INTRODUCING MODE SWITCH IN COMPONENT-BASED SOFTWARE DEVELOPMENT

Hang Yin

2015

School of Innovation, Design and Engineering
INTRODUCING MODE SWITCH IN COMPONENT-BASED SOFTWARE DEVELOPMENT

Hang Yin

Akademisk avhandling

som för avläggande av teknologie doktorsexamen i datavetenskap vid
Akademin för innovation, design och teknik kommer att offentligen försvaras
torsdagen den 29 oktober 2015, 14.15 i Beta, Mälardalens högskola, Västerås.

Fakultetsopponent: Professor Carlo Ghezzi, Politecnico di Milano, Milano, Italy
Abstract

Self-adaptivity, characterized by the ability to dynamically adjust behavior at runtime, is a growing trend in the evolution of modern embedded systems. While self-adaptive systems tend to be flexible and autonomous, self-adaptivity may inevitably complicate software design, test, and analysis. A strategy for taming the growing software complexity of self-adaptive systems is to partition system behaviors into different operational modes specified at design time. Such a multi-mode system can change behavior by switching between modes at runtime under certain circumstances. Multi-mode systems can benefit from a complementary approach to the software development of complex systems: Component-Based Software Engineering (CBSE), which fosters reuse of independently developed software components. However, the state-of-the-art component-based development of multi-mode systems does not take full advantage of CBSE, as reuse of modes at component level is barely addressed. Modes are often treated as system properties, while mode switches are handled by a global mode manager. This centralized mode management entails global information of all components, whereas the global information may be inaccessible in component-based systems. Another potential problem is that a single mode manager does not scale well, particularly at design time, for a large number of components and modes.

In this thesis we propose a distributed solution to the component-based development of multi-mode systems, aiming for a more efficient and scalable mode management. Our goal is to fully incorporate modes in software component reuse, supporting reuse of multi-mode components, i.e., components able to run in multiple modes. We have developed a generic framework, the Mode-Switch Logic (MSL), which not only supports reuse of multi-mode components but also provides runtime mechanisms for handling mode switch. MSL includes three fundamental elements: (1) a mode-aware component model with the formal specification of reusable multi-mode software components; (2) a mode mapping mechanism for the seamless composition of multi-mode components; and (3) a mode-switch runtime mechanism which is executed by each component in isolation from its functional execution and coordinates the mode switches of different components without the need of global mode information. The mode-switch runtime mechanism has been verified by model checking in conjunction with mathematical proofs. We also provide a mode-switch timing analysis for the runtime mechanism to respect real-time requirements.

MSL is dedicated to the mode aspect of a system irrespective of component execution semantics, thus independent of the choice of component models. We have integrated MSL in the ProCom component model with the extension of support for reuse of multi-mode components and distributed mode-switch handling. Although the distributed mode-switch handling of MSL is more flexible and scalable than the conventional centralized approach, when components are deployed on a single hardware platform and global mode information is available, centralized mode-switch handling is more efficient in terms of runtime overhead and mode-switch time. Hence, MSL is supplemented with a mode transformation technique to enhance runtime mode-switch efficiency by converting the distributed mechanism to a centralized mechanism. MSL together with the mode transformation technique has been implemented in a prototype tool where one can build multi-mode systems by reusing multi-mode components. The applicability of MSL is demonstrated in two proof-of-concept case studies.
To Hongwan and our parents
Self-adaptivity, characterized by the ability to dynamically adjust behavior at runtime, is a growing trend in the evolution of modern embedded systems. While self-adaptive systems tend to be flexible and autonomous, self-adaptivity may inevitably complicate software design, test and analysis. A strategy for taming the growing software complexity of self-adaptive systems is to partition system behaviors into different operational modes specified at design time. Such a multi-mode system can change behavior by switching between modes at runtime under certain circumstances. Multi-mode systems can benefit from a complementary approach to the software development of complex systems: Component-Based Software Engineering (CBSE), which fosters reuse of independently developed software components. However, the state-of-the-art component-based development of multi-mode systems does not take full advantage of CBSE, as reuse of modes at component level is barely addressed. Modes are often treated as system properties, while mode switches are handled by a global mode manager. This centralized mode management entails global information of all components, whereas the global information may be inaccessible in component-based systems. Another potential problem is that a single mode manager does not scale well, particularly at design time, for a large number of components and modes.

In this thesis we propose a distributed solution to the component-based development of multi-mode systems, aiming for a more efficient and scalable mode management. Our goal is to fully incorporate modes in software component reuse, supporting reuse of multi-mode components, i.e., components able to run in multiple modes. We have developed a generic framework, the Mode-Switch Logic (MSL), which not only supports reuse of multi-mode components but also provides runtime mechanisms for handling mode switch. MSL includes three fundamental elements: (1) a mode-aware component model with the formal specification of reusable multi-mode software components; (2) a
Abstract

Self-adaptivity, characterized by the ability to dynamically adjust behavior at runtime, is a growing trend in the evolution of modern embedded systems. While self-adaptive systems tend to be flexible and autonomous, self-adaptivity may inevitably complicate software design, test and analysis. A strategy for taming the growing software complexity of self-adaptive systems is to partition system behaviors into different operational modes specified at design time. Such a multi-mode system can change behavior by switching between modes at runtime under certain circumstances. Multi-mode systems can benefit from a complementary approach to the software development of complex systems: Component-Based Software Engineering (CBSE), which fosters reuse of independently developed software components. However, the state-of-the-art component-based development of multi-mode systems does not take full advantage of CBSE, as reuse of modes at component level is barely addressed. Modes are often treated as system properties, while mode switches are handled by a global mode manager. This centralized mode management entails global information of all components, whereas the global information may be inaccessible in component-based systems. Another potential problem is that a single mode manager does not scale well, particularly at design time, for a large number of components and modes.

In this thesis we propose a distributed solution to the component-based development of multi-mode systems, aiming for a more efficient and scalable mode management. Our goal is to fully incorporate modes in software component reuse, supporting reuse of multi-mode components, i.e., components able to run in multiple modes. We have developed a generic framework, the Mode-Switch Logic (MSL), which not only supports reuse of multi-mode components but also provides runtime mechanisms for handling mode switch. MSL includes three fundamental elements: (1) a mode-aware component model with the formal specification of reusable multi-mode software components; (2) a
mode mapping mechanism for the seamless composition of multi-mode components; and (3) a mode-switch runtime mechanism which is executed by each component in isolation from its functional execution and coordinates the mode switches of different components without the need of global mode information. The mode-switch runtime mechanism has been verified by model checking in conjunction with mathematical proofs. We also provide a mode-switch timing analysis for the runtime mechanism to respect real-time requirements.

MSL is dedicated to the mode aspect of a system irrespective of component execution semantics, thus independent of the choice of component models. We have integrated MSL in the ProCom component model with the extension of support for reuse of multi-mode components and distributed mode-switch handling. Although the distributed mode-switch handling of MSL is more flexible and scalable than the conventional centralized approach, when components are deployed on a single hardware platform and global mode information is available, centralized mode-switch handling is more efficient in terms of runtime overhead and mode-switch time. Hence, MSL is supplemented with a mode transformation technique to enhance runtime mode-switch efficiency by converting the distributed mechanism to a centralized mechanism. MSL together with the mode transformation technique has been implemented in a prototype tool where one can build multi-mode systems by reusing multi-mode components. The applicability of MSL is demonstrated in two proof-of-concept case studies.
Populärvetenskaplig sammanfattning

Vi omges dagligen av datorsystem som är inbyggda i produkter—så kallade inbyggda system. De finns i allt från smarta telefoner, medicinska apparater och industrirobotar till bilar och flygplan. Moderna inbyggda system blir alltmer anpassningsbara, med förmåga att dynamiskt anpassa sitt beteende på begäran eller när operativa förhållanden ändras. Anpassningsbara system är vanligtvis flexibla och autonoma, men som sidoeffekt kan de komplicera mjukvarudesignen, liksom testning och analys.

En effektiv metod för att hantera ökad komplexitet i programvaran av anpassningbara system är att dela upp systemet i olika driftlägen. Systemet kan då byta beteenden genom att växla mellan olika driftlägen. Till exempel kan programvaran som styr ett flygplan vara uppdelad i driftlägena taxi, start, flyg och landning. System med multipla driftlägen kan utnyttja en kompletterande teknik för programvaruutveckling av komplexa system: komponentbaserad programvaruutveckling som bygger på återanvändning av tidigare utvecklade programvarukomponenter. Det finns redan komponentbaserade tekniker som använder en centraliserad hantering av driftlägesväxlingar, en av de absolut viktigaste frågorna i system med multipla driftlägen. Men tyvärr är en centraliserad driftlägeshantering inte skalbar och därför olämplig att använda i mer komplexa system.

Vi har utvecklat ett ramverk, Mode-Switch Logic (MSL), som inte bara stödjer återanvändning av komponenter som kan hantera multipla driftlägen, utan också innehåller stöd för att genomföra driftväxling. Till skillnad från centraliserad driftväxling, haneras MSL:s driftväxling av en lokal algoritm som implementerar såväl den egna driftväxlingen som samordningen med driftväxling i andra komponenter. Vi har dessutom utvecklat en analysmetod
VI

för att räkna ut hur lång tid det tar att genomföra driftväxling.

MSL är ett generiskt ramverk som kan anpassas till olika komponentmodeller. Vi har integrerat MSL i komponentmodellen ProCom, som därmed utökats med stöd för återanvändning av komponenter med multipla driftlägen och distribuerad hantering av driftväxling. MSL har även implementerats i ett prototypverktyg som utöver att erbjuda en koppling mellan MSL och den kommersiella komponentmodellen Rubus också implementerar en mappning av MSLs flexibla driftlägeshantering till Rubus centraliserade hantering. Mappningen visar hur MSLs flexibilitet kan kombineras med effektiviteten hos centraliserad driftlägeshantering. Slutligen har vi demonstrerat användningen av MSL i två fallstudier.
摘要

嵌入式系统在日常生活中无处不在。比如智能手机，医疗器械，工业机器人，汽车，飞机等等都属于嵌入式系统。越来越多的嵌入式系统正在逐渐发展成为具有自适应性的系统。自适应性使得一个系统能够实时动态改变自身行为和功能以适应用户的移动调整和运行环境的改变。自适应系统较为灵活和自治，但容易导致更加复杂的软件设计，测试和分析。

系统设计阶段将系统功能划分为多个运行模式是一个有效的处理自适应系统软件复杂度的方法。多模式系统可以在特定情况下实时切换模式来改变运行功能。例如飞机的控制软件通常可以执行模式切换：起飞模式，飞行模式，和降落模式等等。多模式系统的开发还能额外受益于基于组件的软件工程。基于组件的软件工程是另一个有效管理复杂软件的方法。其特色在于重复使用独立开发的软件组件。现有的基于组件的软件开发技术对于多模式系统而言并没有将基于组件的软件工程的优势充分利用起来。组件的模式在组件重用中很少被提及。处理模式转换是多模式系统最重要的问题之一。现有的方法通常使用一个全模式管理器来处理模式转换。这样的集中式模式管理可扩展性不够理想，从而不适合构建较为复杂的系统。

本论文中我们提出一个集散式的方法来实现对多模式系统的基于组件的开发。和其它方法相比，这种集散式的方法具有更好的可扩展性，并且能够更为有效的管理组件和系统模式。我们将组件模式完全整合到软件组件重用中，支持多模式组件的可重用（多模式组件即为可以运行多个模式的组件）。这种集散式的开发方法被称为模式转换逻辑 (MSL: Mode-Switch Logic)。MSL不仅支持多模式组件的可重用，而且提供一系列算法来有效处理模式转换。这些算法运行在每一个组件里并且协调不同组件的模式转换。此外，我们对模式转换进行时间分析，计算模式转换所需时间，保证模式转换及时完成和满足系统的实时要求。

MSL对于组件模型的选择没有特别要求，适用于很多不同的组件模型。我们已经将MSL应用到ProCom组件模型，使得ProCom支持多模式
组件的可重用和集散式的模式转换处理。目前我们已经在一款正在开发的工具中实现了MSL的一些主要功能。该工具为多模式组件的重用提供了一个开发平台。MSL的实用性在本文中通过两个案例被演示出来。
Acknowledgements

What can be experienced and achieved over a half-decade? Apparently, a vast number of things could come to mind, developing a better career, learning a new language, acquiring new skills, writing a book, building a family, traveling around the world, experience new cultures, making new friends...you name it. In retrospect, I am truly proud of the most splendid episode of my life over the last five years—the odyssey to a PhD! While doing research as a PhD student is deemed to be dreary and tedious in the eyes of some other people, it is one of the coolest vocations that I could anticipate. Before I was enrolled as a PhD student at Mälardalen University (MDH), attaining a PhD degree seemed like an insurmountable mission for me. Yet I am fortunate enough to come this far, ready for the defense and on the verge of accomplishing the mission. This would certainly not have been possible without the help and support of those fantastic people that I ever met during my PhD studies.

My deepest gratitude undoubtedly goes to my main supervisor Hans Hansson who guided and inspired me all the way to the end of my PhD journey. Thank you for leading me along the path from a fresh student innocent of the profound academic world to an independent researcher! You are not only a mentor for me but also an academic superstar I respect from the bottom of my heart. I would also like to express my immense appreciation to my co-supervisors Paul Pettersson and Thomas Nolte. You have helped me and influenced me in various ways since my first year in Västerås. I am always encouraged by your wisdom, charisma, and professional attitude.

Life is accompanied by decision making all the time. Starting a PhD in Sweden is probably the best decision that I have ever made. Another decision which is equally sensible is moving to Sweden in the summer of 2008. Sweden is a wonderful country. The longer I stay here, the more affection accumulates. I am sincerely grateful to Damir Isovic, without whom I would have never thought of going abroad. Your visit to East China University of Science and Technology (ECUST, my home university in China) in 2007 is a blessing for
me, as it sets up the initial bridge between China and Sweden right in front of me. Many thanks also go to Huifeng Wang, my ex-supervisor at ECUST, who encouraged my overseas study at MDH.

Apart from my supervisors, there are many other people who ever offered technical advice at different stages of my PhD studies. I convey my heart-felt thanks to Tomas Bures for being the opponent for my Licentiate seminar, Mikael Sjödin for being the examiner for my Licentiate proposal, and Cristina Seceleanu for being the examiner for my PhD proposal. I am deeply indebted to Jan Carlson who has invariably been a reliable source for consultation not only on CBSE, ProCom, and UML metamodels, but also on the technical details of my own research. Thank you for your great contributions to the fruitful discussions and the papers that we co-authored! Another person that merits my wholehearted gratitude is Etienne Borde, who actively guided me after I just started my PhD journey. Another thanks goes to Paul Pettersson and Cristina Seceleanu for enlightening me about formal verification in my research. In addition, I would like to acknowledge Barbara Gallina and Björn Lisper for the advice beneficial to my research.

During the last year of my PhD studies, with the intention of exploring the industrial value of our research results, I visited Arcticus Systems AB several times and received generous support from Kurt Lundbäck and John Lundbäck, the founders of the Rubus component model. My warmest appreciation to you both! Moreover, many thanks go to Kaj Hänninen, Saad Mubeen and Mikael Sjödin for helping me gain so much insight into Rubus.

I used to have a tough time with UPPAAL modeling and verification. UPPAAL is an excellent tool, however, due to my complex models, the verification turned out to be so computationally expensive and memory-demanding that I became desperate for additional hardware assistance. I would like to thank Daniel Flemström for giving me remote access to MDH-Ericsson lab, Mahnaz Malekzadeh and Irfan Sljivo for sharing a workstation that completed my experiments in time.

The completion of a PhD betokens the commencement of a brand new career. It is my absolute honor to be accepted as a postdoc at Chalmers University of Technology (CTH) before my PhD degree is completed. My special thanks goes to Christian Berger and Michel Chaudron. Thank you for believing in me and providing me with such a superb opportunity to work on self-driving vehicles! Christian, even though I have not started my official employment in your team, I have been truly impressed by your devotion to research and passion for coding. It would be really exciting to work with you soon! Another special
thanks is dedicated to Ivica Crnkovic not only for his recommendation for my postdoc application but also for all the great times that he brought to me. Thank you for taking me to the magnificent CBSE community, taking good care of me during conference trips, and getting me involved in organizing ASE’14! It is a pity that I did not have a chance to work with you at MDH, but what a privilege it is to join your research group at CTH as a postdoc!

Since the end of 2014, I joined a local postdoc mentorship program to gain a preliminary understanding of the essentials for a postdoc. I warmly thank Wasif Afzal for arranging such an informative program. In particular, I am beholden to Kristina Lundqvist, my mentor in this program. Thank you for your guidance, encouragement, and genuine support for my job seeking!

My research has been accompanied by a variety of PhD courses that cover a broad spectrum of topics and have significantly enriched my knowledge and mindset. I would like to express my sincere gratitude to those who ever contributed to these courses, including (but not limited to) Gordana Dodig-Crnkovic, Jan Gustafsson, Lars Asplund, Cristina Seceleanu, Paul Pettersson, Hans Hansson, Thomas Nolte, Emma Nehrenheim, Ivica Crnkovic, Séverine Sentilles, Aneta Vulgarakis, Iain Bate, Sasikumar Punnekkat, Phil McMich, Ricardo J. Rodriguez, Reinder J. Bril, Hongyu Pei Breivold, Moris Behnam, Seif Haridi, Elisabeth Uhlemann, Lucia Lo Bello, Mats Björkman, Luis Almeida, Maria Lindén, and Johan Åkerberg.

In 2014, I received a kind invitation from Thomas Nolte to join the powerful CORE (Complex Real-Time Embedded Systems) research group. A big thank you to all the past and current CORE members, including (but not limited to) Thomas Nolte, Moris Behnam, Mohammad Ashjaei, Sara Afsar, Nima Moghaddami Khalilzad, Meng Liu, Saad Mubeen, Matthias Becker, Hamid Reza Faragardi, Reinder J. Bril, Luis Almeida, Rafia Inam, Mikael Åsberg, Yue Lu, and Farhang Nemati. Moris, thank you again for all the help and for being such a nice neighbor next to my office over these years!

The school of Innovation, design and engineering (IDT) at MDH, which I belong to, is a harmonious international family. We have colleagues from different countries with different cultural backgrounds, speaking different languages. Your kindness makes me enjoy every single working day. I am gonna miss all the happy moments with you during conference trips, coffee beaks, lunch chats, parties, and departmental social activities! In addition to the aforementioned thank lists, my warmest appreciation goes to (in alphabetical order): Abhilash Thekkilakkattil, Abu Naser Masud, Adnan Causevic, Aida Causevic, Alessio Bucaini, Andreas Gustavsson, Andreas Hjertström, Andreas Johnsen, Antonio Cicchetti, Baran Cürüklu, Batu Akan, Dag Nyström, Daniel Kade,

I owe a debt of gratitude to the administrative staff and research coordination team, in particular, Carola Ryttersson, Susanne Fronnå, Sofia Jäderén, Ingrid Runnérus, Jenny Hägglund, Annika Havbrandt, Anna Juto Andersson, Malin Åshuvud, Malin Rosqvist, Gunnar Widforss, and Caroline Blomberg. I express my utmost appreciation to your tremendous help that makes my daily affairs a lot easier.

If I have to prioritize my thank list, I shall put my family on top with no hesitation. I am deeply indebted to my parents for their everlasting parental love and care. Although I have been away from home for 14 years and family reunion has become an annual event, the distance between us never shakes our deep-rooted bond. You always respect my own decisions and devote everything to my happiness. Thank you for your unfailing support and encouragement all these years!

Last but foremost, I deliver my deepest thanks and love to my wife Hongwan Qin who has brought the true jubilance in my life, switching me from single mode to married mode. Thank you for being so tolerable, considerate and supportive all the time. Nothing else could make me feel more fortunate than to have you with me! Finally, I sincerely thank my parents-in-law for their unselfish care and tenderness. You are so kind to me that Hongwan is getting jealous:)

This work has been performed within the ARROWS project at MDH, funded by Vetenskapsrådet (Swedish Research Council).

Hang Yin
Västerås, August 18, 2015
## Contents

1 **Introduction**  
   1.1 Background and motivation .......................... 1  
   1.2 Research questions .................................. 7  
   1.3 Contributions ........................................ 8  
   1.4 Publications .......................................... 10  
   1.5 Research methodology ................................ 13  
   1.6 Thesis outline ........................................ 16  

2 **The mode-aware component model**  
   2.1 UML meta-model ........................................ 19  
   2.2 The formal specification of a multi-mode primitive component  
   2.3 The formal specification of a multi-mode composite component  
   2.4 An example ............................................ 24  
   2.5 Summary ................................................. 28  

3 **Mode mapping**  
   3.1 A motivating example ................................... 29  
   3.2 Mode mapping automata .................................. 31  
   3.3 MMA composition ......................................... 38  
   3.4 Summary ................................................ 44  

4 **Mode-switch handling at runtime**  
   4.1 Mode-switch triggering ................................. 48  
   4.2 Mode-switch propagation ............................... 49  
   4.3 Guarantee of mode consistency ......................... 66  
   4.4 Atomic execution ....................................... 68  
   4.5 Concurrent mode-switch triggering ..................... 74  

XIII
### Contents

4.5.1 MSR and MSQ queues ........................................ 74
4.5.2 The validity of pending scenarios ......................... 77
4.5.3 Analysis of the MSR/MSQ queue sizes ..................... 91
4.5.4 Priority MSR/MSQ queues ................................... 95
4.6 Emergency mode switch ........................................ 99
4.6.1 The handling of an emergency scenario ................... 100
4.6.2 Issues due to the concurrent triggering of emergency
and non-emergency scenarios .................................... 108
4.6.3 Solutions to the identified issues ........................... 110
4.6.4 The complete structure of the MSRM ....................... 123
4.6.5 Discussion ..................................................... 128
4.7 Summary ......................................................... 129

5 Verification of the mode-switch runtime mechanism.......... 131
5.1 Overview of the verification ................................... 131
5.2 The model checker UPPAAL .................................. 134
5.3 Verification of a non-emergency scenario ..................... 136
5.3.1 Modeling and verification in UPPAAL ...................... 137
5.3.2 Generalization of the verification results .................. 142
5.4 Verification of an emergency scenario ......................... 148
5.4.1 Modeling and verification in UPPAAL ...................... 148
5.4.2 Generalization of the verification results .................. 152
5.5 Verification of concurrent non-emergency scenarios ......... 154
5.5.1 Modeling and verification in UPPAAL ...................... 154
5.5.2 Generalization of the verification results .................. 161
5.6 Verification of concurrent emergency and non-emergency sce-
narios ............................................................. 164
5.6.1 Modeling and verification in UPPAAL ...................... 165
5.6.2 Generalization of the verification results .................. 171
5.7 Summary ......................................................... 174

6 Mode-switch timing analysis .................................... 175
6.1 Overview ......................................................... 175
6.2 Phase 1—Identifying the MSDM of a scenario ................. 178
6.3 Phase 2—Collecting the current states of Type A components 180
6.4 Phase 3—Mode-switch execution ................................ 182
6.5 Demonstration of the timing analysis .......................... 184
6.6 Deriving atomic execution time ................................ 187
6.6.1 The AEG model .............................................. 187
6.6.2 UPPAAL modeling ................................... 189
6.6.3 Verification ........................................ 193
6.6.4 Generalization ..................................... 196
6.7 Summary ............................................. 196

7 Mode switch for the ProCom component model 199
7.1 The ProCom component model ......................... 200
  7.1.1 The ProSave layer ................................ 200
  7.1.2 The ProSys layer ................................ 203
  7.1.3 The integration of ProSave and ProSys ............. 204
7.2 Multi-mode ProCom components ......................... 205
7.3 Handling mode switch in ProCom ....................... 206
  7.3.1 Handling mode switch in ProSave ................. 206
  7.3.2 Handling mode switch in ProSys ................. 213
7.4 Managing the variability of component connections ...... 216
  7.4.1 Managing the variability of component connections in
        ProSave ...................................... 217
  7.4.2 Managing the variability of component connections in
        ProSys ...................................... 218
7.5 Demonstrating development of multi-mode systems in ProCom 220
  7.5.1 System description ............................. 220
  7.5.2 Mode-switch handling ........................... 221
  7.5.3 Managing component connections for different modes 222
7.6 Summary ............................................. 227

8 Improving mode-switch efficiency 229
8.1 Mode transformation ................................. 230
8.2 Construction of the mode combination tree ............. 232
8.3 Deriving the mode transition graph .................... 238
8.4 Mode transformation example .......................... 240
8.5 Mode transformation verification ......................... 247
8.6 Discussion ........................................... 250
  8.6.1 Industrial value ................................ 250
  8.6.2 Merging system modes ........................... 251
  8.6.3 Partial mode transformation ....................... 251
  8.6.4 Implementing mode switch after mode transformation 252
  8.6.5 The concurrent triggering of multiple scenarios .... 253
8.7 Summary ............................................. 253
## Contents

9 **MCORE: the Multi-mode COmponent Reuse Environment** 255

10 **Case studies** 261
   10.1 An Adaptive Cruise Control system 261
      10.1.1 System description 261
      10.1.2 Component specification 264
      10.1.3 Mode mapping specification 267
      10.1.4 Mode switch at runtime 269
      10.1.5 Mode-switch timing analysis 271
      10.1.6 Discussion 275
   10.2 A healthcare monitoring system 275
      10.2.1 System description 276
      10.2.2 Component specification 278
      10.2.3 Mode mapping specification 280
      10.2.4 Mode switch at runtime 281
      10.2.5 Mode transformation 284
   10.3 Summary 285

11 **Related work** 289
   11.1 Design and modeling methods 289
      11.1.1 MECHATRONICUML 289
      11.1.2 The oracle-based approach for mode switch using property networks 291
      11.1.3 Input-output blocks for modeling multi-mode systems 292
      11.1.4 The modeling of multi-mode systems in MARTE 292
      11.1.5 Component-based coordination of autonomic managers 293
      11.1.6 Top-down mode specification using Event-B 294
   11.2 Component models supporting mode switch or dynamic reconfiguration 294
      11.2.1 BlueArX 294
      11.2.2 COMDES-II 295
      11.2.3 Koala and SaveCCM 295
      11.2.4 MyCCM-HI 296
      11.2.5 Rubus 296
      11.2.6 AUTOSAR 297
      11.2.7 Fractal 297
      11.2.8 Limitations 298
   11.3 Languages supporting mode switch 298
      11.3.1 Ada 298

12 **Conclusions and future work** 305
   12.1 Summary and contributions 305
   12.2 Future research directions 309

**Bibliography** 313

**Index** 327
Contents XVII

11.3.2 AADL ........................................ 299
11.3.3 Giotto and TDL .............................. 299
11.3.4 Darwin ....................................... 300
11.3.5 Modechart .................................. 300
11.3.6 Mode-automata ............................ 300
11.3.7 Limitations ................................. 301
11.4 Dynamic software product line ............... 301
11.5 Multi-mode real-time systems ................ 302

12 Conclusions and future work .................. 305
  12.1 Summary and contributions ................... 305
  12.2 Future research directions ................... 309

Bibliography .................................. 313

Index ........................................ 327
List of Acronyms

AADL  Architecture Analysis & Design Language, page 299
ACC  Adaptive Cruise Control, page 261
AEG  Atomic Execution Group, page 70
AUTOSAR  AUTomotive Open System ARchitecture, page 297
CBD  Component-Based Development, page 4
CBSE  Component-Based Software Engineering, page 4
COMDES-II  COMponent-based design of software for Distributed Embedded Systems-version II, page 295
CTM table  Component Target Mode table, page 239
DSPL  Dynamic Software Product Line, page 301
ECU  Electronic Control Unit, page 250
EMS  Emergency Mode Switch, page 100
EMSP protocol  Emergency Mode-Switch Propagation protocol, page 101
ETS  Emergency Transition State, page 104
LMC  Local Mode Combination, page 232
MCORE  Multi-mode COmponent Reuse Environment, page 255
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCT</td>
<td>Mode Combination Tree, page 232</td>
</tr>
<tr>
<td>MDE</td>
<td>Model-Driven Engineering, page 200</td>
</tr>
<tr>
<td>MDP</td>
<td>Mode-Dependent Property, page 20</td>
</tr>
<tr>
<td>MIP</td>
<td>Mode-Independent Property, page 20</td>
</tr>
<tr>
<td>MMA</td>
<td>Mode Mapping Automaton, page 33</td>
</tr>
<tr>
<td>MMT</td>
<td>Mode Mapping Time, page 178</td>
</tr>
<tr>
<td>MSA</td>
<td>Mode-Switch Abort, page 110</td>
</tr>
<tr>
<td>MSC</td>
<td>Mode-Switch Completion, page 66</td>
</tr>
<tr>
<td>MSD</td>
<td>Mode-Switch Denial, page 51</td>
</tr>
<tr>
<td>MSDM</td>
<td>Mode-Switch Decision Maker, page 50</td>
</tr>
<tr>
<td>MSDT</td>
<td>Mode-Switch Detecting Time, page 178</td>
</tr>
<tr>
<td>MSI</td>
<td>Mode-Switch Instruction, page 51</td>
</tr>
<tr>
<td>MSL</td>
<td>Mode-Switch Logic, page 8</td>
</tr>
<tr>
<td>MSP protocol</td>
<td>Mode-Switch Propagation protocol, page 52</td>
</tr>
<tr>
<td>MSQ</td>
<td>Mode-Switch Query, page 51</td>
</tr>
<tr>
<td>MSR</td>
<td>Mode-Switch Request, page 51</td>
</tr>
<tr>
<td>MSRM</td>
<td>Mode-Switch Runtime Mechanism, page 47</td>
</tr>
<tr>
<td>MSS</td>
<td>Mode-Switch Source, page 48</td>
</tr>
<tr>
<td>MST</td>
<td>Mode-Switch Time, page 178</td>
</tr>
<tr>
<td>MyCCM-HI</td>
<td>Make Your Component-Container Model-High Integrity, page 296</td>
</tr>
<tr>
<td>NTS</td>
<td>Normal Transition State, page 104</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service, page 3</td>
</tr>
<tr>
<td>QRT</td>
<td>Query Response Time, page 178</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------------------------------------------------</td>
</tr>
<tr>
<td>RCT</td>
<td>ReConfiguration Time</td>
</tr>
<tr>
<td>SCT</td>
<td>State-Checking Time</td>
</tr>
<tr>
<td>TCTL</td>
<td>Timed Computation Tree Logic</td>
</tr>
<tr>
<td>WCET</td>
<td>Worst-Case Execution Time</td>
</tr>
</tbody>
</table>
List of Notations

PC: the set of primitive components in a component-based system

CC: the set of composite components in a component-based system

Top: the component at the top of the component hierarchy in a component-based system

\( \tilde{CC} \): the set of composite components except \( \text{Top} \) in a component-based system

\( p_{\text{MS}} \): the dedicated mode-switch port of a multi-mode component for exchanging mode information with the parent

\( p_{\text{MS}} \) in \( c_i \): the dedicated mode-switch port of a multi-mode component for exchanging mode information with the subcomponents

\( P_{c_i} \): the parent of a component \( c_i \)

\( c_k \rightarrow m_i \rightarrow m_j \): \( c_k \) triggers a mode-switch scenario, requesting for a mode switch from mode \( m_i \) to mode \( m_j \)

\( \sigma_{c_i} \): the maximum number of scenarios that \( c_i \) can trigger as an MSS while \( c_i \) runs in the same mode

SC: the set of subcomponents of \( c_i \)

\( T_k_{c_i} = A \) or \( T_k_{c_i} = B \): \( c_i \) is a Type A or Type B component for a scenario \( k \)

\( \text{SC}_A(c_i)(k) \): the set of Type A subcomponents of \( c_i \) for a scenario \( k \)

\( \text{msr}_k \): an MSR for a scenario \( k \) (the same notation goes for all the other primitives)

\( \text{msr}_k_{c_i} \): an \( \text{msr}_k \) from \( c_i \)

\( \text{MMA}_{p_{c_i}} \): the parent Mode Mapping Automaton of \( c_i \in \text{CC} \)

\( \text{MMA}_{c_{c_i}} \): the child Mode Mapping Automaton of \( c_i \)

\( S_k_{c_i} \): the current state of \( c_i \) allows the mode switch for a scenario \( k \) (\( \neg S_k_{c_i} \) otherwise)
List of Notations

\(PC\) the set of primitive components in a component-based system

\(CC\) the set of composite components in a component-based system

\(Top\) the component at the top of the component hierarchy in a component-based system

\(\tilde{CC}\) the set of composite components except \(Top\) in a component-based system

\(p^{MS}\) the dedicated mode-switch port of a multi-mode component for exchanging mode information with the parent

\(P_{sna}^{MS}\) the dedicated mode-switch port of a multi-mode component for exchanging mode information with the subcomponents

\(P_{ci}\) the parent of a component \(c_i\)

\(c_k : m_i^{C_k} \rightarrow m_j^{C_k}\) an MSS \(c_k\) triggers a mode-switch scenario, requesting for a mode switch from mode \(m_i^{C_k}\) to mode \(m_j^{C_k}\)

\(\sigma_{c_i}\) the maximum number of scenarios that \(c_i\) can trigger as an MSS while \(c_i\) runs in the same mode

\(SC_{c_i}\) the set of subcomponents of \(c_i\)

\(T^{k}_{c_i} = A\) or \(T^{k}_{c_i} = B\) \(c_i\) is a Type A or Type B component for a scenario \(k\)

\(SC_{c_i}^A(k)\) the set of Type A subcomponents of \(c_i\) for a scenario \(k\)

\(msr^{k}\) an MSR for a scenario \(k\) (the same notation goes for all the other primitives)

\(msr^{k}_{ci}\) an \(msr^{k}\) from \(c_i\)

\(MMA_{p}^{c_i}\) the parent Mode Mapping Automaton of \(c_i \in CC\)

\(MMA_{c}^{c_i}\) the child Mode Mapping Automaton of \(c_i\)

