that again decrease the chance of introducing new faults. We capture this concern in the characteristic separation of concerns (see Sect. 2.3). We also briefly discuss its relation to fault removal in Sect. 2.2.4.

Only approaches intended for creating new systems are considered in this survey. That is, an approach may well provide support for fault forecasting, but methods and tools to create only dependability analysis models [BMP08] are not considered.

The failure of a service appears as a fault to the (sub-)systems using that service. Hence, we can also compare approaches in terms of what failure semantics [CF99] of the underlying service they aim to tolerate. Another word for this concept is the fault model of a system [FB08], and we include this as a separate characteristic named precisely fault model (see Sect. 2.3). Failures can be classified according to their domain, i.e., failures can be in terms of content, timing or both. By that we mean, the environment or a system component may communicate the correct or incorrect values at the correct or incorrect time. Examples are message corruption (incorrect value at correct time), message reordering (correct value at incorrect time) and process crash/message loss (incorrect value at incorrect time).

2.2.3 Fault Tolerance

Fault tolerance is defined as a combination of error detection and system recovery. System recovery is further split into error and fault handling [Avi+04]. Fault
handling has to do with how to prevent the same fault from causing another error, but hardly any of the approaches surveyed go into this, so we focus on error handling here.

In [Avi+04], Avizienis et al. list three types of error handling:

- **Rollback.** To bring the system back to a saved state that existed prior to error occurrence, for instance, restoring an earlier checkpoint.

- **Rollforward.** To bring the system into a new state that is free from detected errors, for instance, using exception handling to recover from a foreseeable error.

- **Compensation.** To mask the error using redundancy in the erroneous state, for instance, voting on outputs from a set of active replicas.

While we cannot cover every possible fault-tolerance mechanism, we give some examples of mechanisms that can be used to perform these types of error handling.

Checkpointing is often used to do rollback and can help to overcome intermittent faults that may no longer be active when the system tries to go forward from the checkpointed state again. This is useful both in stateful reactive systems [Pnu86] and in long-running transformational systems\(^2\) where starting over from scratch may waste a lot of time. In a single-node system, checkpointing is relatively straightforward, whereas in a distributed system, one must be careful that the set of local checkpoints capture a globally consistent state. For example, node A may take its checkpoint before sending message M1 to node B, while node B takes a checkpoint after receiving M1. If both nodes now crash and recover from their last checkpoint, the system will be in a state where node B has received a message that node A never sent! There are algorithms to ensure consistent checkpointing in distributed systems, but for those we refer to [KK07][Chap. 6].

Exception handling is a structuring mechanism to separate normal from exceptional behaviour. While exception handling in general can be used to trigger the restoration of a checkpointed state, it is more often used to do rollforward. An exception is raised by a software entity (e.g., object, component or procedure) when it detects an error, which activates an appropriate exception handler. The exception handler can, for example, have the entity make another attempt at the task it was doing, or it may delegate the task to another entity, perhaps one that delivers a lower level of service, but is not dependent on the part of the system that

\(^2\)Transformational systems are started and given some input to process. Once the resulting output has been produced, the system terminates. Examples of such systems are program compilers and the Unix diff program for calculating the difference between two files. Reactive systems are typically long-running and wait for input from their environment, on which they react. Examples are web servers, burglar alarm systems and the cruise control system of a car. Many systems comprise a combination of transformational and reactive parts.
2.2. Background and Scope

now has an error. The handler may also forward the exception to a higher-level exception handler, which is useful if the error must be dealt with in a broader scope than the one covered by the current handler. When the software entities in question are components, exception handling is often seen in relation to *idealized fault-tolerant components* [AL81], components that explicitly distinguish between their normal and exceptional behaviour, while also declaring in their interfaces all exceptions they may raise. The drawback of rollforward handling of exceptions is that it requires the designers to have anticipated the error and to have specified how to detect it and recover from it. On the other hand, this also means that recovery can be tailored to very specific error situations.

Replication of software processes means to have a group of identical processes, so that if one fails, the others can still provide the service to their environment. If all replicas process client requests in parallel, we call it *active replication* [FNL07]. Depending on the fault model (see Sect. 2.2.2), the client or a dedicated voter can vote on the outputs (if the output might be wrong) or just accept the first response (if replicas can only crash, but not produce faulty results, for example). This is a typical example of compensation, in that the redundant information is already there to be used when an error is detected.

The state machine approach [Sch90] specifies the replicated processes as deterministic state machines. Given that these state machines receive the same inputs in the same order, they will produce the same sequence of outputs and end up in the same state (i.e., they will be consistent). If each replica is on a different processing node, identical ordering of inputs is not trivial to ensure, as the communication delay to different replicas may vary from different clients. There are various ways to ensure that the ordering of inputs is the same across all working replicas, but they all boil down to solving the problem called consensus [Gue+00], where all the participants need to decide on the same value among several proposed ones.

Process replication can also be done differently. One may designate a primary replica that handles all client requests as long as it is operational. If the primary regularly transfers its state (in the form of a log or a checkpoint), to the backup replicas, we call it *warm-passive replication* [FNL07]. The primary can also store its state on a disk or in another persistent way, so that a backup loads the entire state when detecting the failure of the primary. This is called *cold-passive replication* [FNL07]. These types of replication are perhaps best classified as applying rollback, since we reuse the existing state of the primary. However, when considering the set of replicas together, we can say that we are rolling the system forward by changing which replica is the primary. There is also a strong element of compensation in warm-passive replication, as the backup is ready to take over quickly, possibly masking the error from the clients.

[Avi85] employs a notation NT/MH/PS, which means that there are N executions, on M hardware channels, of P programs. This is useful to express the configuration of a fault-tolerance mechanism in a short-hand way.

For software faults, the best known fault-tolerance mechanisms are recovery blocks
and N-version software. Recovery blocks \cite{Ran75} are a fault-tolerance mechanism where multiple pieces of software are supplied to do the same task. The output is evaluated by an acceptance test. If the output is OK, the system moves on to the next task; if not, the system rolls back to the state before the task was started and selects another block to do the task. These systems are usually implemented as NT/1H/NS.

The N-version software approach \cite{Avi85}, a.k.a. N-version programming, employs N different implementations of the same software which are usually executed in parallel, i.e. as 1T/NH/NS systems. The results are then voted upon to filter out output from a failed version (compensation). This approach rests on the assumption that independently designed and implemented versions will fail independently of each other, something that has been questioned by some experimental studies (referenced in \cite{FB08}).

These two approaches overlap when recovery blocks are done in parallel or N-version software is done in sequence. If the acceptance test of the recovery block is also implemented as a voting on the results from N blocks, the approaches converge.

### 2.2.4 Fault Removal

This section introduces some core techniques for fault removal and briefly discusses their relation to fault prevention. We focus mainly on the verification part; diagnosis and correction are briefly mentioned at the end.

To remove software faults, they must first be detected. The process of doing this is often, somewhat imprecisely, coined verification and validation. We start by explaining the difference between verification and validation, as these concepts are sometimes confused or used as synonyms.

**Verification** is the process of checking that a system or system model adheres to a set of explicit requirements or properties, possibly expressed as a more abstract model of the system. If the requirements are expressed in a notation with formal semantics, this process can often be automated, as there is no human interpretation required.

**Validation** is the process of checking that a system, or its formal requirements, are indeed what was intended by the stakeholders who made the informal requirements. This is basically a check of whether the requirements were correctly expressed, as well as whether the transition from informal natural language to a machine-readable format was done correctly. Hence, it requires human interpretation. That is not to say that it cannot be heavily aided by tool-supported simulation or testing, for example.

Another angle of differentiating the two concepts can be found by defining the validation problem as the decision of “... whether the formalized problem statement...
Background and Scope

(model + properties) is an adequate description of the actual verification problem. [BK08] This points to checking things like accuracy of the model, along with completeness and consistency of requirements. We limit our scope by assuming that requirements are correctly expressed by stakeholders and correctly formalized by developers, hence the following gives an introduction to only the verification part of fault detection.

As discussed in Sect. 2.2.1, a system model normally describes both the structure and the behaviour of a system, often in different sub-models using different notations. Structural models may be syntactically checked for adherence to given constraints, like having at most one connection to each port of a component. On UML models, for example, such constraints can be expressed with the Object Constraint Language (OCL) [Obj12b], if they cannot be directly specified in the diagrams. OCL is a formal language, but rather than using notation from the domain of mathematics and logic, it employs a notation similar to that of many modern programming languages. Behavioural models may also be checked for syntactic constraints, but our main interest is in analysing the resulting behaviour itself.

There are many formal techniques that can be used to accurately describe and reason about the behaviour of software-intensive systems. Some formalisms let (tool-aided) verification experts apply certain axioms and proof rules for proving the fulfilment of a property, while others usually unfold the behaviour model into a transition system, for automatic inspection of every possible state and/or transition.

Any system that can be interpreted as a state transition system, can be verified with respect to safety and liveness properties [AS85]. A safety property requires that something bad never happens, while a liveness property requires that something good eventually will happen [Lam77]. Examples are “never allow action A to happen once action B has happened” for safety and “once event A has happened, eventually event B must happen” for liveness. A special case of a safety property is an invariant. An invariant is a predicate on the state of a system that must hold in every state of the system.

The aforementioned OCL supports expressing invariants over a model, as well as pre- and postconditions on operations. However, plain OCL is not suitable for expressing safety and liveness properties [KT13].

One way to express such safety and liveness properties is through the use of temporal logic. This is a form of modal logic that extends propositional or first-order logic with modalities like □ meaning always and ◻ meaning eventually. There are linear-time temporal logics and branching-time temporal logics. Linear-time temporal logics, like the Linear Temporal Logic (LTL) [Pnu77] or the Temporal Logic of Actions (TLA) [Lam94], consider all properties over all possible paths (sequences of states) going forward from a certain initial state. Branching-time temporal logics, like Computation Tree Logic (CTL) [CES86], consider properties with regard to states and has extra modalities stating whether a property must hold for all
possible branches of the timeline or at least one of them. However, CTL is not strictly more expressive than LTL \cite{Lam80}, so both kinds have their merits. There exist more expressive temporal logics that combine these approaches, but as the approaches surveyed do not use them, we do not delve into those here.

Model checkers can verify the behaviour of systems specified in their input language with regards to properties specified in temporal logic. For example, the TLC model checker \cite{YML99} can verify the behaviour of a system specified in TLA+ \cite{Lam02}, the specification language of TLA, while the Spin model checker \cite{Hol03} takes C-like Promela specifications along with LTL properties as its input. In the event that a property is violated, model checkers give an error trace showing the behaviour of the system up to the point where the violation occurs. Model checking can be highly automated and is hence a popular choice if non-experts are to do formal verification.

The drawback of model checking is that only systems with a finite state space can be verified, as the model checker typically has to visit every possible state of the system. The state space grows exponentially with the number of communicating processes, so model checking of large distributed systems is often not feasible without abstracting away much behaviour. One method to mitigate this problem is to decompose the system into modules that can be verified correct with regard to abstracted interface contracts, so-called compositional model checking \cite{CLM89}. Another option is to use a model checker to do simulation, which we discuss later.

Theorem proving is another way to formally verify that a system specification adheres to the properties formalized from its requirements. This is usually done interactively, by an expert that uses a tool to discharge proof obligations. Some proof obligations are discharged automatically by automated theorem provers, while others need user input. The benefit of theorem provers is that they can handle systems with infinite state spaces, so that e.g. infinite queues or variables with an infinite domain can be used in the system specification. Their main drawback is that using them requires considerable effort and expertise.

Both model checking and theorem proving are formal verification methods that can be used to both disprove and prove a property with regards to a model. They can also be used to prove the refinement between two formal models. Model A refines model B if all the behaviour in model A can be mapped to the more abstract behaviour of model B \cite{AL91}. Typically, this is used to build systems in a top-down fashion where an abstract specification is iteratively refined until reaching a sufficiently detailed specification, from which code can be generated. When building systems in a top-down fashion using refinement, one could argue that the use of formal methods constitute fault prevention, rather than fault removal: By checking

\[ \text{That is, we split the verification into two: We verify that a module will behave as specified in its contract (a formal interface description), and we verify that the contract fits together with the rest of the system. The contract is typically much more abstract than the module, so this reduces the state space and hence saves verification time.} \]
every step along the way from the abstract to the concrete system, faults will be discovered as soon as they are introduced. However, we still consider this to be fault removal, as faults still exist before being detected and, in consequence, removed. Also, both model checking and theorem proving can also be used in a bottom-up fashion where basic components are composed together to form more advanced components and eventually the whole system. In this case, some faults may only be detected once the components are composed. An example is components with interfaces that do not match properly.

In the context of verification, simulation is similar to model checking except that this method does not exhaustively check every possible path of system states. Instead, paths through the state space are chosen either randomly or based on some guiding principles to obtain a good coverage of different sequences of events. The benefit of this method is that it can be applied when model checking is not possible due to a large or infinite state space. This is typically the case when you have an implementation-level model, or add real time and probabilities. The obvious drawback of simulation is that it cannot provide a guarantee that a property will hold, just state that a path or state in the state space, which violated the property, has not been found, so far.

Simulation may also be done in an interactive way, where a viewer chooses among a range of possible next behaviour steps. This is typically used for validation, since it can give an early indication of whether the modelled system behaves as intended. Interactive simulation is usually combined with model animation, here meaning that the behaviour of a model is displayed in terms of the model elements. We also use model animation to denote that the output from a tool like a model checker is translated back in terms of the original model elements, so that error traces or simulation output can be displayed on the model, rather than in terms of the language of the model checker.

Testing is perhaps the most well-known form of verification. It consists of feeding the system with one or more inputs and checking that the outputs are as expected. Like simulation, this is typically an incomplete method that cannot guarantee the absence of faults. Tests can be created manually, but in the context of model-driven engineering, one can take advantage of model-based testing \cite{UL06,Bak+07}. This is a method by which tests are automatically created from a more abstract model and applied to the implemented system (or parts of it). This is especially useful if the system is implemented manually, rather than generated from the model, as the manual development process can introduce new faults not present in the design model. Even with code generation, the previously formally verified model most likely abstracted away from details that could contain faults (e.g., the contents of local operations), so testing is useful also here.

Returning to the scope of the survey, we focus on model-driven approaches for creating systems that are both reliable and fault-tolerant. While reliability can be said to imply fault tolerance, we stress both properties, so as to exclude approaches only aimed at verifying functional properties when assuming no run-time faults at
all. This means that any approach that does not provide built-in fault-tolerance mechanisms must have a feature for detecting design faults in systems that are to execute in a faulty environment. To elaborate, there must be a way to analyse the system design under semantics where faults from the environment affect the system, so that users can determine how well their self-designed or self-integrated fault-tolerance mechanisms work. We do not include approaches where manual testing once the system is deployed is the only way to uncover design faults that only manifest themselves in a faulty environment. Additionally, we require that approaches included in the survey have a way to ensure the consistency between the models and the executable implementation of the system.

Fault removal consists of more than just verification; any problems found must be diagnosed and corrected [Avi+04]. These tasks depend on the kind of verification method used: While model checking gives error traces to look at, an automated theorem prover will typically not give any explanation as to why it failed to discharge a proof [Sch01]. There exist techniques for partially automating the diagnosis task when using model checking (see, for example, [GV03; BNR03]). However, both diagnosis and correction of design-time faults are mainly manual tasks, just like the initial creation of the system model.

2.2.5 Models of System Semantics

A model-driven software engineering approach usually assumes a certain model regarding how the system fundamentals (e.g., processes, communication channels, computation nodes, clocks) are structured and behave. These assumptions can be both in terms of built-in fault-tolerance mechanisms and in terms of how verification is done. In the literature, these models are usually called system models. To avoid confusion with the model of a system built using model-driven engineering, we call them models of system semantics. In the following, we introduce a few such models, focusing on the aspect of synchrony [LL90] (i.e., focusing on how the concept of time influences the system semantics). We present the models with regards to distributed systems; single-node systems can be seen as a special case.

The synchronous system model requires an upper bound on all communication delays and local task executions. In addition, it requires the drift rate of distributed clocks to be bounded. Together, this allows distributed systems to operate in lock-step, meaning that every node is able to receive input, do state transitions and send output in each step or round. Many problems are easy to solve in this model of system semantics, since one can use time to deduce the absence of events, but most real distributed systems are not easily made compliant with its requirements. Systems running on locally controlled networks with predictable loads, as well as many single-node systems, can adhere to this model, though.

To utilize the synchronous system model, system specifications need to be timed. That is, not only does one need to specify the relationship between input, state and output, but also how much time it takes from receiving an input until sending
an output and/or performing a state transition. This makes system specifications more tied to the deployment platform, so it increases the effort of both making the original specification and redeploying on another platform.

The more relaxed the model of system semantics is, the more likely it is that the assumptions in the model hold true in the real system. Under the asynchronous system model, “we make no assumptions about the relative speeds of processes or about the delay time in delivering a message. We also assume that processes do not have access to synchronized clocks, so algorithms based on time-outs, for example, cannot be used” \cite{FLP85}. Any properties proved to hold for a system in this model also holds under stricter models of system semantics. In particular, if a system specification combined with the asynchronous model implies certain properties, those properties will also hold when combining that system with a stricter model. Hence, one would ideally model systems as completely asynchronous, as done in the asynchronous system model. However, the impossibility result of \cite{FLP85} states that under such conditions, even a single crashed process can prevent a group of replicas from reaching consensus, which makes it impossible to use e.g. active replication for fault tolerance. Hence, we usually need slightly stricter requirements on a practical model of system semantics.

The timed asynchronous distributed system model \cite{CF99} includes hardware clocks, which allow to specify application-level timeouts. The clocks are assumed to have a bounded drift rate in regard to real time, but do not necessarily need to be synchronized with each other, although this improves performance. The model further assumes an unreliable datagram service with omission/performance failure semantics (messages can be lost/delayed) that provides both unicast and broadcast message sending. Thus, message corruption or reordering is assumed not to happen, or to be handled perfectly at lower layers. The processes are assumed to have crash/performance failure semantics and may be restarted after having crashed. Data being corrupted in memory or by the processor is omitted from the model. The model makes no assumption on how often communication and processes may fail, i.e., they can fail arbitrarily often. This core model is extended with a progress assumption stating that a majority of processes will infinitely often be stable. To be stable means that all but a bounded number of messages arrive within a bounded time, like in a synchronous system. The cited paper refers to extensive experimentation with real hardware to support their assumptions. I.e., this model is likely to provide a good coverage for real systems. Just as with the synchronous system model, system specifications need to be timed to take full advantage of the timed asynchronous distributed system model.

Another approach is to enhance an asynchronous system with unreliable failure detectors \cite{CT96}. These are software modules that give a list of processes suspected to have crashed. One can then reach consensus using an indulgent consensus algorithm that guarantees its safety properties (all correct processes choose the same value) even if the failure detector sometimes falsely suspects a correct process of having crashed. In this case, time is not modelled in the application, but the implementation of the failure detector is likely to use local clocks. Note that
even if the notion of real time is absent from a model of system semantics, this does not mean that timers cannot be used. It just means that we cannot state how long a timer will take to expire. Instead, a timer is simply interpreted as expiring eventually. We refer to [Ray05], for a more detailed introduction to the failure detector concept.

Although most computer systems handle only discrete events, some embedded systems may deal both with discrete events and continuous values from sensors, making them hybrid systems [GH06]. Such systems are not the main focus in this survey, but one approach surveyed supports building them.

**Process Communication**

In distributed systems, communication between processes is done via message passing. Message passing means that all the information to be shared between processes is copied from the memory of one process and transferred to another process, possibly on a different node, via explicit sending and receiving of messages. It should not be confused with synchronous or asynchronous communication (boundedness of delays) that message passing between processes may also be synchronous or asynchronous. Synchronous message passing means that the message is passed directly between the sender and the receiver without being buffered in-between. This requires the sender to block until the receiver has received the message, before continuing doing other things. Similarly, a receiver calling a receive operation blocks until a message is sent. Asynchronous message passing involves buffering the message between the sending and receiving process. This allows a sender to continue doing other things as soon as the message has been sent.

Another way of achieving inter-process communication is to use shared memory. Here, processes can read from and write to shared variables, avoiding duplication of data. However, this has drawbacks in that one must carefully synchronize access to shared data to avoid that data is simultaneously read and written. Shared memory is easily available on a single-processor system, but can also be implemented for distributed systems, on top of message passing. Here, an underlying layer takes care of copying data across nodes and provides virtual memory addresses to reference data. Unfortunately, ensuring that data is consistent across nodes has quite a bit of overhead, reducing performance of the overlying application.

For intra-process communication, the function or procedure call (also known as method call, in object-oriented programming) can be used to communicate between logically separated components or objects. This paradigm also uses shared memory and likewise passes references to memory locations as parameters to the calls, rather than duplicating data. Function calls can usually be compared to synchronous message passing with the additional constraint of a call–return protocol, meaning that the sender blocks until the receiver responds. Alternatively, it is also possible for the call to spawn a new thread for carrying out the function, so that the call can return straight away. The calling entity must then usually provide a callback
function as a parameter to the call, so as to be notified when the call is finished. Hence, also function calls can be synchronous (blocking) or asynchronous (non-blocking).

Since function calls are the norm for non-distributed programming, many developers prefer to use the same paradigm when developing distributed systems. One approach for this, is the Remote Procedure Call (RPC) abstraction, where a local piece of code, called a proxy or stub, masks the distribution of the called function by transforming the call to message sending and receiving. However, using RPC does not mask failures resulting from the hidden distribution. Hence, the RPC abstraction forces programmers to deal with some of the effects of distribution anyway, called a “leaky abstraction” in [Spo02]. In general, RPC can make for a simpler application if the problem does not require concurrency (i.e., multiple ongoing executions at the same time), whereas asynchronous message passing can take advantage of concurrency to improve performance of distributed applications, as well as make them more resilient to faults. RPC is easily implemented on top of a platform supporting message passing, by having the sender go into an intermediate state where it does nothing but wait for the response. The opposite is also possible, but for asynchronous message passing this requires that the execution platform supports multiple program threads that can run (semi-)concurrently, so that each send and receive can be done in its own thread.

2.3 Characteristics to Classify Approaches

In order to make a comparison between approaches for model-driven engineering of reliable systems feasible, we focus on some key characteristics that we believe are of interest. The characteristics are the following:

**Type of model** This characteristic comprises the modelling paradigm and language. Examples in terms of the modelling paradigm are finite state machine (FSM), Petri-net and temporal logic. Examples of specific modelling languages are UML state machines, Statecharts, B, TLA+, MSCs and UML sequence diagrams.

**Fault removal** The purpose of this characteristic is to describe what kinds of validation and verification features are provided by each approach. Examples are automatic model checking, tool-supported theorem proving, automatic simulation and model animation. If there are additional features that aid in constructing a good model of the system, like dependability analysis, we also describe them here.
Property types and specification style  This characteristic describes in more detail what kind of properties can be verified and how they are specified. For potential users, knowing that properties important to their problem domain can be verified is naturally a key point. It can also be of great importance whether the properties are a fixed set or specified by the user, as the former will require less of the user, while the latter allows for checking application-specific properties. If users can specify properties to be verified, the notation used for this can determine whether this task can be done by everyone or just by experts.

Property types can, for example, be LTL properties, CTL properties, only safety properties or only invariants. With regards to the property specification style, properties may be a fixed set, specified directly in LTL or specified as an abstract model.

Model–implementation consistency  For all approaches in this survey, we require that they have some sort of means to ensure the consistency between the models and the executable implementation of the system, and this characteristic describes how this is achieved for each approach. The two most common ways are either to generate the implementation automatically from the models, or to employ model-based testing, i.e., generate test cases from the models.

Fault-tolerance expressiveness (inspired by [FB08])  Another important property of an approach is what types of fault-tolerance mechanisms are supported. Depending on the system domain, there may be many fault-tolerance mechanisms that are sufficient, or just a few. Similarly, an approach may offer a wide or narrow range of mechanisms. The wider the range of the approach is, the more likely it is to fit with the requirements of the domain. However, approaches that focus on one or just a few mechanisms are often able to automatically integrate these so that the user only needs to configure some parameters. Open-ended approaches, on the other hand, typically allow users to build whatever fault-tolerance mechanisms are desired, but perhaps from scratch. A middle ground is found in approaches that allow to build mechanisms from scratch, but save such solutions in libraries, either as full implementations or patterns to be instantiated.

If ready-made mechanisms are provided, we can categorize them as being instances of either rollforward (e.g., exception handling), rollback (e.g., checkpointing) or compensation (e.g., consistent replicas). If the approach is not tied to specific fault-tolerance mechanisms, we categorize the examples published.

Separation of concerns (inspired by [FB08])  One of the main challenges when building reliable fault-tolerant systems is to handle the increase in complexity that stems from having to both design for functional requirements as well as reliability requirements (i.e., both system features and fault-tolerance mechanisms). When the artefacts designed to satisfy the different types of requirements
are connected together, the resulting complexity is much greater than the sum of the parts. In an ad hoc approach one is likely to quickly lose track of which parts influence what and hence which parts must be changed should, for instance, the anticipated fault model change. This characteristic extracts how this problem is dealt with in each approach.

The separation can be in terms of syntax so that one can see whether an artefact is primarily about fault tolerance or not; time so that one can e.g. add fault tolerance after specifying functionality; or people so that different experts can design for different types of requirements (typically requires interfaces that convey only what developers of other components need to know).

**Solution reuse** Closely related to how well concerns are separated, is how easily solutions produced, both functional features and fault-tolerance mechanisms, can be reused in other systems. The reuse features we are interested in involve not just reusing an informal design pattern, but really reusing a finished piece of the solution, without having to spend almost as long to understand the piece to reuse as it would have taken to build it from scratch. Interface descriptions help greatly with this, and if they are formally verified to be consistent with an underlying component, we can also reuse the component without looking at its detailed design at all. If compositional verification is used for fault removal, also the verification effort may be reused.

**Model of system semantics** This characteristic refers to an instance of a model for the semantics of the fundamental building blocks of a system, e.g., communication channels, processes and clocks. (see Sect. 2.2.5).

The model of system semantics may be expressed in terms of concepts like the upper bound on communication delay (asynchronous or synchronous communication); whether message passing is asynchronous or synchronous (see Sect. 2.2.5); the upper bound on task execution time; whether time is present, and if so, whether it is in the form of local durations or real time; assumptions about the drift rate of distributed clocks; whether application specifications are deterministic or non-deterministic; whether events take place along a discrete or continuous timeline.

**Fault model** This is a characteristic that describes what types of faults are assumed possible on processes and communication channels. As with the previous characteristic, it is important to compare such assumptions to the realities of the system domain, to ensure a match with the approach.

The fault model may make assumptions about the possibility of situations like loss of messages, partitioning of network, reordering of messages, corruption of messages, node crashes (with or without restart), or it may simply assume arbitrary faults (all faults possible).
Platform constraints  This characteristic is aimed at comparing the platform technology used or assumed by different methods or their tools. Constraints can be specific technologies like programming languages, operating systems or middleware platforms, but also to particular properties of these, like providing certain messaging primitives or scheduling capabilities.

Some examples are programming languages like Java, C or C++; or middlewares and application frameworks like .NET, J2EE or CORBA. Libraries may constrain the systems to operating systems like GNU/Linux or Windows, but constraints on operating systems may also be requirements for deadline-driven scheduling.

Tool support  We aim to briefly describe which tools are available for each part of the approach, as well as some tool properties like availability for use, common model formats and common tool platforms.

Tools can support various parts of the development process like editing, verification or code generation. Examples of other properties are: is open source, is free of charge, has commercial support, has a demo available or stores models in a standard format.

Case studies  We include this characteristic to say something more about the maturity of an approach than just its tool support. Methods and tools that have been employed in large-scale projects are more likely to meet user expectations than those that have only been used on small academic examples. This also allows to quickly see whether the ideas and principles of the approach have been tested empirically or are still in the “should work” category.

System domain targeted  Some approaches are aimed at a specific domain. Although the model of system semantics will help to determine the applicability of an approach, we include this characteristic for a less technical description of the intended domain of an approach. Examples of system domains are embedded systems, real-time systems, distributed systems, reactive systems and combinations of these.

We divide the surveyed approaches into two groups: Those that are general model-driven engineering (MDE) approaches with specific support for fault-tolerant systems and those that have added MDE support to a specific set of fault-tolerance mechanisms (i.e., the model is bound by what already exists at the implementation level). We present the first group of approaches in the following section, while the second one is introduced in Sect. 2.5.
2.4 Approaches with Verification of User-Designed/Integrated Fault-Tolerance Mechanisms

The approaches listed in this section have a common characteristic: It is left to the developer to create and to integrate fault-tolerance mechanisms. However, the approaches provide verification support to ensure that the resulting behaviour is as required. This also means they have verification support for the functional part of the system specification.

2.4.1 Extending Charmy for Fault Tolerance

The authors of BMP07 present an approach where they consider both fault tolerance and fault removal important means to increase the reliability of systems. The described approach was not fully implemented at the time of its writing, but is based on the extension of an existing tool-supported development method called Charmy IMP05. Charmy has later been extended, in PIM09, but this work does not consider fault tolerance. However, some of the new features would likely benefit the approach described in BMP07.

**Type of model:** Requirements are specified as UML use cases and validation scenarios as Property Sequence Charts (PSC) AIP06, which is a similar notation to UML sequence diagrams. Effectively, these are verification scenarios, but the authors seemingly use validation and verification as synonyms throughout BMP07. The structure of the system is given by UML component diagrams and abstract behaviour is described by UML state machines.

**Fault removal:** The authors plan to use model checking to verify that the system design adheres to the fault-tolerance requirements. In PIM09, the authors also describe plug-ins for model simulation and animation.

**Property types and specification style:** Linear-time Temporal Logic (LTL) properties are specified as PSCs. These are more expressive than many traditional scenario notations, so that they can be unambiguously interpreted as either safety or liveness properties. The scenario-based notation comes at the cost of having the full expressibility of LTL, but the authors have found PSCs powerful enough to express all well-known property specification patterns AIP06. The model checker used by Charmy is Spin Hol03, which supports checking LTL properties (both safety and liveness).

**Model–implementation consistency** The authors intend to use the system model to generate test cases for a manually implemented executable system. In PIM09, the authors also describe Charmy plug-ins for generating skeleton code.

**Fault-tolerance expressiveness:** The approach is based on the concept of idealized fault-tolerant components AL81, see also Sect. 2.2.3. The idea is to
first attempt to handle a fault internally for each component and, if this is not possible, to raise an external exception. Hence, we consider exception handling (i.e., rollforward) the main FT mechanisms here, but it does not rule out using other mechanisms in the internal handling.

**Separation of concerns:** Each component has one state machine describing normal behaviour and another one describing exceptional behaviour (i.e., behaviour that results from the detection of an error). In the structural view, the normal and the exceptional state machine reside within different sub-components, communicating through coupled provided and required component interfaces. Hence, there is a syntactic separation, even if the state machines may be mutually dependent.

**Solution reuse:** Since components are given an abstract behavioural description, it should be easy to reuse these. However, it is unclear if the manual implementation of each component could also be reused.

**Model of system semantics:** Components communicate via message passing. While not stated explicitly in [BMP07], the message passing is most likely synchronous, as there is no mention of delays in exception handling or concurrency issues due to asynchronous message passing. Distribution is not mentioned, neither is time. We note that the CHARMY version presented in [PIM09], allows both synchronous and asynchronous message passing between possibly distributed components, but we cannot say whether the approach presented in [BMP07] would.

**Fault model:** It is not mentioned which kind of faults the approach is able to deal with.

**Platform constraints:** No restrictions are mentioned for the application platform. To be clear, the test cases generated by the TeSTOR plug-in for CHARMY are not directly executable [Pel+05], so they do not constitute additional platform constraints. The code generation described in [PIM09] is for ArchJava [ACN02], an extension of Java where concepts related to software components and explicitly represented in the code.

**Tool support:** The authors have already developed a tool-supported system development method, CHARMY [IMP05], that does not take fault tolerance into account. They expect to be able to extend this with the ideas described in [BMP07], without major changes. CHARMY can be downloaded free of charge from its public web page [Cha06]. We note that the web page states that the latest version is from 2006, indicating that active development has ceased.

**Case studies:** As the approach described in [BMP07] was not implemented at the time of its writing, there are no case studies to report on. However, they do illustrate the approach using a mining control system as an example.

**System domain targeted:** The approach seems aimed at component-based fault-tolerant systems in general.
2.4. Approaches with Verification of User-Designed/Integrated Fault-Tolerance Mechanisms

2.4.2 SPACE

This approach is a combination of the SPACE method [KSH09] and the corresponding Arctis tool suite [KBH09]. SPACE combines model-driven engineering with decomposing systems into collaborative services. A system is built by composing building blocks that describe the distributed behaviour of two or more roles, forming a service. Local building blocks describing the behaviour of one role are also common. These models have a formal semantics in temporal logic [KSH09, KH10], making verification through model checking possible. Once the service-oriented model is complete and verified, it is transformed into separate state machines for every component, from which Java code is generated [KHB06].

**Type of model:** The SPACE method uses UML collaborations to describe structure, UML activity diagrams to specify service behaviour and External State Machines (ESMs, similar to UML protocol state machines [Obj11d, p. 544]) to model service interface behaviour. The system is composed from services that may be running on a single node or several ones, and in the latter case they are called *collaborations*. The approach also introduces Extended ESMs (EESMs) to describe the interface of multi-instance collaborations [SH11] and External Reliability Contracts (ERCs) [SKH11] specifying interface behaviour in the presence of faults.

**Fault removal:** The Arctis tool provides automatic verification of properties using internal model checking algorithms or by translation to TLA+ to be input to the TLC model checker [YML99]. Arctis also supports user-guided simulation by animation of the model [KBH09].

**Property types and specification style:** The internal model checker is used to check a fixed set of safety properties, including freedom from deadlocks and that services are composed so that their interfaces contracts are respected. Further, it is verified that the internal design of a service implements its interface contract. If one transforms the model to TLA+ (similar to LTL, but with the advantages of full first-order logic, as well as a mature way to express state transition systems), one can also manually add both safety and liveness TLA properties directly to the specification.

**Model–implementation consistency** The Arctis tool provides code generation to Java, Android and Java ME.

**Fault-tolerance expressiveness:** This approach features a library of building blocks, but also allows creating fault-tolerance mechanisms from scratch. Since the blocks are part of the application, not an underlying layer, this follows the end-to-end principle [SRC84]. The drawback is that it is impractical to add some mechanisms globally, e.g., to take a checkpoint on *every* state change. The fault-tolerance mechanisms published do rollforward and compensation.
Separation of concerns: In [SKH10; SKH11], a two-phase development process for fault-tolerant systems is described. Here, a system designed for an idealized (i.e., fault-free) environment is made first, then augmented with fault-tolerance mechanisms to be verified under the semantics of a realistic environment. Fault-tolerance mechanisms are, just like services providing normal functionality, encapsulated by building blocks, hence providing a degree of syntactic separation. In addition to the separation by time and syntax, the authors argue that the building block concept lowers the complexity sufficiently so that the functionality can be created by domain experts and the fault-tolerance mechanisms integrated by fault-tolerance experts.

Solution reuse: The approach strongly facilitates the reuse of building blocks, as they can be reused on the basis of their interface contracts alone, reusing also the verification effort of each block (see [KH09]). Building blocks are stored in a library for easy retrieval.

Model of system semantics: The model of system semantics is quite close to the asynchronous system model: unbounded communication delays and unbounded task execution time, but with local timers. For verification, timers are only “eventually triggered”, matching the semantics of temporal logic, whereas for code generation they have real-time durations. Message passing is asynchronous. Concurrency between processes on different nodes is modelled as non-deterministic interleaving of events, without any bound on how many steps in a row each process may take (asynchronous concurrency). However, fairness constraints can be added, to rule out system runs where some processes never get to take any steps. Events are discrete.

Fault model: The fault model allows node crashes (with or without restart, with or without persistent state storage), message loss and message reordering.

Platform constraints: The Arctis tool currently has code generators for Java, Java ME and Android. In addition, Java is used as action language in the model operations.

Tool support: The Arctis tool is implemented as a plug-in to the Eclipse platform. It also uses the UML2 package from Eclipse, such that the model is compatible with the Eclipse Modeling Framework (EMF) and can be manipulated by compatible tools. The tool is available for academic and commercial use upon request [Bit13].

Case studies: In [SKH11], the authors describe how they apply the approach when adding fault-tolerance features to a system of automatic work wear lockers that track the amount of clothes used by each employee in a hospital. The system resulting from that work is currently running in several hospitals.

System domain targeted: The approach is devoted to fault-tolerant, distributed, reactive systems.
2.4. Approaches with Verification of User-Designed/Integrated Fault-Tolerance Mechanisms

2.4.3 Event B

Event B is an extension of the B Method [Abr96], a formal software engineering method based on the notion of refining abstract machines. Event B is aimed at the specification of event-driven systems in general, as it includes events that can trigger at any time, provided their guard conditions are satisfied, making Event B more suitable than the B method for e.g. distributed reactive systems. In [Ili+12], the authors describe an approach where they use Event B to develop and verify an attitude control system for a satellite through a series of refinement steps, each step proved correct using automated or interactive theorem proving. Both B and Event B have been around for a while and are used by various research groups, so these formalisms are not tied to the approach surveyed here. That being said, most of the research efforts surrounding Event B, seem to come from partners in the DEPLOY research project [Dep13].

**Type of model:** The modelling language used in this approach is the textual Event B language, which is in turn an extension of the B notation. In Event B, abstract machines are described as a set of variables, their initial state, guarded transitions and any invariants that must hold. Note that in Event B, the same notation is used for highly abstracted and non-deterministic machines as for those refined into deterministic implementation-level machines. Although not used for the approach described here, there exists a graphical front-end for Event B that provides a syntax similar to UML [UMLB12].

**Fault removal:** Each refinement step in Event B can be verified using the formal method of theorem proving. The Rodin support tool for Event B automatically generates proof obligations and attempts to prove them using automatic theorem provers. However, up to about 20% of the proof obligations (17% in this particular case study) require human assistance through an interactive theorem prover. Interactive theorem proving is usually considered an expert task, although the authors do claim the method to have “become more accessible for industry practitioners” [Ili+12]. Although theorem proving is the main verification method of Event B, and the one used in this case study, there also exists a model checker and animator for Event B, ProB [ProB10].

**Property types and specification style:** In Event B, properties are stated as invariants (see Sect. 2.2.4). In the case of refinement, these invariants relate the state of more detailed components to that of more abstract ones by stating relationships between their variables. “For a refinement step to be valid, every possible execution of the refined machine must correspond to some execution of the abstract machine.” [Ili+12]. Liveness properties are not mentioned, but there are papers that propose how to add this to Event B [YB09, HA11].

**Model–implementation consistency:** Although code generation is not used in [Ili+12], there is tool support for generating code [Cod12].
Fault-tolerance expressiveness: Event B does not provide any pre-made fault-tolerance mechanisms, the focus is on verification. Hence, anything can be specified, but the examples typically anticipate error situations and then detail how to bring the system back into a normal state, or into a degraded state. This constitutes rollforward of the system, and we see no inherent problems stopping someone from adding compensation (e.g. replicated processes) or rollback (e.g. go back to the last checkpoint). Of course, developing this would require much effort if starting from scratch, but with the introduction of modules (see Solution reuse), one could imagine such mechanisms being reused.

The work done in [LIR10] is a first step in such a direction. Here, the authors introduce a fault-tolerance view and some simple patterns to refine the models of this view.

Separation of concerns: This approach takes faults into account at once, but postpones detailing how to deal with them until later refinements. One typically starts out with a transition that takes the system from any state to a state that belongs to the set of error states. How this happens and which error state it leads to is detailed later. Hence, any mechanisms for dealing with it are also left unspecified, before starting to refine the system.

Solution reuse: The authors have recently extended Event B to support modularization [Ili+10]. This enables to split components into module interfaces and module bodies. Modules can be parametrized and instantiated multiple times, which allows for reuse of both application logic and verification effort. In the case study, the authors used the module mechanisms to create what is effectively a component template for mode-rich control systems.

Model of system semantics: Just as there are no built-in fault-tolerance mechanisms in this approach, there are no implicit assumptions about the model of system and fault semantics. There are, for example, no built-in communication channels. Thus, these can be modelled as synchronous or asynchronous, and with any kind of ordering properties.

Fault model: A developer is also free to model any kind of fault semantics. In the examples shown in [Ili+12], faults can trigger errors at any time, and the kind of faults are not detailed.

Platform constraints: So far, Event B can be transformed to Ada, C and Java [Cod12].

Tool support: Tool support for Event B is primarily provided by the Rodin platform [EvB13], based on Eclipse. Rodin is open-source and free to use. Atelier B, a well-known commercial tool suite for classic B, now also supports Event B. Atelier B has recently become free to use as well, but offers better support and more frequent updates for paying users.

Case studies: In [Ili+12], the authors develop an attitude and orbit control system for a satellite. This is a system that was first implemented in a traditional way, without the use of formal methods, and later re-implemented
2.4. Approaches with Verification of User-Designed/Integrated Fault-Tolerance Mechanisms

using the approach described in [Ili+12]. The original B method has been used in many industrial projects, particularly in the train domain, such as Line 14 of the Paris Metro [Beh+99].

System domain targeted: As implied by its name, Event B targets the modeling of event-based (reactive) systems. Given the good support for theorem proving, one can say that it also targets systems with very high requirements for reliability, where model checkers typically cannot cope with the state space of the detailed system.

2.4.4 Mechatronic UML – FUJABA Real-Time Tool Suite

The FUJABA project started in 1997 and has since evolved into a tool suite with contributors from many different research groups. Its plug-in architecture allows different contributors to focus on different problem domains, so the tool suite has branched out to cover a wide variety of them. As a result, we have chosen to focus on one aspect that sets this approach apart from most of the others we survey: The support for mechatronic systems (i.e., real-time hybrid systems). There is currently one project aimed specifically at this problem domain, the Fujaba Real-Time Tool Suite [Pri+11], which supports the approach they call Mechatronic UML [BGT05].

Type of model: The approach uses a refined (and formally defined) version of UML component diagrams to describe the structure of systems. Behaviour is described using Real-time Statecharts [BGS05], an extended version of timed automata that include deadlines and worst-case execution times (WCETs). To support controllers of mechanical system parts, they also use Hybrid Reconfiguration charts [BGH05]. These allow to reason about continuous dynamic behaviour, described by differential equations, by embedding block diagrams from CAMeL-View [CAM13]. Story diagrams (graph transformations embedded in extended UML activity diagrams) [Fis+00] are used to specify on-line reconfiguration and fault-tolerance patterns.

Fault removal: Verification is done by the UPPAAL model checker (see [UP413] (commercial) or [UPP12] (academic)). Composition of components does not violate previously verified properties, i.e., verification is compositional. A hazard analysis can be done, to pinpoint weak points with respect to reliability. However, this requires additional manual modelling, as the dependability analysis model is not derived from the functional model. The authors mention automatically creating the dependability model or at least checking consistency between the models as future work, though. Using this method of fault forecasting, they can also calculate the probability of certain failure scenarios, assuming one can find good estimates for the reliability of the hardware components that different software components are mapped to. If weak points are found, fault-tolerance patterns can be applied to change the base model into a more reliable design.
The approach has also been integrated with the commercial CAMeL-View tool for simulation of the production code. This allows checking low-level properties that are not captured at the model level.

**Property types and specification style:** There is no mention of verifying anything other than safety properties. Properties are specified in a fragment of timed computation tree logic (TCTL) called ATCTL, which only contains the *always* path operator.

In addition to this verification of safety properties, the story diagrams used to specify reconfigurations of the system support calculating WCET of the reconfiguration, to ensure timeliness in this phase.

**Model–implementation consistency:** The tool currently supports code generation for C++ and real-time Java (RTSJ, see [Dib06]). The generated code takes execution times and deadlines into account, in order to fulfill the same timing properties as the model.