\(S^{k}_{c_i}\) the current state of \(c_i\) allows the mode switch for a scenario \(k\) (\(\neg S^{k}_{c_i}\) otherwise)
<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C^i_j$</td>
<td>the set of components between $c_i$ and $c_j$ on the same path of the component hierarchy</td>
</tr>
<tr>
<td>$c_i.Q_{msr}$</td>
<td>the MSR queue of $c_i$</td>
</tr>
<tr>
<td>$c_i.Q_{msq}$</td>
<td>the MSQ queue of $c_i$</td>
</tr>
<tr>
<td>$c_i.Q_{ems}$</td>
<td>the EMS queue of $c_i$</td>
</tr>
<tr>
<td>$Q[1]$</td>
<td>the first element in an MSR/MSQ/EMS queue $Q$</td>
</tr>
<tr>
<td>$x \in Q$</td>
<td>$x$ is an element in the MSR/MSQ/EMS queue $Q$</td>
</tr>
<tr>
<td>$Q = \emptyset$ or $Q \neq \emptyset$</td>
<td>an MSR/MSQ/EMS queue $Q$ is empty or non-empty</td>
</tr>
<tr>
<td>$l_{c_i}$</td>
<td>the composition level of $c_i$</td>
</tr>
<tr>
<td>$d_{c_i}$</td>
<td>the depth level of $c_i$</td>
</tr>
<tr>
<td>$c_i.t_{msr}$</td>
<td>the transmission time of an MSR within $c_i$ (the same notation goes for all the other primitives)</td>
</tr>
<tr>
<td>$c_i.msdt$</td>
<td>the mode-switch detecting time of an MSS $c_i$</td>
</tr>
<tr>
<td>$c_i.mmt$</td>
<td>the mode mapping time of $c_i \in CC$</td>
</tr>
<tr>
<td>$c_i.sct$</td>
<td>the state-checking time of $c_i$</td>
</tr>
<tr>
<td>$AE_{G_i}$</td>
<td>the atomic execution time of an AEG $G_i$</td>
</tr>
<tr>
<td>$c_i.qrt$</td>
<td>the query response time of $c_i$</td>
</tr>
<tr>
<td>$c_i.rct$</td>
<td>the reconfiguration time of $c_i$</td>
</tr>
<tr>
<td>$c_i.mst$</td>
<td>the mode-switch time of $c_i$</td>
</tr>
<tr>
<td>$S_{mode}$</td>
<td>a dedicated service of a multi-mode ProSave component for handling its mode switch</td>
</tr>
<tr>
<td>$MSL^A_{c_i}$ and $MSL^B_{c_i}$</td>
<td>the dedicated pair of ProSave subcomponents of $c_i \in CC$ for handling its mode switch</td>
</tr>
<tr>
<td>$MSL_{c_i}$</td>
<td>the dedicated ProSys subcomponent of $c_i \in CC$ for handling its mode switch</td>
</tr>
<tr>
<td>$V_{c_i}$</td>
<td>a valid local mode combination (LMC) of $c_i \in CC$</td>
</tr>
<tr>
<td>$V_{c_i,m_{c_i}}$</td>
<td>a valid LMC of $c_i$ for mode $m_{c_i} \in CC$</td>
</tr>
<tr>
<td>$W_{c_i,m_{c_i}}$</td>
<td>the set of all valid LMCs of $c_i \in CC$ for mode $m_{c_i}$</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Mode switch illustration</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>A multi-mode system with multi-mode software components</td>
<td>6</td>
</tr>
<tr>
<td>1.3</td>
<td>Research methodology</td>
<td>15</td>
</tr>
<tr>
<td>2.1</td>
<td>The meta-model of a multi-mode component</td>
<td>21</td>
</tr>
<tr>
<td>2.2</td>
<td>The mode-aware component model of a primitive component</td>
<td>22</td>
</tr>
<tr>
<td>2.3</td>
<td>The mode-aware component model of a composite component</td>
<td>25</td>
</tr>
<tr>
<td>2.4</td>
<td>Port name marking for components $a$, $b$, $c$</td>
<td>26</td>
</tr>
<tr>
<td>2.5</td>
<td>Port name marking for components $c$, $e$, $f$</td>
<td>26</td>
</tr>
<tr>
<td>3.1</td>
<td>The inner component connections of $c$ in different modes</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>The role of mode mapping at runtime</td>
<td>32</td>
</tr>
<tr>
<td>3.3</td>
<td>The role of the mode mapping of $c$ at runtime</td>
<td>33</td>
</tr>
<tr>
<td>3.4</td>
<td>The parent MMA of $c$</td>
<td>35</td>
</tr>
<tr>
<td>3.5</td>
<td>The child MMA of $e$</td>
<td>37</td>
</tr>
<tr>
<td>3.6</td>
<td>The parent MMA of $a$</td>
<td>38</td>
</tr>
<tr>
<td>3.7</td>
<td>The child MMA of $b$</td>
<td>38</td>
</tr>
<tr>
<td>3.8</td>
<td>The child MMA of $c$</td>
<td>39</td>
</tr>
<tr>
<td>3.9</td>
<td>The child MMA of $d$</td>
<td>39</td>
</tr>
<tr>
<td>3.10</td>
<td>MMA composition—Case 1</td>
<td>42</td>
</tr>
<tr>
<td>3.11</td>
<td>MMA composition—Cases 2 and 6</td>
<td>43</td>
</tr>
<tr>
<td>3.12</td>
<td>MMA composition—Case 3</td>
<td>43</td>
</tr>
<tr>
<td>3.13</td>
<td>MMA composition—Case 4</td>
<td>44</td>
</tr>
<tr>
<td>3.14</td>
<td>MMA composition—Case 5</td>
<td>45</td>
</tr>
<tr>
<td>4.1</td>
<td>The MSP protocol—Case 1</td>
<td>55</td>
</tr>
<tr>
<td>4.2</td>
<td>The MSP protocol—Case 2</td>
<td>55</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.3</td>
<td>The MSP protocol—Case 3</td>
<td>56</td>
</tr>
<tr>
<td>4.4</td>
<td>A scenario $k$ triggered by the MSS $e$</td>
<td>57</td>
</tr>
<tr>
<td>4.5</td>
<td>Demonstration of the MSP protocol—Case 1</td>
<td>57</td>
</tr>
<tr>
<td>4.6</td>
<td>Demonstration of the MSP protocol—Case 2</td>
<td>58</td>
</tr>
<tr>
<td>4.7</td>
<td>Demonstration of the MSP protocol—Case 3</td>
<td>58</td>
</tr>
<tr>
<td>4.8</td>
<td>Demonstration of the improved MSP protocol</td>
<td>65</td>
</tr>
<tr>
<td>4.9</td>
<td>Demonstration of the mode-switch dependency rule</td>
<td>67</td>
</tr>
<tr>
<td>4.10</td>
<td>Atomic execution in a pipe-and-filter multi-mode system with multi-mode components</td>
<td>71</td>
</tr>
<tr>
<td>4.11</td>
<td>Demonstration of the MSQ delaying rule</td>
<td>73</td>
</tr>
<tr>
<td>4.12</td>
<td>Concurrent scenarios</td>
<td>79</td>
</tr>
<tr>
<td>4.13</td>
<td>Valid and invalid pending MSR primitives</td>
<td>80</td>
</tr>
<tr>
<td>4.14</td>
<td>The structure of algorithms implementing the task $Check_queues(c_i)$</td>
<td>91</td>
</tr>
<tr>
<td>4.15</td>
<td>Priority assignment to different scenarios</td>
<td>96</td>
</tr>
<tr>
<td>4.16</td>
<td>Demonstration of the EMSP protocol</td>
<td>102</td>
</tr>
<tr>
<td>4.17</td>
<td>Demonstration of a pending MSR valid for two rounds</td>
<td>106</td>
</tr>
<tr>
<td>4.18</td>
<td>Issue 1: An upstream EMS makes a previous MSR invalid</td>
<td>108</td>
</tr>
<tr>
<td>4.19</td>
<td>Issue 2: An EMS blocked by the handling of a non-emergency scenario</td>
<td>109</td>
</tr>
<tr>
<td>4.20</td>
<td>Issue 3: An upstream EMS makes a pending MSQ invalid</td>
<td>110</td>
</tr>
<tr>
<td>4.21</td>
<td>A phenomenon when an MSR queue contains two MSR primitives from the same subcomponent which subsequently sends an MSA</td>
<td>112</td>
</tr>
<tr>
<td>4.22</td>
<td>Demonstration of the MSA handling rule</td>
<td>113</td>
</tr>
<tr>
<td>4.23</td>
<td>Demonstration of the EMS receiving rule</td>
<td>120</td>
</tr>
<tr>
<td>4.24</td>
<td>Demonstration of the preliminary EMS handling rule</td>
<td>121</td>
</tr>
<tr>
<td>4.25</td>
<td>Demonstration of how the MSRM facilitates the handling of an emergency scenario</td>
<td>124</td>
</tr>
<tr>
<td>4.26</td>
<td>Algorithm structure of the MSRM for the handling of concurrent emergency and non-emergency scenarios</td>
<td>125</td>
</tr>
<tr>
<td>4.27</td>
<td>The workflow of $Check_queues2(c_i)$</td>
<td>127</td>
</tr>
<tr>
<td>4.28</td>
<td>An example revealing the flaw in [53]</td>
<td>129</td>
</tr>
<tr>
<td>5.1</td>
<td>Illustration of parent and child stubs</td>
<td>133</td>
</tr>
<tr>
<td>5.2</td>
<td>The overall model structure</td>
<td>133</td>
</tr>
<tr>
<td>5.3</td>
<td>The UPPAAL model of a lamp</td>
<td>136</td>
</tr>
<tr>
<td>5.4</td>
<td>The UPPAAL model structure assuming the triggering of a single non-emergency scenario</td>
<td>137</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>4.21</td>
<td>A phenomenon when an MSR queue contains two EMSs</td>
<td>112</td>
</tr>
<tr>
<td>4.22</td>
<td>Demonstration of the MSA handling rule</td>
<td>113</td>
</tr>
<tr>
<td>4.23</td>
<td>Demonstration of the EMS receiving rule</td>
<td>120</td>
</tr>
<tr>
<td>4.24</td>
<td>Demonstration of the preliminary EMS handling rule</td>
<td>121</td>
</tr>
<tr>
<td>4.25</td>
<td>Demonstration of how the MSRM facilitates the handling of concurrent scena-</td>
<td>122</td>
</tr>
<tr>
<td>4.26</td>
<td>Algorithm structure of the MSRM for the handling of concurrent scenarios</td>
<td>123</td>
</tr>
<tr>
<td>4.27</td>
<td>The workflow of Check_queues</td>
<td>124</td>
</tr>
<tr>
<td>4.28</td>
<td>An example revealing the flaw in [53]</td>
<td>129</td>
</tr>
<tr>
<td>4.29</td>
<td>A scenario</td>
<td>130</td>
</tr>
<tr>
<td>4.30</td>
<td>The MSP protocol—Case 1</td>
<td>131</td>
</tr>
<tr>
<td>4.31</td>
<td>The MSP protocol—Case 2</td>
<td>132</td>
</tr>
<tr>
<td>4.32</td>
<td>The MSP protocol—Case 3</td>
<td>133</td>
</tr>
<tr>
<td>4.33</td>
<td>The structure of algorithms implementing the task primitives</td>
<td>134</td>
</tr>
<tr>
<td>4.34</td>
<td>Valid and invalid pending primitives</td>
<td>134</td>
</tr>
<tr>
<td>4.35</td>
<td>Concurrent scenarios</td>
<td>135</td>
</tr>
<tr>
<td>4.36</td>
<td>Triggered by the MSS</td>
<td>136</td>
</tr>
<tr>
<td>4.37</td>
<td>A single non-emergency scenario</td>
<td>137</td>
</tr>
<tr>
<td>4.38</td>
<td>A non-emergency scenario</td>
<td>137</td>
</tr>
<tr>
<td>5.1</td>
<td>Illustration of parent and child stubs</td>
<td>138</td>
</tr>
<tr>
<td>5.2</td>
<td>The overall model structure</td>
<td>139</td>
</tr>
<tr>
<td>5.3</td>
<td>The UPPAAL model of a lamp</td>
<td>140</td>
</tr>
<tr>
<td>5.4</td>
<td>The UPPAAL model structure assuming the triggering of a single emergency</td>
<td>141</td>
</tr>
<tr>
<td>5.5</td>
<td>The UPPAAL model of a target component which handles a non-emergency scenar-</td>
<td>142</td>
</tr>
<tr>
<td>5.6</td>
<td>The UPPAAL model of a parent stub which handles a non-emergency scenario</td>
<td>143</td>
</tr>
<tr>
<td>5.7</td>
<td>The UPPAAL model of a child stub which handles a non-emergency scenario</td>
<td>144</td>
</tr>
<tr>
<td>5.8</td>
<td>The UPPAAL model structure assuming the triggering of a single emergency sce-</td>
<td>145</td>
</tr>
<tr>
<td>5.9</td>
<td>The UPPAAL model of a target component which handles an emergency scenario</td>
<td>146</td>
</tr>
<tr>
<td>5.10</td>
<td>The UPPAAL models of the parent stub and a child stub for handling an emergen-</td>
<td>147</td>
</tr>
<tr>
<td>5.11</td>
<td>The UPPAAL model structure assuming the concurrent triggering of non-emergen-</td>
<td>148</td>
</tr>
<tr>
<td>5.12</td>
<td>The UPPAAL model for receiving concurrent non-emergency scenarios</td>
<td>149</td>
</tr>
<tr>
<td>5.13</td>
<td>The UPPAAL model of a target component which handles concurrent non-emergen-</td>
<td>150</td>
</tr>
<tr>
<td>5.14</td>
<td>The UPPAAL model of a parent stub which handles concurrent non-emergency sce-</td>
<td>151</td>
</tr>
<tr>
<td>5.15</td>
<td>The UPPAAL model of a child stub which handles concurrent non-emergency sce-</td>
<td>152</td>
</tr>
<tr>
<td>5.16</td>
<td>The UPPAAL model structure assuming concurrent emergency and non-emergen-</td>
<td>153</td>
</tr>
<tr>
<td>5.17</td>
<td>The UPPAAL model for receiving concurrent emergency and non-emergency scenar-</td>
<td>154</td>
</tr>
<tr>
<td>5.18</td>
<td>The UPPAAL model of a target component which handles concurrent emergency an-</td>
<td>155</td>
</tr>
<tr>
<td>5.19</td>
<td>The UPPAAL model of a parent stub which handles concurrent emergency and no-</td>
<td>156</td>
</tr>
<tr>
<td>5.20</td>
<td>The UPPAAL model of a child stub which handles concurrent emergency and non-emergency scenarios</td>
<td>157</td>
</tr>
<tr>
<td>5.21</td>
<td>The UPPAAL model of a target component which handles concurrent emergency an-</td>
<td>158</td>
</tr>
<tr>
<td>5.22</td>
<td>The UPPAAL model of a parent stub which handles concurrent emergency and no-</td>
<td>159</td>
</tr>
<tr>
<td>5.23</td>
<td>The UPPAAL model of a child stub which handles concurrent emergency and non-emergency scenarios</td>
<td>160</td>
</tr>
<tr>
<td>5.24</td>
<td>The UPPAAL model of a target component which handles concurrent emergency an-</td>
<td>161</td>
</tr>
<tr>
<td>5.25</td>
<td>The UPPAAL model of a parent stub which handles concurrent emergency and no-</td>
<td>162</td>
</tr>
<tr>
<td>5.26</td>
<td>The UPPAAL model of a child stub which handles concurrent emergency and non-emergency scenarios</td>
<td>163</td>
</tr>
<tr>
<td>5.27</td>
<td>The UPPAAL model of a target component which handles concurrent emergency an-</td>
<td>164</td>
</tr>
<tr>
<td>5.28</td>
<td>The UPPAAL model of a parent stub which handles concurrent emergency and no-</td>
<td>165</td>
</tr>
<tr>
<td>5.29</td>
<td>The UPPAAL model of a child stub which handles concurrent emergency and non-emergency scenarios</td>
<td>166</td>
</tr>
<tr>
<td>5.30</td>
<td>The UPPAAL model of a target component which handles concurrent emergency an-</td>
<td>167</td>
</tr>
<tr>
<td>5.31</td>
<td>The UPPAAL model of a parent stub which handles concurrent emergency and no-</td>
<td>168</td>
</tr>
<tr>
<td>5.32</td>
<td>The UPPAAL model of a child stub which handles concurrent emergency and non-emergency scenarios</td>
<td>169</td>
</tr>
<tr>
<td>5.33</td>
<td>The UPPAAL model of a target component which handles concurrent emergency an-</td>
<td>170</td>
</tr>
<tr>
<td>5.34</td>
<td>The UPPAAL model of a parent stub which handles concurrent emergency and no-</td>
<td>171</td>
</tr>
<tr>
<td>5.35</td>
<td>The UPPAAL model of a child stub which handles concurrent emergency and non-emergency scenarios</td>
<td>172</td>
</tr>
<tr>
<td>6.1</td>
<td>The 3 phases of a complete mode switch</td>
<td>173</td>
</tr>
<tr>
<td>6.2</td>
<td>Mode-switch timing analysis—Phase 1</td>
<td>174</td>
</tr>
<tr>
<td>6.3</td>
<td>Mode-switch timing analysis—Phase 2</td>
<td>175</td>
</tr>
<tr>
<td>6.4</td>
<td>Mode-switch timing analysis—Phase 3</td>
<td>176</td>
</tr>
<tr>
<td>6.5</td>
<td>Demonstration of the mode-switch timing analysis</td>
<td>178</td>
</tr>
<tr>
<td>6.6</td>
<td>An AEG $G_i$ with multiple components</td>
<td>179</td>
</tr>
</tbody>
</table>
### List of Figures

6.7 UPPAAL models: Data source and data forwarder ........... 190
6.8 UPPAAL model: AEG \( G_i \) ........................................ 191
6.9 UPPAAL model: Component \( f \) ..................................... 192
6.10 UPPAAL model: Component \( a \) ..................................... 193
6.11 UPPAAL model: Component \( b \) ..................................... 194
6.12 UPPAAL model: Component \( e \) ..................................... 194

7.1 A ProSave component ................................................. 201
7.2 Typical connectors in ProSave ........................................ 202
7.3 The connections between ProSave components.................... 202
7.4 A ProSys component ................................................... 203
7.5 The connections between ProSys components....................... 204
7.6 A ProSys component composed by ProSave components .......... 205
7.7 Multi-mode ProCom components ...................................... 206
7.8 The port definition of \( MSL_{ci}^A \) and \( MSL_{ci}^B \) ................. 207
7.9 The connections around \( MSL_{ci}^A \) and \( MSL_{ci}^B \) ............... 209
7.10 The port definition of \( MSL_{ci} \) ....................................... 214
7.11 The connections around \( MSL_{ci} \) .................................... 216
7.12 Managing the variability of ProSave component connections . 219
7.13 Managing the variability of ProSys component connections ... 220
7.14 The inner ProSys component connections within \( a \) .......... 221
7.15 The inner ProSave component connections within \( c \) .......... 221
7.16 The component hierarchy in ProCom ................................ 221
7.17 The port definition of \( MSL_c^A \) and \( MSL_c^B \) .................. 222
7.18 The port definition of \( MSL_a \) ....................................... 223
7.19 The complete view of inner component connections of \( c \) ... 224
7.20 The complete view of inner component connections of \( a \) ... 226

8.1 The overview of mode transformation ................................. 231
8.2 The mode transformation process .................................... 231
8.3 The component hierarchy of an example ........................... 240
8.4 The mode mapping tables of \( a, c, d \) ............................... 240
8.5 Demonstration of the MCT ............................................ 243
8.6 The global configurations of different system modes ............. 244
8.7 The set of MMAs of \( a \) ............................................. 245
8.8 The set of MMAs of \( c \) ............................................. 246
8.9 The set of MMAs of \( d \) ............................................. 247
8.10 Deriving the mode transition graph ................................. 248
8.11 Mode transformation facilitates a mode switch .................... 248
8.12 Partial mode transformation for distributed nodes .......................... 252

9.1 The workspace of MCORE ........................................ 256
9.2 The workflow of MCORE together with Rubus ICE ....................... 258

10.1 The component hierarchy of the ACC system ............................. 263
10.2 The ACC system in different modes .................................... 264
10.3 The ACC Controller in different modes ................................... 265
10.4 The parent MMA of Top .............................................. 269
10.5 The child MMAs of OR, BA, ACC, SL and HMI ......................... 269
10.6 The parent MMA of ACC ............................................. 270
10.7 The child MMAs of DC and SC ........................................ 270
10.8 The mode switch for an emergency scenario \( k_4 \) .......................... 271
10.9 The mode switch for a non-emergency scenario \( k_5 \) .................... 272
10.10 The mode-switch timing analysis for a non-emergency scenario \( k_5 \) .............................................. 273
10.11 The component hierarchy of the monitoring subsystem of a healthcare monitoring system .............................................. 277
10.12 The port names of MuD and its subcomponents ........................... 279
10.13 The parent MMA of MoS ............................................. 281
10.14 The child MMAs of MoS ............................................. 282
10.15 The parent MMA of MuD .............................................. 282
10.16 The child MMAs of MuD .............................................. 283
10.17 MMA composition for MoS ............................................ 283
10.18 Mode switches due to the concurrent triggering of scenarios \( k_2 \)
    and \( k_3 \), with \( k_3 \) being handled first .................................. 284
10.19 The mode combination tree of the monitoring subsystem ............... 286
10.20 The global configurations of different system modes in the
    monitoring subsystem ................................................... 287
10.21 Deriving the mode transition graph of the monitoring subsystem 288
List of Tables

1.1 The mapping of research questions and research contributions 11
1.2 The mapping of key publications and research contributions 14

3.1 Modes to be mapped during component composition 30
3.2 The mode mapping table of a 31
3.3 The mode mapping table of c 31

4.1 The mode mapping table of a 77
4.2 The mode mapping table of c 78
4.3 The minimum required MSR and MSQ queue sizes 95

5.1 A summary of the verification time for the concurrent triggering of four non-emergency scenarios 161
5.2 A summary of the verification time for the concurrent triggering of one emergency scenario and three non-emergency scenarios 171

6.1 Timing factors assignment 185
6.2 Verification results for different data rates 195
6.3 Maximal buffer usage for different data rates 196

10.1 The mode mapping table of the ACC system 268
10.2 The mode mapping table of the ACC Controller 268
10.3 Mode-switch timing factors of the ACC system 272
10.4 The mode mapping table of MoS 280
10.5 The mode mapping table of MuD 280
List of Algorithms

1. New_scenario\(c_i\) .................................. 60
2. Collect_response\(c_i \in CC, k\) ................. 61
3. Propagation\(c_i \in PC\) .................................. 61
4. Propagation\(c_i = \text{Top}\) .......................... 64
5. Mode_switch\(c_i, k\) .................................. 69
6. Collect_MSC\(c_i \in CC, k\) ..................... 69
7. MSQ_propagation\(c_i \in CC, k\) ................... 74
8. New_scenario2\(c_i\) .................................. 82
9. Scenario_reception\(c_i\) ..................... 83
10. Clean_queues\(c_i, k\) ...................... 83
11. Check_queues\(c_i\) .......................... 84
12. Handle_MSQ\(c_i \in PC\) .......................... 85
13. Handle_MSR\(c_i \in PC\) ...................... 85
14. Handle_MSQ\(c_i = \text{Top}\) .................... 86
15. Handle_MSR\(c_i = \text{Top}\) .................... 87
16. Mode_switch2\(c_i, k\) ...................... 89
17. Collect_MSC2\(c_i \in CC, k\) ................... 89
18. Update_queues\(c_i, k\) ...................... 90
19. PriorityQ\(\text{primitive}, c_i\) ..................... 98
20. EMSP\(c_i, k\) .................................. 103
21. Check_queues2\(c_i\) .................................. 105
22. Update_queues2\(c_i, k\) ..................... 107
23. New_scenario3\(c_i\) ....................... 111
24. Handle_MSA\(c_i \in CC, \text{msa}_k c_j\) ............. 114
List of Algorithms

1. New_scenario($c_i$) ........................................... 60
2. Collect_response($c_i \in CC, k$) ............................ 61
3. Propagation($c_i \in PC$) ..................................... 61
4. Propagation($c_i \in \overline{CC}$) ............................. 62
5. Propagation($c_i = Top$) ..................................... 64
6. Mode_switch($c_i, k$) ....................................... 69
7. Collect_MSC($c_i \in CC, k$) ................................. 69
8. MSQ_propagation($c_i \in CC, k$) ............................ 74
9. New_scenario2($c_i$) ......................................... 82
10. Scenario_reception($c_i$) .................................... 83
11. Clean_queues($c_i, k$) ....................................... 83
12. Check_queues($c_i$) ......................................... 84
13. Handle_MSQ($c_i \in PC$) ................................... 85
14. Handle_MSR($c_i \in PC$) ................................... 85
15. Handle_MSQ($c_i \in \overline{CC}$) ............................. 86
16. Handle_MSR($c_i \in \overline{CC}$) ............................. 87
17. Handle_MSQ($c_i = Top$) ................................... 88
18. Handle_MSR($c_i = Top$) ................................... 88
19. Mode_switch2($c_i, k$) ...................................... 89
20. Collect_MSC2($c_i \in CC, k$) ............................... 89
21. Update_queues($c_i, k$) .................................... 90
22. PriorityQ(primitive, $c_i$) .................................. 98
23. EMSP($c_i, k$) .............................................. 103
24. Check_queues2($c_i$) ....................................... 105
25. Update_queues2($c_i, k$) ................................... 107
26. New_scenario3($c_i$) ....................................... 111
27. Handle_MSA($c_i \in CC, msa_{c_j}^k$) ...................... 114
XXXIV List of Algorithms

28 Collect_response2\((c_i \in \mathcal{C}, k)\) \hspace{1cm} \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldot
List of Publications

Papers related to the thesis:

Licentiate thesis


Conference and workshop papers


Technical reports


Papers not related to the thesis:

Conference and workshop papers


Technical reports

Chapter 1

Introduction

Embedded systems are evolving into self-adaptive systems that can dynamically adjust behavior at runtime. However, self-adaptivity often calls for more complex software. To tame such growing software complexity, self-adaptive systems can be developed as multi-mode systems which run in multiple operational modes, with distinctive behaviors for different modes. Apart from multi-mode systems, another complementary solution to effective software management is component-based development by reusing independently developed software components. The prime goal of this thesis is to investigate the combination of these two approaches, i.e., multi-mode systems and component-based development, with the aim to facilitate reuse of multi-mode software components and provide effective runtime mechanisms for handling mode switch. In this first chapter, the background and motivation of this thesis will be introduced. We also identify our research challenges and research questions, followed by a summary of the thesis contributions, and a mapping of these contributions to the publications that the thesis is based on. Finally, we introduce our research methodology and present the thesis outline.

1.1 Background and motivation

Embedded systems are playing a vital role in modern society. Our daily life is full of embedded systems, from tiny portable electronic devices to huge aircrafts. Moreover, it has been witnessed in recent years that embedded systems are gradually evolving to become more self-adaptive and autonomous,
Chapter 1. Introduction

characterized by their abilities to dynamically change configuration at runtime without human intervention. Despite the runtime flexibility of self-adaptive systems, software complexity may grow as a consequence, which is an impediment to software design, modeling, analysis, test, verification and validation.

To tame growing software complexity, self-adaptivity can be realized by partitioning system behaviors into different mutually exclusive operational modes so that each mode corresponds to a specific system behavior. On account of diverging viewpoints and perspectives apropos mode, there is still no official unanimous definition of mode to date, though multi-mode systems have existed for decades. Degani et al. [33] proposed a succinct and sensible definition: "Mode is the manner of behavior of a given system". A multi-mode system usually runs in one mode and can switch to another mode under certain conditions. Herein we define mode switch as follows:

**Definition 1. Mode switch:** For a multi-mode system that supports a set of modes \( \mathcal{M} \), a mode switch is an intermediate transition of the system from the current mode \( m_1 \) to a new mode \( m_2 \), where \( m_1, m_2 \in \mathcal{M} \).

Figure 1.1 illustrates the mode switch of a system from mode \( m_1 \) to \( m_2 \). The mode switch is initiated at time \( t_1 \) and completed at \( t_2 \). During a mode switch, some particular actions need to be executed to change system configurations. For instance, some functionalities of the old mode may be deactivated while some other functionalities of the new mode may be activated. Hence, a mode switch is not instantaneous. Since a mode switch often engenders certain degree of service interruption, it is important to complete a mode switch within a predefined interval.

![Mode switch illustration](image)

Figure 1.1: Mode switch illustration

A representative example of multi-mode systems is the control software of an airplane which could run in the modes taxi (the initial mode), taking off, flight and landing. Different subsystems run in different modes. For instance, the subsystem for controlling the wheels only runs in taxi mode whereas the navigation subsystem may run only in flight mode. Multi-mode systems have been motivated by various reasons:

1. More efficient software design, testing, and verification. A multi-mode system allows for the separate design and parallel testing of different modes, thus facilitating the development process.
2. Diversity of system functionality. A multi-mode system exhibits distinctive functionalities in different modes. Therefore, it is usually easier for a multi-mode system to provide more diversified services compared to a single-mode system.
3. Adaptivity. A multi-mode system can be considered as a type of adaptive system that actively adjusts its behavior and performance to accommodate new conditions. For instance, an adaptive media player of an embedded device with constrained resource may switch between the degraded Quality of Service (QoS) mode and normal mode, depending on its runtime resource provision.
4. Saving resources. For those systems with a lot of tasks which only need to run under certain conditions, the constant running of all tasks would be a waste of resources. It is more efficient to deactivate the tasks that are not currently used in the system. Different modes could be specified based on the set of running tasks.
5. Fault tolerance. Most safety-critical systems are designed to be fault-tolerant so as to reduce the risks for catastrophic consequences. A practical fault-tolerant strategy is to switch to a Safe mode in case a fault occurs.
6. More precise analysis of system properties. Many system properties are associated with a wide range of values, however, the maximum and minimum could have extremely low probability of occurrence. A typical example is the Worst-Case Execution Time (WCET) of a task in a real-time system. It is often the case that this WCET is much larger than the average-case execution time. For a single-mode system, the WCET calculation is typically rather pessimistic. By contrast, for a multi-mode system, if a task runs in multiple modes, its WCET can be calculated or measured for each mode. Compared with a single WCET of the task for all modes, a set of mode-dependent WCETs yield more precise WCET analysis. The same benefit applies to other system properties as well, such as stack usage.
7. Extensibility and scalability. For a single-mode system, adding a new function risks polluting the software structure of the original system and
Chapter 1. Introduction

characterized by their abilities to dynamically change configuration at runtime without human intervention. Despite the runtime flexibility of self-adaptive systems, software complexity may grow as a consequence, which is an impediment to software design, modeling, analysis, test, verification and validation.

To tame growing software complexity, self-adaptivity can be realized by partitioning system behaviors into different mutually exclusive operational modes so that each mode corresponds to a specific system behavior. On account of diverging viewpoints and perspectives apropos mode, there is still no official unanimous definition of mode to date, though multi-mode systems have existed for decades. Degani et al. [33] proposed a succinct and sensible definition: ”Mode is the manner of behavior of a given system”. A multi-mode system usually runs in one mode and can switch to another mode under certain conditions. Herein we define mode switch as follows:

Definition 1. Mode switch: For a multi-mode system that supports a set of modes \( M \), a mode switch is an intermediate transition of the system from the current mode \( m_1 \) to a new mode \( m_2 \), where \( m_1, m_2 \in M \).

Figure 1.1 illustrates the mode switch of a system from mode \( m_1 \) to \( m_2 \). The mode switch is initiated at time \( t_1 \) and completed at \( t_2 \). During a mode switch, some particular actions need to be executed to change system configurations. For instance, some functionalities of the old mode may be deactivated while some other functionalities of the new mode may be activated. Hence, a mode switch is not instantaneous. Since a mode switch often engenders certain degree of service interruption, it is important to complete a mode switch within a predefined interval.

Figure 1.1: Mode switch illustration

A representative example of multi-mode systems is the control software of an airplane which could run in the modes taxi (the initial mode), taking off, flight and landing. Different subsystems run in different modes. For instance, the subsystem for controlling the wheels only runs in taxi mode whereas the navigation subsystem may run only in flight mode. Multi-mode systems have been motivated by various reasons:

1. More efficient software design, testing, and verification. A multi-mode system allows for the separate design and parallel testing of different modes, thus facilitating the development process.

2. Diversity of system functionality. A multi-mode system exhibits distinctive functionalities in different modes. Therefore, it is usually easier for a multi-mode system to provide more diversified services compared to a single-mode system.

3. Adaptivity. A multi-mode system can be considered as a type of adaptive system that actively adjusts its behavior and performance to accommodate new conditions. For instance, an adaptive media player of an embedded device with constrained resource may switch between the degraded Quality of Service (QoS) mode and normal mode, depending on its runtime resource provision.

4. Saving resources. For those systems with a lot of tasks which only need to run under certain conditions, the constant running of all tasks would be a waste of resources. It is more efficient to deactivate the tasks that are not currently used in the system. Different modes could be specified based on the set of running tasks.

5. Fault tolerance. Most safety-critical systems are designed to be fault-tolerant so as to reduce the risks for catastrophic consequences. A practical fault-tolerant strategy is to switch to a Safe mode in case a fault occurs.

6. More precise analysis of system properties. Many system properties are associated with a wide range of values, however, the maximum and minimum could have extremely low probability of occurrence. A typical example is the Worst-Case Execution Time (WCET) of a task in a real-time system. It is often the case that this WCET is much larger than the average-case execution time. For a single-mode system, the WCET calculation is typically rather pessimistic. By contrast, for a multi-mode system, if a task runs in multiple modes, its WCET can be calculated or measured for each mode. Compared with a single WCET of the task for all modes, a set of mode-dependent WCETs yield more precise WCET analysis. The same benefit applies to other system properties as well, such as stack usage.

7. Extensibility and scalability. For a single-mode system, adding a new function risks polluting the software structure of the original system and
necessitates the test of the complete new system. Alternatively, if a new mode is introduced for the new function, the system behavior in existing modes will not be affected. Additional development and test effort is limited to the new mode and the mode switch from or to this new mode. This also implies good scalability of a multi-mode system.

Multi-mode systems can gain additional benefits from another effective technique for improving software complexity management: Component-Based Software Engineering (CBSE) [115, 30], also known as Component-Based Development (CBD). As a promising design paradigm for development of complex systems, CBSE advocates reuse of independently developed software components. In other words, a system does not have to be developed from scratch. Instead, some of its components\textsuperscript{1} or subsystems may be directly obtained from a repository of pre-developed components. Therefore, system development and component development become two separate activities. CBSE boasts several appealing features, such as

- Software complexity management. When a component is reused, there is no need to know its internal details. The only information essential for its reuse is its interfaces and provided/required services. Hence software complexity can be tamed by component reuse which avoids the management of global information.

- Shortened time to market. Usually, it takes much shorter time to reuse a component than to develop its functionality from scratch. The more components that are reused, the higher productivity, thus the shorter time to market.

- Improved software quality. Reusing well tested and certified components is instrumental in building a more reliable system.

The success of CBSE has been evidenced by a variety of component models [80] proposed both in industry and academia, such as Microsoft Component Object Model (COM) [17], Enterprise JavaBeans (EJB) [89] and Open Services Gateway Initiative (OSGi) [3] for general purpose systems, as well as Koala [95], Rubus [57], ProCom [110], and AUTOSAR [43] for embedded systems. A thorough and extensive classification of various component models can be found in [31] while an overview and comparison of existing component models dedicated to embedded systems is provided in [67].

\textsuperscript{1}Hereafter any occurrence of "component" in the text implies "software component" unless explicitly specified otherwise.
There are already some existing works which take the advantage of CBSE for developing multi-mode systems. For instance, Rubus [57] is an industrial component model developed by Arcticus Systems\(^2\) for software development of ground vehicles. In Rubus, a system running on an Electronic Control Unit can support multiple modes while each mode is associated with a unique global configuration. The mode switch of the system is handled by a global mode-switch manager. Since mode is a global concept of the system and each Rubus component is unaware of mode, the mode-switch handling of Rubus has the following limitations:

1. The information of components at all nested levels must be globally accessible for the global mode-switch manager to handle a mode switch.

2. The entire system must be reconfigured when an individual component is added, removed or replaced.

3. Once a multi-mode system is built, the system itself cannot be used as a component in another system due to the lack of support for the composition of multi-mode components.

These limitations can be lifted by the use of multi-mode components, which can be considered as reusable multi-mode systems. Figure 1.2 illustrates a multi-mode system built from multi-mode components. The system, i.e. Component \(a\), consists of components \(b\), \(c\) and \(d\). Component \(c\) is composed by \(e\) and \(f\). Among these components, \(b\), \(d\), \(e\), and \(f\) are primitive components which are directly implemented by code, while \(a\) and \(c\) are composite components composed by other components. The tree structure of the component hierarchy implies a parent-child relationship between each composite component and the components directly composing it. For instance, \(a\) is the parent of \(b\), \(c\), \(d\) which in turn are the subcomponents or children of \(a\). What makes this system distinctive compared with traditional component-based systems is that some of its components are multi-mode components. For example, depicted in Figure 1.2, \(a\) is a multi-mode component which can run in two modes: \(m^1_a\) and \(m^2_a\), with each mode being represented by a unique local configuration. Here each component has a number of ports represented by squares, with its input ports on the left and output ports on the right. Component connections are represented by arrows from an input/output port of a component to an input/output port of another component. When \(a\) runs in mode \(m^1_a\), \(d\) is deactivated (represented by

\(^2\)http://www.arcticus-systems.com/
the dimmed color); when a runs in $m_a^2$, d becomes activated and extra connections are established within a. In addition, b exhibits different mode-specific behaviors (distinguished by black and grey colors) when a is running in different modes. Similar to a, some other components in Figure 1.2 may also be multi-mode components. For instance, c can run in three modes: $m_c^1$, $m_c^2$, and $m_c^3$.

Reuse of multi-mode components raises additional challenges. First, there is still no clear definition of reusable multi-mode components. Second, mode-related information must be properly integrated during the composition of multi-mode components. Moreover, the mode switch of one component may affect the modes of some other components. For instance, the mode switch of a in Figure 1.2 from mode $m_a^1$ to $m_a^2$ requires the activation of d, the behavior change of b, and possibly affects the modes of c, e, and f. The mode switches of different components must be properly synchronized and coordinated to guarantee a correct system mode switch.

These challenges entail new techniques for allowing reuse of multi-mode components in software development of multi-mode systems. In addition, a key issue for a multi-mode system with multi-mode components is how to handle mode switch properly, whereas mode-switch handling in the presence of multi-mode components has hardly been addressed before. Hence, we are motivated to overcome these challenges in this thesis.
1.2 Research questions

Reuse of multi-mode components is beneficial to development of multi-mode systems by virtue of more flexible design choices, more efficient software complexity management, and better scalability. However, to the best of our knowledge, a comprehensive framework supporting reuse of multi-mode components is still lacking, despite the existence of a handful of initial research results relating to multi-mode components. Hence our overall research goal is:

To develop a framework which not only allows the flexible and efficient reuse of multi-mode components for building multi-mode systems but also provides efficient runtime mechanisms for handling mode switch.

The research goal above exhibits a variety of challenges. To achieve the research goal and overcome different challenges separately, the research goal is decomposed into a set of research questions. The first research question is rather straightforward:

Research Question (RQ) 1: What distinctive features and what techniques should a multi-mode component support in order to ensure flexible and efficient reuse?

A fundamental requirement imposed by this research question is the formal definition of a reusable multi-mode component. We must capture the distinguishing features of a reusable multi-mode component as opposed to a single-mode component. Also, particular rules should be specified for mapping component modes during the composition of multi-mode components.

The next research question is about mode switch at runtime:

Research Question (RQ) 2: How do we handle mode switch at runtime at both component and system levels?

Unlike traditional multi-mode systems where a mode switch is performed by changing system configurations, a multi-mode system with multi-mode components can switch mode at both component and system levels. On the one hand, some individual components may switch mode concurrently. On the other hand, a system mode switch can be represented by the joint mode switches of different components. Since no component has the global information, it is necessary to provide suitable runtime mechanisms to coordinate the mode switches of different components.

The expected mechanisms should not only be functionally correct, but also
meet certain timing constraints:

**Research Question (RQ) 3:** How can we ensure that the mode-switch handling mechanisms in RQ 2 do not violate the system timing requirements?

All multi-mode systems are supposed to tolerate certain maximum mode-switch time. Therefore, the mode-switch time of a system must be bounded and analyzable. A mode-switch timing analysis would play a crucial role in evaluating the mode-switch handling mechanisms for this framework.

Aside from the correctness and efficiency of this framework, its applicability is a more practical issue:

**Research Question (RQ) 4:** How do we evaluate the applicability of such a framework mentioned in the overall research goal?

There are various alternatives for evaluating the applicability of this framework, e.g. by demonstrating that this framework can be applied to an existing technique whose practical value has been ascertained, or that we are able to develop a real-world multi-mode system guided by this framework or a tool for building multi-mode systems with reusable multi-mode components.

### 1.3 Contributions

Driven by the set of research questions in Section 1.2, we have developed a generic framework, the Mode-Switch Logic (MSL) for the component-based development of multi-mode systems. MSL comprises the following eight contributions that jointly achieve our research goal:

**Research Contribution (RC) 1:** A mode-aware component model.

This component model provides a formal definition of reusable multi-mode components and specifies their essential features. According to the mode-aware component model, a multi-mode component should have a unique local configuration for each of its modes, dedicated mode-switch ports for exchanging mode information at runtime, and a built-in Mode-Switch Runtime Mechanism (MSRM) for handling its mode switch.

**Research Contribution (RC) 2:** A mode mapping mechanism for each multi-mode composite component.
1.3 Contributions

During the composition of multi-mode components at design time, the modes of different components must be mapped without ambiguity. Since a composite component knows the mode information of its subcomponents but not the other way around, it is the responsibility of each composite component to maintain the local mode mapping between its own modes and the modes of its subcomponents. Our mode mapping mechanism proposes Mode Mapping Automata (MMAs) for the formal representation of the mode mapping of each composite component.

**Research Contribution (RC) 3:** An MSRM for each multi-mode component.

Most traditional mode-switch techniques use a global mode-switch manager for the entire system to handle mode switch. Such a centralized solution usually does not scale well, and requires global information which is often unavailable in a component-based system. By contrast, MSL adopts a distributed mode-switch handling mechanism, allowing each individual component to have its own MSRM and communicate with other components during a mode switch.

**Research Contribution (RC) 4:** A mode-switch timing analysis for our MSRM.

The purpose of the timing analysis is to calculate the worst-case mode-switch time of a multi-mode system. Under the influence of multi-mode components and their MSRM, the mode-switch time of a system depends on both the mode-switch time of each individual component and the inter-component communication time.

**Research Contribution (RC) 5:** The integration of MSL in the ProCom component model.

ProCom [110] is a component model for developing distributed real-time embedded systems, yet without support for multi-mode components. We have integrated the main principles of MSL in ProCom such that ProCom supports reuse of multi-mode components with slight extension. This work provides evidence for the applicability of MSL.

**Research Contribution (RC) 6:** A mode transformation technique for improving mode-switch efficiency at runtime.

As a distributed runtime mechanism, the MSRM of MSL has two unde-
sirable side effects: additional runtime overhead imposed on each component and long mode-switch time due to inter-component communication. When component mode information is globally accessible, the mode transformation technique is able to transform the distributed mode-switch handling of MSL to a centralized solution, thus improving its runtime mode-switch efficiency in terms of runtime overhead and mode-switch time, while still retaining the design-time advantages of MSL.

**Research Contribution (RC) 7: Two proof-of-concept case studies of MSL—An Adaptive Cruise Control (ACC) system and a healthcare monitoring system.**

An ACC [2] system is a common vehicle subsystem for autonomous speed maintenance. Many existing ACC systems are developed as multi-mode systems without multi-mode components. A healthcare monitoring system is used to monitor the health condition of a patient from a health centre. To demonstrate the applicability of MSL, we design both systems by the use of reusable multi-mode components.

**Research Contribution (RC) 8: MCORE: the Multi-mode COmponent Reuse Environment**

MCORE is tool for developing multi-mode systems with multi-mode components. A user can develop and reuse multi-mode components conforming to the mode-aware component model of MSL. Mode mappings between components can be manually specified in accordance with the mode mapping mechanism of MSL. Our mode transformation technique has also been implemented, ready to be integrated in MCORE. The initial goal of MCORE is to export multi-mode system models to Rubus ICE [114], an IDE for the Rubus component model [57]. In this way, other tasks such as software testing, analysis, and code generation can be performed in Rubus ICE.

Table 1.1 maps each research contribution to the corresponding research question. All research questions have been covered by our research contributions.

### 1.4 Publications

This thesis stems from a number of publications. This section gives a brief introduction to the key publications contributing to the backbone of the thesis. The complete list of publications can be found in "List of publications" at the
beginning of the thesis.


  **Contribution:** In this paper, an initial overview of our Mode-Switch Logic (MSL) is presented, including the mode-aware component model, the mode mapping problem, and an early version of our MSRM. The paper also includes a comparison between MSL and a number of component models and languages with mode support.


  **Contribution:** This article provides a comprehensive description of the mode-aware component model and the mode mapping mechanism, which are both demonstrated by a proof-of-concept implementation of an Adaptive Cruise Control system. Moreover, it presents a tool that implements MSL, MCORE: the Multi-mode COmponent Reuse Environment.


  **Contribution:** In a multi-mode system with multi-mode components,
multiple mode-switch events could be concurrently triggered by different components, thus complicating the mode-switch handling at runtime. This paper extends our MSRM by supporting mechanisms for handling concurrently triggered mode-switch events.


  **Contribution:** A mode switch can be triggered as either an emergency event or a non-emergency event. An emergency mode switch must be performed as soon as possible due to its high criticality. This paper extends Paper C by supporting mechanisms for handling the concurrent triggering of both emergency and non-emergency mode-switch events. An emergency mode-switch event can be handled swiftly even in the presence of other concurrently triggered non-emergency mode-switch events.


  **Contribution:** This paper provides a mode-switch timing analysis for our MSRM, assuming no concurrent mode-switch events. It also extends the MSRM by supporting atomic execution, an ongoing execution that should not be interrupted by a mode switch. Contiguous components with atomic execution are identified as an atomic execution group, whose worst-case execution time is calculated by model checking.


  **Contribution:** This article refines the mode-switch timing analysis in Paper E. The refined timing analysis is demonstrated by an Adaptive Cruise Control system. In addition, the correctness of the MSRM is proved, under the assumption of no concurrent mode-switch events.

- **Paper G.** Mode switch handling for the ProCom component model. Hang Yin, Hongwan Qin, Jan Carlson, Hans Hansson. In Proceed-
Chapter 1. Introduction

Multiple mode-switch events could be concurrently triggered by different components, thus complicating the mode-switch handling at runtime. This paper extends our MSRM by supporting mechanisms for handling concurrently triggered mode-switch events.


  **Contribution:** A mode switch can be triggered as either an emergency event or a non-emergency event. An emergency mode switch must be performed as soon as possible due to its high criticality. This paper extends Paper C by supporting mechanisms for handling the concurrent triggering of both emergency and non-emergency mode-switch events.


  **Contribution:** This paper provides a mode-switch timing analysis for our MSRM, assuming no concurrent mode-switch events. It also extends the MSRM by supporting atomic execution, an ongoing execution that should not be interrupted by a mode switch. Contiguous components with atomic execution are identified as an atomic execution group, whose worst-case execution time is calculated by model checking.


  **Contribution:** This article refines the mode-switch timing analysis in Paper E. The refined timing analysis is demonstrated by an Adaptive Cruise Control system. In addition, the correctness of the MSRM is proved, under the assumption of no concurrent mode-switch events.


  **Contribution:** This paper serves as a theoretical guidance for integrating MSL in the ProCom component model. The outcome is an extended ProCom model supporting reuse of multi-mode components.


  **Contribution:** This paper presents a mode transformation technique which transforms our distributed mode-switch handling into centralized mode-switch handling when components are deployed on a single physical hardware platform and global mode information is accessible. The MSRM of each component is replaced with a single global mode-switch manager. In this way, a mode switch is simply a direct transition between two system-wide modes, thus improving runtime mode-switch efficiency and reducing mode-switch time.

I am the main contributor and author of all these publications under the supervision of Hans Hansson. The tool MCORE presented in Paper B was developed by Daniele Orlando, Francesco Miscia, and Simone Di Marco while I worked as their supervisor. Paper G is based on the master thesis of Hongwan Qin under my supervision. Jan Carlson has been involved in discussions and has given valuable comments for Paper A and Paper G.

Table 1.2 maps the aforementioned publications to our research contributions. Apparently, RC 3, i.e. the MSRM, has been addressed by most of these papers.

1.5 Research methodology

Our research methodology is essentially in conformity with the generally accepted research strategies in software engineering identified by Shaw [112]. Illustrated in Figure 1.3, our research has been conducted through five major stages:

1. Identifying the general research goal.
Chapter 1. Introduction

Table 1.2: The mapping of key publications and research contributions

<table>
<thead>
<tr>
<th>Papers</th>
<th>RC 1</th>
<th>RC 2</th>
<th>RC 3</th>
<th>RC 4</th>
<th>RC 5</th>
<th>RC 6</th>
<th>RC 7</th>
<th>RC 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paper A</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Paper B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Paper C</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Paper D</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Paper E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Paper F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Paper G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Paper H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

2. Dividing the research goal into a set of research questions.
3. Exploring solutions to one of these research questions.
4. Validating the solutions to ensure technical soundness.
5. Publishing the validated solutions as scientific papers.

As the inception, identification of the general research goal necessitates a comprehensive literature review of related research topics, such as mode switch, CBSE, and adaptive embedded systems. The purpose of literature review is twofold: (1) to get acquainted with state-of-the-art techniques related to our research; and (2) to inspect the novelty of our research topic. Since the research goal is often too general and abstract to be achieved with concrete solutions, following the divide and conquer policy, we divide the research goal into a number of research questions, each research question addressing a specific challenge.

Our research questions identified in Section 1.2 exhibit certain interdependencies. For instance, RQ 3 is dependent on RQ 2, while RQ 4 is dependent on RQ 1 and RQ 2. Therefore, it is sensible to recognize the fundamental research questions and explore the corresponding solutions which are usually theoretical results. Each solution is subjected to verification and validation. We have adopted sundry techniques to verify the correctness of each solution and to evaluate its applicability. To ensure lucid presentation of each solution, we have a multitude of toy examples throughout the thesis for demonstration purpose. Apart from the toy examples, the Adaptive Cruise Control system and healthcare monitoring system, RC 7 in Section 1.3, are more complex and
Dividing the research goal into a set of research questions.

3. Exploring solutions to one of these research questions.

4. Validating the solutions to ensure technical soundness.

5. Publishing the validated solutions as scientific papers.

As the inception, identification of the general research goal necessitates a comprehensive literature review of related research topics, such as mode switch, CBSE, and adaptive embedded systems. The purpose of literature review is twofold: (1) to get acquainted with state-of-the-art techniques related to our research; and (2) to inspect the novelty of our research topic. Since the research goal is often too general and abstract to be achieved with concrete solutions, following the divide and conquer policy, we divide the research goal into a number of research questions, each research question addressing a specific challenge.

Our research questions identified in Section 1.2 exhibit certain interdependencies. For instance, RQ 3 is dependent on RQ 2, while RQ 4 is dependent on RQ 1 and RQ 2. Therefore, it is sensible to recognize the fundamental research questions and explore the corresponding solutions which are usually theoretical results. Each solution is subjected to verification and validation. We have adopted sundry techniques to verify the correctness of each solution and to evaluate its applicability. To ensure lucid presentation of each solution, we have a multitude of toy examples throughout the thesis for demonstration purpose. Apart from the toy examples, the Adaptive Cruise Control system and healthcare monitoring system, RC 7 in Section 1.3, are more complex and realistic examples, though as proof-of-concept case studies. The verification of the MSRM is based on model checking [26] and manual theorem proving. The mode-aware component model, the mode mapping mechanism, and the mode
transformation technique, are implemented by the tool MCORE.

Finding a satisfying solution to a research question is most likely to be an iterative process. Any insidious flaws of a solution discovered at the validation stage will enforce us to go back to the previous stage, extending, revising, or even discarding the solution. A validated solution contributes to one or more research contributions which are subsequently presented in our scientific papers. This dissemination is not only a confirmation of the validity of our research results but also an opportunity to make more people aware of our research. After publishing a scientific paper, we need to review the list of research questions which may be updated, as some remaining research questions could be modified or removed, or new research questions could be introduced. The next selected research question will be treated in the same manner as the previous one.

The cycle from stages 2 to 5 in Figure 1.3 is constantly influenced by miscellaneous internal and external factors. For instance, reviewers’ feedback from a conference or a journal is instrumental in identifying underlying problems that we have overlooked. Fellow researchers at conferences could suggest improvements and relevant literature in neighboring areas. Other sources of inspiration include literature review, discussion with supervisor and other colleagues, PhD courses, summer schools, and industry contact. They are of great importance to the evolution of the research.

1.6 Thesis outline

The rest of the thesis is organized as follows:

- **Chapter 2 — The mode-aware component model:** captures the distinguishing features of a reusable multi-mode component and presents the formal definition of the mode-aware component model for both primitive and composite multi-mode components. This chapter is based on Paper A [48] and Paper B [55].

- **Chapter 3 — Mode mapping:** describes the mode mapping mechanism of MSL for mapping component modes during the composition of multi-mode components. Mode mapping is locally specified at design time between each composite component and its subcomponents. This chapter is based on Paper B [55].

- **Chapter 4 — Mode-switch handling at runtime:** elaborates how a mode switch is handled at runtime without global mode information.
A mode switch is initiated by a component and may imply the mode switches of some other components. These components need to perform mode switch in a cooperative fashion guided by their own MSRM. The MSRM of each component includes a set of algorithms determining its mode-switch behavior and how it communicates with other components. This chapter is based on Paper C [51], Paper D [53], and Paper F [52].

- **Chapter 5 — Verification of the mode-switch runtime mechanism:** verifies the correctness of our MSRM by model checking and manual theorem proving. This chapter is based on Paper C [51] and Paper D [53].

- **Chapter 6 — Mode-switch timing analysis:** performs a mode-switch timing analysis for our MSRM, with the assumption that there is no concurrent triggering of mode-switch events. The timing analysis takes atomic execution into account. A model-checking approach is devised to obtain the worst-case atomic execution time of a group of components with atomic execution. This chapter is based on Paper E [49] and Paper F [52].

- **Chapter 7 — Mode switch for the ProCom component model:** presents the extended ProCom component model which integrates the main principles of MSL. After extension, ProCom is able to incorporate multi-mode components and handle mode switch with distributed runtime mechanisms. This chapter is based on Paper G [56].

- **Chapter 8 — Improving mode-switch efficiency:** introduces our mode transformation technique to improve runtime mode-switch efficiency for MSL. This technique replaces the MSRM of each component with a global mode-switch manager. As a consequence, component modes are transformed into system modes. Mode transformation not only lowers runtime mode-switch overhead, but also shortens mode-switch time. This chapter is based on Paper H [54].

- **Chapter 9 — MCORE: the Multi-mode COMponent Reuse Environment:** describes MCORE, the tool that implements MSL. MCORE is a web-based environment for the model-based design of multi-mode systems, with reuse of multi-mode components as an exclusive feature. A brief description of MCORE is included in Paper D [53].

- **Chapter 10 — Case studies:** demonstrates the applicability of MSL in an Adaptive Cruise Control system and a healthcare monitoring sys-
tem, both of which are designed as multi-mode systems with multi-mode components. This chapter is based on Paper B [55] and Paper F [52].

- **Chapter 11 — Related work:** provides an overview of research relating to mode switch. We have investigated a broad spectrum of research topics such as design and modeling techniques, component models, dynamic component reconfiguration, frameworks, programming and specification languages, dynamic software product lines, and real-time systems.

- **Chapter 12 — Conclusions and future work:** summarizes the thesis and discusses future work.
Chapter 2

The mode-aware component model

Despite the existence of a multitude of component models [31, 67] for specifying reusable software components, multi-mode components have rarely been addressed. A multi-mode component differs from a traditional single-mode component in terms of interface, internal properties, and many other aspects. In this chapter, we propose a mode-aware component model with the formal definition of multi-mode components. We reveal the similarity and discrepancy of primitive components and composite components. Our mode-aware component model is demonstrated by an example at the end of this chapter.

2.1 UML meta-model

Our mode-aware component mode is not intended to be a complete component model. Instead, it defines the unique and essential features of a reusable multi-mode component. In general, a multi-mode component supports multiple modes and has a unique configuration defined for each mode. A mode switch leads to the reconfiguration of a multi-mode component by changing its configuration in the current mode to a new configuration in the target mode. Furthermore, the mode switch of a component may affect other components. A multi-mode component must be able to communicate with other components during a mode switch. Since no component has the global information of the system where it is reused, a component is only allowed to communi-
cate with its parent and subcomponents via dedicated mode-switch ports. The mode-switch behavior of a multi-mode component is determined by a dedicated built-in Mode-Switch Runtime Mechanism (MSRM).

The characteristics of our mode-aware component model is highlighted in a UML meta-model in Figure 2.1. This meta-model does not include the complete specification of all required concepts. Instead, it provides a high-level illustration of the multi-mode components defined by us. Both a multi-mode primitive component and a multi-mode composite component share some common elements, e.g. the MSRM, ports, properties and mode information. A port can be either a functional port, or a dedicated mode-switch port. The property of a component can be either functional (which is related to the functionality of the component) or extra-functional, e.g. resource consumption, timing characteristics and dependability. Indicated in Figure 2.1, a property can be either a Mode-Independent Property (denoted as MIP) or a Mode-Dependent Property (denoted as MDP). The value of an MIP is independent of the mode of the component, whereas the value of an MDP could be different for different modes. The mode information in Figure 2.1 includes a set of supported modes, among which one mode is defined as the initial mode and one is the current mode. For a multi-mode composite component, its mode information also includes its mode mapping, i.e., a mapping between the modes of this composite component and the modes of its subcomponents. In addition, a multi-mode primitive component may have different behaviors for different modes while a multi-mode composite component should know the activated subcomponents and activated inner component connections for each mode. Each subcomponent is an instance of the multi-mode component and each activated connection connects two functional ports from different components.

2.2 The formal specification of a multi-mode primitive component

A multi-mode primitive component is formally defined as follows:

**Definition 2.** Multi-mode primitive component: A multi-mode primitive component $c_i$ is a tuple:

$$< \mathcal{P}, p^{MS}, \tilde{M}, \mathcal{B}, \mathcal{MI}, \mathcal{MD}, MSRM, MB, MP >$$

where $\mathcal{P}$ is the set of ports of $c_i$ used for communicating with other components; $p^{MS} \notin \mathcal{P}$ is a dedicated bidirectional mode-switch port for exchanging
2.2 The formal specification of a multi-mode primitive component

A multi-mode primitive component is formally defined as follows:

Definition 2. Multi-mode primitive component: A multi-mode primitive component $c_i$ is a tuple:

$$<P, p_{MS}, \tilde{\mathcal{M}}, B, \mathcal{MI}, \mathcal{MD}, \text{MSRM}, \mathcal{MB}, \mathcal{MP} >$$

where $P$ is the set of ports of $c_i$ used for communicating with other components; $p_{MS} \in P$ is a dedicated bidirectional mode-switch port for exchanging mode-related information with the parent of $c_i$; $\tilde{\mathcal{M}}$ is the set of mode-related information of $c_i$ (including $\mathcal{M}$, the set of supported modes of $c_i$); $B$ is the set of mode-specific behaviors of $c_i$; $\mathcal{MI}$ is the set of mode-independent properties of $c_i$; $\mathcal{MD}$ is the set of mode-dependent properties of $c_i$; $\text{MSRM}$ is the Mode-Switch Runtime Mechanism of $c_i$; the function $\mathcal{MB} : \mathcal{M} \rightarrow B$ maps the functional behaviors of $c_i$ to the corresponding modes; the function $\mathcal{MP} : \mathcal{M} \times \mathcal{MD} \rightarrow Q$ assigns values to each mode-dependent property of $c_i$ in each mode, where the set $Q = \bigcup_{e \in \mathcal{MD}} \text{dom}(e)$ is the union of the domains (i.e. range of values) of all mode-dependent properties such that for each mode $m$. 
and mode-dependent property $e$, $MP(m, e) \in \text{dom}(e)$, with $\text{dom}(e)$ denoting the domain of $e$.

Since $p^{MS}$ is dedicated to mode switch, we separate it from $P$. Besides,

**Definition 3.** The set of supported modes $\tilde{M}$ of a multi-mode primitive component $c_i$ is another tuple:

$$< M, m^0, m >$$

where $M$ as explained above is the set of supported modes of $c_i$; $m^0 \in M$ is the initial mode of $c_i$; $m \in M$ is the current mode of $c_i$.

Our formal specification of a multi-mode primitive component is illustrated in Figure 2.2. Just like a single-mode component, the component has a number of input ports $p^0_{in} \cdots p^n_{in}$ and output ports $p^0_{out} \cdots p^n_{out}$ ($m, n \in \mathbb{N}$) to communicate with its neighboring components. Apart from these ports, the component also has a dedicated mode-switch port $p^{MS}$ marked in the blue square in Figure 2.2. It has unique configurations defined for each mode as well as some mode-independent properties. Its mode-switch behavior is controlled by its MSRM.

![Figure 2.2: The mode-aware component model of a primitive component](image)

Note that the input ports and output ports in Figure 2.2 correspond to the *pipes and filters* architectural style [45] of the system composed of our multi-mode components. Our mode-aware component model has no specific component execution semantics because component execution semantics is irrelevant to mode. Hence we only categorize the functional ports of a component into
2.3 The formal specification of a multi-mode composite component

Compared with a multi-mode primitive component, the formal specification of a multi-mode composite component is more complex, since a composite component must know the essential information of itself, its subcomponents and how they are composed.

Definition 4. **Multi-mode composite component**: A multi-mode composite component \( c_i \) is a tuple:

\[
<\mathcal{P}, p^\text{MS}, p^\text{in}^\text{MS}, \bar{\mathcal{M}}, \mathcal{SC}, \mathcal{CN}, \mathcal{MI}, \\
\mathcal{MD}, \text{MSRM}, \text{ASC}, \text{ACN}, \text{MP} >
\]

where \( \mathcal{P}, p^\text{MS}, \mathcal{MI}, \mathcal{MD}, \text{MSRM} \) and \( \text{MP} \) are defined as in Definition 2; \( p^\text{in}^\text{MS} \notin \mathcal{P} \) is a second bidirectional dedicated mode-switch port for exchanging mode-related information with the subcomponents of \( c_i \); \( \bar{\mathcal{M}} \) is the set of mode-related information of \( c_i \); \( \mathcal{SC} \) is the set of the subcomponents of \( c_i \); \( \mathcal{CN} \subseteq (\mathcal{P}_{\mathcal{SC}} \cup \mathcal{P}) \times (\mathcal{P}_{\mathcal{SC}} \cup \mathcal{P}) \) is the set of all inner component connections of \( c_i \) in all modes, where \( \mathcal{P}_{\mathcal{SC}} \) is the set of ports of \( \mathcal{SC} \):

\[
\mathcal{P}_{\mathcal{SC}} = \bigcup_{c_j \in \mathcal{SC}} \mathcal{P}_{c_j};
\]

the function \( \text{ASC} : \mathcal{M} \rightarrow 2^{\mathcal{SC}} \) indicates the activated subcomponents of \( c_i \) in each mode; the function \( \text{ACN} : \mathcal{M} \rightarrow 2^{\mathcal{CN}} \) defines the set of activated inner component connections (connections in use) of \( c_i \) in each mode.

Two dedicated mode-switch ports are defined for a composite component because a composite component must be able to communicate with both its parent and children during a mode switch. Note that a subcomponent behaves as a deactivated subcomponent when all its connections with neighboring components are deactivated. Hence the function \( \text{ASC} \) can be inferred from the function \( \text{ACN} \). However, we explicitly define \( \text{ASC} \) for two reasons: (1) Knowing the activated subcomponents is essential for the mode-switch handling of each
composite component at runtime; (2) A subcomponent can be activated even with all deactivated connections; the power consumption of the system could potentially be reduced by deactivating such a component. Besides,

**Definition 5.** The set of supported modes $\tilde{M}$ of a multi-mode composite component $c_i$ is another tuple:

$$< M, m^0, m, M_{SC}, m^0_{SC}, m_{SC}, MM >$$

where $M$ is the set of supported modes of $c_i$, with $m^0 \in M$ being the initial mode of $c_i$ and $m \in M$ being the current mode of $c_i$; the function $M_{SC} : SC \rightarrow 2^M$ maps each subcomponent of $c_i$ to the corresponding set of supported modes ($\overline{M}$ is the set of all modes supported by the subcomponents of $c_i$); the function $m^0_{SC} : SC \rightarrow 2^{\overline{M}}$ maps each subcomponent of $c_i$ to the corresponding initial mode; the function $m_{SC} : SC \rightarrow 2^M$ maps each subcomponent of $c_i$ to the corresponding current mode; $MM$ is the mode mapping between $c_i$ and its subcomponents.

Figure 2.3 illustrates the mode-aware component model of a composite component. It differs from a multi-mode primitive component in three aspects:

1. Two dedicated mode-switch ports ($p^MS$ and $p_{in}^{MS}$) are used instead of one.

2. The configuration for each mode is specified differently. Since a multi-mode composite component is a composition of other components, it has no specific behavior. Instead, the activated subcomponents and activated inner component connections of a multi-mode composite component can be mode-dependent.