**Fault-tolerance expressiveness:** Mechanisms to implement fault tolerance are described in what they call fault-tolerance patterns. These patterns only deal with the structure of components, not their behaviour. Hence, they are mainly suitable for fault-tolerance mechanisms of the compensation type, such as replicas followed by a generic voter component. Rollforward techniques typically require integration with the functional behaviour of the system and rollback, though general, is usually not an option for real-time systems. The fault-tolerance patterns also include deployment constraints, to be used as input both in an initial deployment mapping and any later reconfigurations.

**Separation of concerns:** It is stated that fault-tolerance patterns are expressed using story diagrams [Fis+00], which are story patterns (graph transformations) embedded in extended UML activity diagrams to allow for sequencing using loops and if-then-else constructs. Detailed examples on the use of these fault-tolerance patterns are not provided. However, the story diagrams are also used for reconfigurations, and in examples showing this, the story diagram refers directly to elements of the base model [Tic+08b]. So even though there is a syntactical separation, there is also a one-way dependency from the story diagram to the base model. If the fault-tolerance patterns are also specified so that they are dependent on the base model, they are not reusable.

**Solution reuse:** The Real-time Statecharts are used to specify real-time coordination patterns, which are descriptions of the protocol over a connection between two ports. These patterns are reusable and so is their verification effort, as composition does not violate previously verified properties, i.e., verification is compositional.

**Model of system semantics:** Unlike traditional UML state machines, Real-time Statecharts do not execute in run-to-completion steps [BGS05], but can be interrupted during a transition. They also have a semantics that allows
communication via message passing over both synchronous and asynchronous channels [GB03]. Further, all transitions and communications are modelled as taking a finite amount of time. Hence, it is vital that the execution platform satisfies the assumptions made in the modelling phase.

**Fault model:** The fault model includes both crash, timing and value failures [Tic+08a]. Clock drift is not considered, as the authors assume a real-time group communication framework to ensure that replicas get messages in the same order and at the same time [Tic06].

**Platform constraints:** The code generation is for Java and C++. As the models in this approach are timed, the execution platform must also be suited for real-time systems, allowing for deadline-oriented scheduling of processes, for instance.

**Tool support:** Fujaba4Eclipse has been around since 2006, while the stand-alone Fujaba tool that it evolved from has been around much longer. Even though it is based on Eclipse, Fujaba4Eclipse defines its own UML meta-model, rather than using the one provided by Eclipse through EMF. There are papers suggesting ways to make the two meta-models interoperable [Bec+08; Joh08], but we do not know how far this process has come in practice. The tool suite itself is free to use, but both the UPPAAL model checker and the CAMeL-View tool require licenses for commercial use. There exist other tool suite configurations that are also based on the Fujaba core. A tool for reverse engineering legacy components with MECHATRONIC UML is also in the making [Hen+10], which can be of interest for anyone wishing to apply the approach to existing systems.

**Case studies:** The approach has been used in different parts of the RailCab project [Rai13], a project where autonomous rail wagons share a common rail and form convoys when possible, to reduce air drag.

**System domain targeted:** This approach is unique in this survey, in its support for modelling also the continuous dynamic behaviour of hybrid systems, which most mechatronic systems are. The approach also targets embedded real-time systems in general.

### 2.4.5 iFTElement

Brito et al. [Bri+09] present an approach built around an architectural abstraction called an idealized fault-tolerant element (IFTElement), which can be seen as a refinement of the idealized fault-tolerant component (IFTC) abstraction (see Sect. 2.2.3). Among other things, an IFTElement can be instantiated as a component (IFTComponent) or a connector (IFTCConnector) with the task of doing computation or coordination, respectively. The approach builds on previous work with the Methodology for the Definition of Exception Behavior (MDCE) and MDCE+ approaches [Rub+05; S B+05], which detail how exception handling can be specified.
and tested throughout the development process. In [Bri+09], the authors focus more on the iFTEElement abstraction and the verification of exception-handling-related properties, before testing the implementation’s consistency with the model. We also note a recent paper [Fer+11] extending this approach (although they use the IFTC abstraction) to include low-level design models for a more detailed exception handling analysis. In the following, we focus on the approach as it is described in [Bri+09].

**Type of model:** Use cases are specified for the scenarios related to abnormal behaviour of the system. Then, a system structure consisting of iFTEs is described via UML component diagrams that explicitly separate normal and abnormal, provided and required, interfaces. The use cases are then formalized into UML sequence diagrams referring to the components and their interfaces. The semantics of the UML components are given in the B-method notation, while the semantics of the sequence diagrams is given as CSP (a process algebra) [Hoa78]. This combination is chosen because B is suitable to specify types and static dependencies, but does not easily allow restricting the order of operation execution [Bri+09].

Finally, UML activity diagrams are used to represent sequence and interaction graphs for unit and integration testing, respectively. These can be generated automatically, so a developer may only need to read them.

**Fault removal:** The approach provides automatic verification by automatically transforming the UML models into B and CSP, to be taken as input by the ProB model checker [BL05]. Note however, that the verification performed is not of the entire state space, but on the sequence of operations specified by the sequence diagrams. Hence, the verification is only exhaustive if the sequence diagrams describe the complete behaviour, which does not align with the guidelines stating to only specify the abnormal behaviour scenarios. Still, the authors do state that scenarios regarding application-specific functional behaviour may also be added, so it seems possible to check every behaviour. Potentially, it is useful to be able to trade the number of scenarios checked for the domain size of variables used in the specification.

The authors also state the verification can be done incrementally (compositionally) by composing iFTEs from other iFTEs. Note, however, that this only holds as long as there are no changes in the sequence diagrams, as that may introduce new behaviour, as opposed to other approaches that check all possible behaviours right away.

**Property types and specification style:** The model of the system is automatically checked for a set of fixed properties. Each iFTE is verified to adhere to a set of syntactic constraints, as well as a set of general behavioural scenarios that apply to every iFTE. To support the verification of behavioural safety properties, a history variable is used to store the sequence of events that happened to reach the current state. One can also add user-defined properties specified as invariants. Combined with the history variable, these
2.4. Approaches with Verification of User-Designed/Integrated Fault-Tolerance Mechanisms

are in effect safety properties. The architecture as a whole is also checked for syntactic adherence to a set of fixed rules. In addition, it is checked for a set of properties such as freedom from deadlocks and that all elements communicate according to a call–return protocol (see Sect. 2.2.5). One can also specify existing exception handling patterns and verify whether these are indeed matched by the behaviour in the model. It is not explicitly stated how the user-defined properties are specified, so they might require the user to specify directly in B or CSP.

Model–implementation consistency: Once a system model is verified to be correct, both unit and integration tests can be generated for the components, so as to verify the consistency between models and manually implemented code. The tests follow the scenarios described in the UML sequence diagrams. As previously mentioned, one can add scenarios for purely functional behaviour, even if the approach focuses on the exception handling part. Thus, one can get test cases for any scenario.

The authors state that the test case generation can be automated, but this was not done in the case study. They cite another paper [PMV07] about a tool being built for this purpose, but that paper is in Portuguese, so that we were not able to extract much information about the tool, except that it takes a model in XMI format [Obj11b] as input and produces test cases in some XML format (i.e., it does not directly output executable test cases).

Thanks to the interface descriptions in the model, the test cases can be generated before the source code is available. The approach enables to perform a dependency analysis, without source code available, to sort the test case execution. For unit tests, this is about not executing tests that start from a given state unless it has already been tested that a component can get into that state successfully. With regards to the integration tests, the sorting ensures the components with the fewest dependencies on other components are tested first, so as to not have to make so many stubs to simulate the untested components.

Fault-tolerance expressiveness: This approach focuses solely on exception handling. As stated in Sect. 2.2.3, this is a very general mechanism that can trigger all sorts of recovery activities, but is usually employed to roll the system forward into a new and error-free (possibly degraded) state.

Separation of concerns: The use of the iFTElement abstraction separates normal and abnormal behaviour both in terms of sub-components and in terms of interfaces of the iFTElement. Further, computation in iFTComponents is separated from coordination in iFTConnectors. Due to this, we consider a separation in time possible, as abnormal behaviour can seemingly be added after the normal behaviour is completely specified. The interfaces of components must be combined with (possibly incomplete) scenarios to get a behavioural contract. If the contract of a component is complete, different people should be able to design any relevant exception handling mechanisms in other components using the contract of the first one.
Solution reuse: An iFTE encapsulates, among others, both a normal and an abnormal sub-component. The business logic is in the normal sub-component and is easily reused, as the adaption to other components or the deployed environment is done by other sub-components. This also makes it relatively easy to use commercial off-the-shelf (COTS) or legacy components, by encapsulating them in an iFTE. Each iFTE should also be reusable. The interfaces provide pre- and post-conditions on operations (specified in B), but not the allowed sequencing of events. The sequencing is provided by the sequence diagrams, and these cover scenario interactions between several iFTEs. Hence, they are not not reusable, at least not at the same granularity as iFTEs. This means that a reuse of the actual verification effort seems unlikely.

Model of system semantics: This approach requires all communication to follow a call/return protocol (i.e., blocking procedure calls). This rules out concurrent execution in the system, as there is only ever one component executing an operation at a time. While this does not rule out distributing the components across multiple nodes (e.g., using remote procedure calls (RPC)), the gain from doing it seems to be limited.

If a component A gives control over to a remote component B via RPC, we assume (no details regarding this are given) the local stub for component B can use a timeout to raise an exception when B does not respond. If timeouts in stubs are not supported, distributing the system would clearly reduce its reliability, when employing the iFTEElement approach.

Fault model: The exception handling mechanism deals well with any kind of fault that produces an error that can be detected by a component. This includes information from sensors, as used in the case study. There is no discussion of whether things like process or node crashes are detected and handled.

Platform constraints: This approach relies on model-based testing, instead of code generation, to ensure that the implementation of the system is consistent with its models. Hence, any platform constraints will be due to the output format of the test cases. As mentioned, there is a tool being built for this, but we have not been able to find details regarding what platforms it supports. The authors of [Bri+09] claim that also existing model-based testing tools can be adapted to work with their models, but there is no discussion of what this entails.

Tool support: A non-specified CASE (computer-aided software engineering) tool is used to create the UML models, and no details are given on the tool that transforms the UML models into B and CSP. Again, a tool for test case generation is being built [PMV07], and one may adapt existing tools for model-based testing to generate test cases from the models.

ProB is freely available [PrBD13] and open-source [PrBL12]. There is also commercial support being offered [For13].
2.4. Approaches with Verification of User-Designed/Integrated Fault-Tolerance Mechanisms

Case studies: In [Bri+09], the authors report on a case study conducted on a mining control system consisting of sensors and actuators, but most likely a single computing node. We note that the system is referred to as *real-time*, but there is no discussion regarding how the approach handles this. There is no mention of actually deploying the system, so we must assume that the case study is purely academic.

**System domain targeted:** Our impression is that the approach targets critical component-based systems without the need for concurrency, and which are executed on a single computation node. Since the approach includes guidelines for how to encapsulate COTS components in an iFTEElement, one could say that it partially targets systems built from these or systems using legacy components.

### 2.4.6 Motorola WEAVR

Model-driven engineering (MDE) has a long history in the telecom industry, as these were some of the first companies to face the complexities of creating large, distributed, reliable systems. In [CBE07], Cottenier et al. present a tool developed at Motorola, the WEAVR, that uses aspect-oriented modelling to support cross-cutting concerns like fault tolerance. Their main motivation is to avoid the replication of effort that arises when teams working on different components all need to implement similar mechanisms to deal with, for example, faults. Because they use highly optimizing code generators, the generated code does not retain the decomposition of the models. Due to this, it is not possible to generate aspect code that is weaved together with the base code. Instead, all the weaving is done at the model level. The Motorola WEAVR is a plug-in to the Telelogic Tau MDE tool suite, now called IBM Rational Tau.

**Type of model:** Structure is given both in terms of UML class diagrams, for an object-oriented decomposition, and hierarchical composite-structure architecture diagrams (similar to UML component diagrams), for a decomposition in terms of layered components. Behaviour is specified using both state-centric state machines providing abstract behaviour specification and transition-centric state machines, which specify behaviour implementation. In addition, there is an action language provided by the Tau tool to specify detailed behaviour such as local operations and decision guards. There is also support for in-line program code in the languages supported by the code generators. Finally, class diagrams are also used to define the scope, priority, and applicability of aspects, using stereotyped connectors.

4 The generated code performs within 5% of manually written code [CBE07].
5 These are abstract state machines similar to UML protocol state machines [Obj11d] [p. 544].
6 These are similar to UML behavioural state machines [Obj11d] [p. 573], like the one shown in Fig. 2.1 (c), page 18.
**Fault removal:** Although not specifically mentioned in [CBE07], the Tau tool suite provides a *model explorer*, which is a model checker that can both do exhaustive and non-exhaustive traversals of the state space [IBM09]. It has many pragmatic solutions for when the state space is too big to search exhaustively (e.g., various features for state space compaction or simply doing a random walk). The model explorer supports, with some restrictions, the detailed analysis of models that contain in-line C/C++ code. If problems are found, error traces can be displayed as sequence diagrams.

Tau also provides a *model verifier*, which allows debugging, tracing and simulation of the models by generating instrumented executable code. The model execution can be driven interactively or by pre-defined test cases. In interactive mode, this feature acts as a model animator.

The authors argue the benefit of writing aspects with respect to relatively stable model elements, instead of writing aspects at the level of the more volatile source code. The WEAVR further provides an aspect effect visualization feature, so that developers can interactively browse the base model and the applied aspects in a way that respects the semantics of the woven model, so as to validate it. Note that the actual woven model is not intended to be manually inspected at all. The visualization is achieved by colouring and annotating model elements, as well as adding links to instances of the applied aspects from the respective places in the affected base model.

**Property types and specification style:** Tau automatically checks that models adhere to a fixed set of syntactic and semantic constraints. The model explorer also checks some unspecified fixed properties, and has the option of checking user-defined invariance properties specified in the action language, which contains a subset of OCL and allows the use of first-order logic.

Test cases for the model verifier can be created manually or derived from abstract specification models, like state-centric state machines. Test cases are either specified as sequence diagrams or in the Textual Testing and Control Notation (TTCN-3) [TTCN13].

**Model–implementation consistency:** The approach features the generation of executable code in multiple programming languages. The test-case-driven simulation mentioned under *Fault removal* is not model-based testing of the implementation, as those tests are applied to the simulation code. However, Tau also supports model-based testing, to some extent, requiring the test cases to be explicitly modelled before automatically creating and executing them [IBM09].

**Fault-tolerance expressiveness:** In this approach, fault-tolerance mechanisms are specified as aspects (see 2.2.1). The aspect advice (called *connector* in [CBE07]) has access to the operation (including parameters of the operation call) or transition that matches the pointcut. Via a reflective API, the advice also has access to information about the instance of the class owning the matched operation or transition. There is no discussion of which, if any,
restrictions this places on what can be specified in an aspect advice, but
the examples range from simple logging aspects to rather complex things
like weaving a two-phase commit protocol into a protocol for requesting re-
sources. In the latter example, the advice introduces new transitions and
operations, which suggests that the expressive powers of the aspects is as
high as when manipulating the base model directly. In that case, all types
of error handling should be possible to implement, but the example given is
only for rollforward.

Separation of concerns: Aspect-oriented modelling provides good separation of
contcerns, as aspects are both syntactically separated from the base model and
can be written after the base model is completed. Components may be given
abstract specifications in the form of state-oriented state machines. This may
enable also those not involved with the base model to easily produce aspects
for it. However, aspects may depend on each other, or affect the same part
of the base model. In both cases, the order in which aspects are woven into
the base model is important to consider. In the WEAVR approach this is
solved by explicitly specifying which other aspects any given aspect should
follow. This means that the developers of one aspect must be aware of all
potentially conflicting aspects.

Solution reuse: We can say that this approach offers solution reuse, just like
other approaches that feature components with an interface description.
However, this kind of reuse just covers the functional part of components.
In addition, the aspects that implement fault-tolerance mechanisms can be
reused separately.

Model of system semantics: Even though UML models are used in this ap-
proach, the SDL profile [Tel07] is applied so that the models can be un-
ambiguously interpreted according to the semantics of SDL [ITU00]. In
practice, this means that communication between components is done using
asynchronous message passing and that local time is available in the form
of timers. State machine transitions are triggered by discrete events, either
the reception of a message or the expiration of a timer. The transitions are
normally executed atomically (i.e., they run to completion without interrup-
tion), but the Tau tool also supports generating a threaded implementation
where high-priority state machines may preempt others. We could not find
whether the model explorer will consider this, or whether it just assumes
complete independence between the preempted and preempting state ma-
chines.

Fault model: There is no assumed fault model in this approach, so it must be
added by developers. Most event-driven systems are open, i.e., they commu-
nicate with a surrounding environment in the form of human users or other
systems. In this approach, the environment can either be modelled explicitly
to create a closed system, or one may list a set of signals with a set of test
parameters that are randomly sent from the environment. This can be used
to also model faults in the communication from the environment by sending
wrong parameters (message corruption), wrong signals (message loss, message reordering, message duplication) or no signals. Modelling faults inside of the system itself is not discussed, but we expect that the aspect mechanism provides a suitable way of doing this as well. Adding non-deterministic decisions that branch into faulty transitions seems to be the most straightforward way.

**Platform constraints:** Code generation to C and C++ is mentioned in [CBE07]. Further, the Tau documentation [IBM09] also describes code generation for Java and C#, as well as a code generator they call AgileC, which generates C code for embedded systems. It is our impression that the woven model can be fed to a standard code generator. If so, all of them should work. However, the models created in this approach are platform-specific. Various code generators have some constraints on what UML elements can be used and how (see e.g. [IBM09 p. 949]). Also, any use of in-line code naturally ties the model to execution platforms for that programming language.

**Tool support:** IBM Rational Tau [IBM13] is an MDE tool suite that provides editing and analysis capabilities. It is commercially available from IBM. The Tau tool suite also provides a two-way integration with IBM Rational DOORS [DOO13], a tool for requirements management and analysis. The Motorola WEAVR [WEA06] is a plug-in to Tau that adds support for aspect-oriented modelling. WEAVR is not publicly available, but free-of-charge licenses are available for research purposes [WeL06].

**Case studies:** The example in [CBE07] concerns a server that grants access to a set of resources, but only if all resources are available at the same time. They ensure the reliability of the system in a distributed setting by adding timeouts and the two-phase commit protocol via aspect-oriented modelling.

At the time of writing, the WEAVR was in production use by the Network and Enterprise Business Unit of Motorola. However, the authors of [CBE07] note that only rather simple aspects like logging were being applied, not the more complex ones, like the two-phase commit protocol in the example. By the authors’ own words, the WEAVR tool was still in an initial state of maturity then, so this may have changed. They express high hopes for applying more complex aspects, estimating they could reduce model sizes by 25 to 40% by using aspects to specify the logic dealing with concerns like fault tolerance and security. The authors also note that 50 to 85% of the telecom infrastructure systems in Motorola are fully automatically generated from SDL and UML models.

**System domain targeted:** According to [CBE07], the target seems to be reactive discrete systems for telecom infrastructure.
2.4. Approaches with Verification of User-Designed/Integrated Fault-Tolerance Mechanisms

2.4.7 MOCAS

The field of self-adaptive systems somewhat overlaps with the field of fault-tolerant systems in that errors can be seen as just another context change to adapt to. However, many MDE approaches for creating self-adaptive systems are not appropriate for dealing with fault tolerance, since they assume a fault-free environment where there is no need for redundancy or distributed control. In addition, the focus in self-adaptive systems is usually on structural adaptation, i.e., the complete replacement of one component with another. This fails to take the current state of a component into consideration, something that often affects the recovery strategy employed. In [BHB09], the MOCAS (Model Of Components for Adaptive Systems) approach is presented. Here, the authors adapt behavioural models and distribute the decision making process across autonomous components, making the approach more suitable for handling fault tolerance. In MOCAS, oblivious business components are wrapped in adaptive containers so that their behaviour may be changed at run time, forming adaptive components. Autonomic components are a subtype of adaptive components that combine sensor, evaluator and effector components to form a control loop where the adaptation is triggered by the components themselves, hence forming a self-adaptive system. MOCAS is implemented as a plug-in to the TopCased toolkit for MDE of critical embedded systems.

**Type of model:** The structure of systems is expressed by UML class diagrams, one class per component. UML state machines give the behaviour of each component. Detailed operations are expressed directly in Java. OCL (see [Obj12b] and Sect. 2.2.4) is used to specify invariants regarding the state of each component.

**Fault removal:** The authors of [BHB09] state that they validate and verify models using the TopCased toolkit, but they do not go into detail. We note that the TopCased toolkit does provide a feature that allows to animate the model in a step-by-step fashion [Com+08], but we are not certain this is what the authors of the MOCAS paper refer to. In [Ber+09], another group of researchers present a feature that allows transforming TopCased models, be it UML, SDL or AADL[7] into model checker input languages. However, at the time of writing in 2009, support for transforming UML models was not yet implemented, and it is not clear if the transformation will work straightforwardly with the MOCAS models.

MOCAS has a model execution engine that runs UML models directly. This engine also does run-time consistency checking to ensure the models are not adapted in a way that violates a set of pre-defined properties. This can be seen as an alternative to verifying that only consistent adaptations are possible up front, or it can be seen as a way of ensuring model–implementation consistency.

A later paper [Car+11] describes the execution semantics of the UML state machines under the MOCAS engine as meta-level contracts specified in OCL on the UML meta-model. The authors describe how this can be used to verify correct model execution both at run time and at design time, although implementing the design-time verification is planned as future work.

**Property types and specification style:** To be clear, the OCL invariants described in [BHB09] are application-specific and must be added by developers, but are not used as properties to verify, rather their violations trigger run-time adaptations. The verification possibilities presented in [Car+11], on the other hand, seem to only concern general properties for UML state machines. These properties are also limited to invariants, but include pre- and postconditions on operations, where each pair of pre- and postcondition are rewritten as multiple invariants.

If the TopCased verification feature presented in [Ber+09] would work with the MOCAS models, there could also be support for verifying LTL properties. At the time of its writing, LTL properties were written directly in the input language of the model checker.

**Model–implementation consistency:** This approach does not provide traditional code generation, instead the MOCAS engine takes the EMF UML models in XMI format [Obj11b] and directly executes them. The engine is written in Java, so calling user-defined Java operations at the appropriate time is straightforward. Since the models are available at run time, the OCL invariants can be continuously monitored. Similarly, the execution engine can ensure that no adaptations result in a system that is not consistent with the MOCAS meta-model.

**Fault-tolerance expressiveness:** OCL expressions are used to specify invariants that should hold as long as a given state of a state machine is active. The invariants only refer to the local state of the component, not the global state of the system. These invariants are checked at run time and trigger adaptations when violated. One such adaptation that is mentioned is to roll back to the previous state. Adaptations are constrained so that the new behaviour has to start from the current state of the state machine and does not interrupt any ongoing behaviour. Apart from this, most reactions are possible to enable the system also rolling forward into an error-free state.

When the adaptations of components may depend on each other, MOCAS supports two ways of keeping components informed. The first one is to organize components in a hierarchy such that components can subscribe to the adaptations of others and react accordingly. Here, there should be no cycles in the subscription graph, as this could cause endless propagations of adaptations. The other way is for a set of components to negotiate what to do up front. No details of how this is achieved are given, but we assume some sort of consensus algorithm is used.

**Separation of concerns:** A business component is made without the need to consider the adaptation to be added afterwards. Then an adaptive container
is wrapped around the business component. This container can intercept messages going to and from the business component, some of which may tell the container to change the behaviour of the business component. Components also have state machines stating in what order different messages may be handled, i.e., constituting behavioural contracts. This should allow for other people to create the adaptive containers than those who make the business components.

**Solution reuse:** The approach supports reuse in the sense that it allows to encapsulate business components, but also in that components have state machines describing their behaviour, allowing whole adaptive components to be reused.

**Model of system semantics:** MOCAS components assume asynchronous communication channels and communicate via asynchronous message passing. All state machine transitions run to completion (i.e., no interruptions), so messages arriving during a transition are buffered and handled afterwards. Although the concept of time is not mentioned explicitly, the example presented contains a scenario where a computing node detects the failure of another one. Thus, we assume either a mechanism like local timers for detecting non-responsiveness, or a built-in service that informs of suspected failures.

**Fault model:** There are no details provided on what faults can be handled. Anything that can be detected in a component can naturally be used to trigger an appropriate adaptation.

**Platform constraints:** The MOCAS execution engine is available for standard Java and for the mobile version, J2ME. The state machine operations are also written directly in Java. While the authors of [BHB09] have made a TopCased plug-in for creating the adaptive MOCAS components, we understand from [Car+11] that the MOCAS engine can also execute plain UML state machines made with other tools, as long as the resulting models are stored in the EMF implementation of UML.

**Tool support:** The TopCased toolkit provides general editing, animation and verification support, is open source and is freely available on-line [Top13]. It is based on Eclipse and use their EMF implementation of UML, like many other tools. A plugin for TopCased enables the specification of MOCAS models and is available for download [MOC10]. The MOCAS execution engine is also freely available [MoE09]. Both the latter have source code available as well.

**Case studies:** The example given in [BHB09] is a sensor network of flood sensor nodes along a river. This system, GridStix, is taken from another paper, so we assume it is only used as an academic example. No evaluation is given.

**System domain targeted:** The TopCased toolkit is said to initially target embedded systems in the aeronautics, space and automotive domains. While the MOCAS paper mentions no such specific industrial domains, MOCAS seems to target distributed, reactive, embedded systems that may be deployed in a changing environment.
2.4.8 rCOS

The authors of [Ke+12], Ke et al., present an approach, rCOS, that aims to combine component-based and object-oriented modelling of systems with formal verification and consistency checking. The authors argue for not only applying formal methods to safety-critical systems, but rather to any complex software or software-intensive system. They further state that the rCOS approach is really a combination of existing theories and techniques, and emphasize the need for good tools for model construction, validation, verification and transformation. The rCOS approach, as presented in [Ke+12], is not particularly focused towards fault-tolerant systems, but handles their specification and verification as well. In particular, it allows to specify processes that trigger faults in the system. In [Xu+12], Xu et al. present the start of an effort to use the rCOS approach specifically for building fault-tolerant systems, guided by an integrated fault tree analysis [KLM03]. Thus, first a purely functional system is developed, on which the fault tree analysis is carried out. Thereafter, based on the analysis results, fault-tolerance mechanisms are added.

Following rCOS, one starts by creating a set of use case models that capture the system requirements. The use cases are then refined by developers, into design-level models. These are, in turn, input for a code generator that outputs the executable program.

**Type of model:** rCOS has a textual modelling notation that is defined in the Unifying Theories of Programming (UTP, [HJ98]) framework extended with concepts from object-oriented modelling. The notation is somewhat similar to Java in style, and the authors argue that this makes it easier to use for developers not familiar with formal methods. In addition, they have created an rCOS UML profile, enabling developers to mostly work with graphical models. In this profile, UML use case diagrams are applied to specify compositions of use cases, while UML component diagrams are used to specify components, their interfaces and their composition. UML sequence diagrams specify component interface protocols (i.e., the allowed ordering of methods calls) and UML state machines describe the reactive behaviour of whole components. The class concept from object-oriented programming is covered by UML class diagrams. Data functionality, i.e., the effect of method calls, is specified textually as Hoare logic, i.e., with preconditions and postconditions. These models are all used in a first iteration specifying high-level use cases. In a second iteration, a more detailed design model is created, also using all the above model types, with the exception of the use case diagrams.

**Fault removal:** Requirements validation is done by feeding the initial use cases, annotated with some OCL constraints, to the AutoPA tool [Li+10]. The tool generates an executable prototype that can be used for validation purposes. The different viewpoint models of rCOS can be automatically checked for consistency with each other, but it is unclear exactly what kind of consistency
2.4. Approaches with Verification of User-Designed/Integrated Fault-Tolerance Mechanisms

checks the authors refer to and how much is implemented. An rCOS system can also be automatically translated into CSP [Hoa78], and analysed using the FDR2 model checker [FSEL13].

Finally, the authors of [Xu+12] present some initial work on doing component fault tree analysis (see [KLM03]) of the rCOS models.

Property types and specification style: The initial requirements model is checked for consistency between viewpoints, absence of deadlocks and safety properties of the reactive behaviour. Apart from the absence of deadlocks, it is not stated what these safety properties are or whether a developer can add application-specific properties.

After transforming the requirements model to the component model, the tool again does static checking of the UML models, to ensure they are complete and consistent. The authors of [Ke+12] state that this is mostly a case of checking OCL constraints. However, it is not stated whether developers can add additional application-specific OCL constraints. Again, CSP is produced from the UML models for FDR2 to verify deadlock freedom of component interactions. The authors also claim that application-dependent safety and liveness properties can be verified by model checking the state machines of the components. However, no details on how this is done are provided.

Data functionality is kept correct by applying refinement laws when creating more detailed designs of methods. Type checking is used to reveal type mismatches, but it is not stated whether an automatic refinement check is carried out with regards to pre- and postconditions. Additionally, model-based testing (see next characteristic) can be used to increase confidence in the data functionality.

Model–implementation consistency: The rCOS tool produces Java code for the executable program. There is also some work on performing model-based testing, i.e., generating test scripts from the rCOS models to test the generated Java classes [Lei+10].

Fault-tolerance expressiveness: The rCOS approach, as presented in [Ke+12], treats fault-tolerance mechanisms like any other functionality. This means that anything can be specified, and one example in the paper shows a voting component with three inputs (i.e., triple modular redundancy or TMR pattern) used in a fault-tolerant memory. In [Xu+12], the authors present two UML templates: the TMR pattern and the Recovery Block pattern. The latter is used to mask transient hardware failures by trying again later, so it is most likely not the same as Randell’s recovery blocks (see Sect. 2.2.3 and [Ran75]), which also employ different software versions to tolerate software faults.

Separation of concerns: In rCOS, provided and required interfaces of components are explicitly specified with both data functionality and ordering of method calls, so that components can be used by third parties without knowing the internal details of the components. Hence, any fault-tolerance
mechanism that can be encapsulated into a single component, can also be syntactically separated from the purely functional logic. It should also be possible to wrap functional components into fault-tolerant components, by adding extra layers of logic around them, providing some syntactic separation internal to a composite component. Note that any faulty behaviour is expressed the same way as normal behaviour, though. In [Xu+12], the authors design a fault-intolerant system first, and then replace some components with fault-tolerant versions. In that paper, there is also some additional syntactic separation in that the fault-tolerant components are stereotyped ≪FTComponent≫.

In the end, fault-tolerance mechanisms are just treated like any other piece of functionality, but the use of component contracts, allows to separate fault-tolerance mechanisms from functionality both in time and with regards to developers, and with some syntactical separation.

**Solution reuse:** The reuse of components also encompasses the proofs done with respect to the properties of the components.

**Model of system semantics:** Although not explicitly stated, it seems the rCOS approach assumes a model where components only communicate via synchronous (i.e., blocking) method calls. There is no discussion on any effects of distributing a system, so a synchronous communication model seems to be assumed.

**Fault model:** Faults can be specified in two ways. A component can have a guarded, internal, autonomous action that can trigger when its guard is satisfied. This action may simulate a range of faults by altering the component state and invoking methods at the wrong time. Alternatively, a component may have “fault methods” in its provided interface, which a fault process can invoke. This allows the fault process to coordinate faults across components to simulate more advanced fault behaviour.

**Platform constraints:** The current code generator produces Java code. The code generator also currently outputs a single monolithic program, not suitable for distributed deployment.

**Tool support:** The rCOS approach has tool support in the form of a set of freely available plug-ins [RCOS13] to the Eclipse platform, using, like many other tools, the EMF implementation of the UML meta-model, UML2 [UML2-12]. These plug-ins provide support for modelling, rapid prototyping and model transformations to other UML models, CSP and Java. The FDR2 model checker [FSEL13] is available for download and free for academic use. Commercial use requires a license.

**Case studies:** Earlier versions of the rCOS approach (with less tool support) have been applied to the CoCoMe example [Che+08; Che+09a], an enterprise sales

---

8In rCOS, a *process* is a component with no provided interface, typically used to pass data between components and coordinate their behaviour through synchronization.
system for which a range of modelling approaches were applied \cite{CoC13}. Parts of the approach have also been applied to medical systems \cite{XLD10} and a service-oriented internet shopping system \cite{LH06}. However, none of these case studies focus on fault tolerance and they all seem to be academic examples. The example presented in \cite{Xu+12}, does focus on fault tolerance, but there is no indication that it was ever implemented. So, while the general model-driven approach has been applied to a range of examples, there is still not much empirical data to support its applicability to fault-tolerant systems.

**System domain targeted:** Both transformational and reactive systems can be built with the rCOS approach, but component-based reactive systems seem to be the main target. Both software and software-intensive systems (e.g., embedded systems) are within scope of the approach. Currently, the approach seems most fitting for non-distributed programs, both with respect to available tool support and with respect to the assumptions made regarding synchronous communication between components via blocking methods calls (see Sect. 2.2.5).

### 2.5 Approaches Tied to Specific Fault-Tolerance Mechanisms

This section presents approaches that are tied to a fixed set of fault-tolerance mechanisms. In these approaches, the functional design or implementation is assumed pre-built outside of the approach. Hence, the modelling mainly consists of configuring the existing fault-tolerance mechanisms to fit the environment the system will be deployed in. Due to this, general verification support may not be necessary. These approaches are also more likely to be tied to a given execution platform, as they typically depend on an existing middleware implementation.

#### 2.5.1 GeneRative Aspects for Fault Tolerance (GRAFT)

The GRAFT approach \cite{TDG09} annotates structural component models with replication requirements so that a model including the necessary replicas can be automatically generated. From the new model, aspect code (see Sect. 2.2.1) is generated to be weaved into a CORBA-compliant middleware, the Component Integrated ACE ORB (CIAO) \cite{Wan+03}. In other words, this approach adds domain-specific knowledge to the middleware instead of the application, a paradigm the authors of \cite{Gok+08} call Model Driven Middleware (MDM). GRAFT is implemented within the CoSMIC tool suite \cite{Gok+08} for engineering distributed real-time and embedded systems.

**Type of model:** GRAFT uses the Interface Definition Language (IDL) from CORBA \cite{Obj11a} to describe the functional aspect of the system. The components
are assumed to be pre-built in an implementation language, IDL just adds a textual static interface description, like, e.g., a Java interface. Hence, there is no behavioural description at this level. The Component Availability Modeling Language (CAML) is used to model fault-tolerance requirements. This is a simple graphical domain-specific language developed by Tambe et al.\cite{TDG09} to annotate the structural model with the degree of replication required of different components.

**Fault removal:** There is no verification step in this approach. However, it may be that the system will be correct by construction, i.e., that the fault-tolerance requirements cannot be expressed in an inconsistent way.

**Property types and specification style:** The fault-tolerance requirements are expressed in CAML, and the only examples given of such fault-tolerance requirements are to require a component to be replicated. Note that these requirements are not really properties to be verified, rather something that guides the model transformation to a fault-tolerant model of the system.

**Model–implementation consistency:** The models are used both to generate an integrated model of the system (e.g., adding replicas to satisfy fault-tolerance requirements) and to generate aspect code to be weaved into the CIAO middleware.

**Fault-tolerance expressiveness:** The GRAFT approach primarily provides component replication as fault-tolerance mechanism. It is stated that warm-passive replication (see Sect. 2.2.3) is used in the example system. Most fault-tolerant CORBA implementations also support active replication and cold-passive replication (see also Sect. 2.2.3 and \cite{FNL07}), so it is probable that these are supported too. The replication can be tailored to the system domain. Components can be grouped to provide failover at a group level, useful for embedded systems where software components are tied to hardware sensors or actuators. Group failover means that a whole set of components take over for another set, when one component in that group fails. There are middleware-level exceptions that trigger failover, but also application-specific exceptions can be created by developers. As far as we can tell, application-specific exceptions are also used to trigger failover to replicas, hence all error handling sorts under compensation.

**Separation of concerns:** The CAML annotations, which express the fault-tolerance requirements, are only visible when looking at the model in the QoS view, whereas a structural view only shows the structure necessary for the functional aspect. Exceptions are also modelled in CAML, but no details are given on how application-specific exceptions are triggered. We suspect the triggering logic is written directly in the application code, even if the handling is modelled in CAML. If this is correct, the separation of concerns depends on whether application-specific exceptions are used. If not, the approach allows to add fault-tolerance requirements without changing the functionality, so concerns are time separated. We assume one could also have a separation of experts in this process, as the fault-tolerance requirements seem
to be constrained to replication, hence not requiring in-depth knowledge of the functionality. If application-specific exceptions are used, and the exception triggers are specified in the application code, there will still be some syntactic separation, and most likely triggers can be added after the functionality. However, writing the exception triggering logic most likely requires insight into the application code, so it should be done by the same person or team who wrote the code.

Solution reuse: We have not come across any claims of reuse for this approach. The application code itself should be reusable, with the possible exception of trigger code for application-specific exceptions. However, there are no features to make the job of creating the fault-tolerant system smaller the next time, nor any that facilitate reuse when building another application.

Model of system semantics: CIAO is built on top of the The ACE ORB (TAO), a CORBA implementation, which supports remote communication both via RPC and via asynchronous method invocations (AMI, [Obj12a]). Since there is no verification support in this approach, a model that the verification would be based on is not mentioned either.

Fault model: With regards to the fault model, the middleware detects and tolerates component crashes, and the application-level exceptions can be raised for any other fault type that can be detected. For example, application-level exceptions are used to detect and tolerate hardware faults in the conveyor belt system example system from [TDG09].

Platform constraints: The approach generates aspect code to modify the Component-Integrated ACE ORB (CIAO), a QoS-enabled implementation of the Lightweight CORBA Component Model (LwCCM). CIAO only offers a C++ API (also know as C++ bindings) to applications [TAO13]. Thus, the application must also be written in C++ or a language that allows to embed C++ code. The effort of porting it to other CORBA-based middlewares is not discussed, but in [Gok+08] the authors state the goal of including a platform-independent model (PIM) above the platform-specific model (PSM), such that they will be able to support different types of middleware.

Tool support: The GRAFT approach is integrated into the open-source CoSMIC tool suite [Gok+08], freely available from its web site [CoS13]. The CIAO middleware is also open-source and freely available for download [CIAO06].

Case studies: A warehouse distribution example system in [TDG09] constitutes a small case study. As there is no mention of industrial use, it is most likely an academic example.

System domain targeted: The approach seems to be appropriate for large-scale, distributed, real-time, embedded systems.
2.5.2 Aurora Management Workbench (AMW)

In [BG07], Buskens and Gonzalez present an approach that allows developers to describe high-availability properties in a declarative way, separated from the functional design of the application. The authors achieve this through a high-level domain-specific language combined with code generators for a high-availability middleware. They motivate their approach by noting that creating highly available distributed systems requires that the developers are experts in the system domain as well as in fault tolerance. To reduce these demands on the developers, the authors therefore wish to simplify the process of implementing and integrating fault-tolerance mechanisms.

**Type of model:** The AMW has its own textual domain-specific language (DSL). This language captures both architectural (structural) and run-time behavioural attributes with respect to fault tolerance. As the language is declarative, it is not used to specify the details of how to achieve the results. Instead, it is used to configure a set of parameters of pre-made fault-tolerance mechanisms. For example, you can state dependencies of components on each other and define what kind of replication strategy should be applied to a component. The application functionality itself is written in C++.

**Fault removal:** As the DSL for specifying availability attributes comprise both behaviour and structure, one can automatically check consistency between the two[9] although it is unclear whether this is already implemented. Integration with tools to build formally verifiable models is only mentioned as future work.

**Property types and specification style:** This characteristic is not applicable for this approach.

**Model–implementation consistency:** Code generators create most of the code to integrate the application with the middleware, which in turn handles things like checkpointing, component migration and error detection. Hence, the risk of manual coding errors is greatly reduced, as is the risk of inconsistencies between actual API calls to the middleware and the configuration of the middleware. Hooks for fault injection are provided in the generated code, to ease testing the system under faulty conditions.

**Fault-tolerance expressiveness:** Heartbeats and application-specific fault monitors are used by the middleware to detect faults. The latter are a result of the application overwriting callback methods that the middleware calls to check for problems. Components can be completely non-protected (if they crash, they are left as they are), restarted after crashing, or replicated. The only replication strategy mentioned in [BG07] is what they call “active standby”;

[9]Example: If component A is dependent on component B, and both are located on node N1, then it does not make sense to specify that the crash of node N1 leads to failover of component A to a replica on a different node, while component B is not replicated.
which is the equivalent of warm-passive replication (see Sect. 2.2.3). In the case of active standby, the middleware ensures that the backup replica is placed on a different processor. The placement of replicas can also be constrained by performance requirements, using the current load of a processor as criterion. Failure groups are supported, to allow groups of components to be treated as an atomic entity with regards to failure (i.e., switch out all in the group, if one fails). The approach also supports declarative fault-escalation policies (i.e., exception handling), even across nodes. However, there is no mention of any additional error handling primitives, leading us to believe that the exception handling feature is used to structure the sequence of error handling attempts, still restricted to restarting a component or failing over to a replica. In summary, rollback and compensation are both supported by this approach. It is not clear whether the middleware automatically compensates by resubmitting ongoing requests once a failover to a backup is completed, or whether this must be done by clients themselves.

One potential drawback of the approach is that there is a centralized configuration manager in each system that handles reconfiguration when errors are detected. While this configuration manager can also be replicated using warm-passive replication, one can imagine scenarios where the failover time suffers from having to first fail over the manager and then start reconfiguring the rest of the system. This strategy relies on infallible communication channels, so that the configuration manager can always reach the nodes it wants to reconfigure. There is also a feature of fault isolation [Avi+04] that uses knowledge of communication links to isolate faulty components, and this also assumes infallible communication channels, as discussed under Model of system semantics below. Hence, this is one the only approach that focuses on fault handling in addition to error handling (see Sect. 2.2.3).

This approach focuses heavily on run-time upgrades, providing advanced features to change the configuration of the system without taking it offline. This enables a system to stay operational even when doing maintenance, an important property for truly high-availability systems.

Separation of concerns: The choice of replication strategy is completely separated from the application logic, so that different instances of an application could be run in different environments with different availability requirements without changing the application code. The application is mainly responsible for returning the data structure to be stored in the checkpoint when a callback method is called by the middleware. The storage, transfer and synchronization of checkpoints are handled automatically by the middleware. As already mentioned, the application may also overwrite callback methods to implement application-specific fault monitors. This is probably a job that requires insight into the application code, hence best left to the people who wrote it.

Solution reuse: We have not come across any claims regarding reuse for this approach. While the application logic itself can be reused for the same
system with different availability requirements, there is no mention of any mechanisms that would make the job easier the second time around.

**Model of system semantics:** Even though there is no formal verification step in this approach, the authors detail their assumptions about the model of system semantics, including the fault model, as the fault-tolerance mechanisms of the middleware rely on these. Components are stated to communicate via message passing, although it is not specified whether the message passing is synchronous or asynchronous. Inter-processor communication links do not fail, and failure detection is accurate.

The last two assumptions are not straightforward. In general, one should not assume infallible communication links unless one has high hardware-level redundancy in the form of multiple independent links [BG07]. Hence, the assumption of infallible links constrains the applicability of the approach. Stating that failure detection (of node crashes) is accurate, builds on top of the infallible communication assumption, but also implies that there is an upper time bound after which it is guaranteed that any events signalled from another node would have been received, i.e. at least a partially synchronous model of system semantics. One may trust such assumptions in a local network where one controls the load, but it is not a suitable assumption across the Internet, for example.

**Fault model:** Processors and processes crash silently and do not generate random output. Any number of processes and components may fail at the same time.

**Platform constraints:** As a consequence of the model of system semantics, the platform must provide perfect communication links. Further, it seems the code generators currently generate C++ code. There is a CORBA API for application messaging, but it is not clear whether using this is required or just an option for messaging.

**Tool support:** At the time of writing in 2007, the AMW is a standalone tool, but integration with Eclipse is mentioned as future work. The tool was developed at Bell Labs and is now used by Alcatel-Lucent. They list AMW on their webpage [AMW13], but there is no mention of licensing or downloading it, implying that it is only available in-house.