3. The mode mapping between a composite component and its subcomponents is only present for composite components.

### 2.4 An example

To ease the apprehension of our mode-aware component model, let’s provide a formal definition for the primitive component $b$ and the composite component $c$ in Figure 1.2. Figure 2.4 displays the inner component connections of $a$ for different modes and labels each component with its port names. Suppose
that the set of supported modes of \( b \) is denoted by \( b.\mathcal{M} = \{m_1^b, m_2^b\} \) such that \( b \) runs in \( m_1^b \) when \( a \) runs in mode \( m_1^a \) and \( b \) runs in \( m_2^b \) when \( a \) runs in \( m_2^a \). The black and grey colors of \( b \) represent two different mode-specific behaviors, which for instance can be two different video decoding schemes for a multimedia application. Let \( \alpha \) and \( \beta \) denote the behaviors of \( b \) for modes \( m_1^b \) and \( m_2^b \) respectively. Component \( b \) has a mode-independent property, CPU consumption (denoted by \( \text{cpu} \)) that is the same for both modes, e.g. \( \text{cpu} = 8 \), and \( b \) also has a mode-dependent property, memory consumption (denoted by \( \text{mem} \)) that is different for different modes, e.g. \( \text{mem} = 5 \) for mode \( m_1^b \) and \( \text{mem} = 10 \) for \( m_2^b \). Then \( b \) can be formally defined by the tuple,

\[
< b.\mathcal{P}, b.p^{MS}, b.\widetilde{\mathcal{M}}, b.\mathcal{B}, b.\mathcal{M}^I, b.\mathcal{D}, b.\mathcal{MSRM}, b.\mathcal{M}D, b.\mathcal{MP} >
\]

where \( b.\mathcal{MSRM} \) can be considered as a dedicated mode-switch algorithm for \( b \) and

\[
\begin{align*}
 b.\mathcal{P} &= \{b.p_{\text{in}}^0, b.p_{\text{out}}^0\} \\
 b.\mathcal{B} &= \{\alpha, \beta\} \\
 b.\mathcal{M}^I &= \{\text{cpu} = 8\} \\
 b.\mathcal{M}D &= \{\text{mem}\} \\
 b.\mathcal{M}B &= \{m_1^b \rightarrow \alpha, m_2^b \rightarrow \beta\} \\
 b.\mathcal{M}P &= \{(m_1^b, \text{mem}) \rightarrow 5, (m_2^b, \text{mem}) \rightarrow 10\} \\
 b.\widetilde{\mathcal{M}} &= < b.\mathcal{M}, b.m^0, b.m >
\end{align*}
\]

where
Figure 2.4: Port name marking for components $a$, $b$, $c$

Figure 2.5: Port name marking for components $c$, $e$, $f$

Figure 2.5 marks the port names of the composite component $c$ and its subcomponents $e$ and $f$. Suppose that the set of supported modes of $c$ is
\(c.M = \{m_1^c, m_2^c, m_3^c\}\). Component \(c\) has Worst-Case Execution Time (denoted as \(wcet\)) as \(c.MD\) such that \(wcet = 70\) for \(m_1^c\), \(wcet = 75\) for \(m_2^c\), and \(wcet = 50\) for \(m_3^c\). Also, \(c\) has a mode-independent property, i.e. its activation period \(T = 100\) for all modes. As a composite component, \(c\) can be formally defined by the tuple,

\[
< c.P, c.p^{MS}, c.p^M_{in}, c.M, c.SC, c.CN, c.MI, c.MD, c.MSRM, c.ASC, c.ACN, c.MP >
\]

where

\[
c.P = \{ c.p^0_{in}, c.p^0_{out}, c.p^1_{out} \}
\]

\[
c.SC = \{ e, f \}
\]

\[
c.CN = \{ (c.p^0_{in}, e.p^0_{in}), (e.p^0_{out}, f.p^0_{in}), (e.p^1_{out}, c.p^0_{out}), (f.p^0_{out}, c.p^0_{out}), (c.p^1_{in}, f.p^1_{in}) \}
\]

\[
c.MI = \{ T = 100 \}
\]

\[
c.MD = \{ wcet \}
\]

\[
c.ASC = \{ m_1^c \rightarrow \{ e, f \}, m_2^c \rightarrow \{ e, f \}, m_3^c \rightarrow \{ f \} \}
\]

\[
c.ACN = \{ m_1^c \rightarrow \{ (c.p^0_{in}, e.p^0_{in}), (e.p^0_{out}, f.p^0_{in}), (e.p^1_{out}, c.p^0_{out}), (f.p^0_{out}, c.p^0_{out}) \},
\]

\[
m_2^c \rightarrow \{ (c.p^0_{in}, e.p^0_{in}), (e.p^0_{out}, f.p^0_{in}), (e.p^1_{out}, c.p^1_{out}), (f.p^0_{out}, c.p^0_{out}) \},
\]

\[
m_3^c \rightarrow \{ (c.p^1_{in}, f.p^1_{in}), (f.p^0_{out}, c.p^1_{out}) \} \}
\]

\[
c.MP = \{ (m_1^c, wcet) \rightarrow 70, (m_2^c, wcet) \rightarrow 75, (m_3^c, wcet) \rightarrow 50 \}
\]

\[
c.M = < c.M, c.m^0, c.m, c.MSC, c.m^0_{SC}, c.m_{SC}, c.MM >
\]

where \(c.MM\) presents the mode mapping of \(c\), and

\[
c.M = \{ m_1^c, m_2^c, m_3^c \}
\]

\[
c.m^0 = m_1^c
\]

\[
c.MSC = \{ e \rightarrow \{ m_1^c, m_2^c, m_3^c \}, f \rightarrow \{ m_1^f \} \}
\]

\[
c.m^0_{SC} = \{ e \rightarrow m_1^e, f \rightarrow m_1^f \}
\]

In the definition of \(c.CN\), each pair "(x, y)" represents a connection from port \(x\) to port \(y\). For instance, \((c.p^0_{in}, e.p^0_{in})\) defines the connection from port \(p^0_{in}\) of \(c\) to port \(p^0_{in}\) of \(e\). The definition above covers not only the mode information of \(c\) itself, but also the mode information of its subcomponents \(e\) and \(f\). For instance, \(e\) supports three modes \(m_1^e\) (the initial mode), \(m_2^e\) and \(m_3^e\).
2.5 Summary

In this chapter, we have presented a mode-aware component model for both primitive and composite components. Each multi-mode component has a unique configuration for each of its supported modes and can reconfigure itself during a mode switch. Dedicated mode-switch ports are introduced for each component to exchange mode information with its parent or subcomponents. The mode-switch behavior of a component is controlled by its own MSRM. Rather than serving as a complete component model, our mode-aware component model emphasizes the mode related features of a multi-mode component, with potential to extend many existing component models for supporting reuse of multi-mode components.
Chapter 3

Mode mapping

Since reusable multi-mode components are independently developed, they typically support different number of modes and name them differently. For that reason, the composition of multi-mode components calls for mode mapping, which establishes the relationship between the modes of a composite component and the modes of its subcomponents. According to our mode-aware component model presented in Chapter 2, each composite component possesses a mode mapping mechanism. This chapter begins with a motivating example showing the importance of mode mapping. Thereafter, we elaborate on our mode mapping mechanism which describes the mapping by Mode Mapping Automata (MMAs), followed by the formal definition of MMA composition.

3.1 A motivating example

Let’s revisit the multi-mode system in Figure 1.2, where two multi-mode components $e$ and $f$ are used to compose $c$, which is further used to compose $a$ together with $b$ and $d$. Table 3.1 lists the expected modes of $a$ and $c$, and the supported modes of the other components. Note that Table 3.1 only lists the selected modes of each component within this particular context. Some components may support more modes which are not displayed in Table 3.1. For instance, Component $d$ may originally support two modes $m^1_d$ and $m^2_d$, whereas only $m^1_d$ is used for composing $a$. After composition, the modes of these components need to be correlated without ambiguity. Such a specification is called mode mapping. Mode mapping must never violate the following principles:
• A primitive component knows its own mode information (supported modes, initial mode and current mode, and the transitions between its modes), but knows nothing about the other components in the system.

• A composite component knows the mode information of itself and its subcomponents, but knows nothing about the other components in the system.

<table>
<thead>
<tr>
<th>Component</th>
<th>Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$m_a^1, m_a^2$</td>
</tr>
<tr>
<td>b</td>
<td>$m_b^1, m_b^2$</td>
</tr>
<tr>
<td>c</td>
<td>$m_c^1, m_c^2, m_c^3$</td>
</tr>
<tr>
<td>d</td>
<td>$m_d^1$</td>
</tr>
<tr>
<td>e</td>
<td>$m_e^1, m_e^2, m_e^3, m_e^4$</td>
</tr>
<tr>
<td>f</td>
<td>$m_f^1$</td>
</tr>
</tbody>
</table>

Table 3.1: Modes to be mapped during component composition

Hence, it is the responsibility of each composite component to map its own modes to the modes of its subcomponents. As an example, tables 3.2 and 3.3 present the basic mode mappings within $a$ and $c$, respectively, in accordance with the inner component connections of $a$ and $c$ given in figures 2.4 and 2.5 in Chapter 2. Modes in the same column are mapped to each other. For instance, in Table 3.2, when $a$ runs in $m_a^1$, among its subcomponents, $b$ runs in $m_b^1$, $c$ can run in either $m_c^1$ or $m_c^3$, and $d$ is deactivated, i.e. not running in any of its modes. Note that $e$ can run in either $m_e^2$ or $m_e^3$ when $c$ runs in $m_c^2$. This phenomenon can be observed by the mode-specific behaviors of $e$ represented by black and grey colors in Figure 3.1. However, the local configuration of $c$ remains unchanged due to the same inner component connections of $c$ when $e$ runs in $m_e^2$ or $m_e^3$.

Basically, tables 3.2 and 3.3 present a set of mode mapping rules that can be specified at design time. However, there are some other mode mapping rules which are beyond the description of both tables. For example, when $a$ switches from $m_a^2$ to $m_a^1$, according to the mode mapping in Table 3.2, the target mode of $c$ can be either $m_c^1$ or $m_c^3$. To eliminate such non-determinism, one of $m_c^1$ and $m_c^3$ must be specified as the default target mode of $c$. To allow the specification of this and other more general mode mappings, we have come up with a more powerful presentation—Mode Mapping Automata (MMAs) for the formal specification of all types of mode mapping rules.
Chapter 3. Mode mapping

3.2 Mode mapping automata

Let $c_i$ be a composite component with $SC_{c_i}$ being the set of subcomponents of $c_i$ and $P_{c_i}$ being the parent of $c_i$. Shown in Figure 3.2, the mode mapping of $c_i$ interacts with its MSRM. When $c_i$ is running in one of its supported modes, it should always know its current mode and the current modes of all $c_j \in SC_{c_i}$.
by its mode mapping, which can be presented by mode mapping tables such as Table 3.2. Moreover, whenever the MSRM of \( c_i \) notices the mode switch of \( c_j \in SC_{c_i} \cup \{ c_i \} \), it will refer to the mode mapping that should tell which other components among \( SC_{c_i} \cup \{ c_i \} \setminus \{ c_j \} \) should also switch mode as a consequence, and the new modes of these components. The complete mode mapping of \( c_i \) can be formally presented by a set of MMAs, which consist of

- One Mode Mapping Automaton (MMA) of \( c_i \), denoted as \( MMA^{p}_{c_i} \);
- One MMA of each subcomponent \( c_j \in SC_{c_i} \), denoted as \( MMA^{c}_{c_j} \).

Since \( c_i \) is the parent, we call \( MMA^{p}_{c_i} \) a parent MMA. Likewise, \( MMA^{c}_{c_j} \) is called a child MMA.

Each MMA of a composite component \( c_i \) can receive and emit internal or external signals. Internal signals are used to synchronize the pair of the parent MMA and its child MMAs while external signals interact with the MSRM of \( c_i \) for requesting and returning mode mapping results. Referring to Chapter 2, \( c_i \) can exchange mode-related information with its parent and subcomponents via its dedicated mode-switch ports. This is achieved by sending and receiving primitives, as will be explained in the next chapter. When \( c_i \) receives a primitive from the parent, being requested to switch mode, the MSRM of \( c_i \) will send an external signal to the parent MMA of \( c_i \). Conversely, when the parent MMA of \( c_i \) emits an external signal to the MSRM of \( c_i \), \( c_i \) will send a primitive to its parent. In a similar manner, when \( c_i \) receives a primitive from a subcomponent which requests to switch mode, the MSRM of \( c_i \) will send an
external signal to the corresponding child MMA. When a child MMA emits an external signal to the MSRM of $c_i$, $c_i$ will send a primitive to the corresponding subcomponent.

Next let’s use the example in Figure 1.2 to illustrate the internal structure of a set of MMAs and the interaction between the MSRM and mode mapping of a component. Figure 3.3 presents the set of MMAs of Component $c$ in Figure 1.2, including a parent MMA $MMA_p^c$ and two child MMAs $MMA_e^c$ and $MMA_f^c$. These MMAs are hierarchically organized in the same way as the corresponding components and are internally synchronized with each other by internal signals. The communication between the MSRM of $c$ and these MMAs is based on external signals.

![Figure 3.3: The role of the mode mapping of $c$ at runtime](image)

The following is the formal definition of an MMA, which applies to both parent MMA and child MMA:

**Definition 6. Mode Mapping Automaton:** An MMA is defined as a tuple:

$$< S, s^0, SI, expr(BV), T >$$

where $S$ is a set of states; $s^0 \in S$ is the initial state; $SI = I \cup E$ ($I \cap E = \emptyset$) is a set of signals received or emitted during a state transition, with $I$ as the set of internal signals and $E$ as the set of external signals; $BV$ is a set of application-dependent variables and $expr(BV)$ is a set of boolean expressions over $BV$; $T \subseteq S \times SI \times expr(BV) \times 2^{SI} \times S$ is a set of transitions of the MMA, where $expr(BV)$ must evaluate to true to enable a transition.
Apart from the formal definition above, we also provide the graphical representation of an MMA:

**Definition 7. MMA graphical representation:** An MMA can be graphically represented as a state machine with states and transitions. Each state is a location represented by a circle, with the initial state being marked by a double circle. If the MMA is a parent MMA of $c_i$, then each state corresponds to a mode of $c_i$. If the MMA is a child MMA of $c_i$ associated with $c_j \in SC_{c_i}$, then each state corresponds to a mode of $c_j$ or the deactivated status of $c_j$ (if $c_j$ can be deactivated), denoted as $D$. A transition $t \in \mathcal{T}$ is represented by an arrow from a state $s$ to a state $s'$, denoted as $s \xrightarrow{\text{In}/\text{Out}} s'$, where In/Out is the label of the transition, "In" is the input that triggers the transition, and "Out" is the output of the transition.

Definition 7 is illustrated in Figure 3.4, which depicts $MMA^p_i$ in Figure 3.3. Three states are included in this MMA, implying that $c_i$ can run in three modes. The state transitions of $MMA^p_i$ and the child MMAs $MMA^c_j$ and $MMA^f_j$ are manually specified to determine the mode mapping of $c$. Shown in Figure 3.4, each transition has a label with input and output separated by "/". The input and output of a transition are typically internal and external signals. We define two types of internal signals and a single type of external signal denoted as follows:

- $x.I(y)$, an internal signal emitted by a parent MMA to the recipient $MMA^c_x$, which is asked to change location to $y$. Here $x$ is a component associated with a child MMA, and $y$ is a mode of $x$, or the deactivated status of $x$, denoted as $D$.

- $x.I(y \rightarrow z)$, an internal signal emitted by $MMA^c_x$ which requests to change location from $y$ to $z$. Here $x$ is the component associated with the child MMA $MMA^c_x$, and $y$ and $z$ are the modes of $x$.

- $x.E(y)$, an external signal requesting $MMA_x$ to change location to $y$, where $MMA_x$ is either a parent MMA or a child MMA. Here $x$ is a component, and $y$ is a mode of $x$.

Using the notations above, we specify the MMA synchronization semantics:

**Definition 8. MMA synchronization semantics:** Let $c_i$ be a composite component with $SC_{c_i} = \{c^1_i, c^2_i, \ldots, c^n_i\}$ ($n \in \mathbb{N}$). Let $\mathcal{A} = \{A_0, A_1, \ldots, A_n\}$ be the set of MMAs of $c_i$, with $A_0$ associated with $c_i$ and $A_h$ associated with
3.2 Mode mapping automata

Each state corresponds to a mode of output of the transition. If the MMA is a parent MMA of location represented by a circle, with the initial state being marked by a double circle. If the MMA is a child MMA of another MMA, then each state corresponds to a mode of output of the transition. "In" is the input that triggers the transition, and "Out" is the label with input and output separated by "\(/\)." The state transitions of MMA are typically internal and external signals. We define the MMA synchronization of MMA: any external signal from the MSRM of MMA at runtime can potentially lead to the MMA synchronization of MMA. An MMA synchronization is an atomic transaction which is always performed between MMA and a child MMA. The synchronization depends on if the external signal arrives at MMA or at a child MMA:

1. An external signal arrives at MMA asking MMA to switch to mode MMA. This triggers a transition of MMA with the input MMA, giving rise to two possible subsequent behaviors:

- **The mode switch of MMA under this condition does not imply the mode switch of any subcomponent of MMA.** Then the output of the transition should be an empty set $\emptyset$, i.e. there is no internal synchronization between the set of MMAs of MMA and no external signals are expected to be emitted from them to the MSRM of MMA.

- **The mode switch of MMA under this condition implies the mode switch of at least one of its subcomponents.** Let $G \subseteq SC_{c_i}$ be the set of its subcomponents which also need to switch mode. Then the output of $t$ should be a set $R$ such that for each $c_j \in G$ which are expected to switch to the mode MMA, there must exist an internal signal MMA.E(M MMA) MMA, synchronizing $t$ with the transition MMA with the label MMA.E(M MMA) MMA. This implies that the MMA in Figure 3.4: The parent MMA of MMA can run in three modes. Hence MMA is the parent MMA while the others are child MMAs.

Figure 3.4: The parent MMA of MMA.

$c_j$ ($k \in [1, n]$). Hence MMA is the parent MMA while the others are child MMAs. Any external signal from the MSRM of MMA at runtime can potentially lead to the MMA synchronization of MMA. An MMA synchronization is an atomic transaction which is always performed between MMA and one or several child MMAs. The synchronization depends on if the external signal arrives at MMA or at a child MMA:

1. An external signal arrives at MMA asking MMA to switch to mode MMA. This triggers a transition of MMA with the input MMA, giving rise to two possible subsequent behaviors:

- **The mode switch of MMA under this condition does not imply the mode switch of any subcomponent of MMA.** Then the output of the transition should be an empty set $\emptyset$, i.e. there is no internal synchronization between the set of MMAs of MMA and no external signals are expected to be emitted from them to the MSRM of MMA.

- **The mode switch of MMA under this condition implies the mode switch of at least one of its subcomponents.** Let $G \subseteq SC_{c_i}$ be the set of its subcomponents which also need to switch mode. Then the output of $t$ should be a set $R$ such that for each MMA $c_j \in G$ which are expected to switch to the mode MMA, there must exist an internal signal MMA.E(M MMA) MMA, synchronizing $t$ with the transition MMA with the label MMA.E(M MMA) MMA. This implies that the MMA in Figure 3.4: The parent MMA of MMA can run in three modes. Hence MMA is the parent MMA while the others are child MMAs. Any external signal from the MSRM of MMA at runtime can potentially lead to the MMA synchronization of MMA. An MMA synchronization is an atomic transaction which is always performed between MMA and one or several child MMAs. The synchronization depends on if the external signal arrives at MMA or at a child MMA:

1. An external signal arrives at MMA asking MMA to switch to mode MMA. This triggers a transition of MMA with the input MMA, giving rise to two possible subsequent behaviors:

- **The mode switch of MMA under this condition does not imply the mode switch of any subcomponent of MMA.** Then the output of the transition should be an empty set $\emptyset$, i.e. there is no internal synchronization between the set of MMAs of MMA and no external signals are expected to be emitted from them to the MSRM of MMA.

- **The mode switch of MMA under this condition implies the mode switch of at least one of its subcomponents.** Let $G \subseteq SC_{c_i}$ be the set of its subcomponents which also need to switch mode. Then the output of $t$ should be a set $R$ such that for each MMA $c_j \in G$ which are expected to switch to the mode MMA, there must exist an internal signal MMA.E(M MMA) MMA, synchronizing $t$ with the transition MMA with the label MMA.E(M MMA) MMA. This implies that the MMA in Figure 3.4: The parent MMA of MMA can run in three modes. Hence MMA is the parent MMA while the others are child MMAs. Any external signal from the MSRM of MMA at runtime can potentially lead to the MMA synchronization of MMA. An MMA synchronization is an atomic transaction which is always performed between MMA and one or several child MMAs. The synchronization depends on if the external signal arrives at MMA or at a child MMA:

1. An external signal arrives at MMA asking MMA to switch to mode MMA. This triggers a transition of MMA with the input MMA, giving rise to two possible subsequent behaviors:
MMA $A_j$. The output $c_j.E(m_{c_i})$ of $t_j$ is sent from $A_j$ to the MSRM of $c_i$ which will send a primitive to $c_j$.

(2) An external signal arrives at $A_k$ ($k \in [1,n]$) requesting $c^k_j$ to switch from mode $m_1$ to $m_2$. This triggers a transition $t_k$ of $A_k$ with the label $c_j.E(m_2)/c_j.I(m_1 \rightarrow m_2)$, synchronized with the transition $t$ of $A_0$ with the input $c_j.I(m_1 \rightarrow m_2)$. The output of $t$ may give rise to four possible subsequent behaviors:

- The mode switch of $c^k_j$ under this condition does not imply the mode switch of any component among $c_i$ and $SC_{c_i} \setminus \{c^k_j\}$. Then the output of $t$ is $\emptyset$ and no further synchronization and external signal are expected.

- The mode switch of $c^k_j$ under this condition implies the mode switch of $c_i$ but not the mode switch of any other subcomponent of $c_i$. Let $m_{c_i}$ be the new mode of $c_i$. Then the output of $t$ is $c_i.E(m_{c_i})$.

- The mode switch of $c^k_j$ under this condition implies the mode switch of at least another subcomponent of $c_i$ but not the mode switch of $c_i$. Then the subsequent behavior will be the same as the second case under Condition (1). Hence the output of $t$ is the set $R$.

- The mode switch of $c^k_j$ under this condition implies the mode switches of both $c_i$ and at least another subcomponent of $c_i$. Let $m_{c_i}$ be the new mode of $c_i$. Then the output of $t$ will be $\{c_i.E(m_{c_i})\} \cup R$, where $c_i.E(m_{c_i})$ is an external signal emitted from $A_0$.

Each transition of $A_0$ can optionally associate its input with a guard $\lambda \in expr(BV)$ of $A_0$. Thereby different outputs can be expected for the same input signal of a transition of $A_0$ depending on the evaluation of the guards.

Our MMA synchronization semantics is partially illustrated by figures 3.4 and 3.5 which provide the graphical presentations for $MMA^p_c$ and $MMA^p_e$ in Figure 3.3, respectively. We exclude the graphical presentation of $MMA^f_j$ because $f$ always runs in mode $m_1$ and hence $MMA^f_j$ contains only a single state without transitions. Let’s use one scenario to demonstrate the synchronization between $MMA^p_c$ and $MMA^p_e$. Suppose that $c$ needs to switch from $m^1_c$ to $m^2_c$ and the current mode of $e$ is $m^1_e$. Then the MSRM of $c$ is supposed to send an external signal $c.E(m^2_c)$ to $MMA^p_c$, triggering one of the two transitions of $MMA^p_c$ from $m^1_c$ to $m^2_c$. These two transitions are associated with two boolean expressions guard 1 and guard 2 which can be manually
3.2 Mode mapping automata

defined based on the desired system functionality. When \( \text{guard} \) 1 is true, the synchronization between the transition \( m^1_c \xrightarrow{e.E(m^2_e) \& \& \text{guard} \ 1} m^2_c \) of \( \text{MMA}^p_e \) and the transition \( m^1_c \xrightarrow{e.I(m^2_e)} m^2_c \) of \( \text{MMA}^c_e \) is enabled. As a consequence, \( \text{MMA}^c_e \) will send the external signal \( e.E(m^2_e) \) to the MSRM of \( e \), which will send a primitive to \( e \), requesting \( e \) to switch to mode \( m^2_c \). Similarly, when \( \text{guard} \) 2 is true, the synchronization between the transition \( m^1_e \xrightarrow{e.E(m^2_e) \& \& \text{guard} \ 2} m^2_e \) of \( \text{MMA}^p_e \) and the transition \( m^1_e \xrightarrow{e.I(m^2_e)} m^3_e \) of \( \text{MMA}^c_e \) is enabled.

![Figure 3.5: The child MMA of e](image)

To simplify the graphical presentation of a parent MMA, if multiple transitions share the same output, starting and ending locations, we combine them into one transition where their different inputs are separated by the notation "\( || \)". For instance, \( \text{MMA}^p_c \) in Figure 3.4 has two transitions \( m^1_c \xrightarrow{e.I(m^1_e \rightarrow m^2_e) \& \& \{e.E(m^2_e)\}} m^2_c \) and \( m^1_c \xrightarrow{e.I(m^1_e \rightarrow m^2_e) \& \& \{e.E(m^2_e)\}} m^2_e \), which are combined into one transition \( m^1_c \xrightarrow{e.I(m^1_e \rightarrow m^2_e) \& \& \{e.E(m^2_e)\}} m^2_e \).

Similarly, figures 3.6-3.9 display the set of MMAs of \( a \) which replace Table 3.2, including \( \text{MMA}^p_a \), \( \text{MMA}^c_a \), \( \text{MMA}^p_b \) and \( \text{MMA}^c_b \). What deserves particular attention is the distinction between \( \text{MMA}^p_e \) in Figure 3.4 and \( \text{MMA}^c_e \) in Figure 3.8. The former is a parent MMA included in the mode mapping of
c, while the latter is a child MMA included in the mode mapping of a. Moreover, the MMAs described for this example have taken all possible cases into account, assuming that an external signal can arrive at any MMA and request the corresponding component to switch to any possible mode. In reality, these MMAs could be substantially simplified if the arrival of the external signal is restricted according to some application-specific requirements. For instance, if $c.E(m^2_c)\$ may never arrive at $\text{MMA}^p_c$ when $c$ is in $m^3_c$, then the transition $m^3_c \xrightarrow{c.E(m^2_c)/e.I(m^2_e)} m^2_e$ of $\text{MMA}^p_c$ in Figure 3.4 can be removed.

**Figure 3.6:** The parent MMA of $a$

**Figure 3.7:** The child MMA of $b$

### 3.3 MMA composition

Section 3.2 shows that the mode mapping of a composite component $c_i$ can be formally defined by a set of MMAs. However, the internal synchronization
3.3 MMA composition

between these MMAs is invisible to the MSRM of $c_i$, which only sees the single composition of the set of MMAs. In this section, we present MMA composition. For each composite component, the basic idea of deriving its MMA composition rules is to analyze the possible outputs of a set of MMAs for each possible input from the MSRM. By Definition 8 in Section 3.2, an external signal from the MSRM of a composite component may give rise to six possible subsequent behaviors with respect to the internal MMA synchronization and the output of the set of MMAs. All these six cases must be taken into account in defining MMA composition.

In the below definition of MMA composition, we assume that the elements of sets are indexed such that we by the indexing can identify the MMA that a specific element is related to, and $x[i]$ will be used to denote the element of $x$ indexed with $i$. We will furthermore use $\biguplus$ to denote a flattening union defined as the set containing all primitive elements of its operands, e.g. $\{a, \{b\}\} \biguplus \{c, \{d\}\} = \{a, b, c, d\}$.

**Definition 9. MMA composition:** For a set $A = \{A_0, A_1, \ldots, A_n\} (n \in \mathbb{N})$
of MMAs, where \( A_0 = < S_0, s^0_0, SI_0, expr(BV_0), \mathcal{T}_0 > \) corresponds to a parent MMA and \( \forall k \in [1, n], A_k = < S_k, s^0_k, SI_k, expr(BV_k), \mathcal{T}_k > \) correspond to the child MMAs synchronized with \( A_0 \), the MMA composition of \( A \) is an MMA defined by the tuple

\[
< S, s^0, SI, expr(BV), \mathcal{T} >
\]

where

\[
S \subseteq S_0 \times S_1 \times \cdots \times S_n
\]

\[
s^0 = (s^0_0, s^0_1, \cdots, s^0_n)
\]

\[
SI \subseteq \bigcup_{i \in [0,n]} E_i
\]

\[
expr(BV) = expr(BV_0)
\]

In defining \( \mathcal{T} \), two cases are considered depending on if an external signal arrives at the parent MMA \( A_0 \) or at a child MMA \( A_i \) (\( i \in [1, n] \)):

1. An external signal \( e_0 \in E_0 \) arrives at \( A_0 \). If

\[
\exists s = (s_0, s_1, \cdots, s_n) \in S \land s_0 \xrightarrow{e_0 \& \& guard/\mathcal{O}_0} s'_0 \in \mathcal{T}_0
\]

where \( guard \in expr(BV_0) \) evaluates to true in state \( s_0 \), then

\[
s \xrightarrow{e_0 \& \& guard/\mathcal{O}} s' \in \mathcal{T} \land s' = (s'_0, s'_1, \cdots, s'_n) \in S
\]

where \( \mathcal{O} \) and \( s''_k (k \in [1, n]) \) are defined by the following

\[
\mathcal{O} \in \bigcup_{k \in [1,n]} \mathcal{O}_k
\]

where \( \mathcal{O}_k \) and \( s''_k \) are given by:

- If \( \exists s_k \xrightarrow{\mathcal{O}_0/k/\mathcal{O}'_k} s''_k \in \mathcal{T}_k \), then \( \mathcal{O}_k = \mathcal{O}'_k \land s''_k = s''_k \);

- Else \( \mathcal{O}_k = \emptyset \land s''_k = s_k \).

2. An external signal \( e_i \in E_i (i \in [1, n]) \) arrives at \( A_i \). If

\[
\exists s = (s_0, s_1, \cdots, s_n) \in S \land s_i \xrightarrow{e_i/\mathcal{O}_i} s'_i \in \mathcal{T}_i \land s_0 \xrightarrow{\mathcal{O}_i, \& \& guard/\mathcal{O}_0} s'_0 \in \mathcal{T}_0
\]

where \( guard \in expr(BV_0) \) evaluates to true in state \( s_0 \), then
where $O$ and $s'_k (k \in [1, n], k \neq i)$ are defined by the following

$$O = \{O_0[0]\} \cup \bigcup_{k \in [1, n], k \neq i} O_k$$

where $O_k$ and $s'_k$ are given by:

- If $\exists s_k \xrightarrow{O_0[k]/O'_k} s''_k \in T_k$, then $O_k = O'_k \land s'_k = s''_k$;
- Else $O_k = \emptyset \land s'_k = s_k$.

Based on Definition 9, we revisit the six cases identified in Definition 8:

1. An external signal arrives at $A_0$ and $O_0 = \emptyset$, when the mode switch of a composite component implies no mode switch among its subcomponents.

2. An external signal arrives at $A_0$ and $O_0 \neq \emptyset$, when the mode switch of a composite component implies the mode switch of at least one of its subcomponents.

3. An external signal arrives at $A_i (i \in [1, n])$ and $O_0 = \emptyset$, when the mode switch of a component $c_i$ implies no mode switch among its parent and its siblings, i.e. components with the same parent as $c_i$.

4. An external signal arrives at $A_i (i \in [1, n])$ and $O_0$ only contains an external signal $e_0$, when the mode switch of a component only implies the mode switch of its parent but not its siblings.

5. An external signal arrives at $A_i (i \in [1, n])$ and $O_0$ only contains a set of internal signals $i_k (k \in [1, n])$, when the mode switch of a component only implies the mode switch of at least one of its siblings but not its parent.

6. An external signal arrives at $A_i (i \in [1, n])$ and $O_0$ contains both $e_0$ and $i_k (k \in [1, n])$, when the mode switch of a component implies the mode switch of both of its parent and at least one sibling.
Next let’s demonstrate each case by a simple example, where a composite component $a$ has two subcomponents $b$ and $c$. Case 1 is demonstrated in Figure 3.10, which includes the mode mapping table of $a$, the MMA of each component and the MMA after composition. Suppose that an external signal $a.E(m^2_a)$ arrives at $MMA^p_a$ (the parent MMA), triggering the transition $m^1_a \xrightarrow{a.E(m^2_a)/\emptyset} m^2_a$. Since this external signal does not imply the mode switch of $b$ or $c$, no state transition will occur in $MMA^c_b$ or $MMA^c_c$. Let $A$ be the MMA after composition, with two states $s_1 = (m^1_a, m^1_b, m^1_c)$ and $s_2 = (m^2_a, m^1_b, m^1_c)$. According to Definition 9, $A$ will undergo the transition $s_1 \xrightarrow{a.E(m^2_a)/\emptyset} s_2$.

Figure 3.10: MMA composition—Case 1

Figure 3.11 demonstrates cases 2 and 6. Case 2 occurs as an external signal $a.E(m^2_a)$ arrives at $MMA^p_a$, triggering the transition $m^1_a \xrightarrow{a.E(m^2_a)/\{b.I(m^2_b),c.I(m^2_c)\}} m^2_a$, which is synchronized with the transition $m^1_b \xrightarrow{b.I(m^2_b)/\{b.E(m^2_b)\}} m^2_b$ of $MMA^c_b$ and the transition $m^1_c \xrightarrow{c.I(m^2_c)/\{c.E(m^2_c)\}} m^2_c$ of $MMA^c_c$. Then $A$ will undergo the transition $s_1 \xrightarrow{a.E(m^2_a)/\{b.E(m^2_b),c.E(m^2_c)\}} s_2$ where $s_1 = (m^1_a, m^1_b, m^1_c)$ and $s_2 = (m^2_a, m^2_b, m^2_c)$.

Case 6 occurs as an external signal $b.E(m^1_b)$ arrives at $MMA^c_b$, triggering the transition $m^2_b \xrightarrow{b.E(m^1_b)/\{b.I(m^2_b \rightarrow m^1_b)\}} m^1_b$. This is synchronized with the transition $m^2_a \xrightarrow{b.I(m^2_b \rightarrow m^1_b) \& \& \ guard/\{a.E(m^1_a),c.I(m^1_c)\}} m^1_a$ of $MMA^p_a$, which further leads to the transition $m^2_c \xrightarrow{c.I(m^1_c)/\{c.E(m^1_c)\}} m^1_c$ of $MMA^c_c$. Then $A$ will undergo the transition $s_2 \xrightarrow{b.E(m^1_b) \& \& \ guard/\{a.E(m^1_a),c.E(m^1_c)\}} s_1$. 


Figure 3.11: MMA composition—Cases 2 and 6

Figure 3.12 demonstrates Case 3. Suppose that an external signal $b.E(m_b^2)$ arrives at $MMA_c^b$, triggering the transition $m_b^1 \xrightarrow{b.E(m_b^2)/\{b.I(m_b^2)\}} m_b^2$. This leads to the transition $m_a^1 \xrightarrow{b.I(m_a^1\rightarrow m_a^2)/\emptyset} m_a^1$ of $MMA_a^p$ without affecting $MMA_c^c$. Then $A$ will undergo the transition $s_1 \xrightarrow{b.E(m_a^2)/\emptyset} s_2$ where $s_1 = (m_a^1, m_b^1, m_c^1)$ and $s_2 = (m_a^2, m_b^2, m_c^1)$.

Figure 3.12: MMA composition—Case 3
Figure 3.13 demonstrates Case 4. Suppose that an external signal \( c.E(m^2_c) \) arrives at \( \text{MMA}^c \), triggering the transition \( m^1_c \xrightarrow{c.E(m^2_c)/\{c.I(m^1_c \rightarrow m^2_c)\}} m^2_c \). This leads to the transition \( m^1_a \xrightarrow{c.I(m^1_c \rightarrow m^2_c)/\{a.E(m^2_a)\}} m^2_a \) of \( \text{MMA}^p \) without affecting \( \text{MMA}^b \). Then \( A \) will undergo the transition \( s_1 \xrightarrow{c.E(m^2_c)/\{a.E(m^2_a)\}} s_2 \) where \( s_1 = (m^1_a, m^1_b, m^1_c) \) and \( s_2 = (m^2_a, m^1_b, m^2_c) \).