**Case studies:** In [BG07], the authors report on an experiment where they made a small high-availability system in two hours and under twenty LOC using the AMW, while needing approximately two weeks and a thousand LOC to do the same using a commercially available high-availability middleware. More significantly, the AMW has already been used to make many field-deployed products: “*AMW is used in large, deployed telecommunications products consisting of several million lines of code comprising a few thousand software components and many tens of thousands of communication links executing on tens of processors*” [BG07].

**System domain targeted:** The approach is devoted to the development of large telecommunications call processing systems. In this domain, the system
owner is usually also the network owner, so the assumptions in the model of system semantics seem to be reasonable.

2.6 Comparison Tables

This section presents a summary of the preceding survey in table form. Naturally, nuanced differences between approaches are not captured in such a form, but the tables should serve as a quick way to filter out approaches that are (or are not) interesting for a given reader. We have omitted two characteristics, Model of system semantics and Case studies, from the tables, as we felt that reducing these characteristics to a few words would present a very inaccurate picture.

A line enclosed in brackets means that the main work cited for the approach did not use or mention this feature, but we found work related to the approach on it. This could be work by the same research group, or even others who work on the same models and tools. From experience, we know that, in academia, a tool-supported approach easily becomes the starting point for many additional developments that may not always be compatible with each other. Hence it could be possible to also take advantage of the bracketed features in the approach, but the bottom line is that we have not seen any explicit demonstration of combining them with those in the main work.
<table>
<thead>
<tr>
<th>Approach</th>
<th>Type of model</th>
<th>Fault removal</th>
<th>Property types and specification style</th>
</tr>
</thead>
</table>
| Extended Charmy | UML use cases  
               Property Sequence Charts  
               UML component diagrams  
               UML state machines   | Model checking  
                       (Model simulation)  
                       (Model animation) | Safety and liveness properties both expressed using Property Sequence Charts. |
| SPACE        | UML collaborations  
               UML activities  
               (Extended) External state machines | Model checking  
                       Model animation | Automatically generates a fixed set of safety properties. Safety and liveness properties can be expressed in TLA. |
| Event B      | Event B (graphic syntax also exists)       | Theorem proving of refinement steps  
                       (Model checking)  
                       (Model animation) | Safety properties expressed via refinement. |
| Mechatronic UML | UML components  
               Real-time Statecharts  
               Hybrid Reconfiguration Charts  
               Block diagrams  
               Story diagrams | Model checking  
                       Hazard analysis  
                       Simulation | (Timed) Safety properties are expressed in ATCTL. |
| iFTElement   | UML use cases  
               UML component diagrams  
               UML sequence diagrams  
               UML activities | Model checking (of scenario state space) | A fixed set of syntactic constraints and safety properties are checked automatically. User-defined safety properties can be added, but it is not clear how they are expressed (possibly in B and CSP). |
| Motorola WEAVR | UML classes  
               UML/SDL component diagrams  
               UML/SDL state-centric state machines  
               UML/SDL transition-centric state machines  
               Action language | Model checking  
                       Simulation  
                       Model animation  
                       Aspect effect visualization | A fixed set of syntactic and semantic constraints are checked automatically. User-defined first-order logic invariance properties can be specified in the action language. Test cases for simulation are created manually or from abstract behaviour specification and expressed as sequence diagrams or in TTCN-3. |
| MOCAS        | UML class diagrams  
               UML state machines | Model execution engine with run-time model consistency checking  
                       Model animation  
                       (Model checking) | (LTL properties can be specified in terms of the input model for the model checker) |

Table 2.1: Type of model, Fault removal, Property types and specification style, part 1
2.7 Discussion

Having presented each approach in detail, as well as summarized them in the tables in the previous section, we now discuss our findings, point out what the future may bring and end with a part discussing the method behind the survey itself. Up to this point, we have tried our best to be as objective as possible in our description of the approaches. In what follows, we allow ourselves to present also our subjective opinions.

2.7.1 Survey Results and Direction of the Field

In the following, we discuss the results of the survey and try to extract any interesting findings. We structure the discussion roughly around the various characteristics, but there is no one-to-one mapping. When we state the number of approaches with a specific feature, we will sometimes put another number in brackets. This is the number of approaches that qualify if we also count the bracketed lines from tables 2.1 to 2.5 (see Sect. 2.6 for explanation of bracketed lines). Where applicable, we also present some thoughts on the future direction of the field. Here, and in the following, the field means the field of research on how to take advantage of model-driven engineering to build reliable and fault-tolerant systems.

Types of models Among the approaches surveyed, UML is used extensively. This is not surprising, since UML may be seen as a consolidation effort of many

<table>
<thead>
<tr>
<th>Approach</th>
<th>Type of model</th>
<th>Fault removal</th>
<th>Property types and specification style</th>
</tr>
</thead>
<tbody>
<tr>
<td>rCOS</td>
<td>rCOS textual language UML use case diagrams UML component diagrams UML sequence diagrams UML state machines UML class diagrams</td>
<td>Automatic prototype for validation Model checking Fault tree analysis</td>
<td>Consistency between viewpoint models checked automatically. Authors claim both safety and liveness properties can be checked, but do not state how they are specified. Data functionality refinement according to refinement laws.</td>
</tr>
<tr>
<td>GRAFT</td>
<td>Interface Description Language (IDL) Component Availability Modeling Language (CAML)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>AMW</td>
<td>AMW-specific textual DSL</td>
<td>Limited verification</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 2.2: Type of model, Fault removal, Property types and specification style, part 2
Table 2.3: Model–implementation consistency, Fault-tolerance expressiveness, Separation of concerns

modelling languages preceding it, and UML embraces all three major graphical description techniques for behaviour, i.e., state machines, sequence diagrams and Petri nets. The latter are represented in UML by the activity diagrams. We notice that if an approach uses executable models at some point, these are mostly specified by some form of state machines.

The popularity of UML probably has something to do with the ease of finding and extending tool support. With the advent of foundational UML (see [Obj11c] and Sect. 2.2.1), the need to provide one’s own formal semantics will be less pressing as well. Hence, we predict that UML will become even more widespread in this field, in the time to come.
<table>
<thead>
<tr>
<th>Approach</th>
<th>Solution reuse</th>
<th>Fault model</th>
<th>Platform constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extended Charmy</td>
<td>Components with contracts (but possibly without implementation)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>SPACE</td>
<td>Collaborations with contracts + verification effort</td>
<td>Supports node crashes (with restart), message loss, message reordering</td>
<td>Code generation for Java, Java ME and Android.</td>
</tr>
<tr>
<td>Event B</td>
<td>Modules with contracts + verification effort</td>
<td>N/A</td>
<td>Code generation for Ada, C and Java.</td>
</tr>
<tr>
<td>Mechatronic UML</td>
<td>Coordination patterns + verification effort</td>
<td>Crash, timing and value failures. Assumes no clock drift.</td>
<td>Code generation for Java and C++. Requires real-time platform.</td>
</tr>
<tr>
<td>iFTElement</td>
<td>Business logic + components with contracts</td>
<td>Application-level exceptions can be raised for anything that can be locally detected by a software component.</td>
<td>N/A</td>
</tr>
<tr>
<td>Motorola WEAVR</td>
<td>Functional components + fault-tolerance aspects</td>
<td>N/A</td>
<td>Code generation for C, C++, C# and Java.</td>
</tr>
<tr>
<td>MOCAS</td>
<td>Business logic + components with contracts</td>
<td>N/A</td>
<td>MOCAS engine available for Java and J2ME.</td>
</tr>
<tr>
<td>rCOS</td>
<td>Components with contracts + verification effort</td>
<td>N/A (Anything that can be expressed as one or more method calls on components)</td>
<td>Code generation for Java and only produces non-distributed programs.</td>
</tr>
<tr>
<td>GRAFT</td>
<td>Business logic</td>
<td>Middleware detects component crashes. Application-level exceptions can be raised for anything that can be locally detected by a software component.</td>
<td>Code generation for CIAO (Lightweight CORBA Component Model). Application must be able to call C++ API.</td>
</tr>
</tbody>
</table>

Table 2.4: Solution reuse, Fault model, Platform constraints
<table>
<thead>
<tr>
<th>Approach</th>
<th>Tool support</th>
<th>System domain targeted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extended Charmy</td>
<td>Charmy: Freely available.</td>
<td>Component-based fault-tolerant systems</td>
</tr>
<tr>
<td>SPACE</td>
<td>Arctis: Eclipse-based and available for academic and commercial use upon request.</td>
<td>Fault-tolerant, distributed, reactive systems</td>
</tr>
<tr>
<td>Event B</td>
<td>Rodin: Eclipse-based, open source and freely available.</td>
<td>High-reliability reactive systems</td>
</tr>
<tr>
<td>Mechatronic UML</td>
<td>Fujaba4Eclipse: Eclipse-based and freely available. UPPAAL and CAMeL-View require licenses for commercial use.</td>
<td>Embedded, real-time, hybrid systems</td>
</tr>
<tr>
<td>iFTElement</td>
<td>Unspecified tools for creating and transforming models. ProB: Open source and freely available. Commercial support available.</td>
<td>Single-node, non-concurrent, component-based systems. Also supports COTS/legacy components.</td>
</tr>
<tr>
<td>Motorola WEAVR</td>
<td>WEAVR: Internal to Motorola, but free research licenses can be obtained. Depends on Tau. IBM Rational Tau: Commercially available.</td>
<td>Reactive discrete systems for telecom infrastructure.</td>
</tr>
<tr>
<td>MOCAS</td>
<td>TopCased, MOCAS plugin for TopCased and MOCAS engine: Open source and freely available. TopCased: Eclipse-based</td>
<td>Distributed, reactive, embedded systems, possibly deployed in a changing environment.</td>
</tr>
<tr>
<td>rCOS</td>
<td>rCOS with plug-ins: Eclipse-based and freely available. FDR2: Free for academic use and commercial license available.</td>
<td>Component-based non-distributed systems</td>
</tr>
<tr>
<td>GRAFT</td>
<td>CoSMIC: open source and freely available.</td>
<td>Large-scale, distributed, real-time, embedded systems</td>
</tr>
<tr>
<td>AMW</td>
<td>AMW: In-house tool at Alcatel-Lucent.</td>
<td>Large-scale, high-availability, telecommunications call processing systems.</td>
</tr>
</tbody>
</table>

Table 2.5: Tool support, System domain targeted
Fault removal  Model checking seems to be the prevalent technique for fault removal in the modelling stage, with six (eight) approaches employing it. This is most likely a result of it being easy to automate, hence lowering its threshold for use. We see that the approaches from Sect. 2.5 do not offer much in the way of fault removal at all. This may be because they, at least the GRAFT approach, restrict the developers sufficiently that it is nearly impossible to specify fault-tolerance mechanisms in an inconsistent way.

Our opinion is that a combination of model checking an abstracted model, combined with later model-based testing, will suffice to ensure their reliability and correctness of all but the most critical of systems. The alternative is to employ theorem proving to prove the correctness of an unbounded model of the system. Should theorem proving become sufficiently automated and accessible, it may become a practical alternative free from the inherent scalability drawbacks of model checking. Even so, we see theorem proving complementing, not replacing, model checking, as the latter provides useful debugging support in the form of error traces.

Safety versus liveness properties  Less than half of the approaches that verify safety or invariance properties also verify liveness properties. For clarity, we give examples of two safety properties and one liveness property for a typical request–response interaction:

S1: Response is not received before request is sent.
S2: Response is only received once.
L1: If request is sent, eventually response is received.

One could argue that verification of liveness properties is more of an academic endeavour, which is of little real-life importance. After all, you could specify most practically important liveness properties as “event A must happen within X steps of event B”, rather than “event B implies that event A eventually happens”. However, this means we must include a notion of “number of steps” or “amount of time”, to track how long has passed since an event. Liveness is then a useful way to specify properties such as boundedness of resource needs, eventual termination and absence of livelocks without having to resort to this extra time tracking, which will complicate specifications. Similarly, when dealing with fault tolerance, we cannot prove properties like “every request gets a response within X steps”, unless we also make some assumptions that messages are only lost a certain number of times within those X system steps. This further complicates our model. Instead, we might use fairness assumptions to state that e.g., a message that is sent infinitely often, is infinitely often received. We may then prove liveness properties based on this assumption. Proving properties under such assumptions is useful, even if the real system may not guarantee the fairness assumptions we make, because we at least get a guarantee that there is nothing but these weak assumptions that keep the property from holding.
Property specification styles  With the exception of Charmy, which features Property Sequence Charts, none of the approaches have a graphical notation to specify application-specific properties to be verified. In the cases where such additional properties are needed, we imagine this to severely increase the threshold for practitioners to use the approaches. On the other hand, many approaches check a set of general properties automatically, perhaps considering this set of properties good enough for what is needed in practise. For critical systems, we believe that developers will want to specify their own additional properties, so that a possibility for this should be provided. In the approaches where properties are generated from component contracts, one could say that this represents a middle ground, in that it provides an indirect way for developers to specify a limited set of application-specific properties, without delving into the input language of the verification tool.

Code generation and platform independence  Many of the approaches that offer code generation use hybrid approaches, which complement models with manually written code where needed. This means that some parts of the code are generated (mostly from some form of state machines and structure models), while detailed operations are expressed directly on code. For instance, SPACE, WEAVR and MOCAS use UML state machines or UML activities to specify concurrent behaviour, while the body of operations is given directly in Java, C or C++. This is interesting, since an often mentioned benefit of model-driven approaches is platform independence, i.e., the fact that higher-level models can be transformed into code for several platforms. One may see this as a practical shortcut that these approaches take; after all, the bodies of operations could also be expressed by some more general action language from which different programming code can be generated. On the other hand, one may argue that platform independence is a less important feature of model-driven engineering than generally assumed.

Execution and scheduling  Concerning the scheduling of executions, the surveyed approaches use a wide spectrum of techniques. At one side of the spectrum is MOCAS, which uses a model interpreter. It takes EMF UML models as input, and can execute both plain UML state machines and the adaptive MOCAS components. SPACE uses a run-time support system that handles scheduling of the events for state machines. The execution is based on run-to-completion semantics of the individual state machine transitions, which makes it possible to execute many state machine instances within the same thread or operating system process. In Mechatronic UML, on the other hand, state machines can be interrupted mid-transition. This scheduling complicates the semantics, but is necessary to enable reacting to new events within the time bounds set in many real-time systems. Approaches that only generate interfaces and let the developer implement manually, leave it up to the developers to ensure that the scheduling semantics matches that of the model. Failing this, properties verified correct in the model may not hold for the executable system. Finally, some approaches require special middleware ded-
icated to fault tolerance, which also determine some aspects of scheduling.

Model–implementation consistency To perform exhaustive verification, all approaches use a model that abstracts away from the executable system. In many cases, this means abstracting away data and focusing on the control flow of the system. By using code generation, one can then guarantee that subtle behavioural properties of the models are preserved, even those that may be hard to test for violations of, like absence of race conditions. However, we must also exercise the detailed operations on data abstracted away in the verification step, to ensure that they do not contain bugs in terms of the data contents, memory accesses or timely termination (e.g., infinite loops through a data structure). Only two (four) approaches feature model-based testing, whereas six (eight) feature some form of code generation to ensure model–implementation consistency. No approaches explicitly integrate both, but there are papers on both Charmy, Motorola WEAVR and rCOS for both code generation and model-based testing. We believe the lack of support for both is typically a matter of maturity of the approaches (elaborated below). As an approach is put into industrial use, the practical advantages of having both will likely become evident, and support for the missing technique will be added.

Fault-tolerance expressiveness We see that the majority of approaches support or have published examples of rollforward fault-tolerance mechanisms. Half of them also support compensation, whereas only two explicitly facilitate rollback. We believe that a reason for the lacking support of rollback may be that this typically affects all application logic, unless it can be hidden in an underlying layer. An alternative could be to employ aspect-oriented modelling, and the WEAVR approach may well support rollback, even if it is not given as an example. We also note that the approaches from Sect. 2.5 do not support rollforward. This is quite natural, as they do not model the functional behaviour. Thus, it is difficult to react to an error in an application-specific way.\footnote{Both approaches allow for application-specific monitors to trigger error handling, but this is different from reacting in a way tailored to the application.}

Separation of concerns All graphical notations have at least some syntactical differentiation between fault-tolerance logic and functional logic, whereas the textual Event B approach does not. Eight approaches also feature some time separation, allowing to mainly add fault-tolerance after completing the functional part of the system, possibly with some tweaking of the functionality afterwards. Support for designing the fault-tolerance logic without studying the details of the functional logic is given by five (six) approaches. This latter point ties in strongly with solution reuse, discussed below.
We are of the opinion that separation of concerns is one of the largest obstacles to overcome in this field, at least for approaches not tied to a specific middleware (see Sect. 2.4). This becomes an issue when incorporating complex fault-tolerance mechanisms, like active replication or checkpointing. One solution could be to plug in middleware-level mechanisms represented by model elements that formally describe how the mechanisms affects the system, so that general verification can still be performed. Aspect-oriented modelling is another alternative that could hold the solution for this. In the future, we hope to see papers describing large-scale case studies that demonstrate good separation of concerns while applying complex fault-tolerance mechanisms. Naturally, many systems need only a basic level of exception handling, in which case many of the surveyed approaches already seem to have adequate solutions in place.

**Solution reuse** In table 2.4, we have tried to separate the approaches into different levels of reuse. Most of the surveyed approaches advertise solution reuse features beyond potentially reusing business logic. That means they really create something within the approach that can be reused in other systems. We also note that four approaches are able to reuse the verification effort of previously verified components, hence greatly reducing the time needed for formal verification. The approaches that are tied to specific fault-tolerance mechanisms (see Sect. 2.5) could claim that they do not need to focus on reuse, as what they already start from the pre-implemented business logic and, in a sense, are already reusing fault-tolerance mechanisms that are embedded in the middlewares they are tied to. We believe that strongly supporting reuse of partial solutions holds the key to the successful application of MDE in real systems.

We believe that a tendency to reuse is one of the advantages of MDE, at least when using graphical models: It typically forces designers to split functionality into small components with clearly defined interfaces, so that they can be viewed in isolation. Such a nicely decomposed system typically offers more reusable pieces than a large monolithic one.

**Formal models of system and fault semantics** Many of the surveyed approaches do not state the assumed model of system semantics, so it is difficult to make much of a comparison. However, we do note that publications that are explicit about it, are mainly the ones from approaches that assume unbounded communication delays and message passing, i.e., the most general ones. This may be surprising, as one would think that the stricter assumptions that are made in an approach, the more explicitly these would be conveyed. However, it may very well be that approaches only considering bounded communication delays (typically non-distributed systems), do not see themselves as part of a bigger field also covering distributed systems without such guarantees. We would hope future work in the field to be more explicit in terms of the model of system semantics, as it would surely aid in understanding and comparing the various approaches.
As we already touched upon in Sect. 2.2.5, we believe confining the developers to a call–return protocol like RPC, is a bad choice if distributed systems are being developed. This entails that CORBA-based approaches should make sure they support asynchronous method invocations (AMI, [Obj12a]), so that developers have an alternative to RPC. The drawback of allowing concurrency is naturally that the complexity of the design can be much higher, but with the aid of formal verification techniques, like most MDE approaches feature, the resulting risks can be mitigated effectively.

With respect to the fault models, we see a clear divide between the approaches in Sect. 2.4 and the two in Sect. 2.5: The latter support a fixed set of fault-tolerance mechanisms, hence also deal with a fixed fault model for which these mechanisms are to work. The former approaches are much less specific with respect to types of faults. Here, the idea is often for developers to design the fault model, just like the system itself, and then use the verification techniques provided to ensure the system is fault tolerant.

**Tool support**  We note that half the tools are based on Eclipse, indicating that this is the most popular platform to build tools on top of. We also note that three out of the four approaches that use UML models and are based on Eclipse, also use the EMF implementation of the meta-model for UML. The approaches using EMF have an advantage in that they may gain extra features from external tools built for EMF UML. An example is the diffing and merging of EMF models [EDM13]. Coupled with the upwards trending use of UML, we imagine that the use of Eclipse and the EMF UML meta-model will also become even more widespread.

We are of the view that better tools are needed to encourage further industry adoption. However, we also recognize that the resources to build such tools are outside the reach of many academic research groups. Hence, we anticipate that the solution lies in joint academia–industry projects ensuring both extra development resources and a calibration of existing tools to industry needs.

**Origin and maturity of the approaches**  The approaches surveyed can be grouped according to whether they started out primarily focusing on formal methods, model-driven engineering or fault tolerance. Only Event B started out focusing on formal methods, while both iFTElement and and AMW approaches started out mainly focusing on fault tolerance. The remaining approaches all started out focusing on MDE.

If we turn it around and look at what is the least mature or last published part of the approach, we get a slightly different view. There are no approaches that employed MDE last, while the MOCAS, GRAFT, AMW and iFTElement approaches included formal methods only at a later stage. The remaining approaches focused on both MDE and formal methods before looking to fault tolerance.
Why is fault tolerance often thought of later on? We think this is quite natural, at least for the approaches that stem from academia. Research is often done in batches that can fit within a PhD, so it is typical to leave something complex like faults out of the scope of an initial approach. Once the underlying theory and some tool support is in place, one then might start looking for larger case studies and find that mechanisms to handle errors are often needed. This may spur new research efforts built on top of the existing results, which seems to be the case for many of the approaches in the survey. We hope that this survey may help (future) researchers quickly get insights into both the work that has been done and the work that has not yet, so as to hopefully start directly at attacking the domain of fault-tolerant reliable systems, rather than starting from scratch again.

With respect to maturity, this is still research. Most approaches are academic; only a few are used in industry or even originate there. However, none of them can be said to be enjoying widespread industrial use. This lack of industry acceptance is common also for model-driven approaches in general, and the use of formal methods.

Does this mean that the field is a dead end? We believe that this is too soon to call. Since neither MDE nor formal methods are very mature yet, seen from an industry standpoint, we cannot expect the combination of them to build fault-tolerant systems to be either. As already mentioned, many approaches started out without considering fault tolerance, focusing instead on MDE or formal verification. In fact, several approaches have just turned to fault tolerance in the last few years. Hence, we cannot expect the resulting approaches to be more mature than the foundations they build upon.

**Missing pieces** From the field of fault-tolerant systems, there are a variety of algorithms, middlewares and mechanisms to aid in making a system fault-tolerant. Although we have only surveyed a limited number of approaches, we were somewhat surprised not to find any approaches that tie in with some well known contributions from the fault-tolerance field. For example, we have not found any approaches that use any form of N-version software (see Sect. 2.2.3) to tolerate software faults. This could simply be because N-version software is seen as redundant if one also uses formal verification of models. Ideally, formal verification will remove all software faults, although in practice they will not, for reasons mentioned above with respect to model-based testing.

Group communication systems (GCSs) \[\text{Bir93}\] provide a range of communication primitives that guarantee the ordering and reliability of (multicast) communication. Such primitives can be used to implement active replication, for example. **Mechatronic UML** assumes a middleware that ensures that replicas get messages in the same order and at the same time, but none of the other approaches seem to take advantage of such middleware. The GRAFT approach does use FT CORBA, and some FT CORBA implementations use a GCS to implement totally ordered multicast \[\text{FNLO7}\]. However, the CIAO ORB does not seem to have been
built on top of a GCS like this.

Another example is the UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms (QFTP, [Obj08]), which is an OMG standard since 2006. The profile can be said to not cover the whole field of fault tolerance, focusing only on replication. Still, not even any of the five approaches that do feature compensation-style fault-tolerance mechanisms mention the profile.\footnote{Note that we are aware of papers referencing the profile (e.g., [EC07] and [RRE04]), but this point, as the rest of the discussion, is only related to the approaches surveyed.}

Much research in the fault-tolerance domain is on how to distribute databases for reliability, while still keeping data consistent. Active replication, combined with deterministic state machines (see Sect. 2.2.3 and [Sch90]) can take care of this, so one could say that the approaches supporting this kind of compensation would also support distributed databases. However, we have found none that actually focus on this. Perhaps it is because the domain of databases has traditionally been separated from that of other reactive systems. Naturally, a database system is reused over many other applications, so the need for building them quickly and reliably for each software project is just not there, but one would think an MDE approach for building applications on top of such databases could be useful. This is especially the case if the database can be configured to provide different trade-offs between performance and data consistency. Then it would be useful to have a model that captures the relevant dependencies and an analysis tool for verifying that problems will not occur.

Similarly to the lack focus on databases, is the lack of focus on fault tolerance in big data processing. Algorithms like MapReduce [DG08] allow large numbers of compute nodes to cooperate on data processing in a distributed manner. Most implementations of MapReduce feature fault-tolerance mechanisms that prevent the failure of a single node to affect the whole computation much, but we have found no approaches, fitting the scope of the survey, that look at how to model such systems. Again, it may simply be because the norm is to build ones application on top of a well-tested middleware providing high-level primitives that are easy to use in a consistent manner. A system only performing a MapReduce computation would also be a transformational system, implying that failures could simply be dealt with by restarting the system. In that respect, having mechanisms for tolerating the failure of nodes can be seen as more of a performance optimization than the fulfilment of a reliability requirement.

Several of the previous points can be summed up in that we were surprised not to see more approaches in Sect. 2.5 using MDE to improve the building of applications on top of proven middleware. On the other hand, our narrow scope may have excluded interesting contributions with respect to some of the things we point out as missing pieces.
Establishment of the field  From the choice of publication venues, the re-
search in this field seems to be quite fragmented. This is a natural starting point,
as the field crosscuts several more established fields of research. We hope to see
some consolidation in the field, leading more new research to be built on top of
existing work, rather than starting from scratch. The already evident gathering
around UML as notation and Eclipse as tool base will help in that. More sur-
veys like this one will also aid in outlining the research field and hopefully lead to
broader collaborations among the research groups within it.

2.7.2 About our Survey

We are ourselves the authors of one of the approaches being surveyed, SPACE.
While we aim for an objective presentation of all approaches, the survey should be
read with this in mind.

In finding work to review, we have mostly employed Google Scholar \cite{Sch13}, but
also gone straight to some papers we were already aware of, as well as searching
for papers citing those we already found to be within scope. Our search criteria
were mainly that the papers should be published in 2006 or later, while also fitting
the remainder of the scope (see Sect. \ref{sct:scope}). Since our scope is quite narrow, there is
much work that is clearly related to the field, yet not quite within the scope of the
survey. Due to time and space limitations, we decided against including shorter
descriptions of this. We may also have missed some publications that are clearly
within the scope, but the scope of all papers published since 2006 is much too big
to search exhaustively.

We have tried to focus on one or two main papers describing each approach, but in
some cases we had to go wider to look for things omitted in the main papers. Doing
this, we sometime discover additional features not necessarily integrated with the
approach described in the main paper, but that nevertheless could be. Hence, one
can argue that the most self-contained papers are punished in that we survey only
their contents, while approaches published more fragmentarily are given a broader
investigation to piece together the puzzle.

Even though we have gone relatively deep into each approach, we have not made
any attempts at ranking \textit{how good} they are, beyond listing their features. This
is partly since we have not had time to actually use them in practise, hence we
are relying on the statements made by the authors and the deductions we draw
from them. Being authors of one approach is another reason not to delve into this.
Finally, we believe a ranking would anyway not be of much use, as the approaches
differ so much that they should be ranked with respect to the specific use case the
reader may have.

We have chosen the characteristics by which we present the approaches in a prag-
matic manner, to make it easy to quickly look up things we believe to be of practical
2.8. Conclusion

To improve the reliability of a system, we can add fault-tolerance mechanisms, so as to tolerate faults that cannot be removed at design-time. However, the resulting rise in complexity increases the probability of software faults being introduced. Hence, unless the process is handled carefully, adding fault tolerance may even lead to a less reliable system. Many research groups are trying to develop practical approaches for incorporating fault-tolerance mechanisms with the functional parts of systems. As a way to deal with the inherently high level of complexity of such systems, some have turned to the paradigm of model-driven engineering. This results in a research field that cross-cuts the established fields of software engineering, system verification, fault tolerance and distributed systems. Many works are presented in the context of one of these traditional fields, making it difficult to get a good overview of what is presently offered. We have therefore surveyed 10 approaches for model-driven engineering of reliable fault-tolerant systems and decided on 13 characteristics that we claim to classify the approaches in a manner useful for both users and developers of such approaches.

We have found it natural to group the approaches into those that allow developers to create their own fault-tolerance mechanisms and those that are tied to a fixed set of mechanisms provided by an existing middleware. The first group allows developers more fine-grained control, but makes the integration of the fault-tolerance mechanisms more difficult. These approaches aid developers by various forms of verification methods, so as to ensure that the system is built correctly. The latter group abstracts away from the system behaviour and focus on fault-tolerance mechanisms that do not need to be tightly integrated with the functional behaviour. They also offer domain-specific languages to easily configure the fault-tolerance mechanisms as desired.

Most approaches are from research groups at universities, and they focus on demonstrating a few new capabilities, rather than consolidating all existing ones into their approach. As a result, none of the approaches can be said to be a superset
of any other one, and choosing any one approach will likely result in a compromise of some sort. For anyone with intentions of using or developing an approach for model-driven engineering of reliable fault-tolerant systems, this survey should therefore provide a good starting point.

References


[AMW13] Aurora Management Workbench. 2013. URL: http://www.alcatel-lucent.com/wps/portal/!ut/p/kcxml/04_Sj9SPykssy0xPLMmZ0vKQjzKld4w3cg3RL8h2VAQAG6J0Gw!!?MSG_CABINET=Bell_Labs_Technologies&MSG_CONTENT_FILE=Technologies/Bell_Labs_Tech_Detail_000028.xml&MSG_PARENT=Technology_Areas/Bell_Labs_Tech_Area_000006.xml&MSG_GPARENT=ROOT (visited on 21/03/2013).


References


References


11. ISBN: 978-1-4503-0289-0. URL: http://doi.acm.org/10.1145/2401736.2401737


References


[WeL06] Motorola WEAVR - LICENSES. 2006. URL: [http://mypages.iit.edu/~concur/weavr/licenses.html](http://mypages.iit.edu/~concur/weavr/licenses.html) (visited on 21/03/2013).


Chapter 3

Context and Research Design

This chapter contains an overview of what we consider prior work to this thesis. Further, we take a more in-depth look at the research questions.

3.1 Starting Point of the Work

The SPACE method (see Paper 3) and its tool suite, Arctis, have been under development since 2005, at the Department of Telematics, NTNU. Most of the work in this thesis has been done in collaboration with Peter Herrmann and Frank Alexander Kraemer, who have worked on the SPACE method since before I began my PhD. Since then, the method has kept evolving, both as a result of contributions I played a part in, and those I did not. It is therefore not trivial to extract the state of the method before my involvement. The following therefore points out what is considered prior work.

We start by answering a question: Why did we consider SPACE a good starting point for a method to create reliable, fault-tolerant, distributed, reactive systems? At the time this work began, in January 2007, the SPACE method could already be described as follows:

SPACE combines model-driven engineering with decomposing systems into collaborative services. That is, in SPACE, a system is built by

---

1 Lately, it has also been commercialized by Bitreactive AS [Bit13b], and the tool has been renamed Reactive Blocks SDK.

2 As already mentioned in Sect. 1.2, we sometimes use the singular personal pronoun, when it fits much better than the plural.
composing building blocks that may describe the distributed behaviour of two or more roles forming a service. Each building block consists of a UML collaboration describing its structure and a UML activity describing its behaviour. Java code is used to express local operations. Moreover, a building block has an external state machine (ESM) that describes the externally visible behaviour from a global perspective. This enables developers to compose (sub-)systems from building blocks without looking inside at their detailed behaviour. SPACE also features a form of fault detection: syntactic inspectors that check for typical syntactic mistakes, like forgetting to connect an outgoing pin from a building block. Once the service-oriented model is complete, it is transformed into state machines for every component, so as to be deployable. From these state machines, Java code can then be generated. The state machines already have a formal semantics in cTLA [HKB06], and work has also begun on giving the collaborative SPACE models the same (later published in [KH07a]). Hence, the SPACE models should be suitable for formal verification. Moreover, many fault-tolerance mechanisms can be thought of as collaborations between two or more roles, possibly physically distributed, which fits nicely with the SPACE decomposition of systems into services. We therefore see SPACE as a good starting point for a method aimed at building reliable, distributed, reactive systems.

Below is a list of some of the most relevant features and the state they were in before my involvement.

- Using activities and collaborations for describing services was already thought of, as published in [KH06].
- Transforming activities into local components with state machines and generating code from these was already done, as described in [KH07b] and [HKB06], respectively.
- Multi-session collaborations with accompanying multi-session activities, were already introduced, but only with respect to model transformation and code generation, not verification. This contribution is described in [KBH07].
- Formal semantics of SPACE activities were not yet finalized, but this work progressed in parallel with making the tool that would automate the generation of TLA$^+$ from the activities. This was later published in [KH07a].

\[3\]Local building blocks describing the behaviour of one role are also common.

\[4\]In [KH06], the external specifications were not yet called ESMs.

\[5\]Multi-session collaborations are a precursor to multi-instance collaborations, and the former allow multiple one-to-one collaborations of the same type to be used in other collaborations.
3.2 Research Questions Revisited

In Sect. 1.1, we have already narrowed our scope to focus on fault removal and fault tolerance, for distributed reactive systems built using the SPACE method. As an extensions to the overview of research questions given in Sect. 1.3, we now elaborate by presenting a set of sub-questions tailored to the scope of extending the SPACE method.

As already stated in Sect. 1.3, our primary concern is that a system does not sabotage itself due to some parts of the system being specified in a way that is inconsistent with other parts. Put shortly, if the environment does not exhibit any failures, neither should the system. We then asked the question:

**RQ1**: How can we enable developers to easily remove faults in the functional design of distributed reactive systems?

We detail this with a set of more specific sub-questions that are tailored to the scope of the work (i.e., extending the SPACE method). These are intentionally phrased as yes/no questions, so as to force the elicitation of ideas to try out. The definition of the sub-questions evolved as results of earlier ones became clear.

As already explained (see Sect. 1.1 and Sect. 3.1), we have chosen to use the model-driven engineering method SPACE as starting point for our work. We have also argued that formal verification methods are effective to uncover the typical design faults in distributed reactive systems. Of the formal verification methods described in [Paper 1], we note that model checking has the potential to be completely automated, once a formal specification describing the system and its properties has been created. We also note that SPACE models have a formal semantics specified in cTLA [HK00], a temporal logic, indicating that no manual interpretation of the models should be required. We therefore ask the following two questions:

**RQ1.1**: Can we automatically create, from SPACE models, formal behaviour specifications that can be directly input to a model checker?

**RQ1.2**: Can a useful subset of the functional properties to be verified be derived automatically from SPACE models?

A potential problem with model checking is that the technique inspects every possible state of a system in order to verify properties. The size of the state space grows exponentially with the number of concurrent stateful elements in its specification, called the state-space-explosion problem. In the worst case, every new building block contains several such concurrent stateful elements. However, the building blocks are equipped with interface contracts that abstract away their internal behaviour. In addition, work on compositional model checking has shown that verifying parts of a system against a contract representing the rest of the system can reduce the total number of states to check [PDH99]. We therefore ask:
RQ1.3: Does model checking allow us to verify properties for SPACE-specified systems of a useful size in a reasonable time? If not, can taking advantage of the interface contracts to perform compositional model checking solve this?

Even though a model checking algorithm executes without the need for user intervention, any resulting error traces need to be interpreted, in order to diagnose the fault and correct it. An error trace refers to the formal behaviour specification that is input to the model checker. To interpret this, competence in the input language of the model checker would be required. Hence, we ask:

RQ1.4: Can error traces from a model checker be expressed in terms of the SPACE UML models?

A positive answer to this would enable developers to diagnose and correct design faults, but this would still be a manual job. We imagine some diagnoses and some corrections to be more frequent than others, so that one may save substantial time if the most common ones could be found and applied automatically. We therefore pose two additional questions:

RQ1.5: Is it possible to automate the diagnosis of a useful subset of design faults?

RQ1.6: Is it possible to provide automatic corrections for a useful subset of diagnosed design faults?

We stated, in Sect. 1.3, that our goal, once the system is free from self-inflicted problems, is to enable it to tolerate faults in its environment. So, we asked:

RQ2: How can we enable developers to easily augment distributed reactive systems with fault-tolerance mechanisms?

We believe there is much sense in the end-to-end argument [SRC84] when it comes to fault tolerance: We cannot completely hide it at a lower layer as there is always the chance of errors that cannot be masked. In this case, the application logic needs to react to the failure, meaning that at least some of the logic for dealing with it must be present in the application itself. This implies that fault-tolerance mechanisms must also be expressed in the SPACE models.

Many fault-tolerance mechanisms are based around replicated services, meaning that the components that provide the service are replicated (i.e., there are additional instances of the same type), so that other instances can keep providing the service, even if some components fail. Hence, we ask:

---

A lower layer here refers to one of the layers of a networked system, from the lower ones responsible for transmitting bits and data packets, to the application at the top, responsible for the overall user experience.

Naturally, being able to encapsulate services using multiple components of the same type also has other uses than fault tolerance, like expressing pure functionality in a scalable manner.
3.2. Research Questions Revisited

**RQ2.1:** Can the SPACE method be used to express replicated services, and if not, what is missing?

When fault-tolerance logic and functional logic is mixed, it quickly becomes hard to see which logic supports the respective kind of requirements. This can impede understanding of the design and increase the risk of introducing faults if requirements of one kind are changed. Currently, the most common way of expressing behaviour in MDE is to use state machines (see Paper 1 Sect. 2.6). The building blocks of the SPACE approach are novel in that they can encapsulate distributed behaviour as UML activities. We imagine this capability could aid in both separating concerns and improving reusability of fault-tolerance mechanisms. We therefore ask:

**RQ2.2:** Are there any benefits to be gained from encapsulating fault-tolerance mechanisms in collaborative building blocks?

In Sect. 1.3, we argued that fault-tolerance-augmented systems were more complex and more prone to contain software faults, particularly in the fault-tolerance mechanisms themselves. We then asked:

**RQ3:** How can we enable developers to easily remove design faults in fault-tolerance-augmented systems?

To fully analyse a system using e.g. replicated services to tolerate process crashes, we must model check a model with several active instances of each type. This can potentially cause a state space explosion, as the state space can grow exponentially with every instance. We ask:

**RQ3.1:** Can we verify sufficiently large models of systems using replicated services, so as to be useful in practice?

Compositional verification has already been pointed out as a possible mitigation strategy for the state-space-explosion problem. However, to enable compositional verification of replicated services, we need to extract the relevant externally visible behaviour of the group of local components constituting the service, something that the interface contracts of SPACE do not currently support well.\(^8\) We therefore ask:

**RQ3.2:** Can we extend the concept of behavioural interface contracts to also encompass replicated services?

---

\(^8\)You could create an ESM for a version of a service that is replicated by exactly \(n\) partition instances of a given type, but would then have to make a different ESM for another version of the same block replicated on \(n+1\) instances. The result would also be syntactically very large, for a large \(n\).
Chapter 4

Summary of Papers

**Paper 1:** Model-Driven Engineering of Reliable Fault-Tolerant Systems – A State-of-the-Art Survey

In this paper, we conduct a survey of 10 recent approaches for model-driven engineering of reliable and fault-tolerant systems. We begin the paper with a thorough explanation of the relevant background concepts from the fields of software engineering, formal verification, fault tolerance and distributed systems. Further, we present 13 characteristics by which we classify the approaches surveyed, before going in detail about each approach and summarizing our findings in a set of tables for easy lookup. We conclude the paper by discussing our findings, summarizing the state of the field and offering some predictions about its future direction. This paper serves as an evaluation of our own work, by comparing it to other approaches that have been developed at roughly the same time. In addition, it is also an attempt at giving an introduction to the field to other researchers, hopefully facilitating new research based upon the current state of the art.

*My contribution:* I did the majority of the work on defining the 13 characteristics for classification. I also did all the data gathering for the survey, including classifying the approaches into the characteristics. I was the lead author and wrote approximately 95% of the text, which also roughly corresponds to my total contribution.

**Paper 2:** Engineering Support for UML Activities by Automated Model-Checking — An Example

In this paper, we show how we can automatically transform collaborative UML specifications of the systems created in Arctis, to the Temporal Logic of Actions, TLA. We also automatically create theorems for a set of safety properties that we wish to be informed of any violations of. These theorems, along with the
corresponding system specifications are then given as input to the model checker TLC, which gives a textual error trace in terms of the TLA specification if any theorem violations are found. We state that error traces can be given graphically, but at this point it is still a manual process. The state-space-explosion problem is mitigated by utilizing the ESMs of building blocks to abstract away states that are not relevant to the behaviour observable at the interface of a building block. The paper shows how message reordering faults can be tolerated by inserting a sequencer block. We also introduce a building block for the resolution of a two-party mixed initiative (a form of race condition).

My contribution: The paper is based on the work done in my project thesis under the supervision of the other authors. I implemented the tool described in the paper and took part in formalizing parts of the SPACE semantics as part of constructing the tool. I did not do much of the actual writing of the paper text, approximately 5%. My total contribution is hard to quantify, as the work was primarily done for my project thesis. In retrospect, it could be said to account for 40%.

**Paper 3** Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services

In this paper, we outline the whole SPACE method and the accompanying tool suite Arctis. Relevant to this work (section 3 of the paper) is how we have integrated the model checking into Arctis, so that any error traces are given graphically, in terms of the original specifications. This completely removes the requirement for the developers to be competent in the temporal logic domain to diagnose error traces. We further show how a theorem violation is taken as a symptom, to be used as input when attempting an automatic diagnosis. For some cases, we also provide automatic fixes that can be automatically applied to the system specification. We also give some numbers showing how much the use of ESMs reduce the state space of an example system.

My contribution: I wrote the majority of Sect. 3, describing existing features and new work done in my master’s thesis under the supervision of the other authors. This new work includes the implementation of a two-way mapping to enable error trace visualization and creating a set of automatic diagnoses and fixes, some of them implemented in the tool. This accounts for writing about 10% of the paper. Again, the work was primarily done for another publication, my master thesis. Hence, it is hard to quantify my total contribution, especially since this paper covers the whole of the SPACE method at the time. Looking back, I estimate my contribution to roughly 15%.

**Paper 4** Model-Driven Construction of Embedded Applications based on Reusable Building Blocks – An Example

In this paper, we introduce multi-instance symmetric building blocks as a way to model multiple instances of the same type collaborating with each other. The dif-
ference of these blocks, from the multi-session blocks in \cite{KBH07}, is that they allow for internal communication among several instances of the same type, rather than being a one-to-one block instantiated many times. This facilitates the encapsulation of complex fault-tolerant behaviour in a single block. \footnote{However, we did not yet have a way to accurately express this behaviour in the contracts, so we had to manually write underspecified contracts to represent the aggregate external behaviour of these blocks when performing model checking.} We further introduce a building block for a fault-tolerant leader election protocol, which in turn uses a building block implementing a failure detector (see Sect. 2.2.5). This paper demonstrates an embedded example system deployed on Sun SPOTs. These are small programmable devices that are equipped with a radio protocol (IEEE 802.15.4) for communication, but do not have the IP protocol on top.

My contribution: I designed the majority of the example system, and was solely responsible for the parts regarding the leader election and failure detector. The actual implementation of the system was done by me. I wrote about 45\% of the paper, which also roughly corresponds to my total contribution.

\textbf{Paper 5:} Towards a Model-Driven Method for Reliable Applications: From Ideal To Realistic Transmission Semantics

In this paper, we outline a two-phase engineering method to incrementally introduce fault-tolerance mechanisms in application models that are initially created assuming ideal, i.e. infallible, communication channels. \footnote{Process crashes are not considered in this paper.} We further demonstrate how the concept of collaborative building blocks allow to encapsulate fault-tolerance mechanisms, and we present a few basic blocks, for reliable notification and inquiry. Further, we motivate the use of activities, rather than using state machines directly, especially in the case of fault-tolerant systems. We also provide some discussion on putting fault tolerance in the application layer versus in a underlying middleware, and how to combine the best of both worlds.

My contribution: I developed the new reliable building blocks. I was also lead author and wrote approximately 70\% of the paper, which also roughly corresponds to my total contribution.