Finally, Figure 3.14 demonstrates Case 5. Suppose that an external signal \( c.E(m^2_c) \) arrives at \( \text{MMA}^c \), triggering the transition \( m^1_c \xrightarrow{c.E(m^2_c)/\{c.I(m^1_c \rightarrow m^2_c)\}} m^2_c \) which is synchronized with the transition \( m^1_a \xrightarrow{c.I(m^1_c \rightarrow m^2_c)/\{b.I(m^2_b)\}} m^1_a \) of \( \text{MMA}^p \). This further leads to the transition \( m^1_b \xrightarrow{b.I(m^2_b)/\{b.E(m^2_b)\}} m^2_b \). Then \( A \) will undergo the transition \( s_1 \xrightarrow{c.E(m^2_c)/\{b.E(m^2_b)\}} s_2 \) where \( s_1 = (m^1_a, m^1_b, m^1_c) \) and \( s_2 = (m^1_a, m^2_b, m^2_c) \).

### 3.4 Summary

Mode mapping is indispensable to reuse of multi-mode components. We have proposed a mode mapping mechanism for mapping component modes at design time. This mechanism enables a composite component to locally map its modes and the modes of its subcomponents. The mode mapping of each composite component can be expressed by a set of MMAs, which interact with the
3.4 Summary

Mode mapping is indispensable to reuse of multi-mode components. We have proposed a mode mapping mechanism for mapping component modes at design time. This mechanism enables a composite component to locally map its modes and the modes of its subcomponents. The mode mapping of each composite component can be expressed by a set of MMAs, which interact with the MSRM of this component. In this chapter we have presented the formal definition of MMA, MMA graphical representation, MMA synchronization semantics, and MMA composition. All these concepts have been further explained by simple examples.
Chapter 4

Mode-switch handling at runtime

Mode switch is one of the most onerous challenges in multi-mode systems due to its dynamic nature. A mode switch triggered at the wrong time is in peril of jeopardizing the execution of the entire system. Since a mode switch is usually performed throughout a well-ordered list of actions rather than being an instantaneous event, it is imperative that a mode switch is handled by appropriate mode-switch protocols which guarantee both the correctness of a mode switch and a bounded mode-switch time. A multi-mode system that performs a mode switch must be able to successfully change its configuration in the old mode to the configuration in the new mode without deadlock or disrupting its functional behavior.

For multi-mode systems with multi-mode components, the handling of a mode switch is even more challenging, as the mode switch of one component may lead to the mode switches of many other components at various levels in the component hierarchy. In particular, the mode information of a component may not be globally accessible for the other components, hampering the mode information exchange between components. The Mode-Switch Runtime Mechanism (MSRM) of our MSL is an efficient distributed mechanism for coordinating the mode switches of different components. Chapter 2 states that each multi-mode component switches mode following its own MSRM. Furthermore, Chapter 3 states that the MSRM of each composite multi-mode component interacts with its mode mapping mechanism while switching mode at runtime. This chapter gives an in-depth description of our MSRM which...
Chapter 4

Mode-switch handling at runtime

Mode switch is one of the most onerous challenges in multi-mode systems due to its dynamic nature. A mode switch triggered at the wrong time is in peril of jeopardizing the execution of the entire system. Since a mode switch is usually performed throughout a well-ordered list of actions rather than being an instantaneous event, it is imperative that a mode switch is handled by appropriate mode-switch protocols which guarantee both the correctness of a mode switch and a bounded mode-switch time. A multi-mode system that performs a mode switch must be able to successfully change its configuration in the old mode to the configuration in the new mode without deadlock or disrupting its functional behavior.

For multi-mode systems with multi-mode components, the handling of a mode switch is even more challenging, as the mode switch of one component may lead to the mode switches of many other components at various levels in the component hierarchy. In particular, the mode information of a component may not be globally accessible for the other components, hampering the mode information exchange between components. The Mode-Switch Runtime Mechanism (MSRM) of our MSL is an efficient distributed mechanism for coordinating the mode switches of different components. Chapter 2 states that each multi-mode component switches mode following its own MSRM. Furthermore, Chapter 3 states that the MSRM of each composite multi-mode component interacts with its mode mapping mechanism while switching mode at runtime. This chapter gives an in-depth description of our MSRM which
considers a series of practical issues such as the triggering of a mode-switch event, the propagation of such a event, the guarantee of mode consistency, atomic execution, the concurrent triggering of multiple mode-switch events, and emergency mode switch.

### 4.1 Mode-switch triggering

Generally speaking, a mode switch in a multi-mode system can be either time-triggered or event-triggered. Time-triggered mode switch is relatively more predictable in that a mode switch is triggered by a periodic event. Typically, a time-triggered mode switch can be easily implemented and analyzed. By contrast, event-triggered mode switch suggests that a mode switch is is triggered by a mode-switch event, which could be an external event, e.g. when a sensor value reaches a pre-defined threshold, or an internal event, e.g. when a system reaches a particular state. Compared with time-triggered mode switch, event-triggered mode switch is much less predictable, thus more difficult to handle. However, event-triggered mode switch is more prevalent than time-triggered mode switch. In fact, time-triggered mode switch can even be considered as a special case of event-triggered mode switch, since timeouts resulting from the advancement of time are events.

Our focus in this thesis is on event-triggered mode switch. For a multi-mode system with multi-mode components, a mode switch is initially triggered by a component identified as the Mode-Switch Source (MSS):

**Definition 10. Mode-Switch Source:** An MSS is the first component (primitive or composite) that needs to switch mode due to the detection of a mode-switch event.

Note that an MSS may not be the first component that detects a mode-switch event. For instance, when a sensor value reaches a threshold, an event is initially detected by the corresponding sensor component. However, the sensor component is not the MSS, as it simply passes the event to higher level components. An MSS is manually specified at design time depending on the actual system requirement. It is allowed to define multiple MSSs in the same system. A component can be an MSS while running in different modes. When an MSS detects a mode-switch event, it will initiate a mode switch by triggering a mode-switch scenario, or simply scenario. Let $\mathcal{M}_{c_i}$ be the set of modes supported by component $c_i$. Then a scenario is defined as follows:
4.2 Mode-switch propagation

Definition 11. **Mode-switch scenario**: A mode-switch scenario is an event triggered by an MSS $c_k$ that $c_k$ requests to switch from its current mode $m^{i}_{c_k}$ to a new mode $m^{j}_{c_k}$, where $m^{i}_{c_k}, m^{j}_{c_k} \in \mathcal{M}_{c_k}$. This scenario is denoted as $c_k : m^{i}_{c_k} \rightarrow m^{j}_{c_k}$.

By Definition 11, a scenario is identified without ambiguity by three parameters: an MSS $c_k$, its current mode $m^{i}_{c_k}$ while triggering the scenario, and the requested new mode $m^{j}_{c_k}$. To distinguish a scenario from other scenarios triggered in the same system, we assign a unique scenario ID to each triggering of a scenario. The triggering of a scenario may give rise to the mode switches of some other components. This can be clearly reflected in the mode mapping tables 3.2 and 3.3 in Chapter 3. Suppose that $a$ triggers a scenario $k$ requesting to switch from $m^{1}_a$ to $m^{2}_a$. Table 3.2 implies that $k$ leads to the mode switches of all the subcomponents of $a$. Suppose that the current mode of $c$ is $m^{1}_c$ when $k$ is triggered. Consequently, $c$ will switch to $m^{2}_c$. Furthermore, Table 3.3 implies that the mode switch of $c$ from $m^{1}_c$ to $m^{2}_c$ implies the mode switch of $e$ from $m^{1}_e$ to either $m^{2}_e$ or $m^{3}_e$, while the mode of $f$ is unaffected. Hence all the components of a system can be grouped into two categories for the triggering of a scenario: Type A components and Type B components:

Definition 12. **Type A/B components**: Consider a scenario $k$ triggered by a multi-mode component and any other component $c_i$ in a multi-mode system. If $k$ implies the mode switch of $c_i$, then $c_i$ is called a Type A component for $k$, denoted as $T^{k}_{c_i} = A$; otherwise, if $k$ has no effect on the mode of $c_i$, then $c_i$ is called a Type B component for $k$, denoted as $T^{k}_{c_i} = B$.

The Type A and Type B components for a scenario can be identified by the mode mappings of all composite components in a multi-mode system. Moreover, the new mode of each Type A component can also be derived by mode mapping. A scenario triggered by an MSS is supposed to be propagated to all the Type A components without disturbing Type B components.

4.2 Mode-switch propagation

There are many different ways of propagating a scenario. The most straightforward option is a direct broadcast from the MSS of the scenario to all the other Type A components. However, this is possible only if the MSS knows the global mode mapping of all components. Such an assumption is against the CBSE paradigm. Irrespective of how a scenario is propagated, the following criteria should always be met:
• Stepwise propagation. In Chapter 2 we have mentioned that due to lack of global mode information a component is only allowed to exchange mode information with its parent or subcomponents via dedicated mode-switch ports. Therefore, the propagation of a scenario must be stepwise, either one step up to the parent or one step down to the subcomponents.

• Precise coverage. A scenario is supposed to be propagated to all Type A components. No Type B components should be informed of this scenario.

• Bounded propagation time. The propagation time of a scenario should be bounded, as short as possible.

These criteria suggest that an efficient propagation protocol is demanded as a vital element of the MSRM of each component. We have developed a Mode-Switch Propagation (MSP) protocol for the propagation of a scenario by transmitting primitives via dedicated mode-switch ports. The MSP protocol respects the current states of all Type A components in the sense that a mode switch is performed based on this scenario only when all Type A components are ready to switch mode. The triggering of a scenario does not necessarily lead to a mode switch. If there is at least one Type A component whose current state does not allow a mode switch (i.e. this component is not ready to switch mode), the scenario has to be rejected. The MSP protocol leaves the authority of approving or rejecting a scenario to a component called the Mode-Switch Decision Maker (MSDM):

Definition 13. Mode-Switch Decision Maker: For a scenario \( k \), the MSDM is a component which decides whether \( k \) is approved or rejected. If the MSDM approves \( k \), all the Type A components will switch mode based on \( k \); if the MSDM rejects \( k \), no components will switch mode based on \( k \).

The MSDM of a scenario is dynamically identified by the MSP protocol at runtime. In general, the MSDM of a scenario is an ancestor or the parent of the corresponding MSS in the component hierarchy. In other words, the MSDM is directly or indirectly composed of the MSS. Otherwise, if the MSS is at the top of the component hierarchy without parent, it will also be the MSDM. Before we expound the MSP protocol, it is necessary to introduce the types of primitives used in this protocol. In general, we distinguish two major types of primitives:

• Upstream primitive: a primitive which is always sent via the port \( p^{MS} \) of a component to its parent.
4.2 Mode-switch propagation

- Downstream primitive: a primitive which is always sent via the port $p_{in}^MS$ of a composite component to its subcomponent(s).

More precisely, the MSP protocol encompasses the following six types of primitives:

- **Mode-Switch Request (MSR):** an upstream primitive originally issued by an MSS while triggering a scenario. Let $msr^k$ denote an MSR for a scenario $k$.

- **Mode-Switch Query (MSQ):** a downstream primitive originally issued by the MSDM of a scenario, asking if a Type A component is ready to switch mode. Let $msq^k$ denote an MSQ for a scenario $k$.

- **MSOK:** An upstream primitive from a Type A component $c_i$ for a given scenario $k$, indicating that $c_i$ is ready to switch mode for $k$. Let $msok^k$ denote an MSOK for a scenario $k$.

- **MSNOK:** An upstream primitive from a Type A component $c_i$ for a given scenario $k$, indicating that $c_i$ is not ready to switch mode for $k$. Let $msnok^k$ denote an MSNOK for a scenario $k$.

- **Mode-Switch Instruction (MSI):** a downstream primitive originally issued by the MSDM of a scenario, triggering the mode switch of a Type A component. Let $msi^k$ denote an MSI for a scenario $k$.

- **Mode-Switch Denial (MSD):** a downstream primitive originally issued by the MSDM of $k$, rejecting a scenario $k$. Let $msd^k$ denote an MSD for a scenario $k$.

Our MSRM in this thesis assumes that there are no transmission errors of any type of primitive. For instance, no omission or duplication occurs during the transmission of a primitive. The sequential order of primitives sent from a component is also preserved on the receiver’s side. This can be assured by the inter-component communication infrastructure. A specific assumption of our MSP protocol is that there is no concurrent triggering of multiple scenarios. This simplifying assumption ensures that an ongoing mode switch based on a scenario is never interrupted by the triggering of another scenario. The handling of concurrently triggered scenarios will be presented in Section 4.5.

In addition to the foregoing primitives, the description of our MSP protocol contains a number of notations used throughout the thesis. Let $PC$ be the
set of primitive components in a system, and let $CC$ be the set of composite components in the same system. Let $Top$ be the component at the top of the component hierarchy in a system. Furthermore, as defined in Section 3.2, the parent of a component $c_i \neq Top$ is denoted as $P_{c_i}$. Let $SC_{c_i}$ be the set of subcomponents of $c_i \in CC$, and $SC^A_{c_i}(k)$ be the set of Type A subcomponents of $c_i$ for a scenario $k$. Let $S^k_{c_i}$ denote that the current state of $c_i$ allows the mode switch for $k$, and let $\neg S^k_{c_i}$ denote that the current state of $c_i$ does not allow the mode switch for $k$. Finally, let $c_i$ be the MSS triggering a scenario $k$, with $c_j$ as the MSDM. We have stated that $c_j$ is an ancestor or the parent of $c_i$, unless $c_i = Top$. Let $C^j_i$ be the set of components between $c_i$ and $c_j$ such that for each $c_o \in C^j_i$, $c_o$ is an ancestor or the parent of $c_i$ and a descendant or a subcomponent of $c_j$. The MSP protocol is described as follows:

**Definition 14. Mode-Switch Propagation protocol:** Consider a scenario $k$ triggered by the MSS $c_i$, with $c_j$ as the MSDM. If $c_i \neq Top$, when $c_i$ triggers $k$, $c_i$ will issue an $msr^k$ sent to $P_{c_i}$. The $msr^k$ eventually reaches $c_j$ through $C^j_i$. Upon receiving the $msr^k$, each $c_o \in C^j_i$, identified as $T^k_{c_o} = A$ and $S^k_{c_o}$, forwards the $msr^k$ to $P_{c_o}$. The MSDM $c_j$ is identified while receiving the $msr^k$ when one of the three following conditions is satisfied: (1) $T^k_{c_j} = B$; (2) $T^k_{c_j} = A$ and $\neg S^k_{c_j}$; (3) $T^k_{c_j} = A$ and $S^k_{c_j}$ and $c_j = Top$. In response to the $msr^k$, $c_j$ makes the following decisions:

- **Under Condition (2),** $c_j$ rejects $k$ by issuing an $msd^k$ that is propagated back to $c_i$ through $C^j_i$. Upon receiving the $msd^k$, each $c_o \in C^j_i$ directly forwards the $msd^k$ to the subcomponent $c_i \in SC_{c_o} \cap C^j_i$ that previously sent the $msr^k$ to $c_o$. The propagation of $k$ is terminated when $c_i$ receives the $msd^k$. No component will switch mode for $k$.

- **Under conditions (1) and (3),** $c_j$ initiates an investigation of the current states of all Type A components by issuing an $msq^k$ that is propagated downstream and stepwise to all Type A components. Upon receiving the $msq^k$, a component $c_u$ needs to suspend its current execution and further propagate the $msq^k$ to $SC^A_{c_u}(k)$ if $S^k_{c_u}$ and $SC^A_{c_u}(k) \neq \emptyset$. Meanwhile, $P_{c_u}$ expects either an $msok^k$ or an $msnok^k$ from $c_u$ as the reply. Component $c_u$ replies with an $msok^k$ if $S^k_{c_u}$ (and $c_u$ has received an $msok^k$ from all components in $SC^A_{c_u}(k)$ when $SC^A_{c_u}(k) \neq \emptyset$). Otherwise, if $\neg S^k_{c_u}$, $c_u$ will directly reply to $P_{c_u}$ with an $msnok^k$ (without propagating the $msq^k$ downstream further to $SC^A_{c_u}(k)$ even if $SC^A_{c_u}(k) \neq \emptyset$). If
c_u receives at least one msnok^k from a subcomponent after propagating the msq^k to SC^A_{c_u}(k), c_u will also reply to P_{c_u} with an msnok^k.

- When c_j has received an msoi^k from all components in SC^A_{c_i}(k), c_j will approve k and trigger a mode switch for k by issuing an msit^k that follows the propagation trace of the msq^k. Each Type A component will start its mode switch upon receiving the msit^k. The propagation of k is completed when all Type A components have received the msit^k. Furthermore, for each composite component c_u which has sent an msnok^k to P_{c_u} due to ¬S^k_{c_u}, if c_u has previously received an msr^k from a subcomponent, then c_u must send an msd^k that is forwarded to c_i through C^u_i. The propagation of k is terminated when the msd^k have reached all the components that have received the msq^k or have sent the msr^k before. A component, which has suspended its execution due to the msq^k, resumes its execution upon receiving the msd^k. No component will switch mode for k.

If c_i = Top, then c_j = c_i and C^i^j = ∅. Then c_i triggers k by directly issuing an msq^k to all components in SC^A_{c_i}(k).

The identification of the MSDM c_j in the MSP protocol is based on a simple logic. Upon receiving an msr^k, the first thing that c_j needs to do is to refer to its mode mapping. If T^k_{c_j} = B, then c_j is apparently the MSDM because k will not affect other components at higher levels. If T^k_{c_j} = A, the current state of c_j decides if c_j is the MSDM of k. If S^k_{c_j} and c_j ≠ Top, then c_j must forward the msr^k to P_{c_j} which may also be affected by k. Otherwise, if ¬S^k_{c_j}, c_j is sure that k will be rejected. Hence there is no need to forward the msr^k further and c_j will be the MSDM of k. Of course, when c_j = Top, c_j is always the MSDM of k.

Note that a component needs to suspend its execution in the current mode upon receiving an msq^k because its execution may change its states. Without suspending the current execution, a component ready to switch mode at time t_0, e.g. while receiving an msq^k, may not be ready to switch mode at a later time t_1, e.g. while receiving an msit^k. For that reason, the execution of each Type A component has to be suspended upon receiving an msq^k such that the MSDM is able to conclude if all Type A components are ready to switch mode at the same time. A component with suspended execution due to an msq^k can resume its execution when it receives an msd^k.
It can be observed with ease from the MSP protocol that the first possible primitive that a component $c_o$ may receive associated with a scenario $k$ is either an $msr^k$ or an $msq^k$. Two special cases are: (1) if $c_o \in \mathcal{P}C$, then the first possible primitive that $c_o$ may receive is always an $msq^k$; (2) if $c_o = Top$, then the first possible primitive that $c_o$ may receive is always an $msr^k$. When $c_o$ receives the first primitive associated with $k$, if $c_o \in CC$, then $c_o$ should refer to its mode mapping to derive $T_{c_o}^k$ and $SC_A^{c_o}(k)$. Thereafter $c_o$ can store the mode mapping result for $k$ for later retrieval when $c_o$ receives a subsequent primitive associated with $k$. This minimizes the number of accesses to the mode mapping, thus facilitating the propagation time of $k$.

From the MSP protocol we summarize three possible cases with respect to the propagation of a scenario $k$:

1. When the MSDM $c_j$ receives an $msr^k$, its current state does not allow the mode switch for $k$. Consequently, $c_j$ directly rejects $k$ by issuing an $msd^k$ that is propagated back to $c_i$ through $C_i^j$. This case only exists when the MSS $c_i \neq Top$.

2. When the MSDM $c_j$ receives an $msr^k$, its current state allows the mode switch for $k$. After $c_j$ issues an $msq^k$ to $SC_A^{c_j}(k)$, $c_j$ receives an $msok^k$ from all components in $SC_A^{c_j}(k)$, thus triggering a mode switch for $k$ by issuing an $msi^k$ to $SC_A^{c_j}(k)$.

3. When the MSDM $c_j$ receives an $msr^k$, its current state allows the mode switch for $k$. Nonetheless, $c_j$ receives at least one $msnok^k$ after issuing the $msq^k$ to $SC_A^{c_j}(k)$, thus rejecting $k$ by issuing an $msd^k$ to $SC_A^{c_j}(k)$.

These three cases are further illustrated by an example in figures 4.1-4.3, where the MSS $e$ (marked in red) triggers a scenario with $a$ as the MSDM (marked in blue). All black components in these figures are Type A components for this scenario while other Type B components are filtered out. Then $b$ is the only component between the MSS and the MSDM, i.e. $C_e^a = \{b\}$. Figure 4.1 is straightforward as $a$ directly rejects the scenario by sending an MSD back. In Figure 4.2, all Type A components are ready to switch mode, thus eventually leading to a mode switch. In contrast to Figure 4.2, $b$ and $h$ in Figure 4.3 are not ready to switch mode, without further propagation of the MSQ. As $a$ receives the MSNOK from $b$ and $c$, an MSD is issued instead of the MSI propagated in Figure 4.2. The MSD is not propagated to $k$ and $l$ since they have not received the MSQ. However, $b$ is obliged to send the MSD to $e$ so that $e$ can abort the handling of $k$. 
It can be observed with ease from the MSP protocol that the first possible primitive that a component \( c_o \) may receive associated with a scenario \( k \) is either an \( msr_k \) or an \( msq_k \). Two special cases are: (1) if \( c_o \in PC \), then the first possible primitive that \( c_o \) may receive is always an \( msq_k \); (2) if \( c_o = Top \), then the first possible primitive that \( c_o \) may receive is always an \( msr_k \). When \( c_o \) receives the first primitive associated with \( k \), if \( c_o \in CC \), then \( c_o \) should refer to its mode mapping to derive \( T_{c_o} \) and \( SC_{A_{c_o}}(k) \). Thereafter \( c_o \) can store the mode mapping result for \( k \) for later retrieval when \( c_o \) receives a subsequent primitive associated with \( k \). This minimizes the number of accesses to the mode mapping, thus facilitating the propagation time of \( k \).

From the MSP protocol we summarize three possible cases with respect to the propagation of a scenario \( k \):

1. When the MSDM \( c_j \) receives an \( msr_k \), its current state does not allow the mode switch for \( k \). Consequently, \( c_j \) directly rejects \( k \) by issuing an \( msd_k \) that is propagated back to \( c_i \) through \( C_{j} \). This case only exists when the MSS \( c_i \neq Top \).

2. When the MSDM \( c_j \) receives an \( msr_k \), its current state allows the mode switch for \( k \). After \( c_j \) issues an \( msq_k \) to \( SC_{A_{c_j}}(k) \), \( c_j \) receives an \( msok_k \) from all components in \( SC_{A_{c_j}}(k) \), thus triggering a mode switch for \( k \) by issuing an \( msi_k \) to \( SC_{A_{c_j}}(k) \).

3. When the MSDM \( c_j \) receives an \( msr_k \), its current state allows the mode switch for \( k \). Nonetheless, \( c_j \) receives at least one \( msnok_k \) after issuing the \( msq_k \) to \( SC_{A_{c_j}}(k) \), thus rejecting \( k \) by issuing an \( msd_k \) to \( SC_{A_{c_j}}(k) \).

These three cases are further illustrated by an example in figures 4.1-4.3, where the MSS \( e \) (marked in red) triggers a scenario with \( a \) as the MSDM (marked in blue). All black components in these figures are Type A components for this scenario while other Type B components are filtered out. Then \( b \) is the only component between the MSS and the MSDM, i.e. \( C_{a e} = \{ b \} \).

Figure 4.1 is straightforward as \( a \) directly rejects the scenario by sending an \( MSD \) back. In Figure 4.2, all Type A components are ready to switch mode, thus eventually leading to a mode switch. In contrast to Figure 4.2, \( b \) and \( h \) in Figure 4.3 are not ready to switch mode, without further propagation of the \( MSQ \). As \( a \) receives the \( MSNOK \) from \( b \) and \( c \), an \( MSD \) is issued instead of the \( MSI \) propagated in Figure 4.2. The \( MSD \) is not propagated to \( k \) and \( l \) since they have not received the \( MSQ \). However, \( b \) is obliged to send the \( MSD \) to \( e \) so that \( e \) can abort the handling of \( k \).

It is noteworthy that the MSDM \( a \) in figures 4.1-4.3 can be any composite component in a system. This implies that a mode switch does not have to be a
global event for the entire system. A mode switch locally performed within a composite component exerts no influence on the rest of the system.

Our MSP protocol resembles to some extent the 2-phase commit protocol for distributed database [12]. The 2-phase propagation of a scenario in our MSP protocol is initiated by the MSDM. Phase 1 starts when the MSDM issues an \texttt{MSQ} and ends when the MSDM has received an \texttt{MSOK} or \texttt{MSNOK} from all its Type A subcomponents. At the end of Phase 1, if the MSDM receives no \texttt{MSNOK}, then Phase 2 starts when the MSDM issues an \texttt{MSI} and ends when all Type A components have received the \texttt{MSI}. Otherwise, if the MSDM receives at least one \texttt{MSNOK}, then Phase 2 starts when the MSDM issues an \texttt{MSD} and ends when the \texttt{MSD} has reached all the Type A components which have received the \texttt{MSQ} in Phase 1 and the components which have previously sent the \texttt{MSR}. 

Figure 4.3: The MSP protocol—Case 3
The 2-phase propagation in the MSP protocol is demonstrated by an example in Figure 4.4, where a scenario $k$ is triggered by $e$ in the system introduced in Figure 1.1, with $a$ as the MSDM. Type A components for $k$ are enclosed within a dotted loop, including $a$, $c$, and $d$. The rest, $b$ and $f$, are Type B components for $k$. Figures 4.5-4.7 depict the three cases of the propagation of $k$. Shown in figures 4.6 and 4.7, the MSDM $a$ initiates a 2-phase propagation as it issues an $msq^k$ to its Type A subcomponents $c$ and $d$. Phase 2 starts as $a$ issues an $msi^k$ or $msd^k$ to $c$ and $d$. The 2-phase propagation is absent in Figure 4.5 because $a$ directly rejects $k$ after receiving an $msr^k$.

The MSP protocol reflects the different behaviors of three types of components with regard to the propagation of a scenario: (1) primitive component; (2) composite component not at the top level; (3) the component at the top level\(^1\).

\(^1\)In principle, the component at the top level can be either primitive or composite. However, if it is primitive, it makes no sense to call it a component-based system because the system itself is a single component. Therefore we assume that it must be composite.
Among the three types, Type (2) exhibits the most general yet most complex behavior, since this type of component must exchange primitives with both its parent and subcomponents. The other two types can be considered as two special cases of Type (2).

We have designed a set of algorithms implementing the MSP protocol for all three types of components. First, Algorithm 1 describes the behavior of an
MSS which triggers a scenario. This algorithm contains the following variables and functions:

- **MS_event_detected** is a local boolean variable with a default value false, returning true when an MSS detects a mode-switch event.
- **Derive_new_mode**(c\textsubscript{i}) is a function returning the new mode of an MSS when the MSS triggers a scenario.
- **Assign_scenario_ID**(c\textsubscript{i}) is a function assigning a unique scenario ID to a triggered scenario.
- **Signal**(c\textsubscript{i}, A, B) is a function for c\textsubscript{i} to send a primitive B via the dedicated mode-switch port A, which is either p\textsubscript{MS} (the port for exchanging primitives with P\textsubscript{c\textsubscript{i}}) or p\textsubscript{inMS} (the port for exchanging primitives with SC\textsubscript{c\textsubscript{i}}). Likewise, **Wait**(c\textsubscript{i}, A, B) is a function for c\textsubscript{i} to receive a primitive B via the port A.

- **msr\textsubscript{k}\textsubscript{c\textsubscript{i}}**(m\textsubscript{c\textsubscript{i}}, m\textsubscript{new\textsubscript{c\textsubscript{i}}}) represents an msr\textsubscript{k} sent by c\textsubscript{i} which requests to switch from mode m\textsubscript{c\textsubscript{i}} to m\textsubscript{new\textsubscript{c\textsubscript{i}}}. Similarly, **msq\textsubscript{k}\textsubscript{c\textsubscript{i}}**(m\textsubscript{new\textsubscript{c\textsubscript{i}}}) represents an msq\textsubscript{k} sent to c\textsubscript{i} which is requested to switch to m\textsubscript{new\textsubscript{c\textsubscript{i}}}. These parameters such as c\textsubscript{i}, m\textsubscript{c\textsubscript{i}} and m\textsubscript{new\textsubscript{c\textsubscript{i}}} are required for doing mode mapping. The other subsequent primitives, which do not need mode mapping, can be presented in a simplified fashion. For instance, **msi\textsubscript{k}\textsubscript{c\textsubscript{i}}** or **msd\textsubscript{k}\textsubscript{c\textsubscript{i}}** represents an msi\textsubscript{k} or msd\textsubscript{k} sent to c\textsubscript{i}, while **msok\textsubscript{k}\textsubscript{c\textsubscript{i}}** or **msnok\textsubscript{k}\textsubscript{c\textsubscript{i}}** represents an msok\textsubscript{k} or msnok\textsubscript{k} sent by c\textsubscript{i}.

- **Mode_mapping**(c\textsubscript{i}, c\textsubscript{j}, m\textsubscript{c\textsubscript{j}}, m\textsubscript{new\textsubscript{c\textsubscript{j}}}) is a function returning the mode mapping results of c\textsubscript{i} ∈ CC based on the mode switch of c\textsubscript{j} from m\textsubscript{c\textsubscript{j}} to m\textsubscript{new\textsubscript{c\textsubscript{j}}}, where c\textsubscript{j} ∈ SC\textsubscript{c\textsubscript{i}} ∪ {c\textsubscript{i}}. Alternatively, **Mode_mapping**(c\textsubscript{i}, m\textsubscript{c\textsubscript{i}}的新) returns the mode mapping results of c\textsubscript{i} based on an incoming downstream primitive such as msq\textsubscript{k}\textsubscript{c\textsubscript{i}}(m\textsubscript{c\textsubscript{i}}的新).

- **Suspend**(c\textsubscript{i}, m\textsubscript{c\textsubscript{i}}) and **Resume**(c\textsubscript{i}, m\textsubscript{c\textsubscript{i}}) are functions for suspending and resuming the execution of c\textsubscript{i} in its mode m\textsubscript{c\textsubscript{i}}.

- **Mode_switch**(c\textsubscript{i}, k) is a function performing the mode switch of c\textsubscript{i} for k or waiting for the mode-switch completion of the subcomponents of c\textsubscript{i}. This function will be further explained in Section 4.3.

- **Collect_response**(c\textsubscript{i}, k), described in Algorithm 2, is a function for collecting the msok\textsubscript{k} or msnok\textsubscript{k} from SC\textsubscript{c\textsubscript{i}}\textsuperscript{A}(k) after c\textsubscript{i} propagates an msq\textsubscript{k} to SC\textsubscript{c\textsubscript{i}}\textsuperscript{A}(k).
• All_ready is a local boolean variable of \( c_i \in CC \) returning \textit{true} when \( c_i \) has received an \( msok^k \) from all components in \( SC^A_{c_i}(k) \), and returning \textit{false} when \( c_i \) has received at least one \( msnok^k \) from a subcomponent.

**Algorithm 1**\ New_scenario\( (c_i) \)

1: loop
2: if MS\_event\_detected then
3: Derive\_new\_mode\( (c_i) \);
4: \( k = \text{Assign\_scenario\_ID}(c_i) \);
5: if \( c_i \neq \text{Top} \) then
6: \( \text{Signal}(c_i, \{ p^{MS}, msr^k_{c_i} (m_{c_i}, m_{new}^c) \}) \);
7: else\( c_i = \text{Top} \)\end{itemize}
8: Mode\_mapping\( (c_i, c_i, m_{c_i}, m_{new}^c) \);
9: Suspend\( (c_i, m_{c_i}) \);
10: \( \forall c_j \in SC^A_{c_i}(k) : \text{Signal}(c_i, p_{in}^{MS}, msq^k_{c_j} (m_{c_j}^{new})) \);
11: Collect\_response\( (c_i, k) \);
12: if All\_ready then
13: \( \forall c_j \in SC^A_{c_i}(k) : \text{Signal}(c_i, p_{in}^{MS}, msi^k_{c_j}) \);
14: Mode\_switch\( (c_i, k) \);
15: else\( \neg \text{All\_ready} \)\end{itemize}
16: \( \forall c_j \in SC^A_{c_i}(k) : \text{Signal}(c_i, p_{in}^{MS}, msd^k_{c_j}) \);
17: Resume\( (c_i, m_{c_i}) \);
18: end if\end{itemize}
19: end if\end{itemize}
20: end if\end{itemize}
21: end loop

Let \( \tilde{CC} = CC \setminus \{ \text{Top} \} \) be the set of composite components except \( \text{Top} \). Algorithms 3-5 implement the MSP protocol for (1) \( c_i \in PC \); (2) \( c_i \in \tilde{CC} \); and (3) \( c_i = \text{Top} \), respectively. In these algorithms, \( msr^k \leftarrow c_j \) denotes that an \( msr^k \) has been previously sent from \( c_j \in SC_{c_i} \) to \( c_i \in CC \).

Note that algorithms 4 and 5 also work when \( SC^A_{c_i}(k) = \emptyset \). For instance, when \( SC^A_{c_i}(k) = \emptyset \), Line 8 of Algorithm 4 will not propagate the \( msq^k \). The function \textit{Collect\_response}\( (c_i, k) \) in Line 9 will be called as usual, setting \textit{All\_ready} to true as per Algorithm 2.