\textbf{Paper 6:} Contracts for Multi-instance UML Activities

In order to specify fault-tolerant systems with crash semantics, we needed a way to express collaborations of multiple instances of the same type, collaborating to provide services in spite of process crashes, and we needed a way to verify the resulting behaviour. As a first step in this direction, this paper introduces extended ESMs (EESMs) for precisely describing the interface contracts of multi-instance UML activities, albeit in the absence of faults. We set the context of this paper in the field of component contracts in general, and argue that EESMs feature most
of the benefits available from other contract types. These are benefits such as
being able to express nesting and interleaving of operations, dependencies between
required and provided interfaces, and support parametrized components, which can
be instantiated with any number of instances. The EESM concept is demonstrated
via an academic example system for a load-balancing router. TLA$^+$ specifications
are given for the example, to show the mapping between the graphical models
and the underlying semantics, and we argue for sufficient scalability by showing
how long it takes to model check the example with varying numbers of activity
instances.

*My contribution:* I designed the example system, wrote the TLA$^+$ specifications
and performed the model checking. I was the lead author and wrote approximately
90% of the paper, which also roughly corresponds to my total contribution.

**Paper 7:** Towards Automatic Generation of Formal Specifications to Validate
and Verify Reliable Distributed Systems: A Method Exemplified by an Industrial
Case Study

In this paper, we contribute a number of things. To start with, the paper describes
a case study conducted on a real system, the lockers for keeping track of work wear,
delivered by Texi AS. The paper also introduces an extension to ESMs and EESMs,
external reliability contracts (ERCs). Inspired by aspect-oriented modelling, we use
ERCs to augment (E)ESMs to specify behaviour in the case of failures. We also
detail how non-trivial application-specific liveness properties can be expressed and
verified, taking advantage of the new ERC concept. Another main point of the
paper is to demonstrate just how much the state space is reduced using composi-
tional verification, substantiating earlier claims of this. We further demonstrate
how compositional verification is even more important in the case of doing model
checking under realistic semantics, as the state space to consider is much larger.
Additionally, we update the development method presented in **Paper 5** no longer
limiting scope to communication faults, but also focusing more on the verification
in phase two, incorporating EESMs and ERCs.

*My contribution:* Together with the second author, I designed and implemented
the actual system described in the paper. I contributed the ERC concept, wrote
the TLA$^+$ specifications and did the verification (model checking) of the system. I
wrote approximately 70% of the paper, which also roughly corresponds to my total
contribution.

**Secondary Papers**

These are papers not included in the thesis, even if written during the PhD period.
The reason for not including them is that the information within is sufficiently
covered by other papers or the thesis introduction itself. Instead, their abstracts
are included in Appendix A along with their bibliographic information.
Paper 8: Model-Driven Engineering of Dependable Systems

This is an extended abstract submitted to a PhD symposium at ICST 2010. It contains a research plan and a preliminary report on results so far. It basically covers a subset of this thesis introduction, although one may notice that the scope of the research questions has been somewhat reduced since then.

My contribution: I was the sole author.

Paper 9: Modeling a Distributed Intrusion Detection System Using Collaborative Building Blocks

In this paper, we demonstrate how we can model a distributed intrusion detection system using the collaborative building blocks of SPACE. The paper’s main relevance to this thesis is that it provides yet another example of a somewhat complex system decomposed by using distributed building blocks and symmetric multi-instance activities/blocks. It also relates to fault-tolerant systems in that this system also has to tolerate changes in its environment, although they are here limited to controlled overload situations, meaning that the overload does not affect the functionality or reliability of the system. The papers also gives additional support to our claim that collaborative building blocks strongly facilitate reuse and hence a reduction of development effort. This effect is further enhanced by the fact that not all domain experts need to be involved in all phases of the development.

My contribution: I developed the non-IDS-specific parts of the example system together with the lead author. The two of us also wrote the sections describing this together (Sect. 3 and 4), so that my part of the writing accounts for approximately 10%, which also roughly corresponds to my total contribution.
Chapter 5

Discussion

In this chapter, we try to answer the various research questions presented earlier, and we give an overview of the existing tool support for the results. We also discuss limitations of the work together with ideas for further work.

5.1 Answers to Research Questions

We structure this part of the discussion around the research questions, attempting to answer each sub-question.

**RQ1**: How can we enable developers to easily remove faults in the functional design of distributed reactive systems?

**RQ1.1**: Can we automatically create, from SPACE models, formal behaviour specifications that can be directly input to a model checker?

We have successfully created a tool that does this, as described in [Paper 2 and 3](#) (and mentioned in most subsequent papers as well). For the actual algorithm used to separate out activity steps and produce TLA+, see [St07].

**RQ1.2**: Can a useful subset of the functional properties to be verified be derived automatically from SPACE models?

Via the included publications, we have shown examples of theorems being automatically generated for absence of deadlocks, bounded queues, respecting ESMs of inner blocks and respecting enclosing ESM (see mainly [Paper 2 and 3](#)). We have further demonstrated that additional application-specific properties can be generated by applying stereotypes to activity elements ([Paper 3](#)). Along with syntactic inspectors, these theorems account for detecting the vast majority of faults we discover in our SPACE models. As shown in [Paper 7](#), we sometimes need to
write manual theorems in addition, but this is mainly in cases where we need to be absolutely sure of some application-specific property.

**RQ1.3:** Does model checking allow us to verify properties for SPACE-specified systems of a useful size in a reasonable time? If not, can taking advantage of the interface contracts to perform compositional model checking solve this?

While we have no publications specifically focusing on the feasibility of non-compositional model checking of purely functional SPACE models, Paper 3 does mention an experiment where we tried to model check the example system without using the ESMs (i.e., using monolithic model checking). We found about a tenfold increase in the state space, but even this was small enough to be easily manageable. The main finding, however, is that while monolithic model checking is subject to a state space that grows exponentially with every stateful element, the state space under compositional model checking seems to increase roughly linearly with every level of decomposition. Hence, a system does not need to be that much larger before monolithic model checking would become impractical. Data to support that compositional model checking does help is given in Paper 3, 6 and 7.

**RQ1.4:** Can error traces from a model checker be expressed in terms of the SPACE UML models?

This was demonstrated in Paper 3. The mapping between activity steps and TLA transitions turned out to be bi-directional, so the main challenge was the implementation.

**RQ1.5:** Is it possible to automate the diagnosis of a useful subset of design faults? 

*and RQ1.6:* Is it possible to provide automatic corrections for a useful subset of diagnosed design faults?

We have given examples and implemented some proof-of-concept diagnoses and corrections (aka. fixes) (see Slt08 and Paper 3). However, our practical experience has shown that the benefits extracted have been limited; we find that visualizing the behaviour leading up to the error is usually sufficient for a diagnosis to quickly be set manually, at least by experienced developers. This may be different in the case where industrial-scale systems are developed and longer error traces are given. We found that to be the case for the system in Paper 7 for example, where it really did take us a long time to diagnose the error manually. We believe that more diagnoses and fixes can be automated, but we find one may want to postpone putting effort into this until practical experience shows that it is necessary. Hence, we find it too early to say whether the diagnoses and fixes that can be automated can be considered a *useful subset* or not.

**RQ2:** How can we enable developers to easily augment their systems with fault-tolerance mechanisms?

**RQ2.1:** Can the SPACE method be used to express replicated services, and if not, what is missing?
5.1. Answers to Research Questions

As demonstrated in Paper 4, 6 and 7, multi-instance activities enable us to model partition instances collaborating with other instances of the same type to provide a service. This then enables the specification of services that mask the failure of some of the participating instances. The foundation for this was laid already in KBH07, and the works in this thesis extended this from using multiple one-to-one collaborations to a general concept allowing to combine any number of partition instances of any type.

RQ2. Are there any benefits to be gained from encapsulating fault-tolerance mechanisms in collaborative building blocks?

Compared to state machines, which according to Paper 1 is the most common way of describing system behaviour in this field, activities provide better separations of concerns. We argue for this in Paper 5, where we contrast the state machines and the activities of the example system. Further, we argue that many fault-tolerance mechanisms are distributed, hence collaborative building blocks enable to encapsulate them in a way that facilitates their reuse.

RQ3: How can we enable developers to easily remove design faults in fault-tolerance-augmented systems?

RQ3. Are we verify sufficiently large models of systems using replicated services, so as to be useful in practice?

Both Paper 6 and 7 give some empirical evidence to suggest we can at least model check with multiple instances of every type, for most systems. When it comes to fault-tolerant systems, the interesting thing is often not to see with how many replicas a system can run, but rather how few. The biggest change in behaviour usually happens going from one to two instances of a given local component type. If you can also verify a system correct with three instances, chances are rather slim that a property will suddenly be violated with four. Hence, we believe that it is not necessary to model check with large numbers of instances of the same type, and we have shown that it is feasible to model check with a few instances of each type. However, this relies on being able to do compositional model checking also for multi-instance activities, as discussed next.

RQ3.2: Can we extend the concept of behavioural interface contracts to also encompass replicated services?

The EESMs introduced in Paper 6 do just this in that they allow to accurately describe the externally visible behaviour of building blocks whose inner behaviour are described by multi-instance activities. In the case where a replicated service is used to provide a fault-tolerant service, the ERCs introduced in Paper 7 allow to extend the contract with failure behaviour. Both these contract types are backed by a semantics in temporal logic allowing to verify them by compositional model

1 A partition instance is the partial behaviour of a local component that contributes to a specific collaboration (i.e., building block).
Discussion

checking. Without this feature, our experiments in Paper 7 indicate that the model checking would not scale to systems of any reasonable size, using multi-instance activities.

5.2 Status of Tool Support

The tool support implemented as part of this thesis can be summarized as follows:

- An Arctis plug-in that transforms SPACE models stored in Arctis into TLA+, while also generating temporal logic theorems representing system properties.

- An extension to this plug-in that allows to map the TLA+ transitions back into steps of the UML activities of the SPACE models, so that they can be visualized in the Arctis editor. It also parses the output from the TLC model checker, so as to be able to play back specific error traces.

- A set of diagnoses and fixes that attempt to automatically determine the cause of a property violation, and in some cases also fix it.

The main benefit of working on a common tool base, such as Arctis, is that one can take advantage of improvements contributed by others, both before starting one’s own work and along the way. The main drawback, however, is that the tool suite is evolving, so that one needs to also put some work into maintenance of already implemented tools, to ensure their continued compatibility with the rest. Such maintenance has not been done in this case: The last time I tried to generate TLA+ from a model in Arctis, it produced buggy TLA+, due to the meta-model of the Arctis models having changed. It may be relatively simple to fix, but I have not committed time to doing so. Instead, the source code has been shared with several colleagues, in case they should find the need and time to fix it. Given that there also exists an internal model checker in the Arctis tool, the translation to TLA+ is these days only necessary when one wants to manually add theorems for advanced properties, like those given in Paper 6 and Paper 7.

Due to the dynamics of the tool suite development, we did not prioritize the implementation of editors for EESMs and ERCs after Paper 6, as planned. Instead, we thought it better to wait, to minimize the risk of further changes to the tool suite. In turn, this lead to simply running out of time before the implementation could be carried out. Hence, the tool for generating TLA+ has not been updated to support EESMs and ERCs either.
5.3. Further work

The following present some parts of the approach that could, or even should, be improved, given more time to do so.

5.3.1 Further Practical Validation of Method and Tools

While the tools produced have been used to develop systems that have other stakeholders than ourselves, it has mostly been the group of researchers working to extend the tools that have also been doing the development with the tools. With the exception of some students using the tools for projects or coursework, there has been little validation of how well the approach works for regular system developers. Recently, the SPACE method and the Arctis tool suite have been commercialized by the startup company Bitreactive AS, so this is likely to promote more exposure to traditional developers working on industrial-scale systems [Bit13a]. However, this mainly applies to the general tool base that is not specifically targeting fault tolerance.

5.3.2 Deployment Model

As pointed out in Paper 7 we see some benefits from having a more fine-grained control over the fault model of a system. This could be in the form of a deployment model where one could easily set which links and components could experience various kinds of failures. For example, one could specify that only some components can crash, whereas others are assumed infallible. This would reduce the state space to model check, and also allow to trade speed of analysis for completeness with respect to multiple concurrent faults, by checking with e.g. one fault type at a time.

5.3.3 Model-Based Testing

As argued in Paper 1 Sect. 2.7 we believe that model-based testing should be included to complement formal verification techniques like model checking. To make testing as practical as formal verification currently is, we would want to automate both test-case generation and execution. Automatic test-case generation could perhaps be facilitated by extending the existing contracts with the legal domains of data variables. Ideally, we would also visualize the events resulting in a failed test directly in the models.
5.3.4 Implement a Consensus Algorithm in SPACE

While this work contains several examples of replicated services, none of them are of the kind that guarantees consistency between replicas. To achieve this, we can implement a consensus algorithm of some sort, and use this to e.g. make a replicated data store. In fact, we previously did start on such a task, using the Paxos consensus algorithm \cite{Lam98} to implement a consistent leader-election service\footnote{Consistent here means to guarantee only one leader at a time, assuming a maximum clock skew.}, but found that concepts such as EESMs and ERCs were needed before it could be completed.

5.3.5 Building Blocks for Middleware

As an alternative to building a complete implementation of e.g. a consensus algorithm at the application layer, we imagine it could be more practical to reuse existing middleware platforms, such as group communication systems (see Paper 1, Sect. 2.7.1). We could then create a few specific building blocks that allow the rest of the application to interface with the middleware, giving it access to e.g. multicast primitives or group membership. This could be combined with the deployment model described above, to specify that the middleware ensures e.g. totally ordered message delivery to all replicas, so that the verification step takes this into account. In order to take full advantage of middlewares for reliability, we would likely also require a code generator tailored to the middleware platform.

5.3.6 Model-Based Specification of Application-Specific Properties

While most properties to be verified are currently derived from the SPACE models, more application-specific ones need to be added directly in the TLA\textsuperscript{+} specification. It would clearly be beneficial if we could add a way to specify such properties, or at least a large subset of them, in some sort of graphical model, hence further reducing the need for competence in temporal logic. A starting point may be to look at the work done with Property Sequence Charts \cite{AIP06}, for example.

5.4 Summary

Increasing the dependability of a computer system increases the acquisition cost so much that many systems are built without a cost-effective level of dependability (for the total lifespan of the system). Further, application of fault-tolerance mechanisms without proper care and competence may actually result in decreasing many
5.4. Summary

dependability attributes, including reliability, which is the main focus of this work. This motivates our decision to work towards the goal of reducing the development effort and competence required to create reliable distributed reactive systems. The scope has been narrowed to focus on improving reliability by the means of fault removal (model checking) and software-implemented fault tolerance. We also limit ourselves to build on the existing model-driven engineering method SPACE.

Within our scope, three research questions have been formulated, along with a number of sub-questions:

**RQ1:** How can we enable developers to easily remove faults in the functional design of distributed reactive systems?

**RQ2:** How can we enable developers to easily augment distributed reactive systems with fault-tolerance mechanisms?

**RQ3:** How can we enable developers to easily remove design faults in fault-tolerance-augmented systems?

In response to the first question, we have found a way to automatically generate a formal behavioural description in TLA\(^+\) from the SPACE activities, and a way to automatically generate a set of theorems to be verified too. Hence, all the input needed by the TLC model checker is created automatically, requiring no effort or competence in formal methods by the system developers. In order for the same to hold when interpreting the results of the model checker, error traces are mapped back to activity steps and visualized in terms of the UML models developers are working on. Additionally, we have found and implemented a set of diagnoses and a smaller set of fixes that can be set or applied automatically.

As part of answering the second question, we have developed and published a set of building blocks showing the feasibility of using SPACE to create fault-tolerant systems. These range from simple encapsulations of “timeout and retry” mechanisms to protocols for leader election. To express the more advanced mechanisms, we extended SPACE with multi-instance activities. These allow to express complex coordination patterns among many, possibly redundant, instances of the same or different types in a single building block, and are a prerequisite to model replicated services. Finally, we have outlined a two-phase method for building reliable, fault-tolerant, reactive systems, where developers can start out simple, without considering fault tolerance, and add more complex solutions afterwards.

In order to allow compositional model checking of multi-instance activities, we have created a graphical notation (EESMs), with semantics defined in temporal logic, that can accurately express the externally visible behaviour of such blocks. Further, we have extended all interface contracts with an aspect-oriented layer that can express failure behaviour of building blocks, and hence allow compositional verification also when realistic failure semantics are assumed. Moreover, we have conducted some simple experiments on some of our published systems to collect
empirical data showing the scalability benefits of compositional model checking compared to monolithic model checking.

In addition to working towards the research questions, we have done a survey of state-of-the-art approaches in the field of model-driven engineering of reliable fault-tolerant systems. In so doing, we have found a set of characteristics by which approaches in the field can be characterized. The characteristics allow to quickly extract key information about approaches and also facilitate their comparison.

We believe the work herein provides well-argued answers to the research questions and, by extension, makes a positive contribution towards achieving the stated goal.
Bibliography


[BB01] B. Boehm and V.R. Basili. “Software defect reduction: Top 10 list”. In: Computer 34.1 (Jan. 2001), pp. 135–137. ISSN: 0018-9162. DOI: 10.1109/2.962984


[KH06] Frank Alexander Kraemer and Peter Herrmann. “Service Specification by Composition of Collaborations — An Example”. In: *Proc. of the 2006 WI-IAT Workshops (2006 IEEE/WIC/ACM Int. Conf. on Web Intelligence and Intelligent Agent Technology)*. 2nd Int. Workshop on Service Composition (Sercomp), Hong Kong. 2006, pp. 129–133.


Bibliography


Part II

Included Papers
The following includes the verbatim text of the published papers. The only changes done are that some figures are resized to adapt to the single-column format and some bibliography entries have been altered to fit the non-numerical bibliography style.
Chapter 6

Paper 2

Engineering Support for UML Activities by Automated Model-Checking — An Example

By Frank Alexander Kraemer, Vidar Slåtten and Peter Herrmann.


6.1. Introduction

Engineering Support for UML Activities by Automated Model-Checking — An Example

Abstract. In our approach for the engineering of reactive services, we specify systems as collaborations by means of UML 2.0 activities. In automated and correctness-preserving steps, the collaborative models are transformed into executable code. The semantics of the activities are defined using temporal logic. This formal fundament can be utilized to prove that the collaborations fulfill certain general well-formedness properties which can be verified by the model checker TLC. This is quite relevant since communication delays in the interactions between the participants realizing a collaboration aggravate the design of correct collaborative behavior. The well-known state space explosion problem of model checkers is mitigated by using special external state machines which define the interface behavior of sub-activities. The generation of the formal input for TLC from the activities is completely automated, so that the engineers working on the activities do not need to be experts in temporal logic and model checking. In this paper, we describe the utilization of TLC to detect and correct design errors by means of an example.

6.1 Introduction

In our engineering approach for reactive services SPACE [KBH07; KH06; KH07b; KHB06; HK07], system specifications are composed of building blocks that model functionality related to a certain task. The building blocks are collaborations covering several components. In addition to the necessary interactions, they also define the local behavior of all the participating components. We use UML 2.0 activities to describe the behavior of collaborations. Activities can be divided into several partitions, each identifying the tasks of the individual participating components. Control flows are represented explicitly and may be synchronized by a number of control nodes. Moreover, activities can be decomposed into sub-activities, so that systems may be built from already existing building blocks.

Enabling entire collaborations as the structuring units of service specifications is beneficial in various respects. First, services usually involve several participating components. Describing them by collaborations gives a holistic view of the service which can be understood without combining all the component descriptions. Second, the degree of reuse is potentially higher since a collaboration solves only a certain subtask and is therefore more likely to be useful in other applications than entire components that typically combine several tasks making them very specific (see for example [HK07]).

Figure 6.1 outlines the development process along with the tools supporting it. An engineer works on collaborative service specifications, using a library of reusable building blocks providing solutions to reoccurring problems. The building blocks can be composed together with additional “glue” logic using an editor for activities.
For the execution of the services, however, descriptions of the system components are needed. We hereby follow a specification-driven approach, in which the service specifications composed of the collaborations are automatically transformed to component-oriented service design models in the form of UML 2.0 state machines, as described in [KH07b]. This has the benefit that consistency between the different development stages is ensured, and engineers just have to maintain the service specifications. The state machines are then the input for our code generators that produce executable code for various platforms (see [KHB06]).

For such an approach and its tools to be correct, formal reasoning is needed to guarantee that properties described by the individual collaborative building blocks are preserved by the composed system. Furthermore, the properties must be maintained through the model transformation to state machines and the implementation on the various execution platforms. For this, we use the compositional Temporal Logic of Actions (cTLA, [HK00]). We formalized both the service specifications in terms of activities [KH07a] as well as the state machines [KHB06]. The coupling principle of cTLA supports the property of superposition [BK89], in which properties of a part of the system (i.e., the individual building blocks) are also valid for the composed system. This makes it possible to map the composition of activities and state machines directly to the cTLA couplings. The model transformation and code generation correspond to refinement steps. Thus, we can use cTLA refinement proofs to verify that these steps are correct (see [KH07b] [KHB06]).

This approach is already beneficial for specification quality, since the abstraction level of the models is higher which allows for a better understanding of the behavior. Coding errors cannot be introduced due to the automatic translation. Nevertheless, the created models need to be correct as well. While some properties may be ensured by a purely syntactic analysis, others require us to consider entire behaviors, for example, that interface events of building blocks have to occur in a certain order. This is usually hard to guarantee manually as behavior involving several components can get quite complex due to the unavoidable delays of the communication medium connecting them. To assure correctness of such behaviors, model checking (i.e., the examination of all reachable states a behavioral description implies) can be used. Model checking, however, needs a certain amount of expertise in formal reasoning, which we do not want to claim from the engineers using our approach. A possibility to overcome this situation is, as Rushby suggests in “Dis-
appearing Formal Methods” [Rus00], to wrap formal techniques within tools so
that they are not seen as difficult anymore, and to increase their user-friendliness.
The idea behind this is that a user does not necessarily need to understand the de-
tails of a formal technique and model-checking, if an automated checking tool gives
understandable feedback addressing the problem in the language of the engineer’s
domain.

To follow such an approach, we developed in [Slh07] an automatic transforma-
tion tool from UML 2.0 activities to TLA+, the language of the Temporal Logic
of Actions (TLA, [Lam94]). For this language, the model-checker TLC [YML99]
is available which can check a specification for various temporal properties that
are stated in form of theorems. For each activity, we generate a set of theorems
automatically which claim certain properties to be kept by activities in general.
Examples for these properties are the correct usage of building blocks within the
activity as well as that the activity itself satisfies a certain externally visible be-
havior. When TLC finds that a theorem is violated, it produces an error trace
displaying the state sequence that leads to the violation. This trace can be given
in terms of easily comprehensible token markings within an activity as well. So, an
engineer using our tools does not have to write or understand the temporal logic
formulas.

The presented approach for model checking makes use of the compositional nature
of our service specifications. As described in [KH07a], a system composed of col-
laborations guarantees the properties of the single collaborations to be maintained.
This follows directly from the semantics based on cTLA [HK00] and the principle
of superposition. The activities describing the complete behavior of collaborations
may be specified in a more abstract form by means of special state machines that
refer to externally visible events dedicated for composition. When model checking
a composite specification, only the abstract specification has to be taken into ac-
count, which reduces the state space. Thus, we check each collaboration separately
and do not consider the entire hierarchy which effectively mitigates the likelihood
of state space explosions.

After discussing some related work done on formal checking of UML models, we give
an introduction to temporal logic as well as the model checker TLC in Sect. 3. We
proceed by introducing an example specification based on activities, and explain
the semantics of activities in temporal logic in Sect. 4. Thereafter, we use our
tools in Sect. 5 to develop an example, starting with a naive solution that gets
corrected based on the feedback of the model checking. We close in Sect. 6 with
some concluding remarks.

6.2 Related Work

Formal checks on UML models are done as part of OMEGA [Hoo02], FUJABA [Bur+04]
and HUGO [Bal+04]. However, these approaches mainly concentrate on state ma-
machines or sequence diagrams, but not on activities as in our case. In [GM05], UML activities are translated into PROMELA, the input language for the SPIN model checker [Hol03]. In [Sto05], a mapping from UML 2.0 activities to Colored Petri Nets is described enabling the usage of Petri Net tools for analysis. In [DS03], UML activities are transformed into the $\pi$-calculus where safety and liveness properties can be expressed using the modal mu-calculus and checked using the MWB tool [VM94]. Eshuis [Esh06] uses NuSMV, a symbolic model verifier to check the consistency of activity diagrams. The difference of these approaches to ours lies mainly on the domain that activities are used for and the chosen semantics. While they focus on activities more from a perspective of business processes assuming a central clock or synchronous communication, we need for our activities reactive semantics [KH07a] reflecting the transmission of asynchronous messages between distributed components. This semantics enables us to generate the executable state machines defined in [KHB06].

6.3 The Temporal Logic of Actions

Leslie Lamport’s Temporal Logic of Actions (TLA, [Lam02]) is a linear-time temporal logic in which semantics is expressed by infinite state sequences. The corresponding syntax is TLA$^+$ that enables describing system behavior by special state transition systems and additional fairness properties. Fig. 6.2 is an example of a TLA$^+$ specification. After a frame containing the module name (i.e., HotelWakeUpSystem), it uses the expression EXTENDS Naturals describing the import of a module including definitions, operators and axioms to model the natural numbers. The states of the state transition system are modeled by variables (here $i$, $t$, $h$ and $a$) which are, in general, non-typed. The predicate Init specifies the set of values the variables shall have in the initial state. The transitions are described by actions each specifying a pair of a current state and its successor state. Here, the current state is referred to by variable identifiers in a simple form while the next state is modeled by primed variable identifiers. An example is the action initial which may be executed if the variable $i$ has the value 1 and $h$ has the value “off”. After its execution, $i$ will carry the value 0 which is described by $i' = 0$. In addition, $h$ will have the value “started” in the following state while the two other variables $a$ and $t$ do not change their values during the execution of the action. The set of system transitions is modeled as the disjunction of the system actions which is expressed by the definition Next, the so-called next-state relation. The overall system description is modeled by the canonical formula Spec. The first conjunct of this temporal formula defines that the predicate Init holds in the first state of every state sequence modeled by Spec. The second conjunct uses the temporal operator $\square$ (“always”) specifying that the rest of the conjunct is valid in all states of all state sequences describing the behavior of the system. The TLA expression $[\text{Next}]_{\langle i,t,h,a \rangle}$ determines that a state transition has to be either a stuttering step in which all variables listed in the subscript maintain their values or satisfies the condition Next. Thus, every state sequence begins with a state fulfilling Init and
module HotelWakeupSystem
extends Naturals
variables i, t, h, a

Init \triangleq
\land i = 1 \land t = 0
\land h = “off” \land a = “off”

initial \triangleq
\land i = 1 \land i' = 0
\land h = “off” \land h' = “started”
\land UNCHANGED \langle i, t \rangle

startAlert \triangleq
\land h = “started” \land h' = “alerting”
\land a = “off” \land a' = “active”
\land UNCHANGED \langle i, t \rangle

stopAlert \triangleq
\land h = “alerting” \land h' = “stopped”
\land a = “active” \land a' = “off”
\land UNCHANGED \langle i, t \rangle

aborted \triangleq
\land h = “stopped” \land h' = “off”
\land t = 0 \land t' = 1
\land UNCHANGED \langle i, a \rangle

confirmed \triangleq
\land h = “stopped” \land h' = “off”
\land t = 0 \land t' = 1
\land UNCHANGED \langle i, a \rangle

timeout \triangleq
\land t = 1 \land t' = 0
\land h = “off” \land h' = “started”
\land UNCHANGED \langle i, a \rangle

Next \triangleq
\lor initial \land startAlert \lor stopAlert
\lor aborted \lor confirmed \lor timeout

Spec \triangleq Init \land □ [Next]_{\langle i, t, h, a \rangle}

\begin{align*}
t0 & \triangleq □((i = 1) \Rightarrow (h = “off”)) \\
t1 & \triangleq □((h = “stopped”) \Rightarrow (t = 0)) \\
t2 & \triangleq □((h = “started”) \Rightarrow (a = “off”)) \\
t3 & \triangleq □((h = “alerting”) \Rightarrow (a = “active”)) \\
t4 & \triangleq □((t = 1) \Rightarrow (h = “off”))
\end{align*}

Figure 6.2: TLA Module

corresponds only to state transitions which either meet one of the system actions or are stuttering steps. Further conjuncts may be used to describe liveness properties by fairness assumptions on actions which, however, is not discussed in this paper.

The second paragraph of the specification contains a list of properties \( t_0 \) to \( t_4 \) which shall be kept by the system. As they all start with the always operator, they state invariant behavior (e.g., if variable \( i \) has value 1, \( h \) must be “off”). To verify an invariant, one has to prove that it holds in the initial condition \( Init \) and that it is preserved by every system action.

The compositional Temporal Logic of Actions (cTLA \([HK00]\)) mentioned in the introduction is a derivative of TLA. It resolves a shortcoming of TLA which is limited to compositions based on joined variables \([AL95]\). In contrast, cTLA combines modules by defining joined system actions as simultaneously executed module actions which is a prerequisite for constraint-oriented models \([Vis+91]\). There, one specifies not single physical components but properties describing partial system behavior which spans several components. As the UML 2.0 collaboration and activity-based models used in our approach demand this particular specification style, we used cTLA instead of TLA to define their semantics \([KH07a]\). cTLA uses a process-like specification style which encompasses both simple and compositional
process descriptions. As the compositional process models can be transferred to simple ones (see [HK00]) and the simple processes are basically defined by the same canonical formulas as TLA\(^+\), it was not a major problem to transform the UML activities to TLA\(^+\) modules like the one depicted in Fig. 6.2. This is done by the tool introduced in [Slt07] such that we can use the model checker TLC [YML99] to automatically prove that the activities fulfill certain properties since TLC uses TLA\(^+\) specifications as input. TLC performs an exhaustive exploration of all reachable system states and verifies that invariant properties are maintained by every checked state\(^1\). In the case of a failure, a path of states leading to the one not fulfilling a property is shown which facilitates the search for the error and can be visualized in the UML activities.

### 6.4 UML 2.0 Activities in the SPACE Approach

In order to study an intricate problem in isolation, we consider a system to carry out wake-up alarms for guests of a hotel. The system is partly automated, as the requests for wake-up alarms are noted manually by the receptionist in a book. The guests prefer to be woken by an alarm instead of a direct phone call, to avoid contact with the personnel at an early morning hour. To convince the receptionist that they really are awake, they confirm the alarm by pressing a button. The reception has a control panel with two buttons and a display for each of the guest rooms, illustrated in Fig. 6.3. At wake-up time, the receptionist pushes the alert button which sounds the alarm in the guest room. If the guest confirms, the display shows Confirmed for some seconds so that the receptionist knows that the guest is actually awake. If the guest does not confirm, the receptionist can abort the alert after some time, upon which he or she may visit the room and rouse the guest with more drastic measures.

\(^1\)For liveness proofs not introduced here, TLC checks sequences of states.
6.4.1 Informal Explanation of Activities

The behavior of the example system is described by the UML 2.0 activity shown in Fig. 6.4. It is divided into two activity partitions, one denoting the hotel reception and one for a guest room. On the reception side, the activity contains three operations to control the display by printing the messages Ready, Aborted and Confirmed. On the side of the guest room, an alarm device is represented by a so-called call behavior action. This is a node that may refer to other activities (in the following referred to as sub-activities) and be used for decomposition. In the system here, we do not know about the internals of the alarm, just that it can be started by a token entering via start and stopped by a token via stop. Similarly, h refers to another activity realizing the protocol between the reception and the hotel room. In contrast to the building block for the alarm, h spans over both activity partitions and as such describes a collaboration between the reception and the guest room.

The system activity starts on the side of the reception at the initial node. A token is emitted upon system startup and moved to a fork node, where it is duplicated. One of the tokens continues to operation display Ready, causing the display to show that the system is ready. Afterwards, it ends at a flow final node. The other token leaves the fork and moves into the call behavior action h via input pin start. This activates the Hotel Wakeup sub-activity. On this level, we just need to know about its externally visible behavior, described by the state machine Hotel Wakeup in Fig. 6.5. The stereotype «esm» applied to it marks that the diagram denotes an external state machine (ESM, [Kra08]) for the sub-activity. Its transitions refer to the input and output pins of the corresponding sub-activity, describing in which sequence tokens may be passed. We see that after start, event start alarm will eventually happen, followed by stop alarm. Thereafter, the sub-activity terminates as either aborted or confirmed, depending on the behavior of the guest. On the side of the guest room, the flow leaving start alarm and stop alarm of h is connected to start resp. stop of the call behavior action a modeling the alarm. On the reception side, the display informs the receptionist about the outcome via two distinct display messages once sub-activity h terminates. As soon as the display messages Confirmed or Aborted appear, a timer is started waiting for a certain time, so that the message can be read. Upon a timeout, the display

---

2To keep the discussion simple, we only consider one room. Using the mechanisms presented in [KBH07], this design can easily be expanded to multiple rooms.

3The decision to put the alarm and the display outside of the Hotel Wakeup h was here mainly to ease the presentation of the contents of h as shown in Sect. 5.
A first (naive) solution for the internals of the call behavior action $h$: Hotel Wakeup is shown in Fig. 6.6. Note that the dashed lines are not a concept of UML activities but are here used to illustrate the preliminary state of the model which we will replace later, based on the findings of the model checking. The flows in solid lines remain stable throughout all solutions. The activity is composed from three buttons alert, confirm and abort from our library of reusable building blocks [Kra09]. Their external behavior is described by $\ll esm \gg$ Button in Fig. 6.5. There, a button is activated via start. In this state, it may be pushed by the user, which causes its termination via pushed. It may also be stopped by a token through stop, whereupon any pushes by the user are ignored.

When the Hotel Wakeup collaboration is started, the alert button is activated immediately. Once it is pressed, a token is emitted via pushed, activating the abort button. At the same time, the flow continues towards the partition for the guest room. As the partitions will be implemented by different, physically remote components, we assume a buffered communication between activity partitions. Therefore, a token waits for an arbitrary time in a virtual queue place where a flow crosses partition borders. This corresponds to the transmission through a physical medium. When the flow from the alert button is received by the guest room partition, the confirm button is activated, and a token is branched off towards the output node start alarm to notify the alarm device. If the confirm button is pushed, the alarm is stopped via output node stop alarm and a confirmation is routed back to the reception partition where the collaboration terminates via output pin confirmed. If the receptionist presses the abort button, the guest room is notified to switch off the alarm and the confirmation button, and the collaboration is terminated via aborted.

### 6.4.2 Semantics of UML 2.0 Activities in Temporal Logic

Formally, UML activities are based on Petri Nets and describe as such a state transition system. In [KH07a] we defined the semantics of activities in terms of
6.4. UML 2.0 Activities in the SPACE Approach

cTLA, which can be easily mapped to TLA⁷⁺, the input language for the model checker TLC, as discussed in Sect. 3. The transformer from UML 2.0 activities to TLA⁷⁺ [Sli07] uses UML activity models stored in the UML2 repository of Eclipse as input. Roughly speaking, the tool maps each token movement of an activity to an action in a TLA⁷⁺ formula in which stateful nodes such as timers, sub-activities, joins and accept signal actions are represented by their own variables. The buffering of flows that cross activity partitions is formalized by queue variables which are bags of tokens. Whenever a token leaves a source partition, it is added to the corresponding queue place. In a second action, it is removed from the queue place and continues the flow in the target partition.

As an example, the specification in Fig. 6.2 displays the TLA⁷⁺ code generated for the system activity depicted in Fig. 6.4. It consists of six actions⁴, each modeling a token movement. The module declares a variable for each stateful node of the activity, that is, the initial node by variable \( i \), the timer by \( t \) as well as the sub-activities for the wake-up \( h \) and the alert \( a \). For both the timer and the initial node, we use simply an integer to store the number of tokens that are resting in them. Initially, there is one token in the initial node (which means the activity is ready to start) and no token in the timer (i.e., the timer is idle). This is expressed with the initial predicate \( Init \) by \( i = 1 \land t = 0 \). The variables for the sub-activities store the current states of the ESMs that represent their externally visible behavior. Initially, both ESMs are in their initial state, so that value “off” is assigned to \( h \) and \( a \) by \( Init \). The six actions model the token movements within the activity. Action \( initial \) specifies the start of the activity. The token resting in the initial node is removed from it \( (i' = 0) \) and enters \( h \) via input pin \( start \). The ESM of \( h \) (according to its definition in Fig. 6.5) makes a transition to state \( started \). When \( h \) is in state \( "started" \), action \( startAlert \) is enabled. It models the emission of a token from \( h \) via \( startAlert \) activating the alarm \( (a' = "active") \). Eventually, the alarm will be deactivated again by the execution of \( stopAlert \). After that, the two actions \( aborted \) and \( confirmed \) are enabled, modeling the termination of sub-activity \( h \) (by \( h' = "off" \)). Due to the merge node, both of these actions start timer \( t \) (by \( t' = 1 \)), enabling action \( timeout \), which restarts sub-activity \( h \).

6.4.3 Theorems for Well-Formed Activities

An important property of our activity specifications is that the events of the sub-activities are invoked in the order specified by their ESMs. This means for example that, whenever a token attempts to enter \( start \) of sub-activity \( h \), then \( h \) must not yet be activated, i.e., \( h = "off" \). A token can be released from the initial node whenever it has a token, i.e., \( i = 1 \). So, we want to be sure that whenever there is a token in the initial node, the sub-activity is not yet active. Formally, this is

⁴We adjusted the automatically chosen variable and action names for readability.
⁵The token is further forked into operation \( display Ready \), which we can ignore here since no stateful node is reached.
an implication $i = 1 \Rightarrow h = \text{“off”}$. As this property must always hold, our tool writes the theorem as an invariant $t_0 \triangleq \Box((i = 1) \Rightarrow (h = \text{“off”}))$. The further theorems describe the other cases in which the ESM of a sub-activity must not be violated by its environment. For example, $t_4$ ensures that whenever the timer is active ($t = 1$), sub-activity $h$ may be started again ($h = \text{“off”}$). The violation of ESMs is only one major source for errors. Thus, the current transformation tool also writes theorems to check the boundedness of queues as well as assertions on the execution of operations that can be added with additional stereotypes [SlT07]. This is, however, not discussed here.

### 6.5 Developing and Model Checking the Example

The use of model checking to correct activity-based service specifications is outlined by discussing the improvements of the hotel wakeup system. We start by applying our transformation tool and create the TLA$^+$ specification of the system activity listed in Fig. 6.4. The outcome is the TLA module introduced in Fig. 6.2 which is checked by TLC. The model checker notifies that 5 distinct states were generated and that no errors were found. Given the theorems that are included in our automatically generated formal specification, this means that the contracts of the used building blocks $h$ and $a$ are obeyed. Thus, we can proceed by checking the design of the Hotel Wakeup activity.

#### 6.5.1 Solution 1: A Naive Start

As an initial solution, we consider the activity introduced in Fig. 6.6. On a first glance, it looks quite straightforward. When the alarm button is pushed, the guest room is notified to activate the confirmation button. A push on their button by either the receptionist or the guest stops the alarm and the respective other button. However, when we model check this activity, TLC says that temporal properties are violated and prints a trace of states that describes the behavior up to the moment when the violation took place. This trace may be projected onto the activity, as illustrated in Fig. 6.7. Hereby, the transfer queues are shown as token places where the flows cross partitions, and the activity and its sub-activities are amended with boxes showing the current state of their ESM.

**State 1.** The activity is not yet active and its ESM is in state off. The queues $a$, $b$ and $c$ are empty, and all sub-activities are in state off as well.

**State 2.** A token was moved via the input node of the activity and activated the alarm button, which is now in state active.

**State 3.** After the alarm button was pressed, a token was forwarded into queue $a$ and the abort button is now active. In this state, TLC reports that a theorem is violated. This theorem states that whenever the abort button is active (and may
therefore emit a token at any time), the ESM of Hotel Wakeup is in state stopped, as an outgoing token from abort would pass through parameter node aborted (see Fig. 6.5). So, in the current state, the active abort button could terminate the entire activity through flow $x_1$ and contradict the ESM. In practice this means that the system using Hotel Wakeup could assume the alarm to be aborted after the abort button was pressed, although the alarm was never started. To check for further errors before a redesign, the tool allows us to ignore this error for a moment and let the abort button be pushed.

State 4. By pushing the abort button, a token was emitted via aborted and another one is placed in queue $b$. In this state, the guest room may decide to consume the token from queue $b$, which would then be moved via $x_2$ into the confirm button that is in state off, which is is against the ESM of the button (see Fig. 6.5). Obviously, the activity in Fig. 6.6 does not regard that due to the transfer medium, an abort flow may overtake the alarm flow.

### 6.5.2 Solution 2: Improved Version with a Sequencer

The problem found in state 4 of solution 1, where the confirm button could be stopped before it was even started, can be solved by adding a building block of type **Sequencer** from our library [Kra09] to the new activity in Fig. 6.8. It controls two flows arriving in any order at $i_1$ and $i_2$ such that their respective outputs may only happen in the order $o_1$ followed by $o_2$. The problem found in state 3 of the previous solution, according to which the ESM of Hotel Wakeup was violated, can be solved by an additional flow $f$ that returns from the hotel room after the alarm was started. A new run of TLC on the activity in Fig. 6.8 reveals, however, that there are still flaws in the system. Figure 6.9 shows the new error trace. The two first states are omitted as they correspond to the ones of Fig. 6.7.

State 3. The alert button has been pressed and a token is waiting to cross from
Figure 6.8: Solution 2

Figure 6.9: Error trace of solution 2

partition reception to partition room in queue a. The abort button has also received a token and is in state active.

**State 4.** The token waiting in queue a has passed through the sequencer and activated the confirm button. The token was also forked so that a copy left the activity via start alarm causing the ESM of Hotel Wakeup (Fig. 6.5) to change from started to alerting. Both buttons are now waiting to be pushed.

**State 5.** The confirm button has been pushed sending a token via stop alarm changing the state of the ESM to stopped. The token was also forked into the queue c where it is waiting to enter the reception partition. The confirm button has returned to state off.

**State 6.** The receptionist pushed the abort button, which switched to off and emitted a token into queue b, so that there is now one token in each of the queues b and c. This harms, however, two theorems that protect the contracts of the buttons. The confirm and stop button are both in state off, but tokens are placed in the queues that flow into the stop pin of the buttons via flows x3 and x4, which would violate their ESMs.
Figure 6.10: Correct solution with a building block to handle mixed initiatives

6.5.3 Solution 3: A Building Block for Mixed Initiatives

State 6 of the trace in Fig. 6.9 reveals an intrinsic peculiarity of the system: Due to the communication delay between the reception and the hotel room, both, an abort and a confirmation, can be in progress simultaneously. This is since during the alerting phase, both the receptionist and the hotel guest may take their initiative at nearly the same time. Although not always recognized, this situation occurs frequently in reactive systems, and has several names such as conflicting BH93 or mixed initiative Flo03 as well as non-local choice BL97. As the problem is quite general, our library of building blocks contains a collaboration to handle mixed initiatives. This collaboration has two participants, a primary and a secondary one. These names reflect which of the sides gets priority over the other if both sides take initiative contemporaneously. Two variants of the building block exist, one where the primary participant starts the collaboration, and one where the secondary one starts. In our system, we use the latter one and assign the primary role to the guest room, so that a confirmation from a hotel guest has priority over the abort from the reception. Fig. 6.10 shows the building block already embedded into the new solution while the ESM showing the detailed interleaving of its events is given in Fig. 6.11. For the sake of brevity, we look here just at the externals of the block, as an engineer would do when reusing it. The internals are similar to the building block Tour Request introduced in KBH07.

After the start of the collaboration via start on the secondary side, started notifies the primary side that the state is reached in which it may trigger an initiative. We couple this action with the start of the alarm. Input pin prim. initiative, denoting an initiative taken by the primary participant, is coupled with the pushing of the confirmation button. As the primary side has priority, we know that the confirmation will succeed, and can therefore stop the alarm right away. If the secondary side takes initiative (input pin sec. initiative), the primary side gets notified via sec. action, which is used to stop both the alarm as well as the confirmation button.

On the secondary side we have to take into account that an initiative from the
Figure 6.11: ESM for Mixed Initiative Secondary Starter

Abort button can be overruled by the confirmation of the guest room. Besides the nodes to start the collaboration and to take initiative, the secondary side has therefore three terminating output pins, from which only one will eventually release a token.

- Pin *primary action* releases a token if the primary side took initiative, and the secondary remained passive, i.e., only the guest confirmed. This leads to stop the abort button and to terminate via *confirmed*.

- Pin *sec. overruled* models that both initiatives have been taken, from which only the primary prevails. It is sensible to distinguish this case from the first one, as the reception in this case does not have to switch off the abort button, which already terminated because of its initiative.

- Pin *sec. accepted* emits a token if the secondary initiative was the only one, and the primary side did not start an initiative on its own, i.e., the alarm was aborted without the confirmation button being pressed.

When we translate this activity into TLA+ and start TLC, we get the message that all properties are fulfilled now. Thus, the activity handles all the incorporated building blocks as prescribed by their respective ESMs. Moreover, it respects its own ESM and can be correctly used within the system described in Fig. 6.4. After checking the activity realizing the call behavior action \(a\) modeling alarms we know that the overall service specification is well-formed and can use it as input for the transformation steps producing executable code.

### 6.6 Concluding Remarks

We presented our service development approach SPACE that uses collaborations as building blocks. Their behavior is described by UML 2.0 activities which we can transform automatically into temporal formulas and a number of theorems expressing relevant properties to be fulfilled by an activity. The correctness of these theorems is model checked by TLC and its error messages lead to stepwise improvements of the models. The approach works both bottom/up and top/down.
Sub-services may be arranged and their composition to larger services may be checked. Vice-versa, as done for the hotel wakeup, we may first assume a certain external behavior and then realize the internals of the service. Of course, many real systems are more extensive than the example used for the discussion here. The larger scale of these system results, however, mostly in a higher number of collaborations to be executed than in more complicated interactions. Thus, we will have a higher number of decomposition levels (see, for instance [HK07]), while the complexity of the models describing individual collaborations will remain of manageable size.

Once a collaboration between components in form of activities is model checked, it can be used in other systems without further proof efforts. This is feasible as the building blocks may be abstracted by their ESMs describing their external behavior. Thus, if we check an activity containing a sub-activity, we only have to consider the ESM of the sub-activity which hides the internal states, such that the state space of the model checked activity is reduced. In consequence, model checking is never done on the entire system with all its details, but it is enough to successively check activities on their decomposition level separately. In this way, services and their compositions from sub-services may be verified in a compositional way which effectively rules out state explosions.

With the automatic formulation of the temporal formulas and theorems we created the base for user-friendly model checking of the service specifications based on UML activities. In future versions, we may offer more advanced feedback to the user that may explain error situations further and suggest typical improvements. This work will be performed as part of the research and development project Infrastructure for Integrated Services ISIS, funded by the Research Council of Norway, where we develop methods, tools and building blocks for services in the domain of home automation.

References


Chapter 7

Paper 3

Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services

By Frank Alexander Kraemer, Vidar Slåtten and Peter Herrmann.


The original publication is available at www.sciencedirect.com via http://dx.doi.org/10.1016/j.jss.2009.06.057

Note: There is a small change in syntax, compared to the previous paper, in that we here use slashes to separate whether the ESM transition is triggered from within the building block or is a result of external events. In this paper, we also prefix pins with their direction with respect to the building block.
7.1. Introduction

Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services

Abstract. We present the integrated set of tools Arctis for the rapid development of reactive services. In our method, services are composed of collaborative building blocks that encapsulate behavioral patterns expressed as UML 2.0 collaborations and activities. Due to our underlying semantics in temporal logic, building blocks as well as their compositions can be transformed into formulas and model checked incrementally in order to guarantee that important system properties are kept. The process of model checking is fully automated. Error traces are presented to the users as easily understandable animations, so that no expertise in temporal logic is needed. In addition, the results of model checking are analyzed, so that in some cases automated diagnoses and fixes can be provided as well. The formal semantics also enables the correct, automatic synthesis of the activities to state machines which form the input of our code generators. Thus, the collaborative models can be fully automatically transformed into executable Java code. We present the development of a mobile treasure hunt system to exemplify the method and the tools.

7.1 Introduction

Reactive systems consist of numerous devices like controllers, sensors and computation nodes which must be connected to provide services together that each single unit could not render separately. Unfortunately, the coordination of units often turns out to be more difficult than expected. One reason for that is the reactive nature of most systems dealing with several actuators or users; often, these systems follow a symmetric peer-to-peer structure in which several units may take initiative simultaneously. This makes the modeling of system synchronizations difficult and demands suitable modeling techniques.

Another inherent reason is the so-called cross-cutting nature of services. Obviously, to execute a service, we need a description of its physically deployable components. Their behavior can be expressed by means of state machines, as for example offered by SDL [ITU07] or UML [Obj07]. A service, however, is typically collaborative and spans across several components, and one component participates in several services. This collaborative dimension is orthogonal to that of components [Mik99]. If we only use component descriptions, services are specified only indirectly by the combined behavior of its participating components. In contrast, a more explicit description in the form of collaborations (see, for example [San+05]), not only has the benefit that service behavior can be understood and analyzed in isolation, but also opens new possibilities for the reuse of services as sub-functions provided by several components: Both, local functionality and solutions to problems that require coordination of several components, can be used directly in various applications.
Based on the idea to enable the reuse of collaborative, reactive behavior in the form of building blocks, we developed the engineering method SPACE [Kra08, KH06, KH07a, KH07b], depicted in Fig. 7.1. To build a system, an engineer considers a library of reusable building blocks. In contrast to more traditional components, these building blocks may cover collaborative behavior among several components. They are expressed as a combination of UML 2.0 collaborations, activities and so-called external state machines (ESMs) to document their externally visible behavior. The building blocks are composed to more comprehensive ones, until the system specification is complete. After an analysis and potential corrections, the produced system specification is transformed automatically into state machines which can be implemented via code generation. The approach comprises the following key features that speed up development:

- The design of a service is facilitated by applying reusable building blocks that are general or domain specific collaborations which can be integrated into several system descriptions. Due to the abstract description via external interfaces expressed by the ESMs, the internals of the building blocks do not have to be considered when they are applied.

- Engineers only work on collaborative models expressed by UML activities. The component-oriented models expressed by state machines that are needed for code generation are derived fully automatically; a difficult and time-consuming manual synthesis of state machines is not necessary.

- Due to the mathematical background in temporal logic, the compositions and transformations are sound. For such a proof, see [Kra08 App. B]. Beyond that, model checking is possible also for larger systems. The compositional properties of the method facilitates the analysis of the building blocks in
separation which reduces the state space during model checking significantly. Currently, the analysis focuses on general safety properties that should be fulfilled by any application, for instance that all building blocks an application is composed from are used in a correct manner. Since theorems for these behaviors can be derived automatically, the process of model checking is completely automated, and engineers do not need to deal with any formal technique directly.

The theoretical foundations of the method are detailed in [Kha08, KH07a, KH07b]. In the following, we focus on the corresponding tool support, implemented by the Arctis plug-ins [Arc09], as depicted in Fig. 7.2. Building blocks are composed by engineers using the Arctis editor. The result can either be composite building blocks or entire systems, which are also special forms of building blocks. If desired, building blocks may be archived in the library for later reuse. To analyze a building block, its UML activity is transformed into a temporal logic formula and transferred to the TLC model checker [YML99]. It verifies the specification against theorems that we will explain later. If a theorem is violated, the analyzer tries to identify possible reasons and presents an error trace as animation in the activity to the engineer. Once a system specification is consistent and sound, it may be implemented automatically using a model transformation and code generation.

We will proceed as follows: In the next section, we present how our example of a mobile treasure hunt is composed from building blocks using the Arctis editor. In Sect. 7.3, we present how Arctis supports the analysis of specifications by automating model checking and the provision of corrections in some cases. The transformation from activities to state machines is explained in Sect. 7.4, and the code generation process is summarized in Sect. 7.5. We close with an overview of related approaches and concluding remarks.

### 7.2 Composing Services from Building Blocks

As an example we develop a mobile treasure hunt, first described in [SB08]. In this game, a player receives a riddle via SMS. The answer of the riddle is associated
with a certain location in the town the game takes place. To answer, the player does not reply via an SMS but tries to reach the location. Via GSM/GPS/WLAN positioning of the mobile phone, the player’s position is known to the system; once at the correct goal, the next riddle is sent out until the final place is reached. To make the game more difficult, players must reach the target location within a limited time. For the discussion, we consider the realization for one player at a time. Using the mechanisms described in [KBH07], this specification can be expanded to handle multiple users as well.

The system is specified by a UML 2.0 collaboration as shown in the screenshot in Fig. 7.3. On this level, the collaboration roles (depicted by rectangles) represent the components of the system. The location server is responsible for the positioning of mobile subscribers. The sms gateway provides SMS-based communication from the users into the system and vice versa. We assume, that these components are realized and managed by an external operator; for our specification, they are therefore part of the environment, marked with a corresponding stereotype. In contrast, the three other components are constituents of the system we are going to implement. The game server is responsible for coordinating the game, assisted by the proximity and riddle servers. The collaboration uses (depicted as ellipses) decompose the overall functionality of the treasure hunt system into sub-services. Between the game server and the sms gateway, collaboration uses s1: Single SMS Notification and s2: Send SMS realize the necessary interactions with the player. Collaboration use p: Proximity Alert refers to a three-way collaboration between the location server, the game server and an additional proximity server. Within this collaboration, the proximity server constantly monitors the position of the user.

Figure 7.3: Eclipse workbench with the Arctis library browser and editor
and alerts the game manager once the user is at a specified target. A dedicated collaboration to query riddles from a data base is \textit{r: Riddle Generation}.

### 7.2.1 Elementary Building Blocks

The services offered by the network operator are encapsulated within dedicated, collaborative building blocks. In addition to the interface behavior towards the operator’s servers, such building blocks may also contain local behavior that simplifies the task to implement and integrate them with the rest of the system.

As UML collaborations and collaboration uses focus only on structural issues like role binding, we use a combination of UML activities and ESMs (external state machines) for the description of behavior. Figure 7.4 shows the external representation for the building blocks encapsulating behavior towards the operator. On the right side, they are shown in their instantiated form as call behavior actions. These are

---

1. The decision to realize the proximity server as a separate component can be motivated by different reasons, for example a load analysis as explained in [BH93].

2. The building block for the location tracking is used within collaboration \textit{Proximity Alert}, as we will see below.
constructs of UML activities and can be composed within an enclosing activity, as we will see in Sect. 7.2.2. The pins at their sides are used to control their behavior. As activities can be understood by token flow semantics [Obj07], building blocks (instantiated as call behavior actions) are controlled by tokens passing their pins. The call behavior action \textit{s1: Single SMS Notification} has only two pins: Input pin \textit{subscribe} activates the block, awaiting an incoming SMS. This is issued by a token passing through the terminating output pin \textit{sms}, which in turn deactivates the building block. As parameter, \textit{subscribe} carries the number agreed upon with the operator that subscribers use to send in messages. Pin \textit{sms} provides objects of type \textit{Message} for each incoming SMS.

To document the valid sequences in which these pins may be invoked, we use the ESMs which are expressed by stereotyped UML state machines shown to the left of each building block. The labels of the transitions refer to the pins that a token passes. A slash distinguishes cause and effect, seen from the context instantiating the building block. The prefixes \texttt{in:} and \texttt{out:} are used to refer to input or output pins, respectively.

Following the description from above, the externally visible behavior of \textit{s1: Single SMS Notification} is triggered in its starting transition from the outside via \texttt{in:subscribe/} and eventually terminates via \texttt{/out:sms} which is triggered from the inside of the building block, i.e, it is spontaneous. Similarly, the building block \textit{s2: Send SMS} is started via pin \texttt{init}. From then on, however, the client side may continuously send text messages via \texttt{sms}. As this is a so-called \textit{streaming} node presented in black, tokens may pass while the block is active.

The block for the location tracking is a bit more complex. After a subscription that tells which mobile user (identified by a mobile subscriber ID, MSID) should be tracked, the client continuously receives updates via streaming pin \texttt{update} while the subscriber moves. This is expressed by the spontaneous self-transition \texttt{/out:update} which has state \texttt{active} as source and target. Once the client is not interested in location data anymore, it may invoke pin \texttt{unsubscribe}, upon which the building block unsubscribes from the location server and terminates via \texttt{terminated}. The ESM also allows that an \texttt{update} is combined with an simultaneous \texttt{unsubscribe} via transition \texttt{/out:update+in:unsubscribe}. This is useful when the reception of an update should be taken as trigger to unsubscribe.

The ESMs describe the behavior of the building blocks so that engineers may instantiate and compose them to build more comprehensive services, without looking at their internals. Furthermore, during the analysis of a building block via model checking, the behavior of the building blocks it consists of is abstracted by the ESMs as well, effectively reducing the state space. The internals of building blocks are only needed when components and their state machines are generated and are described by UML activities, as we will see in the next section. The building blocks are stored within a UML repository managed by Arctis. As each building block is a combination of a UML collaboration, an activity and an ESM, Arctis provides an editor that keeps these three views consistent. Syntactic inspections warn if
any conventions are violated. Building blocks may be retrieved via the library of building blocks shown on the left hand side of Fig. 7.3.

7.2.2 Composing Building Blocks

To create more comprehensive services from elementary building blocks, UML activities are used to describe their precise behavioral composition. As an example, we consider the collaboration for the proximity alert as described by the activity in the lower part of Fig. 7.5 (The figure shows a premature design which we will analyze and improve in Sect. 7.3.) The task of this sub-service is to notify the client once a mobile user reaches a certain target position. Each participant of the collaboration is represented by an activity partition. Proximity alert refers to the location tracking service, represented by call behavior action \( t \). The client starts the sub-service by providing the MSID of the player to track and the target location, encapsulated by an object of type Tracking Target. When the tracking target arrives at the proximity server, it passes a fork node which duplicates the token. One copy follows the lower edge to the operation extractLocation in which the location is extracted and stored in the variable target. Within the same step, the other copy follows the upper flow leaving the fork so that MSID is extracted from the tracking target and the track location is started. From then on, the track location emits a token carrying the current location via update every time the subscriber changes position. This updated location is compared in the boolean operation closeEnough with the target location stored in the variable target. If the position is not yet close enough to the target, the false branch is chosen and the flow ends in the flow final node. If, however, the position is close enough to the target, the else branch is chosen, which notifies the client via alert. Within the same step, a token is pushed through unsubscribe of \( t \), so that no more updates are received. Unsubscribe tokens coming from the enclosing context are directly forwarded to the location tracker. Likewise, the termination of \( t \) is forwarded to the client.

To create the specification, the location building block may simply be dragged into the editor. Arctis manages the assignment to activity partitions based on the role binding of the collaborations. As UML does not provide a language syntax to describe actions executed within the operations like closeEnough, the editor also maintains a Java file for each partition that contains corresponding methods which may be edited by the service engineer.

7.3 Automated Model Checking and Analysis

To analyze building blocks and complete systems, the Arctis editor constantly checks the model for a number of syntactic constraints. For a more thorough analysis of the behavior, Arctis employs the model checker TLC \cite{YML99} based
on the Temporal Logic of Actions (TLA, \cite{Lam02}). Fig. 7.2 outlines this process: When a building block is complete and syntactically correct, Arctis transforms the UML activity into TLA\(^+\), the language for TLA, and automatically adds theorems expressing properties that should hold for any application. Then, TLC is started. If TLC reports an error, our tool visualizes the error trace and analyzes the results; in some cases, it provides diagnostics and proposes fixes that the user may apply. In the following, we will describe the details of this process by analyzing the proximity alert collaboration sketched in Fig. 7.5. In Sect. 7.3.1 we show how the semantics of activities are expressed in temporal logic, and discuss in Sect. 7.3.2 how theorems for correct building blocks can be written automatically. In Sect. 7.3.3, we present how error traces are reported back to the developer by means of animations in Arctis, and in Sect. 7.3.4 how diagnosis and fixes can be proposed by the tool in some cases. Sect. 7.3.5 introduces a building block that solves problems of mixed initiatives, and Sect. 7.3.6 presents the complete treasure hunt example system. Finally, we discuss the scalability of the analysis in Sect. 7.3.7.

Figure 7.5: First solution of the proximity alert
7.3. Automated Model Checking and Analysis

Figure 7.6: TLA+ module for the semantics of ProximityAlert

7.3.1 Semantics in Temporal Logic

TLA specifications are structured as TLA+ modules that describe behavior as sequences of steps. This mathematical interpretation fits well with the more graphical representation of activity behavior as token flows; stable states in which tokens rest in places are represented by the variables of a TLA specification, and the token movements are specified by TLA actions (see KH07a). Figure 7.6 lists the TLA+ module for the proximity alert, as generated by Arctis3. In its second line, the module declares the variables representing the states of the specification, followed by their initial values given by Init. After that, the TLA actions are declared. These are predicates on pairs of states each expressing behavioral steps. The Next statement as well as Spec define the actual specification as the disjunction of all of these actions. The last part describes some theorems which we will explain below. For details, we refer to Lam02. As we focus on the analysis of the coordination of concurrent behavior, we ignore in our TLA model the UML variables of the specification (like target in Fig. 7.5) and UML operations on it. We therefore look at the version of the proximity alert in Fig. 7.7 to make the discussion easier to follow.

Due to the refinement semantics employed in SPACE KH07a, TLA actions are formulated in such a way that tokens only rest on places where they wait for other events to happen. This can be the expiration of a timer or the arrival of another token in a partition. For the proximity alert, tokens rest at the flows that cross partitions between client and proximity server, illustrated by the circles q1,q4, and represented in the TLA+ module by the corresponding variables. We assign integers to them storing the number of tokens in the corresponding place. In addition, the ESM state of the location tracking building block is represented by variable t. So, we can effectively reduce the state space when analyzing the proximity alert by

3For readability, we adjusted the automatically derived names of variables and actions.
considering only the more abstract ESM of the location but not its internal details. As the collaboration is open, that means, depends on the interactions from the enclosing context, we represent the state of the enclosing ESM (shown in Fig. 7.5) by the variable $esm$.

In the initial state (declared by $Init$) all queues are empty ($q_1..q_4 = 0$) and the ESM of $t$ as well as the enclosing ESM are in state $off$. In this state, only action $observe$ is enabled and can be executed. Actions refer to pairs of states, where unprimed variables (like $q_1$) model the current state and primed variables (like $q_1'$) refer to the next state. Consequently, action $observe$ describes that the enclosing ESM changes from $off$ to $active$ and a token is placed into queue $q_1$. In the UML activity, this corresponds to a token entering via $observe$ and flowing into the transmission medium between the client and proximity server.

The other actions represent the residual steps our specification describes. Thus, $subscribe$ models the arrival of a token in the proximity server upon which the location tracking is started. Action $update$ represents that a new location arrived, upon which the client is notified via $q_2$ and the location tracking is terminated. Actions $unsubscribe$ and $unsubscribe2$ describe how the client terminates the subscription to the proximity alert. Actions $term1$ and $term2$ model how a termination by the client propagates towards the server, and $alert$ represents the notification of the client once the target is reached.

When used as instantiated building blocks, the initial ESM state and all final states are mapped to the single state $off$, representing an inactive block. For the enclosing ESM of the main activity, we distinguish between the initial state $off$ and the terminated state $finished$ to reason about the life-cycle, as we will later see.

An update not matching the target location corresponds to a step without state change which we left out for brevity.
7.3. Theorems for Correct Building Blocks

There are a number of general behavioral properties that should hold for any building block. To check them, Arctis automatically adds corresponding theorems to the TLA specification, as listed in the last compartment of Fig. 7.6.

- To prevent communication overflows, queue places between partition borders must be bounded. To detect violations, we state with theorem $\_\_\_\text{bounds}$ that $q_1$ to $q_4$ must not exceed a certain number, here chosen to be 5. This has to hold in any state of the specification, which is expressed by the temporal operator $\_\_\_\_\_\text{always}$. (always).

- A building block must be free of deadlocks, in which it does not reach any of its final states. This is covered by theorem $\_\_\_\_\text{deadlock}$, which states that, at any time, the building block must either have reached a final state of its ESM (encoded as finished), or that one of its actions has to be enabled.

- The sub-activities within a building block must be used according to the ESM, so that pins are traversed only in the allowed order. In particular, this means whenever there is a token that can flow into a pin of the sub-activity $t$, its ESM has to be in a state that accepts this token. For our example this means that whenever a token is in queue $q_1$ that could enter $t$ via subscribe, then the ESM of $t$ (see Fig. 7.4) must be in state off, as an entry via subscribe would activate it. Similarly, whenever a token from $q_3$ could unsubscribe, $t$ has to be in state active. These constraints are expressed by theorems $\_\_\_\_\_\text{q1}$ and $\_\_\_\_\_\text{q3}$.

- As the internal behavior of a building block must correspond to its external description, similar theorems are created for the enclosing ESM. For instance, whenever a token in $q_2$ could traverse via alert, the enclosing ESM (see Fig. 7.5) has to be in state active, expressed by $\_\_\_\_\_\text{q2}$. Theorem $\_\_\_\_\_\text{q4}$ works accordingly.

In addition to the general well-formedness properties described above, users may add application-specific constraints in form of assertions expressed by dedicated stereotypes in the UML activities. Examples for such properties are how often certain operations may or must be executed, or if certain operations are mutually exclusive.

7.3.3 Error Trace Animation in Arctis

Arctis generates the TLA$^+$ module as described above and invokes the TLC model checker. During the generation of the TLA$^+$ module, a map from the variable names used in TLA to the elements of the activity is constructed. Therefore, if TLC reports that theorems are violated, our tool parses the textual error trace provided by TLC and maps each state back into the original activity diagram.
Figure 7.8: Visualization of TLC’s error trace in Arctis
Figure 7.3 illustrates how the trace is presented graphically to the user after TLC reported that a theorem was violated. The user can jump through the error trace, animated by tokens in the editor. The state of the activity and each of its building blocks are represented by the corresponding ESM states. In addition, the pins of \( t \) and the parameter node of the enclosing activity are marked based on their corresponding ESM states: A node that may release a token is depicted with a token besides it, while one that must not be passed by a token is shown crossed out.

- In the initial state 1, a token may enter the activity via \textit{observe} and place a token in queue \( q_1 \). This changes the enclosing ESM to \textit{active}, and state 2 is reached.

- In state 2, the client may send an unsubscribe, placing a token into \( q_3 \).

- In state 3, TLC reports that theorem \( t \_q_3 \) is violated. We can see, that the token in \( q_3 \) could enter the ESM of \( t \) via \textit{unsubscribe}. The ESM, however, is in state \textit{off}, because the token that should activate it still resides in \( q_1 \).

### 7.3.4 Automatic Diagnose and Fixes

The presentation of the traces within the editor is already helpful, especially as users do not have to consider any temporal logic formulas. In addition, Arctis can in many cases provide a more distinct diagnosis and suggest improvements. For that, each violated theorem triggers a number of pattern searches that take the UML activity as well as TLC’s error trace as input. In the example, Arctis detected a match for the situation that a token overtakes another one during the transmission between partitions: Between state 2 and state 3, \( q_3 \) is filled while \( q_1 \) has not yet been emptied. This may be intended by the designer. The fact, that the token arriving in \( q_3 \) harms a theorem, however, is a reason for Arctis to report this situation.

As a remedy, Arctis proposes to add a sequencing construct, so that a token in \( q_3 \) can only proceed towards unsubscribe \textit{after} \( q_1 \) was consumed. The altered design is shown in Fig. 7.9 after Arctis added an additional fork, a join node and a timer. Before a token can move from \( q_3 \) into \textit{unsubscribe} of \( t \), it has to wait in the join node until the other incoming flow can offer a token. This may only happen after a token was consumed from \( q_1 \), which enforces the desired sequence. The additional timer prevents that tokens pass through \textit{subscribe} and \textit{unsubscribe}.

---

6Conceptually, UML distinguishes between parameter nodes that are at the border of activities and pins which represent parameter nodes once the activity is instantiated as call behavior action. Both are represented by the same symbols.

7While elements are added and connected automatically, it is up to the user to adjust their layout.

8Since we do not look into real-time behavior during the analysis, timers are just modeled as actions that may execute whenever the timer is active, see also [KH07a].
within the same step, since this would harm the assumptions of the ESM of block \( t \). Further cases for automatic diagnosis and fixes are described in [SLt08].

### 7.3.5 A Building Block to Handle Mixed Initiatives

We let Arctis analyze the improved design. After a new analysis, Arctis reports that a theorem was violated and presents the error trace. For brevity, we directly consider its last state shown in Fig. 7.10. We can see that in this state, a player must have reached the target position, as one token is in queue \( q_2 \). This token may have only arrived there via pin \textit{update} of \( t \). However, in the meantime, the client has chosen to unsubscribe, since the join node following \( q_3 \) contains a token as well. This reveals a situation that is typical for systems in which several active components may take initiatives at the same time, due to the buffered communication. These two initiatives are in conflict. Once identified, such a situation can be handled by assigning primary and secondary priorities to the conflicting partners. An initiative from the primary side is accepted in all cases. For the secondary side, this means that it must be prepared to receive a primary initiative even after it issued an initiative itself, and obey the primary one; the secondary is in this case discarded. The solution may sound trivial, but such situations are intricate to get right, as the generic solution is combined with the complexity of the rest of the application. Thus, often it is not treated with the appropriate care.

As mixed initiatives are so common in this kind of system, we provide special building blocks in our library that solve such situations (see also [KSH07]). In the following, we present one in which the side that starts the interactions has secondary priority, called \textit{Mixed Initiative Secondary Starter}, MISS for short.
The internal behavior is represented by a network of activity nodes that implements the desired behavior, shown in the center of Fig. 7.11. This activity appears quite complex on the first glance. However, an engineer using this building block never has to look at the inside as presented here; the external description is sufficient. It is given by two local ESMs that describe the external behavior on each of the participants of the building block. The side with secondary priority starts the block. If the primary side takes initiative, the block eventually terminates on the secondary side via \textit{primWins}. If the secondary side takes its initiative (in the treasure hunt this means that the timer expired), it has to wait for the primary side to either confirm via \textit{secAccepted} or, if the primary side took initiative in the meantime, be overruled and receive a \textit{secOverruled}. The ESM for the primary side is easier, as its initiative always succeeds, and no waiting for a confirmation in
necessary.

For the proximity alert, we apply the mixed initiative block with the starting secondary side assigned to the client, as shown in Fig. 7.12. In the strict sense, this means a slight advantage for the players, as an arrival is counted if the corresponding notification reaches the proximity server before the timeout. Later, during the usage of the proximity alert, we need to know if the initiative of the client was overruled. Therefore, we propagate this via the ESM of the proximity alert using alertAnyhow. The introduction of a building block to handle this situation makes this design choice explicit. If we want to change this policy (so that for example the timeout should get priority over the arrival of the player), we would simply replace this block by one in which the primary side starts and assigns the corresponding roles to the proximity server and the client.

![Figure 7.12: Correct Proximity Alert with the Mixed Initiative Building Block](image)

7.3.6 The Complete Treasure Hunt System

The complete behavioral system is described by the activity in Fig. 7.13. Each collaboration use from the system collaboration from Fig. 7.3 is represented by a corresponding call behavior action (i.e., $s1$, $s2$, $p$, $r$). In addition, it contains two auxiliary activity blocks $t2$: Timer to measure time and $c$: Countdown as a decrementing counter. These blocks are local to the game server and help to describe the composition between the other collaborations. Therefore, Arctis draws them in blue.

At startup, $s1$, $s2$ and counter $c$ are initialized as tokens are emitted from the three initial nodes $i1..i3$. Then, the single SMS notification $s1$ is waiting for an incoming SMS to start a game. Once it arrives, the player’s MSID is extracted, upon which a welcome message is produced and sent out via $s2$. Within the same step, the riddle generator $r$ is queried for the first riddle. It answers by simultaneously issuing the next target, the granted time for the completion as well as the question in form of an SMS message. The target is used to start the proximity alert collaboration.
In the same step, timer \( t2 \) is started with the granted time as input and the question is sent out to the player. It is now up to the player to move fast enough to the right target, upon which the proximity alert terminates via \textit{alert}, which stops the timer. In addition, a token is sent through the countdown, which determines if more riddles should be sent out. In this case, after decreasing its internal counter, it directs the token to \textit{continue}, which triggers another round. Otherwise, the game is ended successfully and a message is sent to the player. In case the player arrives too late at the target, the timeout from \( t2 \) causes an unsubscribe from the proximity alert and the player is notified that the game is lost.

### 7.3.7 Scalability of the Analysis

Due to the compositional nature of the underlying formalism in temporal logic and the way building blocks are composed with each other, each building block can be analyzed separately. Even when a building block is composed from others, these sub-behaviors are abstracted by their respective ESMs, which leads to a smaller state space than if a block would be analyzed with all its sub-behaviors in place.

To illustrate the reduction of complexity of the analysis through the building blocks
and ESM, we conducted an experiment, in which we compare the size of state spaces of the complete system with those of the strategy of analyzing all building blocks separately.

- When we produce the state space of the complete system without making use of the building blocks and their ESMs (i.e., we expand the behavior contained in all building blocks), the model checking finds 575 distinct states.

- When we analyze each building block separately, the state space for each building block is only a fraction of this number: The treasure hunt system on its highest composition level in Fig. 7.13 has only 7 distinct states. The building block to handle mixed initiatives in Fig. 7.11 requires 22 states, and the collaboration for the proximity alert from Fig. 7.12 has 17 states. The remaining blocks in Fig. 7.13 not further detailed here have similarly few states, with TrackLocation being the most complex with 4 distinct states.

Obviously, real systems tend to be larger than the presented example. However, this does not result in more complex building blocks, but rather in additional levels of decomposition, represented by additional building blocks. Since those can be analyzed separately, as shown, the total effort for validation only increases linearly with the size of the system, since more building blocks have to be analyzed, but not a larger overall state space.

In the FABULA project [KB09], for example, the treasure hunt system is currently extended with chat and instant messaging functions between the users. This is achieved by encapsulating the treasure hunt system as a service, and creating a new system level in which the treasure hunt service is combined with the chat and instance messaging services.

### 7.4 Automated Transformation

The UML activities of the building blocks together with the Java methods for the content of the call operation actions constitute a complete system description. To split this description into separate components, the activities have to be transformed into executable state machines that can be implemented via code generation to run on our execution platforms, as we will detail in Sect. 7.5. Some concepts found in activities have their direct correspondence in state machines. Call operation actions are executed as operations that are part of a transition. Operations on variables stay largely unchanged, and decisions in activities map to choice pseudostates in a state machine. The remaining concepts, however, are fundamentally different (see [KH07b]):

- In contrast to the explicit control states of state machines, activities represent their states indirectly via the different token markings that occur during the
execution. The transformation has to find all reachable token markings and map them to control states.

- The token movements must be mapped to transitions of state machines. There is, however, no one-to-one mapping between activity flows and state machine transitions either. Depending on the markings, one flow may have to be represented by several state machine transitions. This is the case for join nodes, where tokens have to wait until all incoming edges can fire.

- As components communicate via buffered message exchange, flows crossing partition borders have to be split up and translated into corresponding signal transmissions. If a flow carries objects, signal types have to take these objects as payload.

The detailed algorithm to construct the state machine transition identifies the events within a partition that correspond to events in state machines. These are the expiration of timers and the arrival of signals (resp. tokens entering a partition). By traversing the activity graph and taking into account the current marking, the state machine transition is constructed successively. The details of the algorithm are explained in [KH07b]. In the following, we illustrate the transformation process for the game server component.

7.4.1 Scalability of the Transformation

The search for reachable markings implies a state space exploration of the system’s specification. To reduce the state space, we employ a strategy that, similar to our strategy in model checking, utilizes the external behavior of the building blocks as described by ESMs. Only one state machine is produced at a time. Therefore, we only need to consider those building blocks with its internals that directly contribute to the state machine under construction. The other building blocks are abstracted by their ESMs. When we create the state machine for the game server, for example, we may disregard the internals of t: Track Location, while the other building blocks need to be integrated. To generate the state machine for the proximity server, only the building blocks for the mixed initiative and the location tracking have to be seen from the inside.

7.4.2 State Machine for the Game Server

Figure 7.14 shows the state machine automatically produced by the Arctis transformation from the activity of Fig. 7.13. As general UML state machines can be used in various ways, we describe in [KHB06] rules for transitions so that the state machines may be executed efficiently. For example, each transition with exception of the initial one must be scheduled by a signal or a timeout. The application of these rules is noted by the stereotype executable.
The generated state machine handles the mixed initiative between the timer and the alert correctly. This is visible in control state state_4, reached after a timeout, where the state machine is prepared to receive both an acknowledgement of the timeout as well as a primary initiative modeling the arrival of the player at the target.

7.4.3 Correctness of the Transformation

Obviously, it is important that the generated state machines behave exactly as implied by the activities of the specifications. To ensure this, we use temporal logic as well. Similar to the semantics of activities presented in Sect. 7.3.1 we defined formal semantics of the executable state machines in [KHB06], using a compositional variant of TLA, cTLA [HK00]. A system of state machines can therefore be presented as a TLA specification $Spec_E$. As the implementation relation corresponds to logical implication in TLA, we have to prove that $Spec_E \Rightarrow Spec_A$

---

9The signal names are derived from the names of activity edges, not shown here.
7.5. Code Generation from State Machines

holds, where \( \text{Spec}_A \) is the TLA specification of the system as expressed by activities (see [KH07a]). This relationship can be shown by a TLA refinement proof as demonstrated in [Kra08, App. B]. The necessary refinement mapping [AL91] is easy to find using the guidelines described in [KH07b]. Note, however, that such a reasoning is only necessary to ensure the soundness of the transformation once. During the implementation of systems, the service engineers can then rely on the tool to execute the transformation correctly.

7.5 Code Generation from State Machines

The mechanisms for the execution of state machines go back to principles found in telecommunication systems [BHS81] and use a run-time support system that schedules the execution of the state machine transitions, further described in [KHB06]. Through this additional level of multiplexing, many state machine instances can be executed within the same operating system thread, which is important for systems to scale. While these mechanisms can be implemented on a variety of platforms, we focus currently on Java and use the ServiceFrame/ActorFrame execution platforms [BHM02]. These frameworks take care of addressing and routing. The implementation and scheduling of state machine transitions are based on JavaFrame [HM00]. Code for these frameworks is generated automatically with the tool described in [Kra03; Sty04]. The code generator creates OSGi bundles for the components that can be deployed on different machines. In addition, we can generate Java Micro Edition code for the execution on Sun SPOTs [Mer08], targeting embedded devices.

7.6 Related Approaches

A number of other tools combine UML modeling with formal analysis techniques. The majority of these approaches directly uses state machines as the main specification units. HUGO [KM02], for example, verifies UML state machines against UML interactions using the SPIN model checker [Hol03], and UPPAAL [BY03] to check real-time properties. Fujaba [Bur+04] uses so-called real-time state charts that represent behavioral patterns and utilizes HUppaal [Amn+01] for their verification. The specifications in OMEGA [Hoo+08] are based on state machines as well. Using the model checker IF [Boz+04], they are verified against properties expressed by special observer state machines, as described in [OGO04].

Analysis of activities is done for example in [GM05] via SPIN. In [DS03], UML activities are analyzed using the \( \pi \)-calculus. Safety and liveness properties are expressed using the modal mu-calculus and checked using the MWB tool [VM94]. Similarly, Eshuis [Esh06] uses the model checker NuSMV to check the consistency of activity diagrams. The difference of these approaches to ours lies mainly in
the semantics employed for the activities and the domain of application. While they focus on activities more from a perspective of business processes assuming a central clock or synchronous communication, we need for our activities reactive semantics \cite{KH07a} reflecting the transmission of asynchronous messages between distributed components.

In the domain of web services, the tool suite WS-Engineer \cite{Fos+07} enables verification (freedom of deadlocks, safety and liveness properties) of a BPEL implementation and its choreography description. Like in our method, this is accomplished through the use of an underlying formalism (in this case FSP, Finite State Process, \cite{MK06}) and model checking (via the LTSA tool \cite{MK06}). Apart from the differences in chosen formalism and languages, our method differs from that of WS-Engineer in its strong focus on the behaviorally complete definition of reusable, collaborative building blocks and their separate analysis.

There exist also other tools that present the results of a model checker in terms of a graphical model. vUML \cite{LP99} automatically creates PROMELA specifications from UML state charts and model checks them using SPIN. Like us, they mostly check general properties that the users do not specify manually, but they also allow to declare certain states as erroneous or desired goals. Any error traces are presented as sequence diagrams. Another tool is Theseus \cite{Gol+06} which visualizes error traces from the SPIN and SMV model checkers onto UML 1.4 state chart diagrams, and also generates UML sequence diagrams from the trace. While both of the above tools visualize the trace, they do not try to find a reason for the error. Moreover, as error traces are presented as sequence diagrams, the user has to find manually the relation to the original source model. In our case, errors are visible within the same editor used to create the specification.

In \cite{FF06}, a method is proposed for visualizing soundness violations of workflow Petri nets \cite{Aal98}, detected by the Woflan tool \cite{Aal99}, in the WoPeD tool \cite{Wop08}. Soundness violation is separated into five violation classes and a list of eleven error reasons is presented. In the case of a violation, the violating nodes are highlighted with the violation class and the error reason. If a violation is caused by a certain firing sequence of the net, an animation can be shown. Since this approach works on workflow Petri nets, it is quite close to the UML activities used in our case. However, similar to the works on activities mentioned earlier, focus lies on business processes, not on distributed, reactive components with asynchronous communication.

The tool support provided by the SIMS project \cite{SIMS09} uses collaborations as well, albeit in a form that is complementary to the current approach in Arctis. In SIMS, elementary collaborations describe a pair of behavioral interfaces \cite{CFS08}. These can be connected within composite collaborations to describe, how an overall service goal may be achieved. Engineers are supported by validation algorithms that check compliance of state machines with behavioral interfaces. However, these state machines have to be constructed manually.

The SDL pattern tool (SPT, \cite{DEG04}), supports the integration of patterns into
7.7 Concluding Remarks

In the current version of our method and tools we focus on the analysis of general safety properties as detailed in Sect. 7.3.2. Analysis of real-time and liveness properties is subject of future work. Our experience so far, however, shows that checking safety properties, in particular the obligation with the ESMs of the building blocks, uncover many design flaws that should be addressed first. In addition, we study the possibility to ensure application-specific properties (such as mutual exclusion between certain operations, for instance) by applying assertive stereotypes to the activities. Similar to the general safety properties, these assertions are automatically transformed into theorems for model checking [Slt08].

Arctis is used within the applied research project ISIS (Infrastructure for Integrated Services) funded by the Research Council of Norway. In this project we develop methods, tools and platforms for the rapid specification and deployment of services in the domain of home automation. We believe that the collaboration-oriented approach underlying Arctis is ideal in this setting: While there exists a number of rather stable sub-services that provide some basic functionality, the challenge is to compose them quickly, as demonstrated in the example.

Our tool is also used within the FABULA project [KB09], which deals with the creation of learning platforms that make use of location-aware services. In this project, the treasure hunt system as presented here is extended to serve as a general framework for situated learning systems, in which exercises are solved by pupils in a mobile setting. Within these projects, as well as in teaching and student theses [Hau08] [Hei08] [San08], we have tested the effectiveness of our method. The threshold to get started is rather low; a developer familiar with Java and the Eclipse platform can build a running system within a short time, depending on how many existing building blocks may be reused. For the presented system, for instance, we could reuse the blocks for SMS communication (SendSMS and SingleSMSNotification) and location tracking as well as the one to handle mixed initiatives from previous projects. Therefore, editing the entire system as presented in Fig. 7.12 and 7.13 took us less than one hour.

In our opinion, the specification style supported by our method is quite intuitive, due to its way of decomposing systems into collaborative building blocks: The main specification of the system as depicted in Fig. 7.13 is very close to an informal functional description that can be the result of a requirements analysis. It focuses on the distribution of responsibilities and decomposes the system according to its sub-functions. In contrast, state machines (which in our approach are never read
by humans) provide a less comprehensive view. To understand them, detailed
signal transmission must be considered, and elements related to a single function
(like counters, timers or the coordination of mixed initiatives) are mixed with each
other. In activities, on the other side, all elements related to a certain function are
encapsulated within one building block. This supports reuse, since developers may
simply drag existing functionalities in form of building blocks into the editor and
only refer to their externally visible events represented by pins. Since the building
blocks may cover behavior of several participants, entire sub-services may be reused
by referring to a single element. Furthermore, since the building blocks also allow
system specifications to be decomposed into arbitrary many levels, the complexity
of each level (i.e., building block) is quite manageable, which makes the overall
method also scalable from a developer’s point of view.

For the analysis, we follow the strategy proposed by Rushby in “Disappearing
Formal Methods” [Rus00], to hide formal methods in tools in such a way that
users are not directly concerned with them. In our experience, this strategy not
only reduces the threshold to analyze models thoroughly. This is also an incentive
for the use of rigorous modeling in the first place and integrates well with the
paradigms of the Model-Driven Architecture (MDA, [Obj03]).

Based on case studies, we are currently expanding the analytical capabilities of
Arctis, so that more automated fixes and corrections can be offered. That gives
even better assistance to the engineers which, in consequence, reduces development
time further.

Acknowledgement

The research on which this article reports has partly been funded by the Research
Council of Norway through the project ISIS (Infrastructure for Integrated Services,
#180122).

References

Management”. In: The Journal of Circuits, Systems and Computers

lyzer”. In: Syst. Anal. Model. Simul. 35.3 (1999), pp. 345–357. issn:
0232-9298.

Mappings”. In: Theoretical Computer Science 82.2 (May 1991), pp. 253–
284.


References


Frank Alexander Kraemer studied Electrical Engineering and Information Technology at the University of Stuttgart, Germany, and received his Ph.D. from the Norwegian University of Science and Technology (NTNU) in 2008. He is now a postdoctoral researcher at the Department of Telematics, NTNU. His research activities are centered around the development of reactive systems, with a focus on specification styles to enable their incremental construction and verification, formal methods to ensure their correctness, and corresponding tool support.

Vidar Slåtten received his M.Sc. in 2008 from the Norwegian University of Science and Technology (NTNU), Trondheim, Norway. He is now pursuing a Ph.D. degree at the same university, at the Department of Telematics. His research focuses on enabling the rapid engineering and analysis of reliable systems using a combination of model-driven development and formal methods.

Peter Herrmann studied Computer Science at the University of Karlsruhe, Germany, and achieved his diploma in 1990. From 1990 to 1999 and from 2001 to 2005 he worked as a researcher at the University of Dortmund, Germany, and did his doctorate in 1997 on problem-oriented correctness-guaranteeing design of high-speed communication protocols. Since 2005, he is professor on Formal Methods at
the Department of Telematics (ITEM) of the Norwegian University of Science and Technology (NTNU) in Trondheim. Peter works in the areas of formal specification, design, implementation and verification of distributed systems, networked services and continuous-discrete technical systems, functional and security aspects of distributed component-structured software, and trust management. He has represented his institutions in the EU-funded projects iTrust and SIMS as well as in the projects ISIS and UbiComp which are both funded by the Research Council of Norway. He is a member of the IFIP Working Group 11.11 on Trust Management.
Chapter 8

Paper 4

Model-Driven Construction of Embedded Applications based on Reusable Building Blocks – An Example

By Frank Alexander Kraemer, Vidar Slåtten and Peter Herrmann.

Presented at the 14th International System Design Languages Forum (SDL 2009), September 22 – 24, 2009, Bochum, Germany.