As a final remark, the MSP protocol can be improved to facilitate mode switch at runtime. By Definition 14, after \( c_i \in CC \) propagates an \( msq^k \) to \( SC^A_{c_i}(k) \), \( c_i \) has to wait for the reply from all components in \( SC^A_{c_i}(k) \). If \( c_i \) receives an \( msnok^k \) before receiving all the replies, \( c_i \) still has to wait for the
4.2 Mode-switch propagation

Algorithm 2 Collect_response($c_i \in CC, k$)

1: $n := 0$
2: $All_{ready} := true$
3: if $SC^A_{c_i}(k) \neq \emptyset$ then
4: while $n < |SC^A_{c_i}(k)|$ do
5: $Wait(c_i, p_{MS}^{\text{primitive}})$
6: if (primitive = $msok^k_{c_i}$) $\land$ ($c_j \in SC^A_{c_i}(k)$) then
7: $n := n + 1$
8: else ($primitive = msnok^k_{c_j}$) $\land$ ($c_j \in SC^A_{c_i}(k)$)
9: $n := n + 1$
10: $All_{ready} := false$
11: end if
12: endwhile
13: end if

Algorithm 3 Propagation($c_i \in \mathcal{PC}$)

1: loop
2: $Wait(c_i, p_{MS}^{\text{primitive}})$
3: if primitive = $msq^k_{c_i}$($m_{n_{\text{new}}}^c$) then
4: $Suspend(c_i, m_{c_i})$
5: if $S^k_{c_i}$ then
6: $Signal(c_i, p_{MS}^{\text{primitive}}, msok^k_{c_i})$
7: else ($\neg S^k_{c_i}$)
8: $Signal(c_i, p_{MS}^{\text{primitive}}, msnok^k_{c_i})$
9: end if
10: $Wait(c_i, p_{MS}^{\text{primitive}})$
11: if primitive = $ms^k_{c_i}$ then
12: $Mode_{switch}(c_i, k)$
13: else ($primitive = msd^k_{c_i}$)
14: $Resume(c_i, m_{c_i})$
15: end if
16: end if
17: end loop
Algorithm 4 Propagation($c_i \in \widehat{CC}$)

1: \textbf{loop}
2: \hspace{1em} Wait($c_i, p_{MS}^{MS} \lor p_{in}^{MS}, \text{primitive}$);
3: \hspace{1em} if \ ($\text{primitive} = msr_{c_o}^k(m_{c_o}, m_{c_{new}}^c)$) \land \ ($c_o \in SC_{c_i}$) \ then
4: \hspace{1em} Mode\_mapping($c_i, c_o, m_{c_o}, m_{c_{new}}^c$);
5: \hspace{1em} \textbf{if} $T_{c_i}^k = B$ \ then
6: \hspace{2em} \textbf{if} $S_{c_i}^k$ \ then
7: \hspace{3em} $\forall c_j \in SC_{c_i}^A (k) : Signal(c_i, p_{in}^{MS}, msq_{c_j}^k (m_{c_j}^{new})$);
8: \hspace{3em} Collect\_response($c_i, k$);
9: \hspace{2em} \textbf{if} All\_ready \ then
10: \hspace{3em} $\forall c_j \in SC_{c_i}^A (k) : Signal(c_i, p_{in}^{MS}, msi_{c_j}^k)$;
11: \hspace{3em} Mode\_switch($c_i, k$);
12: \hspace{2em} \textbf{else} ($\neg$All\_ready)
13: \hspace{3em} $\forall c_j \in SC_{c_i}^A (k) : Signal(c_i, p_{in}^{MS}, msd_{c_j}^k)$;
14: \hspace{3em} Resume($c_i, m_{c_i}$);
15: \hspace{2em} \textbf{end if}
16: \hspace{1em} \textbf{else} ($\neg S_{c_i}^k$)
17: \hspace{2em} Signal($c_i, p_{in}^{MS}, msd_{c_{new}}^k$); \{msr\_k \leftarrow c_o\}
18: \hspace{2em} \textbf{end if}
19: \hspace{1em} \textbf{else}($T_{c_i}^k = A$)
20: \hspace{2em} \textbf{if} $S_{c_i}^k$ \ then
21: \hspace{3em} Signal($c_i, p_{in}^{MS}, msi_{c_i}^k (m_{c_i}^{new})$);
22: \hspace{3em} \textbf{else} ($\neg S_{c_i}^k$)
23: \hspace{3em} Signal($c_i, p_{in}^{MS}, msd_{c_i}^k$); \{msr\_k \leftarrow c_o\}
24: \hspace{2em} \textbf{end if}
25: \hspace{1em} \textbf{end if}
26: \hspace{1em} \textbf{end if}
27: \hspace{1em} \textbf{else}($\text{primitive} = msq_{c_i}^k (m_{c_{new}}^c)$)
28: \hspace{2em} Suspend($c_i, m_{c_i}$);
29: \hspace{2em} \textbf{if} $S_{c_i}^k$ \ then
30: \hspace{3em} Mode\_mapping($c_i, m_{c_{new}}^c$);
31: \hspace{3em} $\forall c_j \in SC_{c_i}^A (k) : Signal(c_i, p_{in}^{MS}, msq_{c_j}^k (m_{c_j}^{new})$);
32: \hspace{3em} Collect\_response($c_i, k$);
33: \hspace{2em} \textbf{if} All\_ready \ then
34: \hspace{3em} Signal($c_i, p_{in}^{MS}, mso_{c_i}^k$);
35: \hspace{3em} Wait($c_i, p_{MS}^{MS}, \text{primitive}$);
36: \hspace{3em} \textbf{if} \ primitive = msi_{c_i}^k \ then
37: \hspace{4em} $\forall c_j \in SC_{c_i}^A (k) : Signal(c_i, p_{in}^{MS}, msi_{c_j}^k)$;
38: \hspace{4em} Mode\_switch($c_i, k$);
39: \hspace{3em} \textbf{else}($\text{primitive} = msd_{c_i}^k$)
40: \hspace{4em} $\forall c_j \in SC_{c_i}^A (k) : Signal(c_i, p_{in}^{MS}, msd_{c_j}^k)$;
41: \hspace{4em} Resume($c_i, m_{c_i}$);
42: \hspace{2em} \textbf{end if}
4.2 Mode-switch propagation

\begin{verbatim}
43:   else \{ \neg All\_ready \}
44:     Signal(c_i, p^{MS}, msnok^k_{c_i});
45:     Wait(c_i, p^{MS}, primitive);
46:     if primitive = msd^k_{c_i} then
47:       \forall c_j \in SC^k_i(k) : Signal(c_j, p^{MS}_m, msd^k_{c_j});
48:       Resume(c_i, m_{c_i});
49:     else
50:       Report_error;
51:     end if
52:   end if
53: else \{ \neg S^k_{c_i} \}
54:   Signal(c_i, p^{MS}, msnok^k_{c_i});
55:   Wait(c_i, p^{MS}, primitive);
56:   if primitive = msd^k_{c_i} then
57:     if msn^k_j \leftarrow c_o \in SC_{c_i} \text{ then}
58:       Signal(c_i, p^{MS}_m, msd^k_{c_o});
59:     end if
60:     Resume(c_i, m_{c_i});
61:   else
62:     Report_error;
63:   end if
64: end if
65: end if
66: end loop
\end{verbatim}
Algorithm 5 Propagation($c_i = \text{Top}$)

1: loop
2: Wait($c_i, p_{in}^{MS}$, primitive);
3: if (primitive = $msr_{c_o}^k(m_{c_o}, m_{c_o}^{new})$) \land (c_o \in SC_{c_i}) then
4: Mode_mapping($c_i, c_o, m_{c_o}, m_{c_o}^{new}$);
5: if $S^k_{c_i}$ then
6: Suspend($c_i, m_{c_i}$);
7: \forall c_j \in SC_{c_i}(k) : Signal($c_i, p_{in}^{MS}, msq_{c_j}^k(m_{c_j}^{new})$);
8: Collect_response($c_i, k$);
9: if All_ready then
10: \forall c_j \in SC_{c_i}(k) : Signal($c_i, p_{in}^{MS}, msi_{c_j}^k$);
11: Mode_switch($c_i, k$);
12: else ($\neg$All_ready) \begin{itemize}
13: \forall c_j \in SC_{c_i}(k) : Signal($c_i, p_{in}^{MS}, msd_{c_j}^k$);
14: Resume($c_i, m_{c_i}$);
\end{itemize}
15: end if
16: else ($\neg$S^k_{c_i})
17: Signal($c_i, p_{in}^{MS}, msd_{c_o}^k$); $msr^k \leftarrow c_o$
18: end if
19: end if
20: end loop
other replies. A better strategy for \( c_i \) is to immediately make a decision upon receiving an \( msnok^k \) without waiting for the other replies. If \( c_i \) is the MSDM of \( k \), \( c_i \) can immediately reject \( k \) by propagating an \( msd^k \) to \( SC^A_{c_i}(k) \). If \( c_i \) is not the MSDM of \( k \), \( c_i \) can immediately send an \( msnok^k \) to \( P^k_{c_i} \).

As an even more efficient improvement, a component \( c_i \) can directly send an \( msd^k \) to \( P_{c_i} \) in response to an \( msq^k \) if \( c_i \) is not ready to switch mode. When a component \( c_j \) receives the \( msd^k \) from a subcomponent \( c_i \), \( c_j \) can directly propagate the \( msd^k \) to \( SC^A_{c_j}(k) \setminus \{c_i\} \), and even \( P_{c_j} \) if necessary. The improved MSP protocol is demonstrated in Figure 4.8 which slightly changes the propagation of the scenario \( k \) in Figure 4.7. In Figure 4.8, \( e \) is not ready to switch mode for \( k \) after receiving an \( msq^k \). Instead of replying with an \( msnok^k \), \( e \) directly sends an \( msd^k \) back to its parent \( e \) to reject \( k \). Subsequently, \( c \) further propagates the \( msd^k \) to \( a \), which only needs to propagates the \( msd^k \) to \( d \).

![Figure 4.8: Demonstration of the improved MSP protocol](image-url)

The improved MSP protocol demonstrated in Figure 4.8 facilitates the propagation of a scenario if a Type A component is not ready to switch mode, because the rejection of a scenario can be initiated by any component aside from the MSDM. However, such an improvement breaks the 2-phase propagation structure, becoming intractable to analyze and verify. For that reason, we in the thesis stick to the original MSP protocol described in Definition 14.
4.3 Guarantee of mode consistency

According to the MSP protocol presented in the previous section, the triggering of a scenario $k$ does not render a mode switch until the MSDM propagates an $msi^k$ that is propagated to all Type A components. For a component $c_i \in \mathcal{PC}$ or $c_i \in \mathcal{CC}$ and $SC_{c_i}^A(k) = \emptyset$, if $T_{c_i}^k = A$, its mode switch starts when an $msi^k$ arrives; for a component $c_i \in \mathcal{CC}$ with $T_{c_i}^k = A$ and $SC_{c_i}^A(k) \neq \emptyset$, its mode switch starts right after $c_i$ propagates an $msi^k$ to $SC_{c_i}^A(k)$. The mode-aware component model in Chapter 2 specifies that a multi-mode component has a unique local configuration for each of its supported modes. The mode switch of each component is performed by reconfiguration, i.e. by changing its configuration in the current mode to the configuration in the new mode.

From the perspective of a multi-mode system, a mode switch for a scenario is in essence the joint mode switches of all Type A components. The predominant concern after the triggering of a mode switch is to guarantee mode consistency. When the system considers a mode switch to be completed, all Type A components must have already switched to their new modes. The hierarchical component structure of a component-based system implies that the mode-switch completion of a composite component depends on the mode-switch completion of its Type A subcomponents. If $\exists c_j \in SC_{c_i}^A(k)$ which has not completed its mode switch for the scenario $k$, the mode switch of $c_i$ should be considered to be incomplete. Therefore, mode-switch completion must be indicated bottom-up. We have proposed a Mode-switch dependency rule which is immediately applied after the MSP protocol to guarantee mode consistency. This rule introduces a new primitive $\textbf{MSC}$ (Mode-Switch Completion), denoted as $msc^k$ for a scenario $k$. An $msc^k$ is an upstream primitive sent from a component to its parent to indicate mode-switch completion. A detailed description of this rule is given below:

**Definition 15. Mode-switch dependency rule:** Let $c_j$ be the MSDM of a scenario $k$. After $c_j$ issues an $msi^k$ to trigger the mode switch,

- Each $c_i \in \mathcal{PC}$ with $T_{c_i}^k = A$ starts mode switch by reconfiguring itself upon receiving the $msi^k$. The mode switch of $c_i$ is completed as it completes its reconfiguration. After mode-switch completion, $c_i$ will send an $msc^k$ to $P_{c_i}$.
- Each $c_i \in \mathcal{CC}$ with $T_{c_i}^k = A$ starts mode switch by reconfiguring itself after propagating an $msi^k$ to $SC_{c_i}^A(k)$. The mode switch of $c_i$ is completed when $c_i$ completes its reconfiguration and $c_i$ has received an $msc^k$ from
4.3 Guarantee of mode consistency

all the components in $SC^A_{c_i}(k)$. After mode-switch completion, $c_i$ will send an $msc^k$ to $P_{c_i}$ if $c_i \neq \text{Top}$.

- If $T_{c_j}^k = A$, the mode switch of the system for $k$ is completed as $c_j$ completes its mode switch; if $T_{c_j}^k = B$, the mode switch of the system for $k$ is completed after $c_j$ has received an $msc^k$ from all the components in $SC^A_{c_i}(k)$.

Our mode-switch dependency rule is demonstrated in Figure 4.9, where a mode switch is completed after the propagation in Figure 4.6. The reconfiguration of each Type A component for the scenario $k$ is represented by black bars in Figure 4.9. The white bars mean that the mode-switch completion of a composite component is blocked by at least one $msc^k$ which is expected but has not arrived yet. For instance, after $c$ completes reconfiguration, it still cannot complete mode switch until it receives the $msc^k$ from $e$. The system mode switch for $k$ is completed as the MSDM $a$ completes its mode switch.

Figure 4.9: Demonstration of the mode-switch dependency rule
Algorithm 6 implements the mode-switch dependency rule, as a function called by algorithms 3-5. Algorithm 6 contains the following new variables and functions:

- **MSC_complete** is a local boolean variable of $c_i \in CC$ returning true when $c_i$ has received an **MSC** from all its Type A subcomponents.

- **Reconfiguration($c_i, m_{c_i}, m_{c_i}^{new}$)** is a function reconfiguring $c_i$ from its current mode $m_{c_i}$ to the new mode $m_{c_i}^{new}$.

- **Run($c_i, m_{c_i}^{new}$)** is a function starting the execution of $c_i$ in its new mode $m_{c_i}^{new}$.

In addition to Algorithm 6, a composite component should run another parallel task **Collect_MSC($c_i, k$)** for collecting **MSC** primitives from its Type A subcomponents. This task, described in Algorithm 7, can be implemented in different ways. For instance, shown in Line 4 of Algorithm 6, it can be implemented as an aperiodic task activated when a composite component executes the mode-switch dependency rule. Shown in Algorithm 7, when a composite component $c_i$ has received an $msc^k_i$ from all its Type A subcomponents, the boolean variable **MSC_complete** will be set to true so that Algorithm 6 can proceed from lines 9-10.

### 4.4 Atomic execution

A side effect of the MSP protocol is that an ongoing execution of a component may be suspended due to the propagation of a scenario. A component is required to immediately suspend its execution either when it issues an **MSQ** as the MSDM or when it receives an **MSQ**. Such a transient suspension is usually tolerable, however, a potential problem arises if a component has atomic execution which must run to completion without being interrupted. Atomic execution is rather common in real-world applications. For instance, each transaction of the online transfer system in a bank is typically atomic. The transaction should be either successfully completed or aborted before it starts. If the transaction is aborted on the fly, the transferred money may not arrive at the receiver’s account even after it has already been deducted from the sender’s account. This is certainly not desirable for the bank and its customers. To cope with atomic execution, we are going to extend the MSP protocol in this section.

For a component-based system, atomic execution may exist within an individual component or throughout a group of components. The identification of
4.4 Atomic execution

Algorithm 6 Mode_switch\((c_i, k)\)
1: \(MSC\_complete := true;\)
2: if \((c_i \in CC) \land (SC_{c_i}^A(k) \neq \emptyset)\) then
3: \(MSC\_complete := false;\)
4: Activate Collect_MSC\((c_i, k)\);
5: end if
6: if \(T_{c_i}^k = A\) then
7: \(Reconfiguration(c_i, m_{c_i}, m_{c_i}^{new});\)
8: end if
9: while \(!MSC\_complete\) do
10: end while
11: if \((c_i \neq \text{Top}) \land (T_{c_i}^k = A)\) then
12: \(Signal(c_i, p^{MS}_i, m_{c_i}^k);\)
13: end if
14: if \(T_{c_i}^k = A\) then
15: \(Run(c_i, m_{c_i}^{new});\)
16: else \(T_{c_i}^k = B\) \}
17: \(Resume(c_i, m_{c_i});\)
18: end if

Algorithm 7 Collect_MSC\((c_i \in CC, k)\)
1: \(n := 0;\)
2: while \(n < |SC_{c_i}^A(k)|\) do
3: \(Wait(c_i, p_{in}^{MS}_i, \text{primitive});\)
4: if \((\text{primitive} = m_{c_i}^k) \land (c_j \in SC_{c_i}^A(k))\) then
5: \(n := n + 1;\)
6: end if
7: end while
8: \(MSC\_complete := true;\)
atomic execution throughout multiple components is dependent on the system architectural style and component execution semantics. Our focus is on the pipes and filters architectural style [45] which is fairly prevalent in automotive, multimedia, and telecommunications applications. Such an architectural style is very suitable to demonstrate atomic execution. The example in Figure 1.2 is actually a typical pipe-and-filter system, where all components follow the same execution pattern: wait for the input data from its input ports, process the data, and produce the output data at its output ports. We have introduced the concept of Atomic Execution Group (AEG) to specify atomic execution in a pipe-and-filter component-based system:

**Definition 16. Atomic Execution Group:** An AEG is a single component or a group of components with atomic execution. The input and output ports of an AEG belong to its enclosed components. For an AEG $G$ and a component $c_i \in G$, there must exist a data flow from one or more of the input ports of $G$ to one or more of the output ports of $G$ through $c_i$.

For a multi-mode system with multi-mode components, an AEG can be independently specified for different modes. Figure 4.10 illustrates the internal component connections of a multi-mode component $a$, together with specification of AEGs. When $a$ runs in mode $m_1^a$, two AEGs are specified among the subcomponents of $a$. AEG 1 is a single component $f$, while AEG 2 includes two components, $i$ and $j$. When $a$ runs in $m_2^a$, AEG 1 and AEG 2 are replaced with AEG 3 comprised of $b$, $c$, $f$, and $g$. Component $e$, i.e. a subcomponent of $c$, also belongs to AEG 3. However, $d$, the other subcomponent of $c$, is deactivated when $a$ runs in $m_2^a$, ergo excluded from AEG 3.

There are two general rules for the AEG specification among the subcomponents of a multi-mode composite component $c_i$:

- If $c_i \in CC$ has multiple AEGs defined among its subcomponents, there should be no overlap between different AEGs when $c_i$ is running in the same mode. Otherwise, overlapped AEGs should be merged as a single AEG.

- If $c_i \in CC$ is an AEG in one mode, then all its activated subcomponents must also be in the same AEG. However, it is allowed to have an AEG fully contained in a bigger AEG, which then absorbs the smaller one.

In our mode-aware component model, AEG can be regarded as a mode-dependent property of a multi-mode component. Each component can indicate if it has atomic execution for a particular mode. Moreover, each composite
atomic execution throughout multiple components is dependent on the system architectural style and component execution semantics. Our focus is on the pipes and filters architectural style [45] which is fairly prevalent in automotive, multimedia, and telecommunications applications. Such an architectural style is very suitable to demonstrate atomic execution. The example in Figure 1.2 is actually a typical pipe-and-filter system, where all components follow the same execution pattern: wait for the input data from its input ports, process the data, and produce the output data at its output ports. We have introduced the concept of Atomic Execution Group (AEG) to specify atomic execution in a pipe-and-filter component-based system:

Definition 16. Atomic Execution Group:
An AEG is a single component or a group of components with atomic execution. The input and output ports of an AEG belong to its enclosed components. For an AEG $G$ and a component $c_i \in G$, there must exist a data flow from one or more of the input ports of $G$ to one or more of the output ports of $G$ through $c_i$.

For a multi-mode system with multi-mode components, an AEG can be independently specified for different modes. Figure 4.10 illustrates the internal component connections of a multi-mode component $a$, together with specification of AEGs. When $a$ runs in mode $m_1$, two AEGs are specified among the subcomponents of $a$. AEG 1 is a single component $f$, while AEG 2 includes two components, $i$ and $j$. When $a$ runs in $m_2$, AEG 1 and AEG 2 are replaced with AEG 3 comprised of $b$, $c$, $f$, and $g$. Component $e$, i.e. a subcomponent of $c$, also belongs to AEG 3. However, $d$, the other subcomponent of $c$, is deactivated when $a$ runs in $m_2$, ergo excluded from AEG 3.

There are two general rules for the AEG specification among the subcomponents of a multi-mode composite component $c_i$:

- If $c_i \in CC$ has multiple AEGs defined among its subcomponents, there should be no overlap between different AEGs when $c_i$ is running in the same mode. Otherwise, overlapped AEGs should be merged as a single AEG.
- If $c_i \in CC$ is an AEG in one mode, then all its activated subcomponents must also be in the same AEG. However, it is allowed to have an AEG fully contained in a bigger AEG, which then absorbs the smaller one.

In our mode-aware component model, AEG can be regarded as a mode-dependent property of a multi-mode component. Each component can indicate if it has atomic execution for a particular mode. Moreover, each composite component can specify its enclosed AEGs for each mode and the subcomponents belonging to each AEG. For instance, the atomic execution within $a$ in Figure 4.10 can be formally specified as follows:

\[
a.\text{Atomicity} = \{m_1^a \rightarrow \text{Non-atomic}, m_2^a \rightarrow \text{Non-atomic}\} \\
a.\text{AEG} = \{m_1^a \rightarrow \{\{f\}, \{i, j\}\}, m_2^a \rightarrow \{\{b, c, f, g\}\}\}
\]

Here $\text{Atomicity}$ is a function mapping each mode of $a$ to either atomic or non-atomic. $\text{AEG}$ is another function mapping each mode of $a$ to the specified AEGs among the subcomponents of $a$.

The AEG specification plays a fundamental role in handling atomic execution. Although the propagation of a scenario should not suspend the ongoing atomic execution of an AEG, the atomic execution time of each AEG must be bounded. When there is no ongoing atomic execution, it should be allowed to suspend the current (non-atomic) execution of an AEG. Therefore, the basic idea of extending the MSP protocol is to delay the propagation of an MSQ to an AEG until its ongoing atomic execution is completed. This requires that a non-atomic composite component $c_i$ with one or more AEGs among its subcomponents can monitor the execution status of each AEG at runtime. The execution status of an AEG can be either atomic or non-atomic. Component $c_i$
is only allowed to propagate an **MSQ** to the components belonging to an AEG if the execution status of the AEG is non-atomic. Moreover, before monitoring the execution status of each AEG among the subcomponents, \( c_i \) should freeze the inputs of all these AEGs so that no new data will enter. Without this action, an ongoing atomic execution of an AEG may never stop due to incessant input data flow. As \( c_i \) resumes its suspended execution in the old mode or starts execution in the new mode, it will unfreeze the inputs of the AEGs that it has frozen before. To handle atomic execution properly during the propagation of a scenario, the MSP protocol can be extended by the following **MSQ delaying rule**:

**Definition 17.** **MSQ delaying rule:** Let \( c_i \in CC \) be a non-atomic component that is about to propagate an \( msq^k \) to \( SC^{A}_{c_i}(k) \). Suppose that \( c_i \) contains a number of AEGs among \( SC_{c_i} \) for its current mode. Let \( \mathcal{G}_{c_i} = \{G_1, G_2, \ldots, G_n\} \) \((n \in \mathbb{N})\) be the set of these AEGs. Then \( c_i \) immediately propagates an \( msq^k \) to each subcomponent \( c_j \in SC^{A}_{c_i}(k) \), if \( c_j \) does not belong to any AEG \( \mathcal{G}_l \) \((l \in [1, n])\). For each subcomponent \( c_o \in SC^{A}_{c_i}(k) \) and \( \exists l \in [1, n] \) such that \( c_o \in \mathcal{G}_l \), \( c_i \) freezes the input of \( \mathcal{G}_l \) and checks its execution status. If there is no ongoing atomic execution within \( \mathcal{G}_l \), \( c_i \) will immediately propagate an \( msq^k \) to \( c_o \). Otherwise, \( c_i \) delays the \( msq^k \) propagation to \( c_o \) until the ongoing atomic execution of \( \mathcal{G}_l \) is completed. Component \( c_i \) unfreezes the input of \( \mathcal{G}_l \) when (1) \( c_i \) resumes its suspended execution in the old mode; or (2) \( c_i \) starts execution in the new mode.

The MSQ delaying rule does not address how a component should freeze or unfreeze the input of an enclosed AEG, or how a component monitors the execution status of an enclosed AEG. These tasks are both application-dependent and irrelevant to mode switch, thus out of the scope of this thesis.

To demonstrate the impact of the MSQ delaying rule on the MSP protocol, we revisit the example in Figure 4.4 and introduce atomic execution during the triggering of \( k \). Suppose that \( d \) is an AEG when \( a \) is about to propagate an \( msq^k \). The left part of Figure 4.11 depicts the \( msq^k \) propagation of \( a \) while no ongoing atomic execution in \( d \) is observed. Hence \( a \) immediately propagates the \( msq^k \) to its Type A subcomponents \( c \) and \( d \). By contrast, an ongoing atomic execution in \( d \), represented by the grey bar in Figure 4.11, is observed when \( a \) propagates the \( msq^k \). As a consequence, \( a \) can immediately propagate the \( msq^k \) to \( c \), whereas the \( msq^k \) from \( a \) to \( d \) is postponed until the atomic execution in \( d \) is completed.

The MSQ delaying rule is implemented as Algorithm 8, which simply replaces the action \( \forall c_j \in SC^{A}_{c_i}(k) : Signal(c_i, p_{in}^{MS}, msq^k_{c_j}(m_{new})) \) in algo-
rithms 1, 4, and 5. Low-level application-dependent actions such as freezing and unfreezing the input of an AEG is excluded from Algorithm 8 for the sake of lucidity. Moreover,

- *Atomic* is a local boolean variable returning true if \( c_i \) has atomic execution for the current mode. If *Atomic* evaluates to true for \( c_i \), then \( c_i \) itself can be an AEG or \( c_i \) belongs to an AEG. In either case, there is no need for \( c_i \) to apply the MSQ delaying rule, which is applied by the parent of the AEG instead.

- \( \mathcal{G}_l \) (\( l \in [1, n] \)) (see Definition 17) is an AEG among \( SC_{c_i} \), where \( n \) is the number of AEGs enclosed in \( c_i \).

- \( ES_{\mathcal{G}_l} \) returns the current execution status (either *atomic* or *non-atomic*) of an AEG \( \mathcal{G}_l \). If there is an ongoing atomic execution in \( \mathcal{G}_l \), then \( ES_{\mathcal{G}_l} = \text{atomic} \). Otherwise, \( ES_{\mathcal{G}_l} = \text{non-atomic} \).

Algorithm 8 is a clear indication that the only impact of the MSQ delaying rule on the MSP protocol is the delayed MSQ propagation to certain components with atomic execution. As a result, atomic execution may prolong mode-switch time, yet without altering the essence of the MSP protocol.
Algorithm 8  $MSQ\_\text{propagation}(c_i \in CC, k)$

1: if $\neg$Atomic then
2:   for all $c_j \in SC_{c_i}^A(k)$ \& ($\forall l \in [1, n], c_j \notin G_l$) do
3:     Signal($c_i, p_{in}^{MS}, msq^k_{c_j}(m_{c_j}^{new})$);
4:   end for
5:   for all $c_j \in SC_{c_i}^A(k)$ \& ($\exists l \in [1, n], c_j \in G_l$) do
6:     while $ES_{G_i} = \text{atomic}$ do
7:       Signal($c_i, p_{in}^{MS}, msq^k_{c_j}(m_{c_j}^{new})$);
8:     end while
9:   end for
10: else $\{\text{Atomic}\}$
11:   $\forall c_j \in SC_{c_i}^A(k) : \text{Signal}(c_i, p_{in}^{MS}, msq^k_{c_j}(m_{c_j}^{new}))$;
12: end if

4.5 Concurrent mode-switch triggering

The previous sections of this chapter have explained how a mode switch is performed after the triggering of a single scenario, assuming no concurrent triggering of multiple scenarios. However, a system may have multiple MSSs that could trigger different scenarios concurrently. Multiple scenarios can be propagated to the same component at any time, even during a mode switch. This section introduces new techniques for the handling of concurrent scenarios.

4.5.1 MSR and MSQ queues

When multiple scenarios are concurrently triggered, more than one scenario can arrive at a component simultaneously or within a short interval, whereas the component can only handle a single scenario each time. Therefore, it is necessary for each component to buffer incoming scenarios. Section 4.2 already tells that the first primitive that a component may receive for a scenario $k$ is either an $msr^k$ or an $msq^k$, when the propagation of $k$ follows the MSP protocol. To buffer multiple $\text{MSR}$ and $\text{MSQ}$ primitives for different scenarios, we equip each component with an $\text{MSR queue}$ and an $\text{MSQ queue}$:

**Definition 18. MSR and MSQ queues:** An MSR queue of $c_i$, denoted as $c_i.Q_{msr}$, is a FIFO queue storing any $msr^k_{c_j}$ from $c_j \in SC_{c_i}$ (or any $msr^k_{c_i}$ from $c_i$ itself if $c_i$ is the MSS and $c_i \neq \text{Top}$). An MSQ queue of $c_i$, denoted as $c_i.Q_{msq}$, is a FIFO queue storing any $msq^k$ from $P_{c_i}$ (or from $c_i$ itself if $c_i$ is the MSS and $c_i = \text{Top}$).
4.5 Concurrent mode-switch triggering

For a queue $Q$ which is either $c_i.Q_{msr}$ or $c_i.Q_{msq}$, we use $Q[1]$ to denote the first element in $Q$, $x \in Q$ to denote that $x$ is one element in $Q$, and $Q = \emptyset$ or $Q \neq \emptyset$ to denote that $Q$ is empty or non-empty, respectively.

Definition 18 gives the conditions for a component to enqueue an MSR or MSQ. Initially, when an MSS $c_i$ triggers a scenario $k$, $c_i$ will put an $msr^k_{c_i}$ (denoting an $msr^k$ from $c_i$) in $c_i.Q_{msr}$ if $c_i \neq Top$, or $c_i$ will put an $msq^k$ in $c_i.Q_{msq}$ if $c_i = Top$. In addition, when $c_i$ receives an $msr^k_{c_j}$ from $c_j \in SC_{c_i}$, $c_i$ will put the $msr^k_{c_j}$ in $c_i.Q_{msr}$; when $c_i$ receives an $msq^k$ from $P_{c_i}$, $c_i$ will put the $msq^k$ in $c_i.Q_{msq}$. Since both $c_i.Q_{msr}$ and $c_i.Q_{msq}$ are FIFO queues, the earlier an $msr^k$ or $msq^k$ arrives at $c_i$, the closer it is towards the head of the corresponding queue.

When $c_i$ completes a mode switch for $k$ or rejects $k$, then (1) if $c_i.Q_{msr}[1] = msr^k$, $c_i$ will remove the $msr^k$ from $c_i.Q_{msr}$; (2) if $c_i.Q_{msq}[1] = msq^k$, $c_i$ will remove the $msq^k$ from $c_i.Q_{msq}$. According to the MSP protocol and the mode-switch dependency rule, the dequeuing of an MSR or MSQ occurs when any one of the following conditions is satisfied:

1. $c_i$ completes a mode switch for $k$.
   \[(T_{c_i}^k = A)\]

2. $c_i$ has received an $msc^k$ from all components in $SC^A_{c_i}(k)$.
   \[(T_{c_i}^k = B)\]

3. $c_i$ receives an $msd^k$ from $P_{c_i}$.
   \[((SC^A_{c_i}(k) = \emptyset) \lor (c_i.Q_{msr}[1] = msr^k_{c_i}))\]

4. $c_i$ has propagated an $msd^k$ to $SC^A_{c_i}(k)$.
   \[((SC^A_{c_i}(k) \neq \emptyset) \land (c_i.Q_{msq}[1] = msq^k))\]

5. $c_i$ has sent an $msd^k$ to $c_j \in SC_{c_i}$.
   \[((c_i.Q_{msr}[1] = msr^k_{c_j}) \land (c_i.Q_{msq}[1] \neq msq^k))\]

The first two conditions imply that a mode switch has been completed for $k$, while the last three conditions imply that $c_i$ rejects $k$. Under Condition (3), $c_i$ rejects $k$ without propagating the $msd^k$ further. This occurs when $c_i$ has no Type A subcomponents for $k$ or $c_i$ is the MSS for $k$ (i.e. $c_i.Q_{msr}[1] = msr^k_{c_i}$). The difference between Condition (4) and Condition (5) is that $c_i$ has previously propagated an $msq^k$ to $SC^A_{c_i}(k)$ under Condition (4) but not Condition (5).
A component $c_i$ is supposed to periodically check both its MSR queue and MSQ queue and handle multiple scenarios sequentially. However, $c_i$ should focus on the current scenario while $c_i$ is in a transition state:

**Definition 19. Transition state:** A component $c_i$ is in a transition state for a scenario $k$ within the interval $[t_1, t_2]$, where $t_1$ is the time when $c_i$ suspends its execution due to an $msq^k$, and $t_2$ is the time when $c_i$ either rejects $k$ or completes its mode switch for $k$.