The original publication is available at link.springer.com via http://dx.doi.org/10.1007/978-3-642-04554-7_1
Abstract. For the rapid engineering of reactive systems we developed the SPACE method, in which specifications can be composed of reusable building blocks from domain-specific libraries. Due to the mathematical rigor and completeness with which the building blocks are designed, we can provide tool support facilitating a high degree of automation in the development process. In this paper, we focus on the design of embedded Java applications executed on Sun SPOTs by providing dedicated blocks to access platform-specific functionality. These building blocks can be used in combination with other blocks realizing protocols such as leader election to build more comprehensive applications. We present an example specification and discuss its automatic verification, transformation and implementation.

8.1 Introduction

Maybe it is just that engineers still love the LEGO bricks of their childhood, but creating software systems by connecting reusable building blocks seems to be an attractive development paradigm that can facilitate reuse and enable an incremental development style in which problems can be solved block by block. Yet the everyday practice by developers often does not work as smoothly as simply plugging together bricks: Major challenges lie in the nature of reusable modules in the first place, especially in how to encapsulate and how to compose them. Our engineering method SPACE \cite{Kra08,KH06} aims to address these issues. As reusable units we use special building blocks that express their behavior in terms of UML activities. These can be composed by pins, and a system can be constructed as a hierarchy of building blocks. While building blocks can describe local behavior executed by a single component, they can in general also cover collaborative behavior among several components. This facilitates the reuse of solutions to problems that require the coordination of several components, and is especially useful to describe services.

While our method is general and useful in a variety of domains, we demonstrate in this article its application in the area of embedded systems. For that, we present the results of a case study on a sensor network carried out as part of the applied research project ISIS\footnote{Partially funded by the Research Council of Norway, project #180122} (Infrastructure for Integrated Services \cite{ISIS09}), in which we develop methods, platforms and tools for the model-driven development of reactive systems for applications in home network systems. The case study is implemented on small processing devices from Sun Microsystems, called Sun SPOTs \cite{Sun09} that run Java.
In the following, we cover all steps needed to realize deployable code from high-level specifications. We will focus especially on the definition of building blocks for the domain of Sun SPOTs and on a protocol for fault-tolerant leader election. We start with an introduction of Sun SPOTs including the runtime support system, followed by a brief overview of our method. In Sect. 8.2 we present the example system and its high-level specification based on UML activities. The next two sections document our library for Sun SPOTs and the leader election algorithm. In Sect. 8.5 and 8.6, explanations of the automated analysis and implementation follow, in which state machines similar to SDL processes are synthesized, from which code is generated.

8.1.1 Embedded Java on Sun SPOTs

A sketch of a Sun SPOT is shown on the left side of Fig. 8.1. Each SPOT is equipped with two buttons and sensors for temperature, light and acceleration. SPOTs can also carry extension cards to interact with various other devices. A Sun SPOT is controlled by a 32-bit ARM 9 processor that can run the Java virtual machine Squawk [Squ09] executing Java 1.3 code following the CLDC 1.1 specification. SPOTs can communicate among each other using IEEE 802.15.4 radio communication, and build a mobile ad hoc network.

8.1.2 Runtime Support System

To facilitate the execution of many concurrent processes on Sun SPOTs, we have implemented a runtime support system [BH93], sketched on the right side of Fig. 8.1. It includes a scheduler that is responsible for triggering the execution of state machine transitions whenever signals are received or timers expire. Further, a router and an object responsible for the transport of signals support communication using the SPOT’s radio communication. For a detailed description of the execution mechanisms and their formal behavior in temporal logic, we refer to [KHB06]. To generate the state machine classes from UML state machines, we use the code generator [Bje08; Mer08], which produces the necessary Java code.
8.1.3 The SPACE Engineering Method

We developed the method SPACE [Kra08; KH06] for the engineering of reactive systems. This method focuses on the definition of reusable building blocks expressed as UML activities and collaborations, combined with Java code for detailed operations. Building blocks are grouped into libraries for specific domains, as illustrated on the left hand side of Fig. 8.2. Developers can use these blocks by composing them together within UML collaborations and activities: the collaborations describe the structural binding of roles and provide a high-level overview and activities describe the detailed behavioral composition of events, with some additional glue logic where necessary. Each block has an associated external state machine, abbreviated ESM, that provides a behavioral contract describing in which sequence parameters must be provided to or may be emitted by a block. This description is useful for understanding a block without looking at its internal details, and enables compositional model checking, as we describe below.

Once a specification is complete, it is analyzed to ensure various properties that should hold for any application. For example, a composition of blocks should never harm any of the contracts (ESMs) and a collaboration should terminate consistently. For this behavioral analysis, we use model checking. Due to the compositional semantics and the encapsulation of building blocks by their ESMs, the state space needed for model checking tends to be very small, since only one building block on a single decomposition level has to be considered at a time.\footnote{We observe that most building blocks in our libraries require far less than 100 states.}

Complete systems are represented by special system collaborations and activities. When a system is sound, it can be transformed automatically into executable state machines and components, using a model transformation [KBH07; KH07]. From the resulting state machines, code for different platforms can be generated; for the Sun SPOTs (using the code generator described in [Bje08; Mer08] - see 8.1.2 above).
8.2 A Sensor Network for Remote Home Monitoring

An increasingly popular area for home automation is to remotely monitor vacation homes and cabins. Several sensors can be installed in a cabin. One of the assumptions in our project is that embedded sensors with processing capacity similar to Sun SPOTs are so cheap they can also be used in a consumer market. For instance, the sensors can register the temperature at several places, detecting frost or fire. Further, they can detect sudden changes in light or measure acceleration on doors and windows, indicating that somebody is breaking in. With the extension card presented in [AE09], we further assume that each Sun SPOT is capable of GSM communication to set off an alarm to a remote user, for example by means of an SMS.

To improve the quality and robustness of the system, the sensors communicate among each other before sending an alarm via GSM. This serves several purposes: First, multiple sensors can be used redundantly, so that important conditions are monitored by more than one sensor, whereas only one alarm should be issued. Second, some conditions may give rise to alarm if the sensors are triggered in a certain pattern. For example, while changes in light of one sensor could indicate a broken window shutter, a change observed by several sensors may simply be due to a cloud moving in front of the sun. This means that alarms need to be coordinated. For that reason, we use a leader election protocol that points out one SPOT sensor to filter and issue alarms. If the leader runs out of battery or otherwise fails, a new leader takes over. Such a network is illustrated in Fig. 8.3.

Figure 8.4 shows the UML activity describing the behavior of a SPOT sensor as composed from our reusable building blocks. Since the SPOT sensors of the system all have the same behavior, it suffices to specify only one of them. To visualize

---

3 We will not discuss detailed patterns describing when an alarm should be triggered, and we will also disregard the configuration of individual SPOT sensors.
the relationship of a SPOT sensor to the other sensors explicitly, however, we use two activity partitions. The left one, *spot sensor*, describes how a SPOT sensor is composed from building blocks, which defines the behavior. The right partition, *other spot sensors*, enables us to represent the communication with the other sensors. This partition is only sketched, as only the left one will be used for the transformation and code generation.

A sensor consists of a block for GSM communication $g$, the alarm filter $a$ and three building blocks accessing the Sun SPOT’s sensors for motion ($s1$) light ($s2$) and temperature ($s3$). While these blocks encapsulate local behavior, a building block can also comprise collaborative behavior that is executed by several participants. The leader election, contributed by building block $l$ in Fig. 8.4, is a typical example for that. It is a collaboration among several SPOT sensors, and therefore crosses the activity partitions. Internally, the block specifies the establishment of contact between all the sensors and how a leader is selected amongst them. This behavior

---

4Technically, blocks are modeled as UML elements of type *Call Behavior Action*, which can refer to subordinate activities.
is detailed in Sect. 8.4.

The activity also contains references to the operation \textit{create event}. Since UML does not have a concrete language for actions, the details of these operations are specified by Java methods, managed by our editor. The other elements in the activity are initial nodes (●) as well as merge and decision nodes (◇). Decision nodes are followed by flows that are guarded (★).

Upon the start of a SPOT sensor, the initial nodes emit a token and start all blocks, including the collaboration for the leader election. The alarm filter is started as well, so that the SPOT by default uses its own GSM Alarm block to send any SMS notifications, until it finds another leader. The leader election emits a token through \textit{new leader} once it detects a SPOT that is pointed out as the new leader, carrying its ID. In case a SPOT itself is pointed out as leader, a token is emitted through \textit{i am leader}. In both cases, the ID of the leader is stored in variable \textit{leader}. If a SPOT becomes leader, the alarm filter is started, and if the SPOT loses its leader status, the alarm filter is terminated.

Whenever one of the sensors \texttt{s1}, \texttt{s2} or \texttt{s3} registers a condition, it emits a token via its output pin, upon which an event is created containing the kind of condition and ID of the sensor. If the SPOT owning the sensors has the leader role (i.e., guard \texttt{leader==myID} is valid), the event is directly passed to the alarm filter. Otherwise, the SPOT sensor forwards the event to the current leader. In this case, the leader is one of the other SPOT sensors, and sending to it is specified by the transfer edge \textcircled{1}. Since the other SPOTs are potentially many, we have to select which one to address, using the \texttt{select} operator introduced in [KBH07]. It refers to the ID of the leader. Vice versa, if a SPOT sensor has the leader role, it may receive events from other SPOT sensors (at \textcircled{2}).

Figure 8.5 provides an overview of the dependencies between the building blocks used for the specification of the SPOT sensor system. Most of them are taken from our existing libraries (listed here with only those blocks used in the example). The Alarm Filter, the experimental GSM Alarm, and the complete system are specific for the example.

8.3 Building Blocks Specific for Sun SPOTs

Our library for Sun SPOTs contains twelve building blocks dedicated to the specific capabilities of the devices, such as the buttons, all sensors on the SPOTs, and the LEDs. In the following we present some of those that are used in the SPOT sensor system.
8.3. Building Blocks Specific for Sun SPOTs

8.3.1 Building Block for Sensors

Figure 8.6 shows the internal details of the block for the detection of movements. The accelerometers of the Sun SPOTs are accessible via a special API. To react on sudden accelerations that exceed a certain threshold value, a listener is registered at the SPOT classes that provide access to the hardware. To keep the execution of the code reacting upon an event under the control of the scheduler of our runtime support system (RTS), the building block uses an internal signal as buffer, to decouple the processes. For this reason, operation `register listener` creates a listener, which, upon its invocation following a sudden movement, produces a signal `MOVED`, that is fed into the RTS. Once this signal is processed, the behavior following the accept signal action declared for `MOVED` in Fig. 8.6 is executed: a token is emitted via output node `moved`, and the listener is re-activated, to listen for further movements. The blocks controlling the light and temperature sensors access the SPOT API in a similar way.

On the right hand side of Fig. 8.6, the ESM for the motion sensor is shown. As mentioned previously, it documents the behavior visible at the pins of an activity, so that we know its external behavior when it is instantiated as a block as in Fig. 8.4. Due to the ESM, we know that after a token enters `activate`, tokens may be emitted via `moved` until we terminate the block via `stop`. 
8.3.2 SPOT Discovery

To dynamically find other SPOTs in the sensor network, we provide a collaborative building block which uses the Sun SPOT’s broadcasting functions so that they can discover each other. The corresponding activity is shown in Fig. 8.7. The partition beacon describes how a SPOT that wants to be discovered sends out periodic messages. Since these messages are specific for Sun SPOTS, they are sent directly from the Java operation, instead of using our runtime support system. The partition listener describes the logic to be implemented by a Sun SPOT that wants to discover other SPOTs. For that, it listens to the incoming beacon messages. To decouple the receiving processes from the scheduling of state machine transitions, once such a message arrives, it is fed into our RTS via signal FOUND, similar to the listener reacting to the movement of a SPOT explained above. If the ID is not yet known, a token is emitted via found spot. Notice that if a SPOT wants to both discover other SPOTS and be discovered, it instantiates this collaboration twice, once as a beacon and once as a listener.
8.4 Collaborative Building Blocks for Leader Election

To make sure that only one of the SPOT sensors forwards an alarm over GSM, we use a fault-tolerant leader election protocol. Should the leader SPOT run out of battery or otherwise fail, another one must take its place so that alarms are still sent if necessary. To solve this problem, we implemented an algorithm from [Gar02]. The algorithm uses an Infinitely Often Accurate Detector (IOD) as failure detector, a concept from [CT96], which is used by a component to monitor if any of its communication partners have crashed. In Sect. 8.4.1 we provide a dedicated building block for this function.

The collaboration in Fig. 8.8 specifies the structural aspects of the leader election. It depicts the participant candidate as collaboration role, and refers to the sub-services for SPOT discovery and failure detection by collaboration uses d1, d2 and i1, i2. The leader election is a symmetric collaboration, in which all participating roles have the same behavior, and the role for the candidate is therefore represented twice. For the model transformation and the code generation, the left candidate is used. To make the collaboration with the other candidates explicit, we refer to the other candidates on the right hand side, similar to our proceedings with the SPOT sensors in Sect. 8.2.

\footnote{In the fault-tolerance domain, a node is said to crash if it from some point on permanently ceases all operations, but works correctly until then (see [Tan02]).}
8.4.1 Infinitely Often Accurate Detector (IOD)

In our example, we use the Infinitely Often Accurate Detector (IOD, [Gar02]) as specified in Fig. 8.9. The partition on the left side models the observed SPOT, which periodically sends so-called “alive” messages to the observing SPOT, represented by partition observer. These messages are triggered by the periodic timer \( p \) and carry the ID of the observed SPOT. The observer SPOT maintains two variables to store the status of the observed SPOT; observedID for its ID and the boolean isSuspected. Moreover, the observer has a timer \( t \) to determine if the alive message from the observed SPOT is delayed.

Whenever the observer receives an alive message from the observed SPOT, it reacts depending on the current value of isSuspected:

- If the observer does not suspect the observed SPOT sensor of having crashed, it will simply restart timer \( t \) and wait for the next alive message.
- If, on the other hand, the observer currently suspects the observed SPOT of having crashed, the observer will change isSuspected, increment the timeout period\(^6\), and emit the observed’s ID through output node not suspected.

If, however, timer \( t \) expires (i.e., no alive message was received in time), the observer will suspect the observed SPOT of having crashed, set isSuspected accordingly and emit a token carrying the observed SPOT’s ID through output node suspected.

Since a message could also be delayed in the communication medium, a timeout does not always mean that a SPOT has crashed. Hence there may exist transient states in which two SPOTs are both considered the leader. This, however, is acceptable for our application domain. For a detailed analysis and proof of the properties of the Infinitely Often Accurate Detector, we refer to [Gar02].

8.4.2 Composed Building Block for the Leader Election

The detailed behavior of the leader election is expressed by the activity in Fig. 8.10. Similar to the overall system of Fig. 8.4, the leader election is symmetric. The partition candidate on the left side represents one participant and its detailed behavior, while the partition to the right represents its communication partners.

As part of the leader election, a SPOT participates in the Infinitely Often Accurate Detector (IOD) collaboration as both observer and observed entity. This is represented by blocks \( i1 \) and \( i2 \), which both refer to the activity in Fig. 8.9 but which are bound to partition candidate with roles observed resp. observer. Moreover, this collaboration is executed as multiple concurrent sessions (once towards each

---

\(^6\)Incrementing the timeout period upon detecting a false suspicion ensures that the observer will wrongly suspect the observed only a limited number of times.
8.4. Collaborative Building Blocks for Leader Election

When the leader election collaboration is activated, the *SPOT Discovery* collaboration is initialized as both beacon (*d1*) and listener (*d2*), according to the role binding in Fig. 8.8 so that a SPOT sensor can both detect others and be detected by others. For each sensor found, a token with its ID is emitted via pin *spot found* of *d2*. This ID is used to start a new session of the IOD collaboration *i1*, so that a SPOT is observed by any other SPOT it detects. For that we use again the *select* statement, which this time refers to the value provided by the token flow. Vice versa, once a SPOT is detected by other SPOTS, they start a new instance of the IOD collaboration (in this direction represented by *i2*).

Via the output pins *suspected* and *not suspected* on *i2*, a SPOT is notified about perceived changes in the state of each of the other SPOTS. The logic that follows determines the current leader status. For that, hash table *suspects* maps the ID of the other SPOTS to their respective status (suspected or not suspected). Whenever *i2* issues a change in state of another SPOT via one of its output pins, the subsequent operations store this change to the hash table and determine the new leader. If several SPOTS qualify for the leader status, the one with the lowest ID is chosen. If the leader has changed, we store the new leader and check if the

![Figure 8.10: Building block for leader election](image-url)
new leader is this SPOT. Depending on the outcome, a token is emitted through either the *i am leader* or *new leader* output node.

### 8.5 Automated Analysis

The analysis of the specification is based on model checking. This process is automated, since our tool also generates the corresponding theorems to be verified. Currently, we check the following generally desirable system properties [KSH07]:

- A building block must conform with its own ESM. The motion sensor of Fig. 8.6, for instance, may not emit a token via node *moved* after the surrounding context provided one via *stop*.

- A building block must also obey all ESMs of the subordinate blocks it is composed from.

- Building blocks with more than one participant are checked for bounded communication queues. For the IO detector in Fig. 8.9, for instance, we find that the periodic timer could, in principle, overflow the queue between the observing and the observed component.

The analysis focuses on the soundness of interactions among collaboration participants as well as the correct composition of all building blocks with respect to event orderings. The content of operations (that is, the Java code) is not part of the analysis. In cases where decisions are involved that depend on variables, the analysis always examines all alternative branches. If the executions of some branches may harm certain properties, we reason manually if these cases may in fact happen. For instance, in the IO detector of Fig. 8.9, the else branch may restart the timer before it is started. This, however, never happens in the final system because of the value of *isSuspected*.

The results of the analysis are presented to the user by explanatory annotations within the original UML model, so that no expertise in the underlying formalism is required, as demonstrated in [KBH09]. In addition, counter examples illustrating design flaws are presented as animations within the activities. In our experience, checking the above mentioned properties is of great value in the practical development of specifications. Although these properties may appear simple when considered in isolation, even experienced engineers usually harm several of them in initial designs, especially when more complex collaborations are constructed.

Due to the compositional semantics of our method, each building block can be analyzed separately. Internal building blocks are abstracted by their ESMs, so that the global state space of the specification in Fig. 8.4 has only 15 distinct reachable

---

7 In this case, however, we estimate the time needed for the transmission and subsequent processing and conclude that this is not an issue in a real system.
states. Moreover, since most of the building blocks are taken from libraries and are already analyzed, only the new ones created for the specific applications have to be examined. These are the ones for the SPOT Sensor System, the Alarm Filter and the GSM Alarm.

8.6 Automated Implementation

As briefly mentioned in the introduction, the implementation is performed by a completely automated process with two steps: In a first step, executable state machines are synthesized from the activities. In a second step, code is generated. This is possible since the activities provide descriptions that are behaviorally complete, and the details of operations are provided as Java methods as part of the building blocks.

8.6.1 Transformation to Executable State Machines

In Fig. 8.11 and 8.12, we present the state machines as generated by the transformation. In our method, they are only an intermediate result used as input for the subsequent code generation; developers do not have to edit or read them. In the following, we highlight some properties to demonstrate the soundness of the transformation.

For the partitioning of components into state machines (or processes in SDL), our algorithm follows the guidelines from [BH93]. In particular, the algorithm merges all behavior of building blocks that is executed one at a time by the component under construction into one single state machine. All blocks that denote multi-session collaborations (behavior that is executed multiple times towards a changing number of different communication partners) are implemented by dedicated state machines, one instance for each session, as presented in [KBH07]. For the SPOT sensor system, for instance, the algorithm creates the state machine Spot Sensor, depicted in Fig. 8.11 which takes care of the main component behavior. This includes all logic contained in the building blocks used in Fig. 8.4. However, since the behavior of the Infinite Often Accurate Detector is executed concurrently within each SPOT sensor (once for each other sensor detected), its behavior is implemented by dedicated state machines. These are state machines Observer and Observed in Fig. 8.12.

The main state machine Spot Sensor has two\(^8\) distinct control states, 1 and 2. This is because the transition behavior only has to distinguish if a SPOT is the leader or not. When a spot is the leader, the alarm filter is active and the state

\(^8\)This is less than the 15 states from the previous analysis because the analysis also captures the interleaving with other SPOTs and the queues for communication, which do not contribute any control states for a local component.
Figure 8.11: Bird’s eye view of the synthesized state machine for the Spot Sensor
8.6. Automated Implementation

8.6.2 Code Generation for Sun SPOTs

Since the Sun SPOTs execute Java, the code generator described in [Mer08] is largely based on the standard Java code generator, described in [Kra03]. As introduced in Sect. 8.1.2, the execution is based on a runtime support system, which takes care of scheduling, routing and transport of messages. The scheduler (see Fig. 8.1) maintains event queues for each state machine in which incoming messages and active timers are placed. In a round-robin manner, the scheduler triggers the execution of state machine transitions by feeding the event into a dedicated transition method, which is specific for each state machine type. The transition method contains nested if-statements that distinguish the current control state and input event and then execute the effect as specified by the UML transitions in Fig. 8.11 and 8.12. Effects referring to operation calls on the activity level, such as determine new leader in Fig. 8.10, are copied into the transition method. Other actions that are part of a transition effect, such as sending signals or operations on timers, are synthesized from the UML model. The transport module (see Fig. 8.1), responsible for sending and receiving messages from and to other SPOTs, uses the radio stream protocol from the Sun SPOT API to transmit messages. This protocol provides buffered, reliable, stream-based communications over multiple hops on top of the IEEE 802.15.4 radio protocol. The content of the messages sent via the radio channels are SOAP-documents generated with the help of the kSOAP libraries [Kso09], as described in [Bje08]. For the necessary serialization of
objects, the code generator adds methods that convert objects and primitive types to strings.

8.7 Estimation of Reuse Proportions

To estimate the degree of reuse for the exemplified system, we distinguish between the building blocks that are part of our libraries and intended for reuse, and those building blocks constructed specifically for the application. These are shown in Fig. 8.5 with the libraries on the left hand side. As application-specific we count the Alarm Filter, the GSM Alarm and the overall SPOT sensor system. The effort necessary for the construction of a building block consists of the UML models on the one hand and Java code contained within the call operation actions (like determine new leader in Fig. 8.10) on the other hand.

- By counting the lines of code contained in the call operation actions in each building block, we find that there are $l_{\text{blocks}} = 443$ lines of code within the call operation actions for all building blocks used in the system in total. Those building blocks taken from libraries contribute with $l_{\text{lib}} = 333$ lines, so that the reuse proportion $R_{\text{code}} = l_{\text{lib}}/l_{\text{blocks}}$ is 75 %.

- As an estimate for the effort spent UML modeling, we use a simple metric that just counts the number of activity nodes and activity edges $n = n_{\text{nodes}} + n_{\text{edges}}$ within a building block. This metric shows that all building blocks used in the system consist of $n = 219$ edges and nodes in total. Those building blocks taken from the library contribute with $n_{\text{lib}} = 144$ elements, so that the reuse proportion $R_{\text{model}} = (n_{\text{lib}}/n)$ is 71 %.

Of course, these numbers vary for different systems. For the given example, we have programmed a relatively simple logic for the alarm filter, which contributes only 50 lines of code. Since the GSM module is not yet finalized, we estimate another 50 lines for that building block.

To get an impression of the overall gains including the automatic implementation, we consider also the complete code needed for the execution on top of the runtime support system. The code generated automatically for the state machine logic adds up to $l_{\text{stm}} = 634$ lines, and the number of code lines written manually for the Java operations copied from the building blocks as mentioned above is $l_{\text{blocks}} = 443$. This means that the code necessary for the entire application has $l_{\text{total}} = l_{\text{stm}} + l_{\text{blocks}} = 1077$ lines, from which $l_{\text{stm}}/l_{\text{total}} = 59 \%$ are generated automatically. If we add up these numbers, we find that $(l_{\text{lib}} + l_{\text{stm}})/l_{\text{total}} = 90 \%$ of the Java code lines are either reused or generated from the UML models.

\footnote{The underlying runtime support system has about 1900 lines of code. Since it is provided as a library that can be reused also in manual approaches, it is not part of our calculation.}
8.8 Related Work

There exist a number of approaches for the model-based design of reactive systems that are also suitable for embedded applications. Some of them based on SDL such as TIME [Brk+97], SPECS [Öls+94], SOMT [Tel02] and SDL-MDD [KGW06]. Others, such as ROOM [SGW94] (later UML-RT) or Catalysis [DW99], are oriented towards UML as language. As design models that describe the behavior of individual components, these approaches use state machines, either in the form of SDL processes or as UML state charts (called ROOM charts in [SGW94]). To capture collaborative behavior among several components, most of these approaches rely on MSCs. Catalysis [DW99], inspired by the Object-Oriented Role Analysis Method (OOram, [RWL95]) and DisCo [Jar+90], on the other hand, uses collaborations more explicitly in specific diagrams, albeit in a rather informal way that requires manual synchronization by the developers. Micro protocols [FG07] are another approach to capture and encapsulate communication protocols within self-contained units, by using pairs of SDL processes or composite states.

In principle, these approaches are compatible with the one presented here, since all the design models based on state machines with their emphasis on event-driven transitions are quite similar. The difference lies in the models on which developers work: To enable the composition of collaborative behavior as self-contained building blocks, we use UML activities, from which the state machine-based design models are derived automatically. This enables a number of opportunities for the reusability, the analysis and the overall specification style, as we will argue below.

8.9 Concluding Remarks

In our experience, the composition as enabled by activities, shown for example in Fig. 8.4, is quite flexible. We attribute this to two major reasons: First, the complete but cross-cutting nature of UML activities, in which the coordination of several participants can be described within the same diagram. If, for example, we would like to exchange the selected leader election protocol with another one, we would just have to replace the building block l in Fig. 8.4, and its connections to the other blocks, which can be achieved by focusing on one single diagram. Second, the way activities enable the encapsulation of functionality related to a certain purpose as separate, self-contained building blocks. While state machines offer some means of structuring (for example composite states), they do not offer the same degree of flexibility and separation as activities. The functions encapsulated by the building blocks in Fig. 8.4, for example, are dispersed among several transitions in the state machines of Fig. 8.11 and 8.12. One reason for that is that state machines represent their states by explicit control states, while activities use concurrent flows that may execute independently. Although such behavior can to a certain degree be described in state machines by concurrent regions, such a description style gets
intricate once the behaviors in these regions need to be synchronized. However, since state machines are very suitable for the specification of the executable behavior of components, we generate them in the described way, so that we have both the compositional features of UML activities and the efficient scheduling of state machines.

Besides these properties coming from the chosen notation, an important feature of our method is the compositional verification it enables, based on the underlying semantics in cTLA \[HK00\]. Not only does this reduce the state space during model checking, but it also has important effects on the larger scale development process. Since building blocks can be verified individually, proven solutions can be encapsulated in building blocks, and these can be checked and stored in a library. Whenever a building block is reused, the verified properties are enforced automatically and do not have to be re-verified. This enables “true reuse” as mentioned in \[DW99\], in which reuse does not mean to simply copy and paste some parts of a specification, but also ensures that important properties are maintained.

All things considered, we think that the chosen principles and the way they are combined enable a reuse-oriented specification style, one that encourages the use of encapsulated building blocks to a high degree, but that still allows us to adapt systems to match the requirements of the individual application. This is a crucial step towards the cost-effective LEGO-brick like development paradigm.

Acknowledgements We would like to thank Fritjof Boger Engelhardtsen from Telenor for support with the GSM module, as well as Marius Bjerke and Bemnet Tesfaye Merha for their enthusiastic work on the Sun SPOTs in their theses.

References


References


Chapter 9

Paper 5

Towards a Model-Driven Method for Reliable Applications: From Ideal To Realistic Transmission Semantics

By Vidar Slåtten, Frank Alexander Kraemer and Peter Herrmann.


The original publication is available at dl.acm.org via http://doi.acm.org/10.1145/2401736.2401737

Note: Having used the same example system several times already [KH06; KH07], we forgot to attribute the original example to [BH93; BS01].
Towards a Model-Driven Method for Reliable Applications: From Ideal To Realistic Transmission Semantics

Abstract. We present a model-driven method to incrementally introduce fault-tolerance mechanisms into application models that are initially developed with assumptions of ideal transmission semantics. As main structuring units, our models use collaborative building blocks in UML that can encapsulate the behaviour of several participants in order to perform a certain task. Since these building blocks can be designed and analysed separately, fault-tolerance mechanisms can be introduced block by block, which reduces the size and complexity of specifications that have to be understood at a time. Applying fault tolerance at the application layer also brings the benefits of easily porting applications to other platforms and applying model-level analysis tools to the fault-tolerance mechanisms themselves. We illustrate our method through the development of an access control system.

9.1 Introduction

A major challenge when developing distributed, reactive applications is that each component an application consists of typically has to maintain interactions with several other components. This implies a considerable amount of coordinating logic. Such logic gets even more complex once applications should handle situations in which communication is disturbed by flaws like message loss. Luckily, in many cases, already simple fault-tolerance mechanisms can lead to more reliable applications. For example, timers started when waiting for response signals can protect a component from waiting forever for an answer that may have been lost in a channel.

Though the introduction of even simple fault-tolerant behaviour itself is a task that increases development time, it has an additional effect that makes it problematic during the development: Fault-tolerance mechanisms obscure the essence of the actual application logic, making it harder to understand in the beginning and maintain in the long run. For instance, in a finished, fault-tolerant specification it may not be immediately clear if a certain timer is an important application feature or only has the task to monitor a communication channel, since fault-tolerance mechanisms and application logic are mixed together.

Another reason demanding a clear strategy for the introduction of fault tolerance is rather practical. The expertise needed for securing an application with respect to fault tolerance is different from the expertise needed for application development in specific domains. Only few engineers cover both. Our method therefore explicitly addresses two separate groups of experts; one for the specific application domain in which the system should be applied and one for fault tolerance in general. This
is analogous to our method presented in [GHK09] to handle security aspects by a separate team of experts.

For these reasons, we propose a two-phase method, depicted in Fig. 9.1 based on our engineering method for reactive system, SPACE [KSH09]. In a first phase, a system specification is developed by experts for the specific application domain in step D1. As major specification units, we use special self-contained UML building blocks addressing a single task. Several case studies have shown reuse proportions of 71% on average, see [KH09]. Therefore, domain-specific libraries can offer existing solutions that can be composed to more comprehensive units, until a complete system is obtained. We will detail this development phase in Sect. 9.2. Once this specification passes the analysis based on model checking A1, the majority of domain-specific design is complete and fault-tolerance experts become involved. We refer to a specification at this stage as *idealized*, meaning that it is fault-intolerant, but free from the design faults checked for by the analysis step.

![Figure 9.1: Development Method](image_url)

Within the second phase, the initially idealized specification is incrementally im-
proved to match more realistic transmission semantics in which messages in transit can be lost (step D2). This task is performed by an expert on fault tolerance, assisted by an expert on the application domain when necessary. Similar to the initial development in D1, this step is supported by a library of fault-tolerance mechanisms that store solutions to reoccurring problems. This development phase is the main contribution of this paper and is further detailed in Sect. 9.4.

Once the system passes analysis A2 it is considered reliable (fault-tolerant and free from design faults) and can be implemented by our automated process consisting of a model transformation [KH07] with a subsequent code generation step. This step supports different execution platforms such as standard Java, embedded Java for Sun SPOTs [KSH09a], Android [Hau09], as well as Telenor’s Connected Objects Operating System (COOS, [Her+09]).

In the following, we will detail the development steps of phase 1 for the application domain in Sect. 9.2 by the example of an access control system. The models in Sect. 9.2 assume ideal transmission semantics, i.e. perfect channels. In Sect. 9.3, we introduce realistic transmission semantics. This is the starting point for development phase 2, discussed in Sect. 9.4, that incrementally evolves the idealized system to a reliable one by introducing fault-tolerance mechanisms. We present related work in Sect. 9.5 and discuss our method in Sect. 9.6. We end with some concluding remarks in Sect. 9.7.

While our tools also handle activity diagrams with object nodes, operations and flows, we disregard data within this paper for clarity. For system specifications that include data, see [KSH09a; KSH09b].

9.2 Idealized Application Models

To illustrate our approach, we consider an access control system. At the top of Fig. 9.2 the system’s structure is specified by a UML collaboration, with the icons representing its participants (UML collaboration roles). They show that the system consists of the actual door to control, an input panel, a local station located in the vicinity of the door and the panel, a central station and two servers for authentication and authorization. The ellipses (UML collaboration uses) in between refer to collaborations that describe functions between their participants. The collaboration use pc: Panel Control, for instance, covers the behaviour between the local station and the panel, in which users provide their access code.

Since UML collaborations are a structural description without any behaviour, we use UML activities to complement the specification, as shown in the lower part of Fig. 9.2. Here we see that each collaboration role is represented by its own activity partitions for the panel, the door, the stations and the servers. The collaboration uses are represented by call behaviour actions that in turn refer to activities.
At their frames, they have pins attached which are used to control their behaviour.

The system is started via the initial node within the local station, which starts the panel. Once users enter their personal identification (pid), the corresponding pin from block *pc* emits a token containing it onto the flow. The token is forwarded to the central station, where it is forwarded, after the fork node, to both the authenticate and authorize collaborations, more generally referred to as *building blocks*.

The internal behaviour of the authenticate block is shown by the UML activity diagram to the left in Fig. 9.3. It is a simple inquiry pattern, where a server validates the pid and the client interprets its answer as either ok or not. The authorization works in a similar way.

To utilize these blocks within the access control system, however, the internal details are not important. It is sufficient to look at the external behaviour of these blocks, expressed by the so-called external state machine (ESM) to the right in Fig. 9.3. It shows that after a token is provided via input node *pid*, the authentic-
ation terminates either via *ok* or *nok*.

![Authenticate diagram]

**Figure 9.3: Behaviour of the Authenticate building block**

The central station of Fig. 9.2 collects the answers of both the authenticate and the authorize collaborations. Note that they were started simultaneously, but the servers may respond in any order. The results are fed into the block *a:And*, which realizes a logical function that corresponds to a boolean and gate: Only when both the authentication and the authorization are *ok*, *true* is sent back to the local station upon which the door is opened. In all other cases, the door remains closed. The ESMs of *Panel Control* and *Door Control* are depicted in Fig. 9.4.

Since all building blocks are encapsulated by ESMs, the access control system of Fig. 9.2 can be simulated and analysed even if the panel or door control blocks are not yet specified internally. Once all building blocks are complete, the system (although assuming ideal channels) can also be implemented and executed as an early prototype. This can be used to uncover situations not yet considered and to get early feedback from the customers.

![Panel Control and Door Control diagrams]

**Figure 9.4: External descriptions of the Panel Control and Door Control building blocks**

The tool suite that accompanies our method has analytic capabilities to aid the developer in creating a well-formed specification. Syntactic inspectors check that the model is syntactically correct, for example that all output pins have a connected flow. Semantic analysis is achieved through the use of a model checker [YML99]. The semantics of our models are precisely defined so that an automated transformation to a model checking language can be done [KSH09b]. Properties like freedom from deadlocks and that the composition respects all ESMs are then automatically formulated and verified.
The access control system is analysed and does not violate any properties as it is now.

9.3 Transmission Semantics

The activities in Fig. 9.2 and 9.3 use control flows that cross partition borders. Since partitions denote physically distributed components, this implies some form of communication, in which data is transmitted from a sender to a receiver. For all platforms we generate code for, this communication is provided by an asynchronous message bus, in which the sending operation is decoupled from the receiving of a message, so that a sender does not get blocked. This also means that there is no upper bound on the time it may take a transmitted message to reach its receiver. To mirror this in the execution semantics for activities based on token flows, we therefore assume that tokens are buffered between partitions in an implicit waiting place, as illustrated in Fig. 9.5. We assume in the following that there is at most one message corresponding to a certain activity flow under transmission, that means that the place accepts at most one token.

![Figure 9.5: Implicit waiting place for transmissions](image)

**Ideal Transmission Semantics.** The models in Fig. 9.2 and 9.3 assume an ideal form of communication, in which messages are never lost. This means that the transmission between partitions has the semantics described by the building block in Fig. 9.6, with a send and receive operation. Every token sent will eventually be received, as expressed by the ESM to its right.

![Figure 9.6: Ideal transmission semantics](image)

**Realistic Transmission Semantics.** To represent message loss, we define realistic transmission semantics by the building block in Fig. 9.7. It has the same send and receive operations as the idealized transmit in Fig. 9.6 but also models, by pin and ESM transition lost, that tokens can be lost and hence never be received. (Since lost and receive are mutually exclusive, they technically belong to different UML parameter sets denoted by the additional box around them.)
9.4 Reliable Application Models

If we assume realistic transmission semantics, i.e. that channels may lose messages, a single lost message anywhere in the access control system will leave it deadlocked. The analysis step A2 therefore reveals the following error scenarios:

- The initial token from the local station to the central station containing the \textit{pid} is lost, so the local station waits forever for a reply.
- Any one of the responses from the central station to the local station are lost, also leaving the local station waiting forever.
- Any token lost between the central station and one of the servers causes both the local station and the central station to wait forever.

In the event of a message being lost by a channel, the application should be notified so that it can handle the event in the best way possible for its specific domain. An application could, for example, attempt to compensate for a broken channel by using a redundant information source. Otherwise, it could degrade its service, simply informing its users that requests currently cannot be handled. To achieve that the application is notified of message loss, it can either register for such events through a platform API, or use a timeout mechanisms explicitly represented within the model. The rest of this section presents such a mechanism, how it can be integrated with the access control system and how it can be encapsulated for reuse.

9.4.1 Reliable Notify

Unreliable channels do not explicitly notify any party in the event that a message is lost. Hence, detection must be done at the endpoints. The sender of a message can start a timer when sending a message onto a channel, and ask for an explicit acknowledgement of receipt from the receiver. If the acknowledgement message is received on the sender side, the sender knows for sure that the message it sent reached its destination. However, the lack of the acknowledgement message when the timer expires can be interpreted as any of the following situations:

- The sender’s message was lost before reaching the receiver.
- The acknowledgement message has been lost on its way back do the sender.
The sender’s message, or its acknowledgement message, is still in the channel.

Hence, there is no way of being sure that the sender’s message has been lost; all we can do is set a time after which it is unlikely that the acknowledgement message will arrive and consider the sender’s message as lost after that time. To prevent a message or acknowledgement mistakenly assumed to be lost from interfering with later interactions, we assume a numbering mechanisms for filtering them out.

We introduce a solution for a reliable notification service in the form of a building block, Notify R, shown in Fig. 9.8 (We use the naming convention that R is added as suffix to the name to mark that a building block is reliable, i.e. that it has passed analysis A2.) This building block encapsulates the behaviour just described to detect message loss: When a token arrives through pin in, the sender side puts it onto the channel while at the same time starting a timer. Upon receiving the token from the channel, the receiver side delivers it to the enclosing building block via pin out. The receiver also sends an acknowledgement back to the sender side. When the sender receives the acknowledgement, the building block terminates via the ack pin. Terminating the block also removes any token in the timer. Should the timer expire before the acknowledgement arrives from the receiver side, the building block will terminate via the noAck pin, indicating to the enclosing building block that the token may not have reached the receiver (yet).

![Figure 9.8: Behaviour of the Notify R building block](image)

The behaviour of Notify R is abstracted by its ESM, as shown in Fig. 9.9. An important property is that the block always emits a token through ack or noAck before it terminates, even if tokens are lost between partitions.

### 9.4.2 Reliable Authenticate

Following our method from Fig. 1, we start our work on the access control system by looking at the collaborations it consists of, and begin with the one for authentication, as introduced in Fig. 9.3.
We run analysis A2 on it, with realistic transmission semantics. It finds that a deadlock occurs when the token from *pid* is lost in the channel between client and server, represented by the point where the topmost flow crosses the partition border. As a result, we invoke a syntactic substitution action on this flow that inserts an instance of *Notify R*, changes the target of the old flow to its *in* pin and adds a new flow going from *out* to the original target of the first flow (the *validate* operation). The substitution algorithm also connects *ack* to a flow final node and *noAck* to a newly inserted pin in *Authenticate* named *failed*. The result can be seen in Fig. 9.10.

The analysis tool will detect property violations introduced by the changes. For Fig. 9.10, the analysis finds a deadlock to happen in the event that the token returning from the server to the client is lost. In other words, the client is waiting for a response from the server before terminating, and we realize that the block must somehow detect if this could also be lost. Substituting the flow carrying the response for a new *Notify R* would only help in notifying the server side of the possibly lost token, which is not helpful as the server has no interest in what happens after it has sent its response. Instead, we connect the flow from the *ack* pin to a timer, which we merge with the flow going to the *failed* pin, as shown in Fig. 9.11. This way, *Authenticate R* always terminates, something which the analysis in step A2 confirms.
The authorize building block is modified in exactly the same way as the authenticate block, and the description is hence omitted. The blocks for door control and panel control are not further specified in this example, and are assumed to be infallible.

9.4.3 Reliable Inquiry

We notice a pattern in the final version of Authenticate R. The parts that we just added are useful for all instances of this request–response pattern. Hence, we create a new building block, Inquiry R, as shown in Fig. 9.12 to encapsulate this communication pattern for future reuse. The ESM of the block is depicted in Fig. 9.13.

This building block now uses two timers in total, one within n: Notify R waiting for the acknowledgement of the request, and one that waits for the response. One could say that the latter makes the first redundant since sending the response is of course an indirect acknowledgement for the initial request. However, when the
computation time on the server for producing the response is considerably longer than the transmission time for a signal (since it for example could include further communication with other components), the extra acknowledgement would allow the client to detect communication problems earlier. Anyway, if computation for the response would be very short, one could provide an extra version for Inquiry that uses the response as acknowledgement for the request.

9.4.4 Reliable Access Control System

Having created reliable versions of the authenticate and authorize blocks that reflect that messages may be lost, we now make the access control system reliable as well. We see a request–response pattern similar to that of Inquiry R between local station and central station. The difference here is that there are two alternative responses, out of which only one should arrive. This is easily added to our existing Inquiry R building block to produce the Inquiry 2 R block by simply adding a new pair of pins, responseIn2 and responseOut2, and a flow between them.

Using this new block, we obtain the version of the access control system shown in Fig. 9.14 which satisfies analysis A2. In particular, it is free from deadlocks. As the application already has logic to handle a negative authentication or authorization result (from the nok pins), we simply merge the outputs of the failed pins with these. The same is the case when a token is lost between the local station and the central station; the flow from failed is merged with the negative response. Note that this is a design decision that has to involve an expert on the application domain. The application specified under the ideal transmission semantics assumption does not, in the general case, contain the necessary information to algorithmically transform it into a reliable version. Specifying the best action to take upon detecting message loss hence requires manual intervention, just as the specification of the functionality itself.
9.5 Related Work

There are several other approaches that combine model-driven development with fault tolerance and techniques for fault removal, like model checking or testing.

Bucchiarone et al. [BMP07] present plans for an approach that utilizes techniques for both fault tolerance and fault removal to increase the dependability of systems. They specify functional and fault-tolerance requirements by UML use cases and validation scenarios by sequence diagrams. A system architecture is created in the form of UML component diagrams for structure and state machines for behaviour. Each component has one state machine describing normal behaviour and another one describing exceptional behaviour. They already have a tool-supported method, CHARMY capable of model checking and test case generation that they expect to be able to extend for fault-tolerant system architectures without major changes. While combining fault-tolerance mechanisms with model checking to uncover design faults is similar to what we are planning, our approach differs in that we generate the code directly from our specifications, instead of manually implementing the system and then testing it.

In [EKM07], Ernagan et al. specify services using interaction models (sequence diagrams) that are assumed to be complete. The authors propose to add detectors and mitigators to manage these services without altering the services themselves at all. Detectors observe the communication of the service and attempt to detect cases of unexpected behaviour (a message was sent when it should not have been) or of non occurrence behaviour (a message that should have been sent is not).
The idea of keeping the functional service specification separated from the fault-tolerance mechanisms that detect and handle errors seems very elegant, but we suspect it is somewhat restrictive in terms of what solutions for error handling can be employed. This will typically also require a complex platform that can, for example, completely transparently re-route messages between service roles, if mitigators are activated to help the system recover from a process crash.