It can be observed that a transition state implies the suspended execution of a component. An MSS should not trigger any scenario in a transition state. A transition state protects the handling of the current scenario by a component from the interference of other scenarios. We propose an MSR/MSQ queue checking rule for each component to periodically check its MSR queue and MSQ queue:

**Definition 20. MSR/MSQ queue checking rule:** If $c_i$ is not in a transition state, then:

- If $c_i.Q_{msq} \neq \emptyset$, $c_i$ will immediately handle $c_i.Q_{msq}[1]$.
- If $(c_i.Q_{msq} = \emptyset) \land (c_i.Q_{msr} \neq \emptyset)$ and $c_i.Q_{msr}[1]$ has not been propagated to $P_{c_i}$, $c_i$ will immediately handle $c_i.Q_{msr}[1]$.

The handling of each MSR and MSQ follows the MSP protocol and the mode-switch dependency rule.

The MSR/MSQ queue checking rule assigns a higher priority to $c_i.Q_{msq}$. The reason is that if $c_i.Q_{msq}[1] = msq^k$ which comes from $P_{c_i}$, then $P_{c_i}$ must be in a transition state for $k$ by Definition 19. Hence $P_{c_i}$ will not handle any other scenario until it leaves the transition state for $k$. If $c_i$ first handles an $msr^{k'} \in c_i.Q_{msr}$ instead and $c_i$ needs to propagate the $msr^{k'}$ to $P_{c_i}$, a deadlock will arise as the handling of $k'$ by $c_i$ is blocked by $P_{c_i}$ that is waiting for an $msok^k$ or $msnok^k$ from $c_i$.

Once $c_i$ enters a transition state for $k$, it will temporarily ignore all the other incoming scenarios which are enqueued and handled afterwards.

Note that the MSR/MSQ queue checking rule implies a time-triggered pattern. An alternative could be an event-triggered pattern such that the MSRM of a component remains idle until a new primitive arrives. In this way, the event-triggered pattern may reduce the runtime computation overhead of the MSRM. However, the event-triggered pattern can hardly facilitate the handling of concurrent scenarios, since for both patterns, any incoming MSR or MSQ must be enqueued when a component is in a transition state for another scenario.
4.5 Concurrent mode-switch triggering

4.5.2 The validity of pending scenarios

A remarkable issue of the concurrent triggering of multiple scenarios is that the mode switch of a component based on one scenario may invalidate some pending MSR primitives in its MSR queue. To demonstrate the impact of one scenario on subsequent scenarios, consider the example in Figure 1.2, where the mode mappings of $a$ and $c$ are re-specified in tables 4.1 and 4.2, respectively. Note that the mode mapping of $c$ in Table 4.2 is different from its previous mode mapping in Table 3.3. Figure 4.12 depicts three concurrent scenarios triggered by different components: (1) $k_1 = (b : m^1_b \rightarrow m^2_b)$; (2) $k_2 = (e : m^1_e \rightarrow m^2_e)$; (3) $k_3 = (f : m^1_f \rightarrow m^2_f)$. In Figure 4.12, the MSS of each scenario is marked in red while the corresponding MSDM is marked in blue. Type A components for each scenario are enclosed in a dotted loop. Apparently, all components are Type A components for $k_1$ and $k_3$, whereas only $c$ and $e$ are Type A components for $k_2$. For each scenario, the current mode and new mode of each Type A component are both displayed in Figure 4.12. For instance, when $b$ triggers $k_1$, $m^1_f \rightarrow m^2_f$ means that $f$ needs to switch from $m^1_f$ to $m^2_f$ as a Type A component for $k_1$. The mode mappings of $a$ and $c$ imply that when $b$ runs in $m^1_b$, $e$ can run in either $m^1_e$ or $m^2_e$ and $f$ must run in $m^1_f$. Therefore, all three scenarios can be simultaneously triggered.

Table 4.1: The mode mapping table of $a$

<table>
<thead>
<tr>
<th>Component</th>
<th>Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$m^1_a$, $m^2_a$</td>
</tr>
<tr>
<td>$b$</td>
<td>$m^1_b$, $m^2_b$</td>
</tr>
<tr>
<td>$c$</td>
<td>$m^1_c$, $m^2_c$, $m^3_c$</td>
</tr>
<tr>
<td>$d$</td>
<td>Deactivated, $m^1_d$</td>
</tr>
</tbody>
</table>

Suppose that $k_1$ and $k_3$ are triggered at the same time. Then $a$ will receive an $msr^k_b$ from $b$ and an $msr^k_c$ which is originally issued by $f$. If $a$ receives the $msr^k_b$ earlier than the $msr^k_c$, then applying the MSR/MSQ queue checking rule, $a$ will first handle the $msr^k_b$. Suppose that after some time a mode switch for $k_1$ is completed. Then $a$ is supposed to remove the $msr^k_c$ out of its MSR queue and handle the $msr^k_b$. Nonetheless, since $T^k_f = A$, $f$ is already running in its new mode $m^2_f$ when $a$ is ready to handle the $msr^k_b$. As a consequence of the mode switch for $k_1$, $k_3$ becomes invalid because $f$ triggered $k_3$ before its
mode switch for $k_1$. Likewise, as another Type A component for $k_1$, $c$ sent an $msr_{c}^{k_3}$ to $a$ before its mode switch for $k$. The $msr_{c}^{k_3}$ from $c$ to $a$ also becomes invalid after the mode switch for $k_1$. Due to the invalidity of $k_3$ and the $msr_{c}^{k_3}$ propagated from $f$ to $a$ via $c$, components $a$, $c$, and $f$ must remove the $msr_{c}^{k_3}$ from their MSR queues upon mode-switch completion for $k_1$.

Sometimes a pending MSR remains valid in spite of the mode-switch completion of another scenario. Suppose that $k_2$ and $k_3$ are simultaneously triggered. Then $c$ will receive an $msr_{e}^{k_2}$ from $e$ and an $msr_{f}^{k_3}$ from $f$. If the $msr_{e}^{k_2}$ arrives at $c$ earlier, $c$ will first handle the $msr_{e}^{k_2}$. Suppose that a mode switch is completed for $k_2$. The pending $msr_{f}^{k_3}$ in $c.Q_{msr}$ is still valid because $k_2$ does not affect the mode of $f$ which is a Type B component for $k_2$. Hence, upon the mode-switch completion of $k_2$, $c$ must handle the pending $msr_{f}^{k_3}$ in $c.Q_{msr}$ instead of removing the $msr_{c}^{k_3}$ from $c.Q_{msr}$. The validity of a pending MSR is judged as follows:

**Definition 21. The validity of a pending MSR:** Let $c_i$ be a component which has completed a mode switch for a scenario $k$ ($T_{c_i}^k = A$) or has received an $msc^k$ from all components in $SC_{c_i}^A (k)$ ($c_i \in CC \land T_{c_i}^k = B$). If $c_i.Q_{msr} \neq \emptyset$, then for each $msr_{k'} \in c_i.Q_{msr}$ ($k' \neq k$),

- If $c_i \in PC$, then the $msr_{k'}$ must be invalid, since $c_i$ must be the MSS of $k'$.
- If $c_i \in CC$, then the $msr_{k'}$ can come from either $c_i$ or $c_j \in SC_{c_i}$. If the $msr_{k'}$ is from $c_i$ itself, then the $msr_{k'}$ is valid when $T_{c_i}^k = B$ and invalid when $T_{c_i}^k = A$. If the $msr_{k'}$ is from $c_j \in SC_{c_i}$, then the $msr_{k'}$ is valid under one of the two conditions: (1) $T_{c_j}^k = B$; or (2) $T_{c_j}^k = A$ and $c_j$ sends the $msr_{k'}$ to $c_i$ after sending an $msc^k$ to $c_i$ while $c_i$ is waiting for an $msc^k$ from $SC_{c_i}^A (k)$. Otherwise, the $msr_{k'}$ is invalid.
Condition (2) in Definition 21 deserves extra attention. For $c_i \in CC$ and $c_j \in SC_{c_i}(k)$, $c_j$ must send an $msc^k$ to $c_i$ upon mode-switch completion as per the mode-switch dependency rule. After sending the $msc^k$ to $c_i$, $c_j$ runs in a new mode. If $c_j$ triggers another scenario $k'$ in its new mode by sending an $msr^{k'}$ to $c_i$ which has not completed its mode switch for $k$, then this $msr^{k'}$ should be valid even though $T_{c_i} = A$. 

Figure 4.12: Concurrent scenarios
The validity of a pending MSR is further illustrated in Figure 4.13, where a composite component $a$ has three subcomponents $b$, $c$, and $d$. At time $t_0$, $a$ starts propagating an $msi^{k_1}$ to its Type A subcomponents $b$ and $c$ for $k_1$. Meanwhile, there are already three pending MSR primitives in $a.Q_{msr}$: an $msr^{k_1}_b$ from $b$, an $msr^{k_2}_c$ from $c$, and an $msr^{k_3}_d$ from $d$. At $t_1$, $b$ completes its mode switch for $k_1$, sends an $msc^{k_1}$ to $a$, and starts running in its new mode. At a later time $t_2$, $b$ sends an $msr^{k_4}_b$ to $a$ due to the triggering of another scenario $k_4$. This $msr^{k_4}_b$ is valid in that it is triggered after the mode switch of $b$ for $k_1$. The $msr^{k_3}_d$ is also valid because $d$ does not switch mode after it sends the $msr^{k_3}_d$ to $a$. The $msr^{k_2}_c$ is yet invalid because it is sent before the mode switch of $c$ for $k_1$. That is why $a$ removes the $msr^{k_2}_c$ from $a.Q_{msr}$ when it completes its mode switch for $k_1$ at $t_3$. The $msr^{k_4}_b$ is also removed from $a.Q_{msr}$ at $t_3$, as $k_1$ has been completely handled by $a$.

![Figure 4.13: Valid and invalid pending MSR primitives](image)

Compared with a pending MSR, it is much easier to derive the validity of a pending MSQ:

**Definition 22. The validity of a pending MSQ:** Let $c_i$ be a component which has completed a mode switch for a scenario $k$ ($T^k_{c_i} = A$) or has received an $msc^k$ from all components in $SC^A_{c_i}(k)$ ($c_i \in CC \land T^k_{c_i} = B$). If $c_i.Q_{msq} \neq \emptyset$, then for each $msq^{k'}_{c_i} \in c_i.Q_{msr}$ ($k' \neq k$),

- If $(c_i = Top) \land (T^k_{c_i} = A)$, the $msq^{k'}_{c_i}$ is invalid, since $c_i$ must be the MSS for $k'$.
4.5 Concurrent mode-switch triggering

- Otherwise, the msq\(^k\)' is always valid.

Definition 22 implies that if \(c_i \neq \text{Top}\), a current mode switch for \(k\) never invalidates a pending MSR in \(c_i.Q_{\text{msr}}\). Suppose \(\exists \text{msq}^{k'} \in c_i.Q_{\text{msq}}\) when \(c_i\) has completed a mode switch for \(k\) or has received an \(\text{msc}^k\) from all components in \(SC^A_{c_i}(k)\). If \(T_{c_i}^k = A\), then \(c_i\) must not be the MSDM of \(k\) according to the MSP protocol. When \(c_i\) sends an \(\text{msc}^k\) to \(P_{c_i}\), \(P_{c_i}\) must be in the transition state for \(k\), whereas \(P_{c_i}\) is not able to send an \(\text{msq}^{k'}\) to \(c_i\) before leaving the transition state for \(k\). Therefore, \(T_{c_i}^k = B\) and \(c_i\) must be the MSDM of \(k\). Since \(P_{c_i}\) is a Type B component for \(k\), the \(\text{msq}^{k'}\) from \(P_{c_i}\) to \(c_i\) must be valid.

Now that a component is able to derive the validity of the pending MSR or MSQ primitives in its MSR and MSQ queues, each component should update both its MSR and MSQ queues, following the MSR/MSQ queue updating rule:

**Definition 23. MSR/MSQ queue updating rule:** Let \(c_i\) be a component which has completed a mode switch for a scenario \(k\) (\(T_{c_i}^k = A\)) or has received an \(\text{msc}^k\) from all components in \(SC^A_{c_i}(k)\) (\(c_i \in CC \land T_{c_i}^k = B\)). If \(c_i.Q_{\text{msr}} \neq \emptyset\), then \(c_i\) will remove each invalid MSR from \(c_i.Q_{\text{msr}}\); if \(c_i.Q_{\text{msq}} \neq \emptyset\), then \(c_i\) will remove each invalid MSQ from \(c_i.Q_{\text{msq}}\).

The essence of the MSR/MSQ queue updating rule is to remove the pending MSR and MSQ primitives that become invalid due to the mode switch for a previous scenario. Note that if \(c_i\) in Definition 23 has an \(\text{msr}^k\) or an \(\text{msq}^k\) in its MSR and MSQ queues, the \(\text{msr}^k\) or \(\text{msq}^k\) is dequeued based on the five dequeuing conditions listed in Section 4.5.1, but not removed by the MSR/MSQ queue updating rule.

We have devised a set of algorithms which implement our MSRM for the handling of concurrent scenarios. First, the behavior of an MSS implemented as the function \(\text{New\_scenario}(c_i)\) in Algorithm 1 is updated to new function \(\text{New\_scenario2}(c_i)\) in Algorithm 9 which takes MSR and MSQ queues into account. In Algorithm 9, \(\text{enqueue}(A, B)\) is a function enqueuing the primitive \(A\) (either an MSR or an MSQ) in a queue \(B\) (either MSR queue or MSQ queue). We shall also use \(\text{dequeue}(A, B)\) to denote the dequeuing operation. When an MSS \(c_i \neq \text{Top}\) triggers a scenario \(k\), \(c_i\) will first put an \(\text{msr}^k\) in \(c_i.Q_{\text{msr}}\) instead of directly sending the \(\text{msr}^k\) to \(P_{c_i}\). In the same manner, when \(c_i = \text{Top}\) triggers \(k\), \(c_i\) will first put an \(\text{msq}^k\) in \(c_i.Q_{\text{msq}}\) instead of sending the \(\text{msq}^k\) to \(SC^A_{c_i}(k)\). Moreover, \(\text{TransitionS}\) is a local boolean variable of \(c_i\) returning true if \(c_i\) is in a transition state and returning false otherwise. An MSS \(c_i\) is only allowed to trigger a scenario when it is not in a transition state.
Each component \( c_i \) should reserve a dedicated task for receiving incoming **MSR** and **MSQ** primitives which are put in the corresponding queues. Such a task is implemented in Algorithm 10. This algorithm also considers any possible arrival of an \( msd^k \) due to the direct rejection of \( k \) by the MSDM. Algorithm 10 introduces the following new notations:

- \( mscSent_{c_j} \) is a local boolean variable of \( c_i = P_{c_j} \). When \( c_i \) is in a transition state for \( k \), \( mscSent_{c_j} \) returns true if \( c_j \) has already sent an \( msc^k \) to \( c_i \). Otherwise, \( mscSent_{c_j} \) returns false.

- \( valid^k \) is a local boolean variable of \( c_i \) set to true when \( c_i \) receives an \( msr^k_{c_j} \) from \( c_j \in SC_{c_i} \) and \( mscSent_{c_j} \) is true so that the \( msr^k \) will not be removed when \( c_i \) applies the MSR/MSQ queue updating rule.

Apart from these notations, Algorithm 10 calls a function **Clean_queues**\((c_i,k)\) which is implemented in Algorithm 11. This function is called as \( c_i \) leaves the transition state for \( k \), removing any possible \( msr^k \) or \( msq^k \) from its MSR and MSQ queues. In addition, whenever \( c_i \) sends an \( msr^k \) to \( P_{c_i} \), a local boolean variable \( msrSent \) is set to true. Component \( c_i \) resets \( msrSent \) to false by calling **Clean_queues**\((c_i,k)\).

The MSR/MSQ queue checking rule presented in Definition 20 is implemented in Algorithm 12 as another separate task of \( c_i \). The two functions **Handle_MSQ**\((c_i)\) and **Handle_MSR**\((c_i)\) are called as \( c_i \) handles \( c_i.Q_{msq}[1] \) and \( c_i.Q_{msr}[1] \), respectively, following the MSP protocol and the mode-switch dependency rule. Since \( c_i \) has different mode-switch behaviors when \( c_i \in \mathcal{PC}, c_i \in \mathcal{CC} \), or \( c_i = Top \), these functions are implemented in algorithms 13-18,

---

**Algorithm 9 New_scenario2\((c_i)\)**

1: loop
2: if \( \neg TransitS \land MS\_event\_detected \) then
3: \( k = Assign\_scenario\_ID(c_i) \);
4: if \( c_i \neq Top \) then
5: enqueue\((msr^{k}_{c_i}, c_i.Q_{msr})\);
6: else \( \{c_i = Top\} \)
7: enqueue\((msq^{k}_{i}, c_i.Q_{msq})\);
8: end if
9: end if
10: end if
11: end loop
4.5 Concurrent mode-switch triggering

Algorithm 10 Scenario_reception($c_i$)

1: loop
2: \textbf{Wait}($c_i, p^{MS} \lor p^{MS}_\text{in}, \text{primitive}$);
3: if $(\text{primitive} = \text{msr}^k_{c_j}) \land (c_j \in SC_{c_i})$ then
4: if ($c_i \in CC \land \text{mscSent}_{c_j}$) then
5: \text{valid}^k := \text{true};
6: end if
7: \text{enqueue}\left(\text{msr}^k_{c_j}, c_i, \text{Q}_{\text{msr}}\right);
8: else if $\text{primitive} = \text{msq}^k$ then
9: \text{enqueue}\left(\text{msq}^k, c_i, \text{Q}_{\text{msq}}\right);
10: else \{primitive = msd^k\}
11: if ($\neg\text{TransitionS}) \land (\text{primitive} = \text{msd}^k)$ then
12: if ($c_i \in CC \land (c_i, \text{Q}_{\text{msr}}[1] = \text{msr}^k_{c_j}) \land (c_j \in SC_{c_i})$ then
13: \text{Signal}\left(c_i, p^{MS}_{\text{in}}, \text{msd}^k_{c_j}\right);
14: end if
15: \text{Clean_queue}(c_i, k);
16: end if
17: end if
18: end loop

Algorithm 11 Clean_queue($c_i, k$)

1: if $c_i, \text{Q}_{\text{msq}}[1] = \text{msq}^k$ then
2: \text{dequeue}\left(\text{msq}^k, c_i, \text{Q}_{\text{msq}}\right);
3: end if
4: if $c_i, \text{Q}_{\text{msr}}[1] = \text{msr}^k$ then
5: \text{dequeue}\left(\text{msr}^k, c_i, \text{Q}_{\text{msr}}\right);
6: \text{msrSent} := \text{false};
7: end if
covering all cases for \( c_i \). To simplify the presentation, atomic execution is not considered. However, the function \( MSQ\_propagation(c_i, k) \) in Algorithm 8 can be used for \( MSQ \) propagation to support atomic execution. These algorithms divide the MSRM implementation in algorithms 3-6 into two parts, one for handling an \( MSR \), the other for handling an \( MSQ \). Note that the function \( Clean\_queues(c_i, k) \), which is called as \( c_i \) leaves the transition state for \( k \), is used to remove any possible \( msr^k \) or \( msq^k \) from the MSR and MSQ queues of \( c_i \).

The code segments highlighted in red in these algorithms are dedicated to the handling of concurrent scenarios. What is worth further explanation is that both the function \( Scenario\_reception(c_i) \) in Algorithm 10 and the function \( Handle\_MSQ(c_i) \) provide the handling of an incoming \( msd^k \), depending on if \( c_i \) is in a transition state for \( k \). If \( TransitionS \) is true, the \( msd^k \) will be handled by \( Handle\_MSQ(c_i) \); if \( TransitionS \) is false, the \( msd^k \) will be handled by \( Scenario\_reception(c_i) \).

**Algorithm 12 Check_queues\((c_i)\)**

1: loop
2: if \( \neg TransitionS \) then
3: if \( c_i.Q_{msq} \neq \emptyset \) then
4: \( Handle\_MSQ(c_i) \);
5: else
6: if \( (c_i.Q_{msr} \neq \emptyset) \land (\neg msrSent) \) then
7: \( Handle\_MSR(c_i) \);
8: end if
9: end if
10: end if
11: end loop

Algorithms 13, and 15-18 call a function \( Mode\_switch2(c_i, k) \) implemented in Algorithm 19 which updates the function \( Mode\_switch(c_i, k) \) in Algorithm 6. The function \( Mode\_switch2(c_i, k) \) can activate a task \( Collect\_MSC2(c_i, k) \) which replaces the task \( Collect\_MSC(c_i, k) \) in Algorithm 7. Depicted in Algorithm 20, \( Collect\_MSC2(c_i, k) \) is responsible for receiving an \( msr^k \) from \( SC_{c_i}^A(k) \) and updating \( mscSent_{c_j} \) for each \( c_j \in SC_{c_i}^A(k) \). Initially, \( mscSent_{c_j} \) is set to false. After \( c_j \) receives an \( msr^k \) from \( c_j \), \( mscSent_{c_j} \) will be set to true, thereby affecting lines 4-6 of Algorithm 10.

Algorithm 19 also calls another function \( Update\_queues(c_i, k) \) described in Algorithm 21 which implements the MSR/MSQ queue updating rule (Def-
4.5 Concurrent mode-switch triggering

Algorithm 13 \textit{Handle\_MSQ}(c_i \in \mathcal{PC})

1: \( msq_{c_i}^k(m_{c_i}^{new}) := c_i.Q_{msq}[1]; \)
2: \( TransitionS := true; \)
3: \( Suspend(c_i, m_{c_i}); \)
4: \( \text{if } S_{c_i}^k \text{ then} \)
5: \( \text{Signal}(c_i, p^{MS}, msok_{c_i}^k); \)
6: \( \text{else } \neg S_{c_i}^k \text{ } \)
7: \( \text{Signal}(c_i, p^{MS}, msnok_{c_i}^k); \)
8: \( \text{end if} \)
9: \( \text{Wait}(c_i, p^{MS}, \text{primitive}); \)
10: \( \text{if } \text{primitive} = msr_{c_i}^k \text{ then} \)
11: \( \text{Mode\_switch2}(c_i, k); \)
12: \( \text{else } \text{primitive} = msd_{c_i}^k \text{ } \)
13: \( \text{if } TransitionS \text{ then} \)
14: \( \text{Clean\_queues}(c_i, k); \)
15: \( TransitionS := false; \)
16: \( \text{Resume}(c_i, m_{c_i}); \)
17: \( \text{end if} \)
18: \( \text{end if} \)

Algorithm 14 \textit{Handle\_MSR}(c_i \in \mathcal{PC})

1: \( msr_{c_i}^k(m_{c_i}, m_{c_i}^{new}) := c_i.Q_{msr}[1]; \)
2: \( \text{Signal}(c_i, p^{MS}, msr_{c_i}^k(m_{c_i}, m_{c_i}^{new})); \)
3: \( msrSent := true; \)
Algorithm 15 Handle_MSQ($c_i \in \mathcal{C}$)

1: $msq^k_{c_i} (m_{c_i}^{new}) := c_i, Q_{msq}[1]$;
2: $TransitionS := true$;
3: $Suspend(c_i, m_{c_i})$;
4: if $S^k_{c_i}$ then
5: $Mode\_mapping(c_i, m_{c_i}^{new})$;
6: $\forall c_j \in SC^A_{c_i}(k) : Signal(c_i, p_{in}^{MS}, m_{c_j}^k (m_{c_j}^{new}))$;
7: $Collect\_response(c_i, k)$;
8: if $All\_ready$ then
9: $Signal(c_i, p_{in}^{MS}, m_{sok}^k_{c_i})$;
10: $Wait(c_i, p_{in}^{MS}, primitive)$;
11: if $primitive = msi^k_{c_i}$ then
12: $\forall c_j \in SC^A_{c_i}(k) : Signal(c_i, p_{in}^{MS}, msi^k_{c_j})$;
13: $Mode\_switch2(c_i, k)$;
14: else [$primitive = msd^k_{c_i}$]
15: if $TransitionS$ then
16: $\forall c_j \in SC^A_{c_i}(k) : Signal(c_i, p_{in}^{MS}, msd^k_{c_j})$;
17: $Clean\_queues(c_i, k)$;
18: $TransitionS := false$;
19: $Resume(c_i, m_{c_i})$;
20: end if
21: end if
22: else [$\neg All\_ready$]
23: $Signal(c_i, p_{in}^{MS}, m_{sok}^k_{c_i})$;
24: $Wait(c_i, p_{in}^{MS}, primitive)$;
25: if $(primitive = msd^k_{c_i})$ \&\& $TransitionS$ then
26: $\forall c_j \in SC^A_{c_i}(k) : Signal(c_i, p_{in}^{MS}, msd^k_{c_j})$;
27: $Clean\_queues(c_i, k)$;
28: $TransitionS := false$;
29: $Resume(c_i, m_{c_i})$;
30: else
31: $Report\_error$;
32: end if
33: end if
34: else [$\neg S^k_{c_i}$]
35: $Signal(c_i, p_{in}^{MS}, m_{sok}^k_{c_i})$;
36: $Wait(c_i, p_{in}^{MS}, primitive)$;
37: if $(primitive = msd^k_{c_i})$ \&\& $TransitionS$ then
38: if $(c_i, Q_{msr}[1] = msr^k_{c_i})$ \&\& $(c_o \in SC_{c_i})$ then
39: $Signal(c_i, p_{in}^{MS}, msd^k_{c_o})$;
40: end if
41: $Clean\_queues(c_i, k)$;
42: $TransitionS := false$;
43: $Resume(c_i, m_{c_i})$;
44: else
45: $Report\_error$;
46: end if
47: end if
Algorithm 16 \textit{Handle\_MSR}(c_i \in \mathcal{CC})

1: $msr_{c_o}^k(m_{c_o}, m_{c_i}^{new}) := c_i.Q_{msr}[1]$;
2: $Mode\_mapping(c_i, c_o, m_{c_o}, m_{c_i}^{new})$;
3: if $T_{c_i}^k = B$ then
4: \hspace{1em} if $S_{c_i}^k$ then
5: \hspace{2em} Transition$S := true$;
6: \hspace{2em} Suspend($c_i, m_{c_i}$);
7: \hspace{2em} $\forall c_j \in SC_{c_i}^A(k) : Signal(c_i, p_{im}^{MS}, msq_{c_j}^k(m_{c_j}^{new}))$;
8: \hspace{2em} Collect\_response($c_i, k$);
9: \hspace{2em} if $\neg All\_ready$ then
10: \hspace{3em} $\forall c_j \in SC_{c_i}^A(k) : Signal(c_i, p_{im}^{MS}, msi_{c_j}^k)$;
11: \hspace{3em} Mode\_switch2($c_i, k$);
12: \hspace{2em} else \{$\neg All\_ready$\}
13: \hspace{2em} $\forall c_j \in SC_{c_i}^A(k) : Signal(c_i, p_{im}^{MS}, msd_{c_j}^k)$;
14: \hspace{2em} Clean\_queues($c_i, k$);
15: \hspace{2em} Transition$S := false$;
16: \hspace{2em} Resume($c_i, m_{c_i}$);
17: \hspace{1em} end if
18: \hspace{1em} else \{$\neg S_{c_i}^k$\}
19: \hspace{2em} if $(c_i.Q_{msr}[1] = msr_{c_o}^k) \land (c_o \in SC_{c_i})$ then
20: \hspace{3em} $Signal(c_i, p_{im}^{MS}, msd_{c_o}^k)$;
21: \hspace{2em} end if
22: \hspace{2em} Clean\_queues($c_i, k$);
23: \hspace{1em} end if
24: \hspace{1em} else \{$T_{c_i}^k = A$\}
25: \hspace{2em} if $S_{c_i}^k$ then
26: \hspace{3em} $Signal(c_i, p_{im}^{MS}, msr_{c_i}^k(m_{c_i}, m_{c_i}^{new}))$;
27: \hspace{3em} $msrSent := true$;
28: \hspace{2em} else \{$\neg S_{c_i}^k$\}
29: \hspace{3em} if $(c_i.Q_{msr}[1] = msr_{c_o}^k) \land (c_o \in SC_{c_i})$ then
30: \hspace{4em} $Signal(c_i, p_{im}^{MS}, msd_{c_o}^k)$;
31: \hspace{3em} end if
32: \hspace{3em} Clean\_queues($c_i, k$);
33: \hspace{2em} end if
34: \hspace{1em} end if
Chapter 4. Mode-switch handling at runtime

Algorithm 17 Handle_MSQ($c_i = Top$)

1: $msg^k_m(c_i^{new}) := c_i, Q_{msg}[1];$
2: $Mode\_mapping(c_i, m_{c_i}, m_{c_i}^{new});$
3: $TransitionS := true;$
4: $Suspend(c_i, m_{c_i});$
5: $\forall c_j \in SC_{c_i}^A(k): Signal(c_i, p_{in}^{MS}, msq^k_m(c_j^{new}));$
6: $Collect\_response(c_i, k);$ 
7: $if \ All\_ready \ then$
8: $\forall c_j \in SC_{c_i}^A(k): Signal(c_i, p_{in}^{MS}, ms_{c_j}^k);$
9: $Mode\_switch2(c_i, k);$ 
10: $else \neg\ All\_ready\$
11: $\forall c_j \in SC_{c_i}^A(k): Signal(c_i, p_{in}^{MS}, msd^k_{c_j});$
12: $Clean\_queues(c_i, k);$ 
13: $TransitionS := false;$
14: $Resume(c_i, m_{c_i});$
15: $end if$

Algorithm 18 Handle_MSR($c_i = Top$)

1: $msr^k_m(c_o, m_{c_o}^{new}) := c_i, Q_{msr}[1];$
2: $Mode\_mapping(c_i, c_o, m_{c_o}, m_{c_o}^{new});$
3: $if S_{c_i}^k \ then$
4: $TransitionS := true;$
5: $Suspend(c_i, m_{c_i});$
6: $\forall c_j \in SC_{c_i}^A(k): Signal(c_i, p_{in}^{MS}, msq^k_m(c_j^{new}));$
7: $Collect\_response(c_i, k);$ 
8: $if \ All\_ready \ then$
9: $\forall c_j \in SC_{c_i}^A(k): Signal(c_i, p_{in}^{MS}, ms_{c_j}^k);$
10: $Mode\_switch2(c_i, k);$ 
11: $else \neg\ All\_ready\$
12: $\forall c_j \in SC_{c_i}^A(k): Signal(c_i, p_{in}^{MS}, msd^k_{c_j});$
13: $Clean\_queues(c_i, k);$ 
14: $TransitionS := false;$
15: $Resume(c_i, m_{c_i});$
16: $end if$
17: $else \neg S_{c_i}^k \$
18: $Signal(c_i, p_{in}^{MS}, msd^k_{c_o});$
19: $Clean\_queues(c_i, k);$ 
20: $end if$
Algorithm 19 Mode_switch2($c_i, k$)

1: $MSC_{\text{complete}} := \text{true};$
2: if ($c_i \in CC$) $\land$ ($SC^{A}_{c_i}(k) \neq \emptyset$) then
3:   $MSC_{\text{complete}} := \text{false};$
4:   $\textbf{Activate}$ $\text{Collect}_2MSC(c_i, k);$  
5: end if
6: if $T^k_{c_i} = A$ then
7:   $\text{Reconfiguration}(c_i, m_{c_i}, m_{c_i}^{\text{new}});$  
8: end if
9: while $\neg MSC_{\text{complete}}$ do 
10: end while
11: if ($c_i \neq \text{Top}$) $\land$ ($T^k_{c_i} = A$) then
12:   $\text{Signal}(c_i, p^{MS}, msc_{c_i}^k);$  
13: end if
14: $\text{Update}_\text{queues}(c_i, k);$  
15: $\text{Clean}_\text{queues}(c_i, k);$  
16: $\text{Transition}_S := \text{false};$
17: if $T^k_{c_i} = A$ then
18:   $\text{Run}(c_i, m_{c_i}^{\text{new}});$  
19: else ($T^k_{c_i} = B$)
20:   $\text{Resume}(c_i, m_{c_i});$
21: end if

Algorithm 20 Collect_MSC2($c_i \in CC, k$)

1: $n := 0;$
2: $\textbf{for all } c_j \in SC^{A}_{c_i}(k) \text{ do}$
3:   $msc_{sent_{c_j}} := \text{false};$
4: $\textbf{end for}$
5: $\textbf{while } n < |SC^{A}_{c_i}(k)| \text{ do}$
6:   $\text{Wait}(c_i, p_{in}^{MS}, \text{primitive});$
7:   if ($\text{primitive} = msc_{c_j}^k$) $\land$ ($c_j \in SC^{A}_{c_i}(k)$) then
8:     $n := n + 1;$
9:    $msc_{sent_{c_j}} := \text{true};$
10: $\textbf{end if}$
11: $\textbf{end while}$
12: $MSC_{\text{complete}} := \text{true};$
inition 23) in line with the criteria for judging the validity of pending MSR and MSQ primitives (see definitions 21 and 22). A component \( c_i \) applies the MSR/MSQ queue updating rule right before it leaves a transition state for \( k \), if \( k \) has triggered a mode switch. Algorithm 21 also resets two variables \( \text{valid}^{k'} \) and \( \text{msrSent} \) to false. If \( \exists \text{msr}^{k'}_{c_j} \in c_i.Q_{msr} \) with \( c_j \in SC_{c_i} \) and \( \text{valid}^{k'} = \text{true} \), the validity of this \( \text{msr}^{k'}_{c_j} \) should be re-evaluated when \( c_i \) applies the MSR/MSQ queue updating rule for the next time.

\[ \text{Algorithm 21 Update queues}(c_i,k) \]

1: if \( c_i \in PC \) then
2: for all \( \text{msr}^{k'}_{c_i} \in c_i.Q_{msr} \) do
3: dequeue(\( \text{msr}^{k'}_{c_i},c_i.Q_{msr} \));
4: end for
5: else \( \{ c_i \in CC \} \)
6: if \( T^{k}_{c_i} = A \) then
7: for all \( \text{msr}^{k'}_{c_i} \in c_i.Q_{msr} \) do
8: dequeue(\( \text{msr}^{k'}_{c_i},c_i.Q_{msr} \));
9: end for
10: end if
11: for all \( (\text{msr}^{k'}_{c_j} \in c_i.Q_{msr}) \land (c_j \in SC_{c_i}) \) do
12: if \( (T^{k}_{c_j} = A) \land \neg(\text{valid}^{k'}) \) then
13: dequeue(\( \text{msr}^{k'}_{c_j},c_i.Q_{msr} \));
14: end if
15: end for
16: if \( (c_i = \text{Top}) \land (T^{k}_{c_i} = A) \) then
17: for all \( \text{msq}^{k'}_{c_i} \in c_i.Q_{msq} \) do
18: dequeue(\( \text{msq}^{k'}_{c_i},c_i.Q_{msq} \));
19: end for
20: end if
21: for all \( \text{valid}^{k'} = \text{true} \) do
22: \( \text{valid}^{k'} := \text{false} \);
23: end for
24: end if
25: \( \text{msrSent} := \text{false} \);

As a summary of the implementation of our MSRM for handling concurrent scenarios, the MSRM of each component \( c_i \) can be implemented by three periodic tasks running in parallel:

1. \text{New_scenario2}(c_i) (Algorithm 9): This task is used for an MSS \( c_i \) to
trigger a new scenario.