Guelfi et al. present the DRIP Catalyst method in [Gue+04]. Here, coordinated atomic actions (CAAs, [Xu+95]) are used to specify all system behaviour. A CAA is represented by an activity diagram with each role in its own activity partition, similarly to the way we use activity diagrams to describe the collaboration of roles. The authors intend to follow the MDA approach [MM03] of refining a platform-independent model to a platform-specific model (PSM) and then generating code, but at the time of writing they create the PSM directly. Verification of the system behaviour is planned as future work. The main difference from our work is that this approach is built around the concept of CAAs, and the DRIP framework for expressing them in Java, so that all behaviour is specified as CAAs from the start. Hence both normal behaviour and fault-tolerance mechanisms are specified at once. We, instead, allow for a naive initial specification and then utilize tool-assisted analysis to help developers introduce fault-tolerance mechanisms in a following step.

Our approach of adding fault tolerance to a functional model bears similarities to aspect-oriented modelling where such cross-cutting concerns are also specified in separation as aspects and then weaved into the model. An aspect consists of a pointcut model that specifies a matching place to insert the advice model, which is the additional logic for handling the aspect.

Domokos and Majzik [DM05] look at how to incorporate dependability via aspect-oriented modelling. They operate at a purely architectural level, so that the behaviour of the system is not included. The method does, however, output an analysis model in the form of stochastic Petri nets, which can be used to determine the dependability properties of the system, i.e. the failure and repair processes of the system components and how errors propagate between them.

Both Fuentes and Sanchez [FS07] and Cui et al. [Cui+09] use activity diagrams to model system behaviour and aspect advice. The former operates on executable models, like our approach, so that design faults can be found before implementation. However, neither of them apply aspect orientation for fault tolerance, rather persistence, authorization and other aspects whose addition has little consequence for the application.

Kienzle and Guerraoui [KG02] argue that mechanisms for fault-tolerance (they use transactions) may not be suitable to be separated into an aspect of their own. As a reason they identify that the addition of fault-tolerance mechanisms, at least in their case, requires big changes to the application logic that cannot be made automatically. The experiment described is, however, conducted in the context of aspect-oriented programming, not at a modelling level. The applicability to the
works in the previous paragraph could hence be questioned.

There are also approaches that incorporate fault tolerance already at the requirements stage of the development process.

Berlizev and Guelfi [BG09] use UML activity diagrams to detail the behaviour of UML use cases that are again used to express system requirements. In addition to the system functionality, they also incorporate fault tolerance into the requirements, specifying both deviations from normal behaviour and recovery strategies.

Mustafiz et al. [MKB08] present a way to express degraded service outcomes, as well as exceptional modes of operation during the requirements engineering phase of development. They elaborate use cases with UML activity diagrams and mark the possible outcomes of an activity by stereotypes ≪success≫, ≪degraded≫ or ≪failure≫ in order to clearly distinguish them. The authors also advocate the explicit specification of exceptional operation modes that may result from degraded service outcomes. This allows for adjusting the expectations and behaviour of the system users to match the current situation.

Although the syntax of the two papers above is somewhat similar to ours, the semantics are not. For example, Berlizev and Guelfi use activity partitions to separate between normal and abnormal behaviour, not to separate distributed collaboration roles that could be physically distributed, and hence subject to unreliable communication. As these works focus on requirements, they cover an earlier stage of the development process than ours; what must happen, not how.

Both Bucchiarone et al. [BMP07] and Mustafiz et al. [MKB08] structure their specifications similar to ideal fault-tolerant components [AL81], meaning that all behaviour at the interface of a component is specified for both normal and exceptional cases. Our building blocks also completely specify their interface behaviour, but we do not syntactically separate the exceptional from the normal behaviour. This is because there is (currently) no semantic difference between normal and exceptional behaviour; everything is alternative behaviour that is enabled under certain conditions.

There is some work on automating the process of adding fault tolerance to fault-intolerant systems. In [KA00], for example, Kulkarni and Arora automatically transform a fault-intolerant program into a fault-tolerant one. However, if all variables cannot be read and written in a single atomic step, which is the case for our asynchronously communicating components, their algorithms have exponential complexity, limiting their practicality.

To sum up, our work looks for a middle ground between the approaches that require developers to design for functionality and fault tolerance at once and the approaches that would add the fault-tolerance aspect automatically at the end. In our experience, fault-tolerance mechanisms can be encapsulated and reused in another application, but they still need a human developer to make the decisions on how to integrate them with that particular application. Combining this with tools
for analysis that keep developers from introducing design faults in the integration process, as well as tools for code generation, makes for a practical approach that stands apart from the alternatives that we are aware of.

9.6 Discussion

We have described an incremental, corrective approach that takes a potentially unreliable specification and incrementally introduces fault-tolerance mechanisms. For instance, during the development of the access control system, we developed reliable versions of Authenticate and Authorize. From their externals, they differ from their idealized counterparts just by the pin failed. Of course, once the reliable versions of these blocks exist, domain experts can directly refer to the reliable blocks in the first place. This reduces the effort in the second development phase, since fewer blocks have to be made reliable.

Building blocks can be analysed separately for their suitability in systems with realistic transmission semantics, since they are encapsulated by ESMs. Once fault-tolerance mechanisms are added, the external behaviour has to be extended in some cases, such as the additional pin failed for the authenticate block in Fig. 9.11. This additional pin is necessary since the transmission failure cannot be (guaranteed to be) handled within the authenticate block, but has to be propagated to the surrounding application, which must decide what should be done. This can trigger a kind of domino effect, where in the worst case the entire system specification has to be extended, starting at the innermost building blocks until the highest level of composition is reached. To prevent this effect, one could follow a specification style in which building blocks are equipped with failure pins by default, as a default hook for any exceptions related to communication errors.

The soundness of our approach can be demonstrated by the behaviour of the executable state machines that it produces. They expose the same behaviour as state machines that would have been designed manually. For a comparison, we consider the state machines for the central station’s component:

- The state machine for the central station of the unreliable system as specified in Sect. 9.2, for instance, has 1 initial state, 6 control states, 6 decisions and 14 transitions as shown in Fig. 9.15.

- The state machine for the central station of the reliable system as specified in Sect. 9.4 in comparison, consists of 1 initial state, 5 control states, 6 decisions and 40 transitions as can be seen in Fig. 9.16.

These numbers demonstrate how the additional mechanisms for fault tolerance considerably increase the complexity of the resulting logic. We must note that the number of transitions is also due to the automation within our approach. If we would have designed these state machines by hand, some transitions could be
saved by using composite states or transitions that are declared for a group of states. However, this does not change the intrinsic complexity of the problem: Due to the fault-tolerance mechanisms, we must also keep track of 5 timers, and the number of signals to receive by the central station has grown from 3 in the idealized central station to 7 for the reliable one.

The comparison of the activity models and the resulting state machines reveals another benefit of our method: On the state machine level, we are not able to reuse fault-tolerance mechanisms in the way we are on the level of activity diagrams. In state machines, elements belonging to one interaction partner are mixed together with elements responsible for other things. Furthermore, the collaborative nature of our building blocks allows developers to study fault-tolerance properties from a holistic viewpoint, where the behaviour of all participants is contained in a single
Another way to separate fault-tolerance mechanisms from application logic is to move the first into a middleware layer providing robust communication primitives to the application layer. However, no matter how sophisticated this middleware layer is, failure-free communication cannot be guaranteed (since somebody can still unplug the ethernet cable), and at some point the application has to be involved, as described by the end-to-end arguments of Saltzer et al. [SRC84]. Moreover, putting advanced mechanisms into the middleware layer makes it a critical asset that demands development and maintenance resources, especially once it should be provided for a wide range of software platforms and devices. We therefore explicitly propose in this paper a method of including these mechanisms into a model that does not require perfect channels from an implementation. This makes an adaptation to different platforms easier, since they only need to provide simple communication primitives. For this reason, we can quickly create code generators for different execution platforms, as the ones named in the introduction. This strong focus on models, however, does not rule out the possibility to have a middleware layer that offers some form of fault tolerance; while a noAck within Notify R is triggered in our case by a timeout in the model, it could also be triggered by an exception of a middleware layer. The actual integration with the application logic (namely by pin noAck) would be equivalent. Figure 9.17 shows how we would model an Authenticate R 2 like this. This illustrates the flexibility we have in implementing fault-tolerance mechanisms; we can use mechanisms from both
middleware and application-layer libraries as long as the interface behaviour of the mechanisms are described in the model so that the integration with the application logic can be analysed. Middleware will often have a performance advantage, whereas the libraries can provide better portability.

![Authenticate R 2](image)

Figure 9.17: An alternative reliable Authenticate building block using middleware exceptions

To estimate productivity gains from our modelling method based on building blocks, we compared it in [Ber09; Knu09] to manual programming. The numbers indicate that development time when building blocks are provided is only a fraction of that needed to manually program a system. These gains are of course only real when building blocks are actually reused among several systems. This, however, is very likely when we consider the high reuse proportions of building blocks observed within our case studies summarized in [KH09].

### 9.7 Concluding Remarks

For the example, we developed the simple Notify R collaboration that uses a timer to protect the access control system against deadlocks. In the absence of a reliable end-to-end transport protocol at lower layers, one may want to handle message resending or reordering within the models as well. This requires a more advanced version of Notify R, similar to the Alternating Bit Protocol [BSW69], but leaves the application models utilizing it unchanged.

The error handling presented for the example is rather simple, as a failure is simply reported back as any other negative result. For more complex error handling, one could find that the additional behaviour specified amounts to much more. Hence, we would look into separating the specification of this error handling behaviour from the functional one, so that the functionality of the application (in an ideal world) can still be easily understood. An exception handling mechanisms utilizing
9.7. Concluding Remarks

UML exceptions [Obj09] may facilitate this. Aspect-oriented modelling is another possibility. We may also modify the syntax of the error handling behaviour to improve the readability of the diagrams.

We currently ensure a finite state space of our specifications by limiting the number of messages in a channel. This way, we can use model checking to verify some properties of our specifications. Abdulla and Jonsson [AJ93] prove that even with unbounded channels, some verification problems are decidable. In [AJ96], they prove that model checking liveness properties [OL82] is not decidable. More recent work in [BS03] states that using probabilistic system models, this limitation can be overcome. If modelling unbounded channels should be desirable in the future, we will look into incorporating these techniques into our tools.

In the current approach, the error handling mechanisms are introduced manually, motivated by deadlocks identified in analysis A2. For example, in the Access Control System R shown in Fig. 9.14, we manually insert an Inquiry 2 R block to protect the communication between local station and central station from message loss. By utilizing our existing analysis tool, we see the possibility for automatically suggesting inserting these blocks where suitable, based on patterns identified in the state space obtained during model checking.

Further, we want to expand the scope of our method to also deal with unreliable processes that may crash and provide building blocks for detecting this, as well as for maintaining a consistent shared state. Also, in [GHK09] we developed an analogous approach in which unsecured system specifications are extended with security mechanisms in a separate development phase. In an ideal setting, this security enhancement of the system would just follow our development phase that introduces fault tolerance.

In the introduction, we started our argumentation by the observation that fault tolerance mechanisms can draw the attention away from the actual application logic. In the state machines as in Fig. 16, this is clearly the case: The original logic of an access control system is not immediately obvious from it. With our method, however, this state machine representation is generated automatically, and does not have to be understood by humans. Instead, systems are specified in the form shown in Fig. 14, where separate functions are represented by separate blocks, which can be studied in isolation, like the one in Fig. 11. So far, our approach has been tested on several academic examples. For its suitability for real-sized systems, we rely on the scalability of the underlying development method SPACE. Consequently, we have started a larger evaluation based on an industrial system, within the project ARCTIS V, supported by the Research Council of Norway.
9.8 Acknowledgements

The research presented in this paper is partially funded by the Research Council of Norway (projects ARCTIS V and ISIS).

References


Chapter 10

Paper 6

Contracts for Multi-instance UML Activities

By Vidar Slåtten and Peter Herrmann.


The original publication is available at link.springer.com via http://dx.doi.org/10.1007/978-3-642-21461-5_20
Contracts for Multi-instance UML Activities

Abstract. We present a novel way of encapsulating UML activities using interface contracts, which allows to verify functional properties that depend on the synchronization of parallel instances of software components. Encapsulated UML activities can be reused together with their verification results in SPACE, a model-driven engineering method for reactive systems. Such compositional verification significantly improves the scalability of the method. Employing a small example of a load balancing system, we explain the semantics of the contracts using the temporal logic TLA. Thereafter, we propose a more easily comprehensible graphical notation and clarify that the contracts are able to express the variants of multiplicity that we can encounter using UML activities. Finally, we give the results of verifying some properties of the example system using the TLC model checker.

10.1 Introduction

A key to efficient software engineering is the reuse of existing software components that ideally are independently developed and marketed as commercial off-the-shelf (COTS) products. To enable a seamless combination of the components, one needs meaningful descriptions of the component interfaces that specify properties to be kept by both the components themselves and their environments. Due to the tight interaction with their environment, this requirement holds in particular for reactive systems [HP85]. To support the reuse of software components for reactive, distributed software systems, we use collaborative building blocks described by multi-partition UML activities, each augmented with an interface contract in the form of a UML state machine [KH09, KSH07]. We call these contracts External State Machines (ESMs). The contracts not only enable reuse of problem solutions, but also make for a reuse of verification effort as the user can verify a composed system using only the contracts, which in turn have been verified to be correct abstractions of the underlying solutions. This compositional approach helps to reduce the complexity and state space of the system models significantly [KH09]. The ESMs also help another problem with reuse: It may not always be straightforward to look at a reusable activity and see what it does and how to compose it correctly with the rest of the system. As the ESM only describes behaviour visible to the outside of the block, it aids both these tasks.

ESMs constitute what Beugnard et al. [Beu+99] call synchronization contracts, meaning that they can specify the effect of interleaving operations on a component, not just sequential operations. However, up until now we have been limited to collaborations in which only one instance of each type participates, as the contracts could not support collaborations featuring multiple component instances of the same type. If, for instance, a client request may be routed to one of several servers, we could not express an interface behaviour that permits server S to receive the request only if none of the other servers have received it. In systems
that employ load balancing or fault-tolerance mechanisms, however, to specify and
guarantee this kind of behaviour is crucial. Thus, compared with the ESMs, we
need additional concepts and notation for behavioural interfaces.

Any extension of ESMs should ideally keep a key characteristic of SPACE [KSH09]:
The underlying formalism is hidden to the user. According to Rushby [Rus00], this
is a key quality of practical development methods. SPACE relies on automatic
model checking to verify system models. To mitigate the problem of state-space
explosion, we limit our scope to verifying properties dependent only on the control
flow of the system designs. While we could very well include data in the model, the
model checker would not be able to verify properties dependent on data for realistic
systems, as the state space would grow exponentially with every data element we
include. Nevertheless, as pointed out in [KSH07; KSH09], also the model checking
of just control flows is of great practical help, for single-instance activities.

The ESMs are basically Finite State Machines (FSMs) with special annotations of
their transitions. To model multiple entities in a suitable way, we use Extended
Finite State Machines (EFSMs) [CK93] instead, which allow to refer to the indexes
of instances in the form of auxiliary variables. The semantics of these Extended
ESMs (EESMs) is formalized using the Temporal Logic of Actions, TLA [Lam94].
To relieve the software engineer from too much formalism, we further present a more
compact graphical notation in the form of UML state machines where statements
closer to programming languages are used to describe variable operations.

The next section discusses related work on component contracts, particularly work
using UML. Our load balancing example system is presented in Sect. 10.3. In
Sect. 10.4, we formalize the EESM semantics for many-to-many activities in TLA,
and present the graphical notation. We give EESMs for the other types of activities,
one-to-one and one-to-many, in Sect. 10.5. Some results, in particular about the
effects on model checking, as well as future work is discussed in Sect. 10.6, where
we also conclude.

10.2 Related Work

There are several other works that define a formal semantics for UML activities
[Esh06; GM05; Sto05], but neither of them include contracts for use in hier-
archical activities. Eshuis [Esh06] explicitly argues to leave this concept out, as
any hierarchical activity can be represented as a flat one. However, this results in
a much bigger state space.

As Beugnard et al. [Beu+99] point out, we can only expect software compon-
ents to be reused for mission-critical systems if they come with clear instructions
on how to be correctly reused and what guarantees they give under those condi-
tions. UML has the concept of Protocol State Machines [OMG10] to describe the
legal communication on a port of a component. Mencl [Men04] identifies several
shortcomings of these, for example that they do not allow to express dependencies between events on a provided and required interface, nor nesting or interleaving method calls. To remedy this, he proposes Port State Machines, where method calls are split into atomic request and response events. These Port State Machines are restricted to pure control flow, as transition guards are not supported. Bauer and Hennicker \cite{BH09} describe their protocol style as a hybrid of control flow and data state. However, they also cannot express the dependency between provided and required interfaces, and they currently lack verification support for whether two components fit together.

The ESMs have similar properties to Port State Machines in that all interface events are atomic, i.e., an operation is split into a request and response event, to allow for expressing nesting and interleaving of operations. They also essentially combine provided and required interfaces in the same contract, hence allowing to express dependencies between them. The EESMs combine this with data state to allow for compact representations of parametrized components and collaborations.

Sanders et al. \cite{San+05} present what they call semantic interfaces of service components, both described as finite state machines. These interfaces support both finding complementary components and implementations of an interface, hence also compositional verification. While they provide external interfaces for each local component and then asynchronously couple these to other, possible remote, components, our activity contracts act as internal interfaces that can be synchronously coupled with other local behaviour in the fashion of activity diagrams.

Our approach differs from all the above in that it permits the encapsulation of both local components and distributed collaborations between components, described by single-partition or multi-partition activities respectively. Further, our extended interfaces allow to constrain behaviour based on the state of parallel component instances, giving a powerful abstraction of distributed collaborations.

\section*{10.3 A Load Balancing Client – Server Example}

In SPACE, the main units of composition are collaborative building blocks in the form of UML activities that can automatically be transformed to executable code \cite{KSH09}. A system specification consists of a special system activity as the outermost block. This system activity can contain any number of inner activities, referenced by call behaviour actions, as well as glue logic between them. Figure 10.1(a) shows the system activity for our example, a set of clients that can send requests via a Router m-n block to a set of servers. Thus, Router m-n is an inner block, its activity depicted in Fig. 10.1(b). Each activity partition is named in the upper left corner and the bracketed parameter behind the name, $a$ for client and $b$ for server, denotes the number of component instances of this type. While each client only sees one router, each server partition has $a/b$ instances of the router block, as denoted by parameters $(1, a/b)$ after its name and the shade around the
server side of the block. This is also illustrated in Fig. 10.1(c), where we see that each client component only has a single requester sub-component, whereas each server has two responders. Note that the structural view is completely redundant and only serves to illustrate the information in the activities and the EESMs introduced below. The diagonally striped area inside the server partition represents other components of the same type, i.e., other servers. This gives a visual indication that the Router m-n block, in addition to collaborating with clients, also collaborates with other server instances. Each server also makes use of an inner activity called Make Response, which turns requests into responses. We have omitted it in Fig. 10.1(c), as it is completely local.

It is the job of the Router m-n block in Fig. 10.1(b) to balance the load so that a single server does not have to serve requests from all clients at the same time. All requesters can send requests to all responders, as illustrated by the full mesh connectivity in Fig. 10.1(c). Each responder uses the Forward block to forward requests to other responders, if it is currently busy itself. In the structural view, the component of the Forward block is shown as a small nameless rectangle on each responder that can communicate with every other such component. It is important to note that the Router m-n activity encapsulates asynchronous communication between components, while the synchronous interaction between an outer and an inner activity takes place via pins linking flows of the two activities. For instance,

\footnote{They are really activity parameter nodes when seen from the inner activity itself, but
10.3. A Load Balancing Client – Server Example

the block Router m-n is linked with the system activity by the pins reqIn, reqOut, resIn and resOut. Here, reqIn is a start pin initiating an activity instance (really, the corresponding requester instance), whereas resOut is a stop pin terminating the instance. The remaining pins with black background are streaming pins for interacting with active instances.

The semantics of UML activities is similar to Petri-nets, where the state is encoded as tokens resting in token places and then moving along the directed edges to perform a state transition [OMG10]. In our approach, all behaviour takes place in run-to-completion steps [KH10]. That is, all token movements are triggered by either receptions of external events (for instance, tokens resting between partitions) or expiration of local timers, and the tokens move until such an event is needed to make progress again.

Initial nodes start the behaviour of each system-level partition instance. They are fired one by one, but we make the assumption that no token will enter a partition before its initial node has fired. The initial node of the server partition can fire at any time, releasing a token into the start pin of the Make Response block. In the client partition, the token emitted from the initial node will enter the Router m-n block via the reqIn pin. Afterwards, it will be forwarded to a server instance and leave the block via pin reqOut, to enter pin req of Make Response. The Make Response block will eventually emit a token via its res pin, and the server partition will choose one of the Router m-n instances to receive it via its resIn pin, as denoted by the select statement [KBH07]. A select statement takes some data carried by the token and uses it to select either among various instances of a local inner block or of remote partitions. The Router m-n block will eventually emit a token through its resOut pin in one of the client partitions, which will follow the edge to the timer on the client side, where the step will stop. Later, the timer will expire, causing it to emit the token so that the client can perform another request. In this example, the timer is simply an abstraction for whatever the client might be doing between receiving a response and sending a new request. The behaviour of the Router m-n block is described in Sect. 10.4.

When we compose a system by creating new activities and reusing existing ones, we want to be able to verify properties of it. Given that SPACE models have a formal semantics [KH07] [KH09], we can express them as temporal logic specifications and use a model checker to verify properties automatically. To mitigate the problem of the state space growing exponentially with every stateful element, each activity is abstracted by an External State Machine (ESM), which is a description of the possible ordering of events visible on the activity pins. This allows us to do compositional verification of the system specification: We first verify that the activity and its ESM are consistent, then we only use that ESM when verifying properties for a higher-level or system-level block. Note that verifying the consistency of an ESM and its activity cannot be done automatically for all blocks, as some will con-

___

are called pins when an activity is reused via a call behaviour action. We use pins to denote both, to keep it simple.
strain their control-flow behaviour according to data stored in the activity tokens. In this case, the model allows all possible behaviours, and the potentially false positives reported by a model checker (that is, error traces that cannot occur in the real system) can be inspected manually, reducing the verification task. A select statement is an example where data constrains the destination of a token.

Figure 10.2(a) shows the ESM of the local building block Make Response. As discussed in the introduction, the ESM notation has the same expressive power as a finite state machine or Mealy machine [Mea55], to be precise. The transition labels correspond to pins, and the slash character separates the transition trigger event, as seen from the perspective of the block, from the effect. Hence, `start/` means that the transition is triggered from outside by a token entering pin `start` and that no tokens will pass any other pins in the same step. The ESM shows that a response is not output until a request has been sent in, and that this will not happen in the same step. Once the non-empty state is reached, further requests may enter and responses may be emitted from the block. Just looking at the ESM, however, we cannot know exactly how many responses will be sent out, as there is no way of knowing if a `/res` event has caused a transition to the empty state, or if the ESM is still in the non-empty state. This is because there is no way to track the actual number of buffered requests. When verifying properties of a system, such information may sometimes be necessary. For example, if this block was used in a system that sends three requests, we would like to infer from the ESM that exactly three responses can be emitted back out.

10.4 Contracts for Multi-instance Activities

To support multi-instance activities, we extend the ESMs to include transition guards, variables, arithmetic and predicate logic. Hence, they are now formally

\[\text{queue} = 0\] \[\text{queue} > 0\] \[\text{queue} -= 1\] \[\text{queue} += 1\]
EFSMs [CK93]. This enables us to specify event constraints that relate to the state of parallel component instances. Moreover, this increases the general expressiveness, so that we are able to better handle the case of the Make Response block.

Figure 10.2(b) shows the EESM of the Make Response block. We have here added a variable, queue, that tracks the number of requests buffered. To constrain the behaviour based on the queue size, as well as update it, this EESM also contains transition guards in square brackets and variable operations in lined boxes.

Figure 10.1(b) shows the internal activity of the Router m-n block. A request enters through the reqIn pin of the requester partition and is forwarded to a responder partition. The select statement, along with the fact that there are n responder partitions, tells us that a requester expects to have a choice of responders to communicate with, when forwarding the token. When a token crosses a partition border, the corresponding activity step ends, as remote communication is asynchronous. When the token is received by the responder partition, it is passed on to an inner block, Forward. This block may emit the token directly through its out pin to be passed on through the reqOut pin of Router m-n, or it may forward the token to another responder, if this one is busy already serving a request.

When a response token is received via the resIn pin, it is duplicated in the fork node and passed both to pin done and the channel for the originating requester partition.

We now describe the EESM of block Router m-n using the language TLA+ of the temporal logic TLA [Lam94], as shown in Fig. 10.3. The TLA+ module starts by defining the module name on the first line. The extends keyword states that this module imports the modules Naturals, which defines various arithmetic operators on natural numbers, and MatrixSum, defining operators for summing matrices. The variables of the module are declared using the VARIABLES keyword, where req and res represent the requester and responder partitions respectively. Constants are declared by the CONSTANTS keyword. They are the parameters of the model. When creating the building block Router m-n, we do not know how it will be reused in an enclosing activity. Another developer may choose to put multiple instances in both, one or none of the enclosing partitions. So, we need constants for the number of requesters and responders per enclosing partition instance, as well as the number of enclosing partition instances on each side. Hence, the global number of requesters is really no_req * no_req_encl.

A TLA+ specification describes a labelled transition system. This is given as a set of states, a subset of the states that are initial states, a set of actions (labels) and a transition relation describing whether the system can make a transition from one state to another via a given action. The set of initial states is described by the Init construct. Here, the req and res variables are each given records for their corresponding pins (except pin resOut, see below), which in turn are functions in

---

3This behaviour is described by the EESM of the Forward block, which, due to space constraints, we do not show.
two dimensions stating whether a token has passed the pin for each requester or responder instance. That is, a requester or responder instance is identified by an enclosing instance number combined with an inner instance number.

Next in the TLA⁺ module follow the actions, which formally are predicates on pairs of states. Variables denoting the state before carrying out an action use their common identifiers while those referring to the state afterwards are given a prime. The action reqIn states that for a requester(e,i) identified by enclosing instance e and inner instance i, a token can enter pin reqIn only if the given instance has not yet had a token pass through this pin. The next conjunct of the reqIn action says that the values of the req variable will be the same as now, except that the counter for tokens having passed through pin reqIn will be set to 1 for requester(e,i). The UNCHANGED keyword states which variables are not changed by an action, as TLA⁺ requires all next-state variable values to be set explicitly.

The reqOut action also represents that a token is only allowed through pin reqOut of responder(e,i) if it has not already had a token pass through. The second line constrains this further by stating that a token passing event can only happen if the sum of all tokens having passed any reqIn pin is greater than the sum of all tokens having passed any reqOut pin. Hence, this event on responder(e,i) is constrained by the state of parallel components. Action resIn states that a token may only enter a responder(e,i) via pin resIn if the same instance has already emitted a token via pin reqOut, but not already sent a response through pin resIn. The resOut action states that a requester(e,i) can only emit a token through pin resOut if it
has received a token through pin reqIn. This is constrained further by requiring there to be a responder\((f, k)\), that has received a token through its resIn pin. All counters belonging to requester\((e, i)\) and responder\((f, k)\) are then set to 0, to reset their state. As the resOut action also performs the reset, there is no TLA\(^+\) variable for its corresponding pin.

This behaviour is easier seen looking at the graphical notation in Fig. 10.4. Here, the style of the transition operations is closer to programming languages like Java. The number of partition instances is denoted |partition name|. We omit the domain of \(\exists\) statements where this is obvious from the context, and the keyword ALL denotes all indexes in a domain. We also omit specifying which partition a pin belongs to if there is only one pin by that name in the activity. The transition from the initial node to the active state represents the Init construct in TLA\(^+\), and the remaining transitions represent the actions.

Since we do not model that a token keeps the index of its requester instance as data while located at a server, we cannot fully automatically verify that the activity and EESM for Router m-n are consistent. The model checker finds counterexamples where a response is simply sent to a requester that has not yet issued a request, instead of to a requester that has. What we can verify automatically, however, is that whenever a token is sent back to a requester, the EESM is in a state where the token would be allowed through the resOut pin of at least one of them.

Once a building block is complete, we can reuse it like we have reused Router m-n in our system example from Fig. 10.1(a). To verify properties of the system, we generate the TLA\(^+\) module in Fig. 10.5 (see [KSH07]). This module instantiates other modules, namely Router m-n and Make Response. The constants no_clients and no_servers represent the parameters \(a\) and \(b\) from the system activity. We express the actions of the system activity as a composition of constraints and operations on the variables of the system activity, and actions of ESMs of the inner activities. Hence, the Init construct not only sets the timer in all client instances to 0 and all initial nodes to 1, but also calls the Init construct of Router m-n and Make Response as shown by \(r!Init\) and \(m!Init\). Note also that since this is a system
EXTENDS Naturals, MatrixSum

MODULE load_sharing_system

VARIABLES r_req, r_res, m_state, m_queue, client, server

CONSTANTS no_clients, no_servers

no_res \triangleq no_clients \div no_servers

r \triangleq \text{INSTANCE} \text{router}_m \text{with} no_req \leftarrow 1, no\_req\_encl \leftarrow no\_clients,
no_res \leftarrow no\_res, no\_res\_encl \leftarrow no\_servers, req \leftarrow r\_req, res \leftarrow r\_res

m \triangleq \text{INSTANCE} \text{make\_response} \text{with} no\_make\_response \leftarrow 1, no\_enclosing \leftarrow no\_servers,
state \leftarrow m\_state, queue \leftarrow m\_queue

Init \triangleq \text{client} = [\text{timer} \mapsto [i \in 1 \ldots \text{no\_clients} \mapsto 0], \text{initial} \mapsto [i \in 1 \ldots \text{no\_clients} \mapsto 1]]
\land \text{server} = [\text{initial} \mapsto [i \in 1 \ldots \text{no\_servers} \mapsto 1]] \land r!\text{Init} \land m!\text{Init}

start\_client(p) \triangleq \text{client}.initial[p] = 1 \land \text{client}' = [\text{client} \text{EXCEPT} !.\text{initial}[p] = 0]
\land r!\text{reqIn}(p, 1) \land \text{UNCHANGED} \langle \text{server}, m\_state, m\_queue \rangle

start\_server(p) \triangleq \text{server}.initial[p] = 1 \land \text{server}' = [\text{server} \text{EXCEPT} !.\text{initial}[p] = 0]
\land m!\text{start}(p, 1) \land \text{UNCHANGED} \langle \text{client}, r\_req, r\_res \rangle

r\_reqOut\_m\_req(p, i) \triangleq r!\text{reqOut}(p, i) \land m!\text{req}(p, 1) \land \text{UNCHANGED} \langle \text{client}, \text{server} \rangle

m\_res\_r\_resIn(p, i) \triangleq m!\text{res}(p, 1) \land r!\text{resIn}(p, i) \land \text{UNCHANGED} \langle \text{client}, \text{server} \rangle

r\_resOut\_client\_timer(p) \triangleq r!\text{resOut}(p, 1) \land \text{client.}\text{timer}[p] = 0
\land \text{client}' = [\text{client} \text{EXCEPT} !.\text{timer}[p] = 1] \land \text{UNCHANGED} \langle \text{server}, m\_state, m\_queue \rangle

client\_timer\_r\_reqIn(p) \triangleq \text{client.}\text{timer}[p] = 1 \land \text{client}' = [\text{client} \text{EXCEPT} !.\text{timer}[p] = 0]
\land r!\text{reqIn}(p, 1) \land \text{UNCHANGED} \langle \text{server}, m\_state, m\_queue \rangle

Next \triangleq
\forall p \in 1 \ldots \text{no\_clients} : \text{start}\_client(p)
\lor p \in 1 \ldots \text{no\_servers} : \text{start}\_server(p)
\lor p \in 1 \ldots \text{no\_servers}, i \in 1 \ldots \text{no\_res} : r\_reqOut\_m\_req(p, i)
\lor p \in 1 \ldots \text{no\_servers}, i \in 1 \ldots \text{no\_res} : m\_res\_r\_resIn(p, i)
\lor p \in 1 \ldots \text{no\_clients} : r\_resOut\_client\_timer(p)
\lor p \in 1 \ldots \text{no\_clients} : \text{client}\_timer\_r\_reqIn(p)

Spec \triangleq \text{Init} \land \Box[\text{Next}] [\text{r\_req, r\_res, m\_state, m\_queue, client, server}]

P1 \triangleq \Box(\forall p \in 1 \ldots \text{no\_servers} : m\_queue[p, 1] \leq \text{no\_res})
P2 \triangleq \Box(\text{Sum}(r\_res\_reqOut, \text{no\_servers}, \text{no\_res}) \leq \text{Sum}(r\_req\_reqIn, \text{no\_clients}, 1))
P3 \triangleq \Box(\forall p \in 1 \ldots \text{no\_servers}, i \in 1 \ldots \text{no\_res} : 
\langle \text{server}\_\text{initial}[p] = 0 \land \text{ENABLED} r!\text{reqOut}(p, i) \Rightarrow \text{ENABLED} m!\text{req}(p, 1) \rangle)
P4 \triangleq \Box(\forall p \in 1 \ldots \text{no\_servers} : \text{ENABLED} m!\text{res}(p, 1) \Rightarrow 
\exists i \in 1 \ldots \text{no\_res} : \text{ENABLED} r!\text{resIn}(p, i))
P5 \triangleq \Box(\forall p \in 1 \ldots \text{no\_clients} : \text{ENABLED} r!\text{resOut}(p, 1) \Rightarrow \text{client.}\text{timer}[p] = 0)

Figure 10.5: TLA\textsuperscript{+} module for the system activity
activity, we do not need to add an extra dimension for enclosing partition instances when identifying activity elements like the timer, as it cannot be reused in other activities. For a description of each action, we refer back to Sect. 10.3.

The whole system specification is written as a single formula $\text{Spec} \triangleq \text{Init} \land \Box [\text{Next}]_{(\text{vars})}$. This formula states that the transition system has initial states(s) as specified by $\text{Init}$ and that every change to the variables listed in $\text{vars}$ is done via one of the actions listed in the next-state relation, $\text{Next}$. The box-shaped symbol ($\Box$) in front of $[\text{Next}]$ is the temporal logic operator always. It means that what follows must be true for every reachable state of the system model.

The small example system of this paper is chosen to allow us to show the formal semantics of the EESMs in TLA$^+$ and clarify that they are unambiguous, yet expressive enough for our needs. Therefore, the properties that we can verify for this system might seem rather trivial, but for more complex systems, variations of these properties may be very hard to verify without a formal model. The properties we want to verify are written formally below the horizontal bar in Fig. 10.5. All the properties can be verified by model checking. See Sect. 10.6 for further discussion of the results.

**P1** The number of requests queued in any Make Response block is at most equal to the number of responders per server, i.e., the inner queue is finite.

**P2** There are at most as many ongoing requests on the server side as there are on the client side.

**P3** Whenever a server is started and a responder instance of that server is ready to emit a token through the reqOut pin, the Make Response instance of that server is ready to accept a token through its req pin.

**P4** Whenever a token can be emitted from the Make Response block of a server, at least one of the responder instances on that server is able to accept it.

**P5** Whenever a token can be emitted via the resOut pin of a requester instance, the corresponding timer is empty, hence ready to receive a token.

### 10.5 Other Types of Multiplicity

Our formalism for expressing contracts of multi-instance activities also works for one-to-one building blocks without any internal select statements, like Router 1-1 shown in Fig. 10.4(a). This is a special case, where each requester instance is statically mapped to a responder instance and vice versa. As each binary collaboration cannot have any constraints on its behaviour in terms of the state of parallel instances, we can simplify the EESM as shown in Fig. 10.7 without loss of information. This is, in fact, the same notation that we have been using already for ESMs of activities with one instance of each type [KSH09], only augmented with an index $i$. The difference is that the formal semantics now supports multiple
Figure 10.6: The two other variants of the router activity, with respect to select statements

instances globally, instead of requiring such a block to be used in a system with only one instance of each enclosing partition as well.

Figure 10.7: UML notation for the EESM of Router 1-1

Finally, we present a one-to-many variation of the router block, Router 1-n, where a requester is statically mapped to a set of responders, as shown in Fig. 10.6(b). This could be used in a setting where each server from Fig. 10.1(a) has one responder instance per client, so that each client has a choice of any server when issuing a request. When the response is to be routed back, the corresponding requester is already given, due to the static mapping.

The EESM of Router 1-n is shown in Fig. 10.8. Due to the mapping between requester and responder instances, the notation is a bit more complex than for the other variants. For example, the /reqOut(i) transition states that a token may only leave the reqOut pin through instance i if this has not happened already. The rest of the guard constrains this further by stating that a token passing can only occur if the corresponding requester instance has gotten more tokens through its reqIn pin than the sum of tokens having already passed through pin reqOut in all the responders mapping to that requester. The expression reqIn[responder[i]] means the value of the reqIn variable for the requester who can be found by mapping responder[i] to its corresponding requester. Hence, Σ reqOut[requester[responder[i]][k]] means the sum of reqOut values for the k different responders found by mapping responder[i] to its requester and then mapping that requester to the set of corresponding responders.

Note that it is the EESM that holds the information on whether there is a con-

4In addition, there could naturally be a mirror version of the Router 1-n activity, a Router n-1, but this is also a one-to-many activity.
10.6 Concluding Remarks

Table 10.1: Number of states found and time required to verify properties P1–P5

<table>
<thead>
<tr>
<th># of servers</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td># of clients</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>7 states, &lt; 1 sec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>37 states, 1 sec</td>
<td>70 states, 1 sec</td>
<td>707 states, 3 sec</td>
</tr>
<tr>
<td>3</td>
<td>241 states, 2 sec</td>
<td>3410 states, 5 sec</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1713 states, 4 sec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>12617 states, 9 sec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>94513 states, 48 sec</td>
<td>188962 states, 99 sec</td>
<td>283411 states, 155 sec</td>
</tr>
<tr>
<td>7</td>
<td>715001 states, 651 sec</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

strained static mapping or not. In contrast, the EESM of Router m-n, given in Fig. 10.4, has no references to any particular parallel instance or set of instances, only to the current instance and the keyword ALL.

10.6 Concluding Remarks

All the properties from Fig. 10.5 have been verified by the TLC model checker [YML99], for the parameter values shown in Table 10.1. Model checking only verifies properties for a model with some exact parameters. It does not say whether those properties will still hold for different parameters. However, if the model changes behaviour with respect to a property for some specific parameter values, it is often when a parameter is changed from 1 to >1, or it is likely due to highly intentional design decisions. Hence, the fundamental problem remains, but it is not always that great in practice.

5We are aware that the TLA specification for the given example can be optimized by only storing the aggregate number of tokens having passed through a pin on any of the responders in a server. However, this optimization would not work if the EESM required two tokens to pass pin reqOut before a token is allowed though pin resIn.
Given that model checking is automatic, one could say that time is not an issue, as we can just leave a computer at it and check for up to, for example, a thousand instances of each partition. However, as Table 10.1 shows, the time needed grows exponentially as we increase the number of client instances. The linear increase from server instances comes from the fact that more servers reduce the number of responders per server.

There is a high level of parallelism in our system example. This is also the case for other systems using EESMs that we have verified. Hence, we expect partial order reduction \cite{HP95} to alleviate the state-space blowup from increasing the number of instances. We therefore plan to also formalize our semantics in Promela, so we can use the Spin \cite{Hol03} model checker, which implements partial order reduction. The formalisms are compatible, as there is already work for transforming another TLA derivative, cTLA, into Promela automatically \cite{RPK04}. For relatively simple blocks, where the contract must be verified for any number of instances, the TLA formalism allows for writing manual refinement proofs as well \cite{Lam96}.

We already have a tool for generating TLA\textsuperscript{+} from SPACE models \cite{KSH07}. This tool greatly reduces the time required to specify systems, and it automatically generates many types of general properties to ensure the well-formedness of SPACE models. We will extend the tool to work with EESMs, outputting Promela specifications as well. To hide the formalism when specifying application-specific properties, there is work in progress to express them in UML.

To verify properties like “Every request is eventually responded to”, would require adding data to identify each request and adding liveness constraints to the model. Being based on TLA, the formalism can accommodate this quite easily in the form of weak or strong fairness assumptions. The limiting factor is still time needed for model checking.

Having formalized extended ESMs, we are eager to use them in the setting of fault-tolerant systems, where multiple instances of the same type often collaborate to mask failures, and conditions such as a majority of the instances being reachable are often essential to precisely describe the behaviour of a block.

To conclude, contracts encapsulate software components and facilitate both reuse and compositional verification. The SPACE method uses collaborations detailed by UML activities as the unit of reuse. We introduce EESMs, which allow to describe the global behaviour of multi-instance activities, abstracting away internal state, while still having the expressive power to detail when an external event can take place. An example from the load balancing Router m-n block is that a request will only arrive at a server that has free capacity in the form of free responder instances, and only if the number of requests received from all clients is greater than the number of requests forwarded to any server. While the EESMs have a formal semantics in TLA, we give graphical UML state machines as specification tools, so that software engineers themselves need not be experts in temporal logic.
References


[HP95] Gerard J. Holzmann and Doron Peled. “An improvement in formal verification”. In: *Proc. 7th IFIP WG6.1 Int. Conf. on Formal Description Techniques*. 1995, pp. 197–211.


Chapter 11

Paper 7

Towards Automatic Generation of Formal Specifications to Validate and Verify Reliable Distributed Systems: A Method Exemplified by an Industrial Case Study

By Vidar Slåtten, Frank Alexander Kraemer and Peter Herrmann.

Presented at the 10th International Conference on Generative Programming and Component Engineering (GPCE 2011), October 22 – 23, 2011, Portland, USA.


The original publication is available at dl.acm.org via http://doi.acm.org/10.1145/2047862.2047888

Note: In this paper, we refer to a built-in model checker that has been made, in addition to the previously described transformation to TLA$^+$ to be input to the TLC model checker. The built-in model checker supports the analysis with ESMs, so that we here only use TLA$^+$ and TLC for the more advanced verification tasks. Also, there has been a change in the semantics with respect to ESMs and EESMs in that tokens arriving at inner blocks when the (E)ESM is not ready are just ignored, and are hence reported as warnings rather than errors that need to be fixed.
Towards Automatic Generation of Formal Specifications to Validate and Verify Reliable Distributed Systems: A Method Exemplified by an Industrial Case Study

Abstract. The validation and verification of reliable systems is a difficult and complex task, mainly for two reasons: First, it is difficult to precisely state which formal properties a system needs to fulfil to be of high quality. Second, it is complex to automatically verify such properties, due to the size of the analysis state space which grows exponentially with the number of components. We tackle these problems by a tool-supported method which embeds application functionality in building blocks that use UML activities to describe their internal behaviour. To describe their externally visible behaviour, we use a combination of complementary interface contracts, so-called ESMs and EESMs. In this paper, we present an extension of the interface contracts, External Reliability Contracts (ERCs), that capture failure behaviour. This separation of different behavioural aspects in separate descriptions facilitates a two-step analysis, in which the first step is completely automated and the second step is facilitated by an automatic translation of the models to the input syntax of the model checker TLC. Further, the cascade of contracts is used to separate the work of domain and reliability experts. The concepts are proposed with the background of a real industry case, and we demonstrate how the use of interface contracts leads to significantly smaller state spaces in the analysis.

11.1 Introduction

Since nearly half a century ago, theoretical computer scientists have developed a plethora of techniques to model and to verify software in a formal way. In spite of several outstanding results, however, formal methods are still not used that much in practise. A likely reason is the complexity of many methods which tend not only to be laborious but also require a considerable amount of expertise. Hence, to make the application of formal techniques more popular in software development, they must be much simpler and faster to use. An approach to achieve this is through model-driven engineering, for instance on the basis of UML or SDL. These languages can be effectively used to describe software in such a way that implementations can be automatically generated from them. Further, formal methods may be used to analyze them, as they often give an appropriate level of abstraction. In this way, models can be used as front-ends for formal tools, which leads to what Rushby \cite{Rus00} calls “disappearing formal methods.”

Our method SPACE and its tool Arctis \cite{Kra08,KSH09b} are designed with this strategy in mind and optimized for the development of reactive, distributed applications. System behaviour is modelled by UML activities that due to their
Paper 7

Token semantics similar to Petri-nets [KH10; Obj10] can be easily understood. The activities have been provided with a reactive formal semantics [KH10] such that both automatic code generation [Kra08; KSH09b] and formal analysis [KSH09b] are possible. Moreover, this modelling technique is scalable since activities can be composed using UML call behavior actions, which we refer to as blocks. A block may both embed an activity and be a part of another one. The interaction between the two activities is modelled by UML pins through which tokens flow when transferring from one activity to the other. The behaviour at this interface is modelled by contracts in the form of so-called External State Machines (ESMs, [KH09]) and an extended version of them (EESMs, [SH11]). This allows for storing blocks in libraries and re-using them in different software models (see [KH09]).

Arctis facilitates the formal analysis of system models for important, generally desirable software properties by using a model checker [KSH09b]. Since the model checker does its analysis by an exhaustive search of the reachable system states, it works fully automatically. Further, error traces can be animated on the UML activities such that the Arctis users do not need a deeper understanding of the formalism laying behind the analysis.

A disadvantage of model checking is that models of realistic systems often comprise too many system states to be checked in an acceptable amount of time. This is due to the combinatorial blow-up in the number of states when combining component behaviours, known as the state explosion problem. In Arctis, we mitigate this problem by compositional verification (see [KSH09b]). In particular, we use the interface contracts to reduce the system verification to a number of local verification runs each concerning only a single activity as well as a number of (E)ESMs. Thus, the number of states to be model checked grows linearly with the number of the activities in the system model instead of increasing exponentially.

However, while this strategy works to ensure that each building block is well-formed from a local point of view, for some systems we also need to check application-specific system properties. This holds especially for properties with respect to reliability, i.e., to check how a system reacts in the presence of communication and process failures. The need to consider the system with a larger scope and the presence of failures increases the complexity (and hence, the state space) drastically, simply because of the higher degree of concurrency. Furthermore, mechanisms to improve reliability often employ multiple instances of a given type for redundancy (see, for instance [KSH09a]). Here, one has to take also data that distinguishes the individual instances into account, which further escalates the state space to be checked.

This paper is devoted to demonstrating how compositional verification can mitigate the state explosion problem for reliable systems. To achieve that, we validate the concept of EESMs [SH11] by showing how they are applied in an industrial case study where we were contracted to add a fault-tolerant best-effort mutual exclusion protocol to an existing system of intelligent clothes lockers for hospitals (see Sect. 11.2). In Sect. 11.3 we describe our development method for reliable
systems, extended from earlier work considering just message loss \cite{SKH10}, and the specifications created for the case study. In addition, we show how the activities and contracts are transformed into TLA+ \cite{Lam02}, the input language of the TLC model checker \cite{YML99}. In contrast to proofs of system functionality, not all theorems specifying the reliability properties to be verified can be automatically generated at the moment. Instead, we demonstrate the flexibility of the framework by showing how the best-effort mutual exclusion property can be expressed by an expert (see Sect. \ref{sec:bes}). Furthermore, in Sect. \ref{sec:ext} we introduce a novel extension we call External Reliability Contracts (ERCs) as a way to handle the effects of process crashes and message loss in a compositional manner, while still being able to verify larger system sizes under failure-free semantics. We show that our variant of compositional verification gives significant savings over monolithic verification and discuss our findings, in Sect. \ref{sec:imp} We survey some related work in Sect. \ref{sec:rel} and conclude in Sect. \ref{sec:conc}.

\section{Texi Case Study: Lockers that Read RFID}

Texi AS is a company that delivers RFID-based logistics systems to organize work wear, typically for hospitals. All clothes have a small RFID tag sewn into the fabric, which is used to track the clothes during the entire usage cycle. Our case study focuses on the lockers in the hospital that store the clothes and make them available to the hospital staff. The lockers are equipped with antennas that can read the RFID tags, so that they know which clothes are stored in them.

When employees want new clothes, they swipe their employee card through a card reader. The locker then checks if the employee is allowed access to the locker. If the employee has access, the door is unlocked. After the clothes are removed and the door is closed again, the reading process is started to see which clothes have been removed by the employee.

A typical installation in a hospital has many lockers that stand next to each other, and the reading process of the antennas takes several seconds. For that reason it is likely that several employees access closely located lockers simultaneously. This can lead to wrong reading results, since the antennas may interfere once two or more lockers read at the same time.

For this reason, we introduced a solution that delays the reading within lockers so that only one locker may read at a time. Obviously, this can be achieved by introducing a central controller that takes the role of coordinator among all lockers that are in danger of interference. A locker then has to obtain permission before it may activate its antennas. Such a solution, however, introduces several single points of failure. If the central controller goes down, or if a locker does not release its read permission, all further reading requests from other lockers will not be answered and employees will not be able to get their work wear. For that reason, we had to develop a more robust solution, where a locker can carry on alone, if
other parts of the system fail.

11.2.1 System Requirements

According to Texi, the requirements for the improved system are the following: If possible, only one locker shall read its contents at a time. However, availability should still have priority over this mutual exclusion property, since the possible inconsistencies due to concurrent reading can be manually corrected, but a blocked locker would hinder hospital work. Hence, we cannot use a mutual exclusion algorithm that blocks if a locker is unable to contact the central controller. Instead we create a protocol that attempts to provide mutual exclusion of locker reads, but does not necessarily provide it in the presence of process crashes and message loss.

While it is easy to express a mutual exclusion property, “no two lockers should have permission to read their contents at the same time,” it is not straightforward to express the kind of best-effort mutual exclusion property (in the following called BEME property) that our customer requests. For our protocol, the BEME property can be expressed as a mutual exclusion property that is conditional on the absence of message loss and process crashes. We note that message loss is indistinguishable from a long delay. Thus, such events can be explicitly modelled in the form of timeout events. The same goes for process crashes, which we can model as a transition to a state where no further events can take place in the crashed component of the system, except a restart event. With this in place, we might simply state the BEME property to be “as long as no timeout or crash has occurred, the mutual exclusion property must hold.” However, this does not say anything about whether the system will recover from a faulty state and go back to providing the mutual exclusion property once a sufficiently long time has passed without further crashes or timeouts taking place. To include this, we can express the BEME property as “if there is a time after which no further timeouts or crashes occur, then there will eventually be a time where the mutual exclusion property holds forever.” In addition to the BEME property, the customer naturally wants the system to be free from any deadlock scenarios and that all read requests are eventually granted.

Next, we will introduce our method and the specification of the case study before we revisit the BEME property in Sect. 11.3.2.

11.3 The Method for Reliable Systems

As explained above, the main specification element of the SPACE method are building blocks, expressed by UML activities and encapsulated by formal contracts. Within the method, we distinguish three levels of descriptions for the external
contracts of a building block that have different significance with respect to the overall development workflow:

- **External State Machines (ESMs, [KH09])** are UML state machines that describe the order in which parameter pins on the building blocks may be used. ESMs do not consider any data or sessions (multiple instances of a block), which in many cases is sufficient to explain how blocks are to be assembled correctly to more comprehensive applications. The complexity of the analysis is also limited, so that checking whether the blocks are correctly integrated can be performed in the background of the editing process, without interrupting the user.

- **Extended ESMs (EESM, [SH11])** are an extension of ESMs that adds actions and guards on data variables. This is especially useful for mechanisms that increase the reliability of systems, since they often require multiple instances of the same block, i.e., sessions. Since session IDs are data, EESMs may capture relations of several sessions, or simply count how often a certain action has been executed. Thus, EESMs are more expressive than ESMs, but this comes at the cost that the verification with EESMs is more complex. This analysis is therefore carried out in an extra step with temporal logic as the basis (see Sect. 11.3.2).

- **External Reliability Contracts (ERCs, introduced in this paper)** add yet another layer to the contracts. They amend (E)ESMs by describing behaviour that results from communication and process failures. In principle, this behaviour could be directly expressed within the (E)ESMs. However, we have observed that this behaviour is often orthogonal to the original, purely functional behaviour described by (E)ESMs. It can therefore be expressed separately, much like an aspect in aspect-oriented programming. This also has the benefit that systems may be analyzed both with and without failures, as discussed later.

To make our method practical, we also take into account the level of expertise that is needed to fulfil a certain task. Therefore, we identify two groups of engineers:

- **Domain experts**, who are familiar with a certain application domain and relevant technologies, such as for example RFID and embedded systems. Domain experts have programming skills and may also model on the level of UML activities, but do not need the ability to formulate and verify temporal theorems, for instance.

- **Reliability experts**, who are familiar with reliability problems and possible solutions, and who are also familiar with the necessary formal methods to assure system quality with respect to reliability. To optimally utilize their expertise, we assume that reliability experts are hired on a case-by-case basis. Therefore the method should be optimized to keep their overhead regarding non-reliability related questions low, as we will discuss in the following.
Within a verification project \[\text{ArcV}\], in which we apply our tool to industrial cases such as the one presented, we see that this categorization is quite to the point. With respect to these roles, we expect our method to be used in the following way, as illustrated in Fig. 11.1:

1. Domain experts create the main part of the application by composing building blocks. We have measured that up to 71% of blocks may be reused from libraries (see \[\text{KH09}\]). This may already include some building blocks provided by reliability experts, if their necessity is obvious or if the domain expert already knows these blocks from previous projects. For the domain expert it is often sufficient to look at the (simple) ESMs to integrate building blocks correctly into a system.

2. The basic analysis (\(A_1\)) is executed in the background, so that the resulting specification \(S\) is at least well-formed and describes consistent compositions of the building blocks, as far as the ESMs are concerned. This eases the job of the reliability expert, since many inconsistencies and ambiguities are already removed.

3. The consistent model \(S\) is handed over to a reliability expert, who performs an in-depth analysis \(A_2\) using the model checker TLC \[\text{YML99}\]. TLA+ \[\text{Lam02}\], the input language of TLC, is generated automatically from the UML model, now also including the variables, actions and guards contributed by EESMs. Some systems, such as the one presented in our case study, require also application-specific properties for which the reliability expert formulates the corresponding theorems, as explained in Sect. 11.3.2.
4. To verify properties also with realistic assumptions including faulty channels and crashing processes, the ERCs of the building blocks are taken into account, or, where necessary, introduced.

The results of $A_2$ may require that additional fault-tolerance mechanisms are introduced or given functionality is changed. Depending on the extent of the changes, these are either done by the reliability expert alone, or in cooperation with the domain expert. These decisions may also require feedback from the customer, when consequences of failures and remedies have to be taken into account. In our case study, for instance, Texi had to make the trade-off between data consistency and locker availability.

5. From a consistent specification, executable code can be generated by a model transformation from the UML activities to state machines and a subsequent code generation step. This step is completely automated, and we verified that the code generation preserves system constraints [Kra08]. It is hence guaranteed that this code also fulfils the properties of the original specification.

Needless to say, this method is an idealized workflow that serves as an orientation rather than an inflexible corset. In practise, the separation between the experts may be less distinct than described. In addition, since building blocks may be checked back into the library, complete solutions that already take reliability concerns into account may be directly applied by domain experts. In [KSH09a], for instance, we developed a robust leader election protocol. Although the protocol is not trivial, the resulting building block is so easy to handle that it can be integrated by a domain expert without any trouble.

### 11.3.1 System Design in SPACE

In the following, we focus on the parts of the system that deal with mutual exclusion between lockers, and do not further detail other functionality. We assume that steps 1 and 2 have already been carried out, and focus now on the work of the reliability expert.

Figure 11.2 shows the UML activity for the Timed Mutex Distributed building block that we created to implement the mutual exclusion protocol. Partitions represent separate components of the system that are physically distributed. They are named in the upper left corner, in this case *controller* and *locker*. The fact that there can be more than one locker is represented by $[n]$ after the name of the locker partition. To distinguish the different lockers, each of them has a specific ID. In the implementation, we have chosen the IP address of each locker as its ID. The single controller partition contains a building block *Timed Mutex Local*, which implements the locally concentrated part of the protocol.

UML activities have a semantics based on token flows. For the building blocks, we use a reactive variant of them [KH10], in which tokens flow in run-to-completion
steps (so-called activity steps), each of which are triggered by an observable event, either the expiration of a timer or the reception of a signal.

Each locker partition of the building block can receive a request for the read permission via the starting pin named request, through which a token enters the activity, travels along the edge to the fork node and is duplicated. One duplicate enters the Timer block via pin start. The other passes through operation getMyID, which retrieves the ID of the locker, and comes to rest at the partition border between the locker and the controller partition. All tokens rest between partitions, as message passing is asynchronous, meaning that the sending and receiving of a message are two different events.

Two activity steps are now enabled: One step is triggered by the token on the topmost edge arriving at its destination and entering the block Timed Mutex Local via pin request(ID). Another possibility is for the timer to emit a token. To see why this can happen, we must consider the external contract of the Timer block.

The ESM of block Timer is shown in Fig. 11.3. It shows that once a token has passed via the start pin, the block may spontaneously emit a token via its expired pin and will also accept tokens via the pins stop and reset. Therefore, a possible next event is that the timer expires, releasing a token through the expired pin, via the merge node and on through the grant pin of the locker partition. Hence, this timer ensures that no locker is blocked from reading its contents by the controller crashing or a communications failure.
### 11.3. The Method for Reliable Systems

Figure 11.4 shows the EESM of the Timed Mutex Local block, and thus also the behaviour we can expect from it. EESMs are different from ESMs in that their state does not just consist of the control state like `active`, but also the values assigned to any variables they have (i.e., they are *extended* finite state machines [CK93]). Due to the extra variables, EESMs are initialized implicitly, as shown by the transition from the initial state, at the very left. That is, the initial transition is executed together with the startup of the component (modelled by a top-level partition) the building block is part of. In the case of Timed Mutex Local, the EESM tracks two sets, the IDs that have been sent through pin `request` and the same for pin `grant`. An EESM also takes into account that a block can be instantiated as *multi-session*, which means that several instances of a block execute concurrently. Therefore, every variable of the EESM is an array in which each block instance has its own index, \(i\). Note that in Fig. 11.2 only one instance of the Timed Mutex Local block is instantiated, so \(i\) is always 1.

The EESM only allows one request from each ID for each block instance at a time. To express this, the transition labelled `request(i, ID)` has a guard stating that for a `request(i, ID)` event to happen, that ID must not already be in the set `requests[i]`. Further, the transition has an operation, written in a lined rectangle, that specifies that the new ID is added to the set represented by `requests[i]`. A `wait(i, ID)` transition can only take place if the block has already received a request from that ID, but not yet sent out a grant. A `grant(i, ID)` transition has the exact same guard as `wait(i, ID)`, but has an additional operation to update the `grants` variable. The `release(i, ID)` transition is only enabled when ID has been granted. This transition resets the contract with respect to that ID by removing it from both the requests and grants set, allowing new requests with that ID.

Looking back at Fig. 11.2, we see that a token released from the `wait(ID)` pin of block `tml`, denoted `tml:wait(ID)`, will need to rest at the partition border before being received by the locker it is destined for. As there are several lockers, a select statement [KBH07] is used to only send the message to the locker with the address given in the ID parameter of the token. Upon arrival at the locker, the token originating from `tml:wait(ID)` will enter the `reset` pin of the Timer block, hence...
delaying the expiration of the timer. If the timer has already expired when this happens, the semantics of the contract is that the token is discarded as it attempts to enter the block via a pin that the contract, in its current state, does not allow tokens to pass. During our analysis, such a scenario raises a warning so that the developer can make sure it is intentional.

The events following a token being released via \texttt{tml:grant(ID)} are similar to the above, only here, the token is duplicated upon arrival at the locker to both stop the timer and pass through the \textit{grant} pin of the Timed Mutex Distributed block itself. The EESM of Timed Mutex Distributed, shown in Fig. 11.5, does not permit more than one grant without a release and request in-between, for that locker instance. If a grant has already happened, the token will be discarded when trying to leave the block via the grant pin, just like a token trying to enter a block at the wrong time. The EESM also tells us that a token can enter the Timed Mutex Distributed block via pin \textit{release} if that locker instance has received a grant. This will cause the token to be sent towards the controller component and received in a later step to enter \texttt{tml:release(ID)}, resetting its EESM with respect to that ID.

The activity of the Timed Mutex Local block is shown in Fig. 11.6. The protocol to ensure mutual exclusion is implemented by a combination of two blocks: The block of type Mutex ensures that only one locker is given read permission at a time and keeps track of the lockers that have requested permission. The Wait block is taking care of the notification towards each individual locker. It is instantiated as a multi-session block, which means that it is executed with several instances, one for each locker ID, signified by the additional shadow around it and the parameter \((n)\). This session pattern simplifies the modelling of concurrent behaviour, since each session instance only has to keep track of the protocol state of a single locker.

Due to space constraints, we do not show the contracts of Mutex and Wait, but give an informal overview of the behaviour of Timed Mutex Local: The first request is granted right away, whereas subsequent requests are queued at the Mutex block

\footnote{Note that this filtering effect only applies when the contract can decide if a transition is enabled based solely on local information. An EESM transition with a guard that refers to a remote variable (e.g., another component instance being in a specific state) cannot filter out tokens, hence any violations are real errors.}

\footnote{When an EESM does not use any data apart from session IDs and transitions do not reference other sessions, we present it in a simpler way that looks like an ESM except for the additional index \((i)\) on every transition label [SH11].}
11.3. The Method for Reliable Systems

while the corresponding Wait block instances periodically send out tokens via their *keepWaiting* pins. This period is shorter than the duration of the Timer block in Fig. 11.2 to prevent its expiration under non-failure conditions. Once a previously granted locker sends a release, or the Wait block instance for this locker times out, the next ID in the Mutex queue is used to tell its Wait block instance to grant read permission. Hence, timeouts from the Wait block instances ensure that the protocol can continue even if the release of a read permission is never received.

As there is more than one instance of the Wait block, we have to use select statements when communicating with them. More importantly, since each Wait block instance is implemented as its own state machine, the run-time-support system treats messaging between the parent state machine (the controller) and the children state machines (the Wait block instances) in an asynchronous manner, but with the FIFO property.

11.3.2 Analysis $A_2$ of Timed Mutex Local

To verify more detailed properties of our specifications than analysis $A_1$ supports, we use a formalism based on temporal logic, the Temporal Logic of Actions (TLA, [Lam02]). We can generate TLA+, the language for TLA, automatically from the Arctis models [KSH09b], although not all features introduced in this paper are yet supported by the implementation.
Figure 11.7 shows an excerpt of the TLA+ specification of the EESM, the activity and the consistency proof for Timed Mutex Local, focusing on the events related to a token passing through the request(ID) and wait(ID) pins. Each run-to-completion step of the activity, and each EESM transition, is represented as one TLA+ action. The TLA+ actions for the EESM transitions are quite similar to the graphical representation. The main difference is that updates of variables are written in a different style and that variables that are not changed in a transition are explicitly marked as such. The activity part of Fig. 11.7 refers to contracts of inner blocks by (block name)!((E)ESM transition) like m!request_wait(1, ID). It also uses functions sendToWait((pin name), ID) and receiveFromWait((pin name), ID), which are both functions we have defined to asynchronously send and receive tokens with the given ID via the named pins. From the part of the specification headed by “From the proof”, we see that the events of the EESM and the activity are connected so as to take place together in one atomic step. For example in the action requestEvent(ID), whenever a request(i, ID) takes place for the EESM, either a request_m_request m_wait m_wait(ID) or request_m_request m_immGrant m_wait m_immGrant(ID) action must take place in the activity at the same time. In contrast, as modeled by waitEvent(ID), when the activity wants to send a token through the wait pin, we may accept that the EESM does not allow it, due to the semantics of discarding tokens that attempt to travel via pins at the wrong time. This is expressed with the if-then-else construct.

Theorems that verify the consistency of the activity and its (E)ESM can be generated automatically. For example, to verify that a token accepted through pin request by the EESM is also accepted by the activity, we use the model checker TLC [YML99] to check the invariant p1 from Fig. 11.7. The invariant states that whenever the EESM is ready to allow a token through the request pin (i.e., the corresponding TLA+ action is enabled), one of the corresponding transitions of the activity are also enabled. Invariant p2 states the same thing for events where tokens pass through the outgoing wait pin. The difference is that any violation of this invariant is interpreted as just a warning, not something that necessarily has to be corrected. Instead, the developer should make sure that any scenarios where the activity can send a token through the pin, but the EESM does not allow it, are intentional. The scenarios are given automatically by the model checker in the form of an error trace that we can visualize in the Arctis models [KSH09d].

The design of Timed Mutex Local shown in Fig. 11.6 has a bug that can lead to a deadlock situation. When running TLC, it returns an error trace to show that the design allows the following scenario:

1. A request from locker L1 is received, granted and released, but the release

---

3In TLA and TLA+, an action is a predicate on a pair of system states, modelling the changes to the variables that are carried out in a system step.
4The index parameter of the Mutex block is hard coded as 1, since there is only one instance being used in the activity. All EESMs still have an index, so we can use the same TLA+ segment to express them regardless of how many instances are actually used.
module Excerpt_for_request_and_wait

From the EESM
request(i, ID) ≡
∧ ID ∉ requests[i]
∧ requests[i] = [requests EXCEPT ![i] = requests[i] ∪ {ID}]
∧ UNCHANGED (grants)

wait(i, ID) ≡
∧ ID ∈ requests[i]
∧ ID ∉ grants[i]
∧ UNCHANGED (grants, requests)

From the activity
request_m_request_m_wait_w_wait(ID) ≡
∧ m! request_wait(1, ID)
∧ sendToWait(waitPin, ID)
∧ UNCHANGED (w.state, fromWait)

request_m_request_m_immGrant_w_immGrant(ID) ≡
∧ m! request_immediateGrant(1, ID)
∧ sendToWait(immediateGrantPin, ID)
∧ UNCHANGED (w.state, fromWait)

w_keepWaiting_wait(ID) ≡
∧ receiveFromWait(keepWaitingPin, ID)
∧ UNCHANGED (w.state, m_queue, toWait)

From the proof
requestEvent(ID) ≡
∧ eesm! request(1, ID)
∧ ∨ act! request_m_request_m_wait_w_wait(ID)
∧ ∨ act! request_m_request_m_immGrant_w_immGrant(ID)

waitEvent(ID) ≡
∧ act! w_keepWaiting_wait(ID)
∧ IF enabled eesm! wait(1, ID)
    THEN enabled eesm! wait(1, ID)
    ELSE UNCHANGED (requests, grants)

p1 ≡ □!(∀ ID ∈ IDs :
    enabled eesm! request(1, ID) ⇒
    (enabled act! request_m_request_m_wait_w_wait(ID)
     ∨ enabled act! request_m_request_m_immGrant_w_immGrant(ID)))

p2 ≡ □!(∀ ID ∈ IDs :
    enabled act! w_keepWaiting_wait(ID) ⇒
    enabled eesm! wait(1, ID))

Figure 11.7: TLA+ excerpt for the proof of the Timed Mutex Local block
only gets as far as being duplicated by the fork node so that one token reaches \( m:release(ID) \) to remove the request entry, while the other token has not yet been received by the Wait block through its \( \text{release} \) pin.

2. The Wait block for locker L1 emits a token through its timeout pin, which rests in the asynchronous channels from the Wait blocks to the controller.

3. Another request from locker L1 is received, an entry is added in the Mutex block and a token put in the queue to the corresponding Wait block via pin \( \text{immediateGrant} \).

4. The timeout from the L1 instance of Wait is finally received by the controller and reaches \( m:release(ID) \), which removes the new request from locker L1 instead of the original request.

This causes an inconsistency between the state of the Wait blocks and the state of Mutex that later on can deadlock the system. The chance of this sequence of events actually happening is very small, especially as the delay from the local, but asynchronous message passing between the Wait blocks and the rest of the controller partition is expected to be much shorter than the non-local message passing. However, these are the kind of subtle faults that could take down a system after years of operation and be very difficult to pinpoint when trying to figure out the reason for the failure.

One could solve the problem by changing the run-time-support system so that local messages always have priority over non-local ones, or we can insert a First block that ensure that either only the release or the timeout for a certain ID reaches Mutex, as shown in Fig. 11.8. The nice thing about this new version, is that the EESM is exactly the same as before, so the replacement does not trigger a need to redo any of the verification done already. Using the EESM of the Timed Mutex Local block, we just need to verify consistency of the EESM and activity for this new block.

Although we primarily want to prove the BEME property for the Timed Mutex Distributed block, we can also express it for the Timed Mutex Local block. This “BEME Local” property will then be “if there are ever no more timeouts, then eventually the block will grant at most one read request at a time”. This is a liveness property, meaning that it describes something that should happen. In our method, the theorems for these properties are formulated manually. To verify liveness properties, we need to add constraints to filter out behaviours that are unreasonable for a real system, such as a dice that is rolled infinitely many times yet never shows a “six”. Typically this means constraining the behaviours to those where things that can always or infinitely often happen, do actually happen sometimes, known as fairness constraints [Lam02]. To verify the BEME Local property, we also add a liveness constraint to the specification stating that eventually there will never be any more timeouts from any of the Wait block sessions, hence TLC only considers the part of the state space where this holds. We can then express
the BEME local property in TLA$^+$ as $\square (\text{Cardinality}(\text{grants}[1]) \leq 1)$, which reads “eventually always the cardinality of the set of granted IDs is at most one.”

Although advanced properties like this one, at least currently, have to be written manually, the reliability expert only has to add two lines to a TLA$^+$ specification of perhaps hundreds of lines to get TLC to check it. This shows that the automation provided by our method can also be very useful for the cases where some things have to be done manually.

We do not have the space to go in detail also about the verification of the Timed Mutex Distributed block. The only truly new property compared to the Timed Mutex Local block, is that we would like to verify that every request received from a locker is eventually responded to via a grant. Again, we will have to add fairness constraints to filter out unrealistic behaviours, and then we can express the property as “always, for all lockers, a request from a given locker has been received implies that eventually a grant is sent to that locker.”

11.4 Modelling Realistic Semantics

Our interface contracts help developers to understand the behaviour of a block and facilitate compositional verification by describing only the events that are visible to the outside of the block. In order to do compositional verification under realistic
semantics, the contracts must therefore describe the externally visible effects of
process crashes and message loss, too. This could be done by writing a new ESM
or EESM that includes extra transitions caused by these types of events, with
the drawback of having to maintain two, partially overlapping, contracts, one for
ideal semantics and another one for realistic semantics. To avoid this potential for
inconsistency, we use an aspect-oriented notation to express what we call External
Reliability Contracts (ERCs).

The ERC for the Timed Mutex Local block is shown in Fig. 11.9. In the follow-
ing, we use the terminology of aspect-oriented programming, AspectJ in particular
[Asp11]. The part of the ERC in black is the pointcut, the pattern that must
match in order to insert the advice given in blue colour. The first transition has a
pointcut that looks for any transition that starts in an initial state and ends in a
state called active. Looking at the EESM in Fig. 11.4, we see that there is only one
such transition. Hence, the ERC adds an extra variable up (an array, in case there
is more than one instance of the block) to the EESM to denote the state of the
process running the controller partition. The ERC transition to the right of this
matches any state named active and adds a new transition with target state active
so that every controller instance that is not crashed, i.e., up[i] = 1, can crash. The
following transition matches any EESM transition that has state active as both its
target and source. It adds an additional guard stating that the controller must be
up in order for such an EESM transition to take place.

As shown in the top right corner, the ERC adds a transition from state active with
up equal to 0 that restarts the controller. A transition with this guard tells us
whether the existing state of the EESM is remembered after a crash or not. There
is no persistent storage for the state of this block, so both the set of requests and
grants are reset to the empty set upon restarting the component. Exchanging the
Timed Mutex Local block and its ERC for an otherwise identical block that does
use persistent storage, could alter the results of verification with realistic semantics. All local blocks have an ERC with at least these four transitions.

The timeout of a Wait block instance is not explicitly expressed in the EESM of Timed Mutex Local, as it does not trigger any tokens to traverse any external pins. However, the application-specific BEME property of Timed Mutex Distributed is conditional in timeouts eventually not happening anymore, so we want to export this event to the enclosing level through the ERC. As a result we include two more ERC transitions that are specific to exporting the timeout event, placed below the first row of transitions. More precisely, they export the activity step where a token arrives at the controller from $w:timeout(ID)$ and reaches $m:release(ID)$, i.e., the event that can cause a new locker to be granted read permission before the previous holder has released it. The first transition adds a timeout variable to the contract, which is initialized to 0. The second transition states that a timeout for a given ID may happen when the component is not crashed and the ID is an element of the set of grants. It also removes the ID of the timed out Wait block instance from the contract sets, to allow new requests even if the release message is lost. Once the timeout event is explicit in the contract like this, we can use it to verify the BEME property for the Timed Mutex Distributed block.

Note that while the EESM is constructed independently from the ERC, the opposite is not true. The ERC is tailored to the existing EESM, and may have to change if the EESM changes.

We skip showing the ERC of Timed Mutex Distributed. Since each locker has a timer that effectively masks the effects of message loss or the controller crashing, the ERC is as simple as the first row of transitions from the ERC in Fig. 11.9. Not surprisingly, we find this pattern in other blocks as well: Blocks that are built to tolerate failures under realistic semantics have almost the same contract under ideal semantics; the ERC only adds that tokens cannot pass through pins of partitions that have crashed.

In principle, one can match different ERC transitions to the same (E)ESM transition, so that the application of one ERC transition un-matches others. Because of this, we currently demand that any ERC is expressed so as not to have conflicting aspect transitions, meaning that they can be applied in any order and get the same result. We also do not apply ERC transitions to (E)ESM transitions created by other ERC transitions, hence any such transitions must be self-contained, like the last transition in Fig. 11.9 that already includes “up[i] == 1” since this will not be added by the third ERC transition.

As seen in the next section, realistic semantics greatly increase the size of the state space. Having ERCs as aspects to (E)ESMs means that we always have access to

---

5We can also verify the BEME property compositionally under ideal semantics, by first verifying that Timed Mutex Local only gives two or more grants at the same time if a timeout has happened, and then we can refer to whether the set of grants has had more than one element, instead of referring directly to whether a timeout has happened.
a less complex specification with ideal semantics as well, easily trading details in
the behaviour for verifying with more component instances when needed.

11.5 Discussion

To evaluate the usefulness of the EESMs and ERCs in terms of enabling composi-
tional verification, we observed model checking runs of different specifications. Our
hypothesis is that compositional verification will greatly reduce the verification ef-
fort, both when considering only ideal semantics (EESMs) and when checking with
full realistic semantics (EESMs + ERCs). Table 11.1 gives the number of states
found and the time it took to search through them for several variants of the Timed
Mutex Distributed block and the Timed Mutex Local block. The rows of the table
are as follows:

TML gives the results for model checking the Timed Mutex Local block from
Fig. 11.6 compositionally. That is, using the contracts of the inner blocks,
Mutex and Wait, to abstract them.

TMD gives the results for model checking Timed Mutex Distributed from Fig. 11.2
when abstracting Timed Mutex Local by its contract.

TMD Comp. gives the aggregate results for compositional verification of Timed
Mutex Distributed. This row is simply the sum of the results from the two
rows above it. We use these number as a base line to compare the other
numbers to.

TMD Direct represents monolithic verification and gives the results for model
checking the Timed Mutex Distributed block and the Timed Mutex Local
block at the same time. Here, we keep the EESM of Timed Mutex Local to
filter out tokens traversing the boundaries of the block. This is to demon-
strate the effect of analysing both parts at once, instead of separately. The
numbers with unit \( x \) give how many times larger the number of states or
seconds is compared to the base line of TMD Comp.

For one locker, TMD Direct has just over double the state space of TMD
Comp., and for two lockers the monolithic verification needs 89 times the
state space and takes 32 times as long as compositional verification does. For
three lockers, the difference is even greater: More than 282 times the state
space and more than 576 times the time is used for monolithic verification.
All numbers in the table prefixed by “>” are just an indication of the lower
bound, as we terminated the model checking run at that point. Note that
monolithic verification here only refers to removing one layer of abstraction,
the Timed Mutex Local block, not replacing all blocks in Timed Mutex
Distributed by their inner contents.

\(^6\)We use the variant of Timed Mutex Local with the First block as that allows to search
the whole state space without encountering a deadlock.
### 11.5. Discussion

<table>
<thead>
<tr>
<th>Alternative</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number of lockers</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Ideal Semantics</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TML states</td>
<td>20</td>
<td>1 441</td>
<td>125 648</td>
</tr>
<tr>
<td>TML time</td>
<td>0 sec</td>
<td>4 sec</td>
<td>56 sec</td>
</tr>
<tr>
<td>TMD states</td>
<td>54</td>
<td>2 961</td>
<td>157 464</td>
</tr>
<tr>
<td>TMD time</td>
<td>0 sec</td>
<td>3 sec</td>
<td>44 sec</td>
</tr>
<tr>
<td>TMD Comp. states</td>
<td>74</td>
<td>4 402</td>
<td>283 112</td>
</tr>
<tr>
<td>TMD Comp. time</td>
<td>0 sec</td>
<td>7 sec</td>
<td>100 sec</td>
</tr>
<tr>
<td>TMD Direct states, x TMD Comp</td>
<td>166</td>
<td>388 018</td>
<td>&gt; 80 M</td>
</tr>
<tr>
<td>TMD Direct states, x TMD Comp time</td>
<td>3 sec</td>
<td>222 sec</td>
<td>&gt; 16 hours</td>
</tr>
<tr>
<td>TMDD NC states</td>
<td>8 746</td>
<td>&gt; 193 M</td>
<td>-</td>
</tr>
<tr>
<td>TMDD NC states, x TMD Comp</td>
<td>118 x</td>
<td>&gt; 44 296 x</td>
<td>-</td>
</tr>
<tr>
<td>TMDD NC time, x TMD Comp</td>
<td>4 sec</td>
<td>&gt; 29 hours</td>
<td>-</td>
</tr>
<tr>
<td><strong>Realistic Semantics</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TML states</td>
<td>40</td>
<td>5 636</td>
<td>500 408</td>
</tr>
<tr>
<td>TML time</td>
<td>1 sec</td>
<td>5 sec</td>
<td>359 sec</td>
</tr>
<tr>
<td>TMD states</td>
<td>1 152</td>
<td>331 776</td>
<td>95 551 488</td>
</tr>
<tr>
<td>TMD time</td>
<td>3 sec</td>
<td>98 sec</td>
<td>64 101 sec</td>
</tr>
<tr>
<td>TMD Comp. states</td>
<td>1 192</td>
<td>337 412</td>
<td>96 051 896</td>
</tr>
<tr>
<td>TMD Comp. time</td>
<td>4 sec</td>
<td>103 sec</td>
<td>64 460 sec</td>
</tr>
<tr>
<td>TMD Direct states, x TMD Comp</td>
<td>1 920</td>
<td>25 970 688</td>
<td>-</td>
</tr>
<tr>
<td>TMD Direct states, x TMD Comp time</td>
<td>1 6 x</td>
<td>77 x</td>
<td>-</td>
</tr>
<tr>
<td>TMD Direct time, x TMD Comp</td>
<td>4 sec</td>
<td>27 320 sec</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 11.1: Number of states and time to find them using TLC

**TMDD NC** shows the results when attempting to model check the Timed Mutex Distributed block without any contract between it and the contents of the Timed Mutex Local block, i.e., simply adding the contents from the Timed Mutex Local block from Fig. 11.8 into Fig. 11.2. In this case, we do not have the contract of Timed Mutex Local to filter out behaviour between the part that came from Timed Mutex Local and the part from Timed Mutex Distributed. A comparison with the compositional case is thus not correct from the point of view of verification: Such a block not only needs to be verified in one run, but also passes more tokens between its parts, increasing the number of possible behaviours in each part. It is simply not the same
specification. Nevertheless, we include it by the row TMDD NC (No Contract) to show the practical result of attempting to build the Timed Mutex Distributed block in this way. The results show that we can only model check the specification for one locker within reasonable time.

Considering realistic semantics means considering a much bigger state space, hence compositional verification is necessary even for small models. To do compositional verification in this case, we incorporate the ERCs into the contracts. The results of the model checking are as follows:

**TML** When analysing under realistic semantics, the state space of Timed Mutex Local almost quadruple for more than one locker. This is as expected since there are two new Boolean variables, \( up \) and \( timeout \), to keep track of. We only track timeouts that allow another locker to get the read permission, hence this variable is never changed for the case with only one locker.

**TMD** It gets more complex for the Timed Muted Distributed block, as it contains message channels that can drop messages. Since it has more than one partition type and can have more than one instance of the locker partition, there are many combinations of crashes possible. Together with message loss, this greatly increases the state space under realistic semantics.

**TMD Direct** Just like with ideal semantics, we see an increase in the state space when model checking with both parts of the system at once with realistic semantics. The difference is that the numbers for the compositional verification are already much higher under realistic semantics, so the exponential blowup due to monolithic verification has a much greater effect in practise. This is especially so for the time taken to search the state space, as TLC tends to search fewer states per second for larger state spaces.

As we can see, the number of lockers contributes to the state space in an exponential manner, when not using any state space reduction techniques.\(^7\) Hence, it is important to keep the starting numbers low, so that we can verify the specifications for a large enough number of lockers that we gain confidence in their correctness.\(^8\) This is where the benefit of our compositional approach comes in: By analysing tightly coupled parts of the behaviour while abstracting the rest, we can avoid the exponential growth in the state space that stems from analysing too many parts of the behaviour at once. This in turn, allows us to reach further with respect to the number of instances of the same type the model checker can handle. To sum up, we see that compositional verification performs significantly better than monolithic...

---

\(^7\)We present data from unoptimized specifications, as the main point is to show the relationship between numbers of different rows.

\(^8\)Model checking cannot prove properties of general models, only the model instances that are actually checked. Hence, there could be a number of lockers for which the properties do not hold. The best we can do is to check with a few instances for which most bugs will manifest themselves.
verification with contracts, and that trying to build the system without contracts most likely would lead to a state space that is infeasible to model check at all with multiple lockers.

11.6 Related Work

The idea of compositional verification of temporal logic specifications, whether it is by manual proofs [HK00] or model checking [CLM89], is not a new one. There are also several approaches that automate the compositional verification process. However, it is not trivial how a system is decomposed. Cobleigh et al. [CAC06] use the L* learning algorithm coupled with a model checker for automatic assume–guarantee reasoning [CGP03] about the properties of systems. They report that “the vast majority” of the 2-way decompositions found for each example system actually did not improve on monolithic verification. In fact, their results were not very promising: Only about half of the examples studied could be improved by assume–guarantee reasoning, and even in these cases the gains were mostly limited to expanding the model by one instance. However, later studies on the topic report better results [Pas+08; Che+10], although the improvements over monolithic verification are seldom by more than factor 4. So why are our experimental results so different? Although these works and ours both deal with compositional verification, they are not directly comparable: First of all, we report on a single, albeit real, system. This is not enough to say anything precise about the performance. Further, these works find a new assumption for each property to verify, while we use a static contract for all properties. While they are looking for an assumption that perfectly abstracts the other part of the system for a given property, we take advantage of the fact that we control the resulting implementation and carry any extra constraints from the contracts into the actual implementation. Also, our development method naturally leads to tightly coupled clusters of behavioural logic with looser coupling between them, due to the inherent goal of creating reusable building blocks.

We use UML activities to model software components. There are other works giving UML activities a formal semantics [Esh06; GM05; Sto05], but these all omit contracts to enabled hierarchical activities. As pointed out in [Beu+99], we can only expect software components to be reused for critical systems if they come with clear instructions on how to be correctly reused and what guarantees they give under those conditions. UML already provides the concept of Protocol State Machines [Obj10] to detail how a component can communicate with its environment. Mencl [Men04] proposes Port State Machines to improve on several shortcomings of Protocol State Machines, for example that they do no allow nesting or interleaving method calls, nor dependencies between a provided and required interface. His Port State Machines split method calls into atomic request and response events to allow for nesting and interleaving method calls, but they are restricted to pure control flow, as transition guards are not supported. Bauer and Hennicker
BH09 introduce a protocol description that is a hybrid of control flow and data state styles. However, this approach also lacks the ability to express dependencies between required and provided interfaces.

Like Port State Machines, our contracts have atomic interface events to allow for the expression of nesting and interleaving method calls. As they abstract both the block and its environment, they also express the provided and required interfaces in the same structure, hence allowing to express dependencies between them. In addition, our EESMs combine this with data variables so that we can more accurately express the behaviour of blocks with many instances SH11 or blocks whose behaviour is otherwise strongly data dependent.

Sanders et al. San+05 present semantic interfaces of service components, using finite state machine notation to describe both. Semantic interfaces can be used to find both complementary and implementing components, hence they support compositional verification. The main difference from our contracts is that semantic interfaces abstract local components that are asynchronously connected to remote or local components, while our contracts are mainly used internally in one process to connect sub-components, described as activity diagrams, synchronously together. The fact that our contracts allow encapsulation of both local components and distributed collaborations between components, sets them apart from all the above.

11.7 Concluding Remarks

While we currently either analyse a specification under completely ideal or realistic semantics, we see an advantage in having more fine-grained control over the execution semantics of individual system parts. This could be achieved by extending our method and tool with a deployment model where one could easily alter which components and channels should have ideal or realistic semantics, or even the ordering properties of a channel. For the scenario above, such a tool would enable faster analysis for one failure source at a time, but not reveal any problems caused by a combination of failure sources.

All ERCs written for this case study have been made manually. However, we could automate most of their construction for local blocks, which all have the three first transitions from Fig. 11.9 in common. If we enable developers to tag elements of the existing model with a ≪persistent≫ stereotype, we should be able to automatically generate the restart transition as well.

ERCs are used to export reliability-related events that are not directly visible as tokens passing through pins. It may be that other non-functional properties of our models can be described in the same manner, adding them as aspects to (E)ESMs. For example, there is work to analyse security aspects of SPACE models GKH11.

9As seen from the Wait blocks in Fig. 11.6 asynchronous coupling is also supported.
and we can imagine a use for a concept like this to export security aspects from inside a block to a higher level.

If we are to apply several aspect-oriented contracts to each base contract, perhaps even created by different people, the problem of conflicting aspects transitions is likely to increase. In such a case, we might need to develop a conflict resolution mechanism to ensure that there is no ambiguity in the result of the aspect weaving.

In summary, we have shown that encapsulation using our contracts can reduce the state space to verify by at least factor 100 compared to monolithic verification. We have introduced ERCs to allow compositional verification also under realistic semantics. ERCs allow to easily switch between analysis under ideal or realistic semantics. Since the state space under realistic semantics is larger than under ideal semantics, this allows trading realistic behaviour descriptions for larger model sizes, when convenient.

References


References


Part III

Appendices
Appendix A

Secondary Papers
A.1 Paper 8

Model-Driven Engineering of Dependable Systems

By Vidar Slåtten.


The original publication is available at ieeeexplore.ieee.org via http://dx.doi.org/10.1109/ICST.2010.49

Abstract. Improving the dependability of a computer system increases the acquisition cost so much that many systems are built without a cost-effective level of dependability. This motivates our decision to work on reducing the development effort and competence required to create dependable, distributed, reactive systems. The scope is narrowed to extending the SPACE method with software-implemented fault-tolerance mechanisms and providing tool-supported fault removal in the form of model checking. The results so far mainly cover fault removal, but we also have some early results on providing fault-tolerance mechanisms at the application layer. We discuss future work as well.
Abstract. Developing complex distributed systems is a non-trivial task. It is even more difficult when the systems need to dynamically reconfigure the distributed functionalities or tasks. Not only do we need to deal with the application-specific functionalities that are intricate, but we also have to handle the complex logic of coordinating the distribution and relocation of tasks. In this paper, we model an intrusion detection system that distributes its analysis units to a number of hosts and assigns fine-grained analysis tasks to these hosts in order to cope with the rapid increase of audit data from today’s IT systems. The system is further capable to react to overload situations and to shift tasks to other hosts. To develop this complex system, we apply the model-based engineering method SPACE. In particular, we show that the collaborative specification style of the method can significantly reduce the development effort. Also, the formal semantics of SPACE ensures the correctness of important design properties.