2. *Scenario_reception*(c_i) (Algorithm 10): This task puts an incoming *MSR* or *MSQ* in the corresponding queues. In addition, it also handles an incoming *MSD* when c_i is not in any transition state.

3. *Check_queues*(c_i) (Algorithm 12): This task handles pending *MSR* and *MSQ* primitives by the MSR/MSQ queue checking rule.

Among these tasks, *Check_queues*(c_i) has dependency on many other algorithms. The relation between these algorithms and *Check_queues*(c_i) are depicted in Figure 4.14. Note that *Collect_MSC2*(c_i, k) is an aperiodic task activated by the function *Mode_switch2*(c_i, k).

![Figure 4.14: The structure of algorithms implementing the task *Check_queues*(c_i)](image)

### 4.5.3 Analysis of the MSR/MSQ queue sizes

Our MSRM satisfies properties such as deadlock freeness. A comprehensive verification of the MSRM will be presented in Chapter 5. In this section, we focus on a key property relating to the MSR queue and MSQ queue introduced
in Section 4.5.1: queue size, a parameter which must be determined with particular care. On the one hand, the queue size must be sufficiently large to avoid overflow; on the other hand, too large queue size with superfluous reservation is a waste of resource.

The queue size of the MSR/MSQ queue of a component highly depends on the scenario triggering regulation of a MSS. Suppose that an MSS \( c_i \) triggers a scenario \( k_1 \) at time \( t_1 \). At a later time \( t_2 \), \( c_i \) has not switched mode and \( c_i \) still has an \( m_{sr}^{k_1} \) in \( c_i.Q_{msr} \). Is \( c_i \) allowed to trigger another scenario \( k_2 \) within the interval \( [t_1, t_2] \)? When \( c_i \) is in a transition state with suspended execution, \( c_i \) should not trigger \( k_2 \). Nonetheless, if there exists a point in \( [t_1, t_2] \) when \( c_i \) is not in any transition state, \( c_i \) could be allowed to trigger \( k_2 \), though the \( m_{sr}^{k_2} \) is in peril of being removed by the MSR/MSQ queue updating rule.

Consider an extreme case that \( c_i \) triggers three consecutive scenarios, putting three primitives \( m_{sr}^{k_1}, m_{sr}^{k_2}, \) and \( m_{sr}^{k_2} \) in \( c_i.Q_{msr} \) before \( c_i \) starts to handle the \( m_{sr}^{k_1} \). If \( c_i \) switches mode for \( k_1 \) or another scenario, the \( m_{sr}^{k_2} \) and \( m_{sr}^{k_3} \) will be both removed.

Without loss of generality, we allow the rapid triggering of multiple scenarios from the same MSS. For each component \( c_i \), let \( \sigma_{c_i} \) be the maximum number of scenarios that \( c_i \) can trigger as an MSS while \( c_i \) runs in the same mode. The analysis of the MSR queue size of a primitive component is given in Theorem 1:

**Theorem 1.** For a component \( c_i \in \mathcal{PC} \), if \( c_i \) is never an MSS, then \( c_i \) requires no MSR queue; if \( c_i \) is an MSS while running in at least one mode, then the minimum required MSR queue size of \( c_i \) is \( \sigma_{c_i} \).

**Proof.** Each \( m_{sr}^k \) in \( c_i.Q_{msr} \) comes from either a subcomponent of \( c_i \) or \( c_i \) itself. Since \( c_i \in \mathcal{PC} \), each \( m_{sr}^k \) can only come from \( c_i \) itself. If \( c_i \) is not an MSS, \( c_i \) will never trigger any scenario. Hence \( c_i \) requires no MSR queue. If \( c_i \) is an MSS while running in at least one mode, in the worst case, \( c_i \) may trigger \( \sigma_{c_i} \) consecutive scenarios before \( c_i \) starts to handle the first scenario. These \( \sigma_{c_i} \) consecutive scenarios require \( \sigma_{c_i} \) slots in \( c_i.Q_{msr} \). Hence the minimum required size of \( c_i.Q_{msr} \) is \( \sigma_{c_i} \).

The analysis of the MSQ queue size of a component \( c_i \) corresponds to two theorems 2 and 3, depending on if \( c_i = Top \) or not.

**Theorem 2.** The minimum required MSQ queue size of a component \( c_i \neq Top \) is 2.

**Proof.** Let \( c_j \) = \( P_{c_i} \). When \( c_i.Q_{msq} = \emptyset \), \( c_i \) can receive an \( m_{sq}^k \) from \( c_j \). By the MSP protocol, after sending the \( m_{sq}^k \) to \( c_i \), \( c_j \) must be in the transition
4.5 Concurrent mode-switch triggering

Theorem 1. For a component in Theorem 1:

- The minimum required size of \( c \).MSQ queue size of a component

Proof. Let \( i \) be the maximum \( i \) before \( c_i \) leaves the transition state for \( k \). If \( c_j \) propagates another \( msq^k \) to \( c_i \) immediately after \( c_j \) leaves the transition state for \( k \) and starts running in the new mode, the \( msq^{k'} \) may arrive at \( c_i \) before \( c_i \) removes the \( msq^k \) from \( c_i \).Q.msq, thus leading to the co-existence of the \( msq^k \) and the \( msq^{k'} \) in \( c_i \).Q.msq. Suppose that \( c_i \) receives the \( msq^{k'} \) at time \( t_1 \) and removes the \( msq^k \) at \( t_2 > t_1 \). Since \( c_j \) is in the transition state for \( k' \) within \([t_1, t_2]\), still waiting for an \( msok^{k'} \) or \( msnok^{k'} \) from \( c_i \), \( c_j \) will not propagate another \( MSQ \) to \( c_i \) at any time in \([t_1, t_2]\). Hence there are always two elements in \( c_i \).Q.msq within \([t_1, t_2]\). After \( c_i \) removes the \( msq^k \) at \( t_2 \), the the \( msq^{k'} \) becomes the only \( MSQ \) in \( c_i \).Q.msq. The number of elements in \( c_i \).Q.msq can never exceed 2.

The reasoning above proves that the minimum required size of \( c_i \).Q.msq is 2 when \( c_i \neq Top \).

Theorem 3. For a component \( c_i = Top \), if \( c_i \) is never an MSS, then \( c_i \) requires no MSQ queue; if \( c_i \) is an MSS while running in at least one mode, then the minimum required MSQ queue size of \( c_i \) is \( \sigma_{c_i} \).

Proof. Since \( c_i \) has no parent, any \( MSQ \) in \( c_i \).Q.msq must come from \( c_i \) itself as the MSS. If \( c_i \) is never an MSS, then \( c_i \) will never trigger any scenario. Hence \( c_i \) requires no MSQ queue. If \( c_i \) is an MSS while running in at least one mode, in the worst case, \( c_i \) may trigger \( \sigma_{c_i} \) consecutive scenarios before \( c_i \) starts to handle the first scenario. These \( \sigma_{c_i} \) consecutive scenarios require \( \sigma_{c_i} \) slots in \( c_i \).Q.msq. Hence the minimum required size of \( c_i \).Q.msq is \( \sigma_{c_i} \).

The MSR queue size of a composite component depends not only on \( \sigma_{c_i} \) but also the number of subcomponents of \( c_i \). Lemma 1 derives the maximum number of elements from the same subcomponent in \( c_i \).Q.msr:

Lemma 1. For \( c_i \in \mathcal{C} \), at any instant, there are at most two MSR primitives from \( c_j \in \mathcal{SC}_{c_i} \) in \( c_i \).Q.msr.
Proof. Suppose that \( \exists \text{msr}^k_{c_j} \in c_i.Q_{msr} \) and \( c_j \in SC_{c_i} \). This implies that \( c_j.Q_{msr}[1] = \text{msr}^k \). Component \( c_j \) will not send another \( \text{msr}^k_{c_j} \) to \( c_i \) until it removes the \( \text{msr}^k \) after leaving the transition state for \( k \) or another scenario \( k'' \).

If \( c_j \) has completed a mode switch, it is supposed to send an \( \text{MSC} \) to \( c_i \). The mode-switch dependency rule guarantees that \( c_j \) removes the \( \text{msr}^k \) from \( c_j.Q_{msr} \) earlier than \( c_i \) removes the \( \text{msr}^k_{c_j} \) from \( c_i.Q_{msr} \). If \( c_j \) triggers another scenario \( k' \) right after the mode switch, then there is a chance that \( c_i \) receives another \( \text{msr}^k_{c_j} \) from \( c_j \) before \( c_i \) removes the \( \text{msr}^k_{c_j} \). Consequently, there are two \( \text{MSR} \) primitives from \( c_j \) in \( c_i.Q_{msr} \). This has been demonstrated in Figure 4.13. Before \( c_j \) removes the \( \text{msr}^k \) or \( \text{msr}^k_{c_j} \) from \( c_i.Q_{msr} \), it is impossible for \( c_j \) to send another \( \text{MSR} \) to \( c_i \) because \( c_j \) has the \( \text{msr}^k_{c_j} \) at the head of \( c_j.Q_{msr} \). Hence the number of elements from \( c_j \) in \( c_i.Q_{msr} \) is at most 2.

If \( k \) is rejected, then \( c_i \) must remove the \( \text{msr}^k_{c_j} \) from \( c_i.Q_{msr} \) before \( c_j \) removes the \( \text{msr}^k \) from \( c_j.Q_{msr} \) due to the downstream propagation of the \( \text{msd}^k \). Consequently, at each instant, there can be at most one element from \( c_j \) in \( c_i.Q_{msr} \).

Therefore, at any time, there are at most two \( \text{MSR} \) primitives from \( c_j \) in \( c_i.Q_{msr} \).

Based on Lemma 1, Theorem 4 analyzes the minimum required MSR queue size of a composite component:

**Theorem 4.** For \( c_i \in CC \), if \( c_i \) is never an MSS or \( c_i = \text{Top} \), then the minimum required MSR queue size of \( c_i \) is \( 2 * |SC_{c_i}| \); if \( c_i \) is an MSS while running in at least one mode and \( c_i \neq \text{Top} \), then the minimum required MSR queue size of \( c_i \) is \( 2 * |SC_{c_i}| + \sigma_{c_i} \).

**Proof.** If \( c_i \) is never an MSS or \( c_i = \text{Top} \), then any \( \text{MSR} \) in \( c_i.Q_{msr} \) must come from a subcomponent of \( c_i \). Lemma 4 has revealed that for each \( c_j \in SC_{c_i} \), there are at most two \( \text{MSR} \) primitives from \( c_j \) in \( c_i.Q_{msr} \) at any instant. The number of \( \text{MSR} \) primitives in \( c_i.Q_{msr} \) reaches the maximum when all the subcomponents of \( c_i \) have sent two \( \text{MSR} \) primitives to \( c_i \). Therefore, the minimum required size of \( c_i.Q_{msr} \) is \( 2 * |SC_{c_i}| \).

If \( c_i \) is an MSS while running in at least one mode and \( c_i \neq \text{Top} \), then apart from the \( \text{MSR} \) primitives from \( SC_{c_i} \), there may be up to \( \sigma_{c_i} \) other \( \text{MSR} \) primitives from \( c_i \) itself in the worst case. Therefore, the minimum required size of \( c_i.Q_{msr} \) is \( 2 * |SC_{c_i}| + \sigma_{c_i} \).
The analysis of MSR/MSQ queue sizes is summarized in Table 4.3, showing the minimum required sizes of both the MSR queue and MSQ queue of an arbitrary component \(c_i\). Here \(c_i \neq MSS\) denotes that \(c_i\) can never be an MSS while \(c_i = MSS\) denotes that \(c_i\) is an MSS while running in at least one mode. Since the queue sizes in Table 4.3 are derived based on the worst-case analysis, our analysis results are safe but also pessimistic. It is assured that no component suffers from overflow in its MSR/MSQ queues. However, since a mode switch is typically a sporadic event, the MSR/MSQ queues of a component should be empty most of the time.

<table>
<thead>
<tr>
<th>(c_i)</th>
<th>MSR queue</th>
<th>MSQ queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) (c_i \in \mathcal{PC}, c_i \neq MSS)</td>
<td>Not required</td>
<td>2</td>
</tr>
<tr>
<td>(2) (c_i \in \mathcal{PC}, c_i = MSS)</td>
<td>(\sigma_{c_i})</td>
<td>2</td>
</tr>
<tr>
<td>(3) (c_i \in \mathcal{CC}, c_i \neq MSS)</td>
<td>(2 \ast</td>
<td>SC_{c_i}</td>
</tr>
<tr>
<td>(4) (c_i \in \mathcal{CC}, c_i = MSS)</td>
<td>(2 \ast</td>
<td>SC_{c_i}</td>
</tr>
<tr>
<td>(5) (c_i = \text{Top}, c_i \neq MSS)</td>
<td>(2 \ast</td>
<td>SC_{c_i}</td>
</tr>
<tr>
<td>(6) (c_i = \text{Top}, c_i = MSS)</td>
<td>(2 \ast</td>
<td>SC_{c_i}</td>
</tr>
</tbody>
</table>

### 4.5.4 Priority MSR/MSQ queues

The MSR queue and MSQ queue of a component are both FIFO queues, implying that all incoming \(\text{MSR}\) and \(\text{MSQ}\) primitives are enqueued based on the arrival time. Notwithstanding, it is often desired that \(\text{MSR}\) and \(\text{MSQ}\) primitives associated with more important scenarios can be placed closer to the head of a queue. To distinguish the importance of different scenarios, we assign a priority to each scenario. Let \(I_k\) denote the priority of a scenario \(k\). If the specification of all scenarios in a system is globally accessible, a global priority assignment for all scenarios is feasible. For instance, Figure 4.15(a) shows three scenarios \(k_1, k_2,\) and \(k_3\) triggered by \(b, d,\) and \(f,\) respectively. The expression \(I_{k_1} > I_{k_2} > I_{k_3}\) signifies that \(k_1\) has a higher priority than \(k_2\) which has a higher priority than \(k_3.\)

Sometimes no global information is available in a component-based system, hindering the global priority assignment of different scenarios. Then the priority of a scenario can be locally assigned within each composite compo-
This is demonstrated by the expression paring the head of pose higher priority than 96 Chapter 4. Mode-switch handling at runtime. Consider a component. Note that scenarios within the composite component may receive an Any incoming instances guarantee that if components e or f. We allow different scenarios to have the same local priority. This is demonstrated by the expression \( I_{k_3} = I_{k_1} \) in Figure 4.15(b). The local priority assignment in Figure 4.15(b) is only valid within a. For example, suppose a belongs to a bigger system with many other components not shown in Figure 4.15(b). Although \( k_2 \) has a higher priority than \( k_1 \) for a, \( k_1 \) may have a higher priority than \( k_2 \) for some other components.

The MSR/MSQ queues of each component must be upgraded from FIFO queues to priority queues so as to take the advantage of local priority assignment. Consider a component \( c_i \) with two priority queues \( c_i.Q_{msr} \) and \( c_i.Q_{msq} \). Any incoming \( msr^k \) (or \( msq^k \)) is enqueued in \( c_i.Q_{msr} \) (or \( c_i.Q_{msq} \)) by comparing \( I_k \) with the priority of other scenarios associated with the primitives in the queue. Taking \( c_i.Q_{msr} \) as an example, we should under most circumstances guarantee that if \( \exists msr^{k_1}, msr^{k_2} \in c_i.Q_{msr} \) with \( msr^{k_1} \) closer to the head of \( c_i.Q_{msr} \), then \( I_{k_2} \leq I_{k_1} \). Nonetheless, since an ongoing mode switch should not be interrupted by another scenario, there are exceptions when \( c_i.Q_{msr}[1] \) cannot be replaced with another MSR associated with a scenario with higher priority. We propose the following MSR enqueuing rule for enqueuing an MSR in a priority MSR queue without interrupting an ongoing mode switch:

**Definition 24. MSR enqueuing rule:** Let \( c_i \) be a component which receives an \( msr^k \). If \( c_i.Q_{msr} = \emptyset \), then \( c_i \) will put the \( msr^k \) at the head of \( c_i.Q_{msr} \).
Otherwise, if $c_i.Q_{msr} \neq \emptyset$, let $c_i.Q_{msr}[1] = msr^{k'}$. Component $c_i$ enqueues the $msr^k$ in $c_i.Q_{msr}$ by comparing $I_k$ with all the other $I_{k''}$ where $msr^{k''} \in c_i.Q_{msr}$ such that for any two $msr^{k_1}$ and $msr^{k_2}$ in $c_i.Q_{msr}$ with $msr^{k_1}$ closer to the head of $c_i.Q_{msr}$, $I_{k_2} \leq I_{k_1}$ holds. However, the $msr^{k'}$ should not be affected by the $msr^k$ when: (1) $c_i$ is in the transition state for $k'$; or (2) $c_i$ has sent the $msr^k$ to $P_{c_i}$ ($c_i \neq Top$). Under each condition, if $I_k > I_{k'}$, the $msr^k$ should be put right after the $msr^{k'}$.

Similarly, a component can apply the following **MSQ enqueuing rule** for enqueuing an **MSQ** in a priority MSQ queue without interrupting an ongoing mode switch:

**Definition 25. MSQ enqueuing rule:** Let $c_i$ be a component which receives an $msq^k$. If $c_i.Q_{msq} = \emptyset$, then $c_i$ will put the $msq^k$ at the head of $c_i.Q_{msq}$. Otherwise, if $c_i.Q_{msq} \neq \emptyset$, let $c_i.Q_{msq}[1] = msq^k$. Then,

- If $c_i = Top$, $c_i$ enqueues the $msq^k$ in $c_i.Q_{msq}$ by comparing $I_k$ with all the other $I_{k''}$ where $msq^{k''} \in c_i.Q_{msq}$ such that for any two $msq^{k_1}$ and $msq^{k_2}$ in $c_i.Q_{msq}$ with $msq^{k_1}$ closer to the head of $c_i.Q_{msq}$, $I_{k_2} \leq I_{k_1}$ holds. However, the $msq^{k'}$ should not be affected by the $msq^k$ if $c_i$ is in the transition state for $k'$. In this case, if $I_k > I_{k'}$, the $msq^k$ should be put right after the $msq^{k'}$.

- If $c_i \neq Top$, $c_i$ enqueues the $msq^k$ after the $msq^{k'}$.

The MSQ enqueuing rule suggests that $c_i.Q_{msq}$ is only treated as a priority queue when $c_i = Top$. When $c_i \neq Top$, we know that there can be at most 2 elements in $c_i.Q_{msq}$. When $c_i$ receives an $msq^k$ with $c_i.Q_{msq}[1] = msq^{k'}$, $c_i$ must be in the transition state for $k'$. Hence the $msq^k$ should not disturb the $msq^{k'}$ irrespective of $I_k$ and $I_{k'}$.

Our MSR enqueuing rule and MSQ enqueuing rule are implemented in Algorithm 22, a function $PriorityQ(primitive, c_i)$ which can replace the function $enqueue()$ called by algorithms 9 and 10. In Algorithm 22, $TS_{c_i} = k'$ denotes that $c_i$ is in the transition state for a scenario $k'$. In addition, it calls a new function $enqueue2(A, B, C)$ which puts a primitive $A$ (either an **MSR** or **MSQ**) in $B$ (either a priority MSR queue or a priority MSQ queue). The boolean variable $C$ is set to true if the first primitive of the priority queue can be replaced with a new primitive associated with a scenario with higher priority.
Algorithm 22 PriorityQ\((\text{primitive}, c_i)\)

1: \textbf{if} \(\text{primitive} = \text{msr}^k\) \textbf{then}
2: \hspace{1em} \textbf{if} \(c_i.Q_{\text{msr}} = \emptyset\) \textbf{then}
3: \hspace{2em} \text{enqueue}(\text{msr}^k, c_i.Q_{\text{msr}});
4: \hspace{1em} \textbf{else}
5: \hspace{2em} \textbf{if} \((c_i.Q_{\text{msr}}[1] = \text{msr}^{k'}) \land \text{Transition}S \land (\text{TS}_{c_i} = k')\) \textbf{then}
6: \hspace{3em} \text{enqueue2}(\text{msr}^k, c_i.Q_{\text{msr}}, \text{false});
7: \hspace{2em} \textbf{else if} \((c_i \neq \text{Top}) \land \text{msrSent}\) \textbf{then}
8: \hspace{3em} \text{enqueue2}(\text{msr}^k, c_i.Q_{\text{msr}}, \text{false});
9: \hspace{2em} \textbf{else}
10: \hspace{3em} \text{enqueue2}(\text{msr}^k, c_i.Q_{\text{msr}}, \text{true});
11: \hspace{1em} \textbf{end if}
12: \textbf{end if}
13: \textbf{end if}
14: \textbf{end if}
15: \textbf{if} \(\text{primitive} = \text{msq}^k\) \textbf{then}
16: \hspace{1em} \textbf{if} \(c_i.Q_{\text{msq}} = \emptyset\) \textbf{then}
17: \hspace{2em} \text{enqueue}(\text{msq}^k, c_i.Q_{\text{msq}});
18: \hspace{1em} \textbf{else}
19: \hspace{2em} \textbf{if} \((c_i = \text{Top})\) \textbf{then}
20: \hspace{3em} \textbf{if} \((c_i.Q_{\text{msq}}[1] = \text{msq}^{k'}) \land \text{Transition}S \land (\text{TS}_{c_i} = k')\) \textbf{then}
21: \hspace{4em} \text{enqueue2}(\text{msq}^k, c_i.Q_{\text{msq}}, \text{false});
22: \hspace{3em} \textbf{else}
23: \hspace{4em} \text{enqueue2}(\text{msq}^k, c_i.Q_{\text{msq}}, \text{true});
24: \hspace{2em} \textbf{end if}
25: \hspace{2em} \textbf{else}
26: \hspace{3em} \text{enqueue}(\text{msq}^k, c_i.Q_{\text{msq}});
27: \hspace{2em} \textbf{end if}
28: \hspace{1em} \textbf{end if}
4.6 Emergency mode switch

In time-critical systems, a scenario may be triggered due to an emergency event. It is vital that a mode switch for such an emergency scenario is performed and completed swiftly, even in the presence of other concurrent scenarios. A vivid example of emergency mode switch is road trains [11] which provide the opportunity for a vehicle to join or leave a platoon of other vehicles which automatically follow a leading vehicle maneuvered by a licensed driver on a highway. To be a member of a platoon, a vehicle should approach the platoon and send a request to the leading vehicle. If the request is approved by the leading driver, the vehicle will become part of the platoon and enter a semi-autonomous control mode in which the vehicle is able to automatically follow the platoon, releasing the driver from the tedious driving. A driver who intends to leave the platoon can simply send a leaving request to inform the leading driver and drive out of the platoon as the car switches to manual mode. Imagine the triggering of an emergency scenario when the leading driver brakes sharply to elude an imminent danger, e.g. when a wild animal leaps onto the road. Then all members of the platoon are expected to immediately switch to an emergency braking mode to avoid rear-end collision. If the triggering of this emergency scenario coincides with the leaving request of a driver, emergency braking must be enforced for the entire platoon before any driver switches to manual mode and leaves the platoon.

Our MSRM presented in the previous subsections becomes less effective to handle emergency scenarios for various reasons:

- When an emergency scenario is propagated by the MSP protocol, the scenario can be rejected even if only a single Type A component is not ready to switch mode.

- Even if we assign the highest priority to an emergency scenario, it may still be delayed by the handling of another scenario.

- There is a risk that a pending MSR or MSQ associated with an emergency scenario is invalidated by the MSR/MSQ queue updating rule.

Therefore, in this section we extend the MSRM to guarantee that an emergency scenario always triggers a mode switch, which is neatly performed. The extension provides a dedicated handling of emergency mode switch without altering the handling of non-emergency scenarios. The extended MSRM is based on the following assumptions:
1. A system has at most one emergency scenario.

2. When an emergency scenario arrives at a Type A component, a direct switch from its current mode to the new mode is always possible, even when the component is not ready to switch mode.

3. An emergency scenario cannot interrupt the ongoing reconfiguration of a component.

Assumption 1 is made to simplify the problem. Even the triggering of a single emergency scenario can substantially complicate the runtime mode-switch handling of each component. It is our future work to support the concurrent triggering of multiple emergency scenarios with different criticality levels. Assumption 2 prevents an emergency scenario from being rejected by forcing all Type A components to switch mode. This is a sensible choice in that it is more important to perform an emergency mode switch than to respect the execution state of an individual component. Assumption 3 implies that the handling of an emergency scenario may be delayed by an ongoing component reconfiguration, however, aborting an ongoing component reconfiguration is a dangerous action which may break the mode consistency between different components.

In this section, we first present a new mechanism for the handling of an emergency scenario without the concurrent triggering of other non-emergency scenarios. Then we identify the issues arising from the concurrent triggering of both emergency and non-emergency scenarios. Finally, we provide the corresponding solutions to the identified issues.

4.6.1 The handling of an emergency scenario

In order to make all components recognize an emergency scenario, we introduce a new primitive EMS (Emergency Mode Switch) for the propagation of an emergency scenario. Let $ems^k$ denote an EMS associated with an emergency scenario $k$. When a component $c_i$ receives an incoming $ems^k$ or triggers an emergency mode switch by issuing an $ems^k$, the $ems^k$ is first put in an EMS queue of $c_i$, denoted as $c_i.Q_{ems}$. Since we assume that at most one emergency scenario is specified in a system, the required size of an EMS queue is 1.

The propagation of an emergency scenario is less complicated than the propagation of a non-emergency scenario, since there is no need to check the current state of each Type A component which must switch mode as requested. We have proposed an Emergency Mode-Switch Propagation (EMSP) protocol dedicated to the propagation of an EMS:
Definition 26. Emergency Mode-Switch Propagation protocol: Let $c_i$ be the MSS of an emergency scenario $k$. When $c_i$ triggers $k$, $c_i$ puts an $ems^k$ in $c_i.Q_{ems}$. When $c_i$ starts to handle the $ems^k$, it will suspend its execution in the current mode. Moreover,

- If $c_i \in \mathcal{PC}$, it will send the $ems^k$ to $P_{c_i}$.
- If $c_i \in \widetilde{cC}$, it will send the $ems^k$ to $P_{c_i}$ and $SC^A_{c_i}(k)$.
- If $c_i = Top$, it will send the $ems^k$ to $SC^A_{c_i}(k)$.

Upon receiving the $ems^k$, each $c_j$ puts the $ems^k$ in $c_j.Q_{ems}$. When $c_j$ starts to handle the $ems^k$, it will suspend its execution in the current mode. Moreover,

- If $c_j \in \mathcal{PC}$, $c_j$ will not propagated the $ems^k$ further.
- If $c_j \in \widetilde{cC}$, it will propagate the $ems^k$ depending on the sender $c_n$ and $T^k_{c_j}$: (1) If $c_n = P_{c_j}$, $c_j$ will propagate the $ems^k$ to $SC^A_{c_j}(k)$; (2) If $c_n \in SC_{c_j}$ and $T^k_{c_j} = A$, then $c_j$ will propagate the $ems^k$ to $\{P_{c_j}\} \cup SC^A_{c_j}(k) \setminus \{c_n\}$; (3) If $c_n \in SC_{c_j}$ and $T^k_{c_j} = B$, then $c_j$ will be the MSDM of $k$, propagating the $ems^k$ to $SC^A_{c_j}(k) \setminus \{c_n\}$.
- If $c_j = Top$, then $c_j$ will propagate the $ems^k$ to $SC^A_{c_j}(k) \setminus \{c_n\}$, where $c_n \in SC_{c_j}$ is the sender of the $ems^k$.

After the propagation of an $ems^k$, each Type A component immediately starts its mode switch while the MSDM, the only Type B component for $k$ with the $ems^k$ in $c_i.Q_{ems}$, waits for an $msc^k$ from $SC^A_{c_i}(k)$. The mode-switch dependency rule in Definition 15 also applies to an emergency mode switch. If atomic execution exists in a system, the designer can make the optimal decision based on the system requirement, e.g. either delaying the propagation of an EMS to protect atomic execution or aborting atomic execution by force to speed up the propagation.

The EMSP protocol is demonstrated in Figure 4.16, where an MSS $e$ triggers an emergency scenario $k$, leading to the mode switches of Type A components $c$, $d$, and $f$. When $e$ triggers $k$, an $ems^k$ is sent to its parent $e$ which further propagates the $ems^k$ upstream to $a$ and downstream to the other Type A subcomponent $f$. After $a$ receives the $ems^k$, it propagates the $ems^k$ to the other Type A subcomponent $d$. All Type A components start mode switch after the
propagation of the $ems^k$. The emergency mode switch is completed as $a$ receives an $msc^k$ from both $c$ and $d$. It is self-evident from this example that the propagation of a scenario following the EMSP protocol is faster than the propagation of the same scenario following the MSP protocol. The EMSP protocol accelerates the propagation from three perspectives. Firstly, Phase 1 of the MSP protocol is removed for the propagation of an emergency scenario because all Type A components must switch mode no matter whether they are ready or not. Secondly, the EMSP protocol allows a component to propagate upstream and downstream EMS primitives simultaneously. For instance, $c$ in Figure 4.16 propagates an $ems^k$ to $a$ and $f$ at the same time. Moreover, a component can start mode switch promptly after the propagation of an $ems^k$ without waiting for the decision of the MSDM, which must always approve an emergency scenario.

Algorithm 23 describes a function $EMSP(c_i, k)$ that implements the EMSP protocol. This function is called by a component $c_i$ running in a mode $m_{c_i}$ when $c_i$ starts to handle an $ems^k$ from $c_i, Q_{ems}$. The expression $ems^k \leftarrow c_i$ means that $c_i$ is the MSS of the $ems^k$. Otherwise, the $ems^k$ may come from either $P_{c_i}$ or a subcomponent $c_l \in SC_{c_i}$. When $c_i$ sends an $ems^k$ to $P_{c_i}$, we use $ems^k(m_{c_i}, m_{new}^{c_i})$ to denote the mode switch of $c_i$ from mode $m_{c_i}$ to mode $m_{new}^{c_i}$. This information is necessary for the mode mapping of $P_{c_i}$. When $c_i$ receives an $ems^k$ from $P_{c_i}$, the $ems^k$ is denoted as $ems^k(m_{new}^{c_i})$ which indicates the new mode $m_{new}^{c_i}$ of $c_i$.  

Figure 4.16: Demonstration of the EMSP protocol
Algorithm 23 $EMSP(c_i, k)$

1: $Suspend(c_i, m_{c_i});$
2: if $ems^k \leftarrow c_i$ then
3: if $c_i \in \mathcal{PC}$ then
4: $Signal(c_i, p^{MS}, ems^k(m_{c_i}, m_{c_i}^{new}));$
5: else
6: $Mode\_mapping(c_i, m_{c_i}^{new});$
7: $\forall c_j \in SC_{c_i}(k): Signal(c_i, p^{MS}, ems^k(m_{c_j}^{new}));$
8: if $c_i \neq Top$ then
9: $Signal(c_i, p^{MS}, ems^k(m_{c_i}, m_{c_i}^{new}));$
10: end if
11: end if
12: else if $ems^k \leftarrow P_{c_i}$ then
13: if $c_i \in \mathcal{PC}$ then
14: return ;
15: else{$c_i \in \mathcal{CC}$}
16: $Mode\_mapping(c_i, m_{c_i}^{new});$
17: $\forall c_j \in SC_{c_i}(k): Signal(c_i, p^{MS}, ems^k(m_{c_j}^{new}));$
18: end if
19: else{$ems^k \leftarrow c_i \in SC_{c_i}$}
20: $Mode\_mapping(c_i, c_i, m_{c_j}, m_{c_i}^{new});$
21: $\forall c_j \in SC_{c_i}(k) \setminus \{c_i\}: Signal(c_i, p^{MS}, ems^k(m_{c_j}^{new}));$
22: if $(T^k_{c_i} = A) \land (c_i \neq Top)$ then
23: $Signal(c_i, p^{MS}, ems^k(m_{c_i}, m_{c_i}^{new}));$
24: end if
25: end if
Apart from the EMSP, if the mode information of all components is globally accessible, the propagation of an emergency scenario can be further facilitated by the concurrent triggering of the same emergency scenario from different components, provided that all EMS primitives share the same scenario ID, and the emergency scenarios triggered by different components result in the same group of Type A components and the same new mode of each Type A component. When the same emergency scenario \( k \) is concurrently triggered from different components, a component \( c_i \) may receive an \( ems^k \) from different senders during the propagation of the \( ems^k \). It is sufficient for \( c_i \) to only put the first incoming \( ems^k \) in \( c_i.Q_{ems} \), while all subsequent \( ems^k \) primitives that arrive before \( c_i \) completes the mode switch for \( k \) will be discarded. However, a more efficient strategy for \( c_i \) is to remember the identities of the senders of each subsequent \( ems^k \) that arrives before \( c_i \) starts to propagate the \( ems^k \). Then there is no need for \( c_i \) to propagate the \( ems^k \) to these senders. For instance, if \( e \) and \( f \) in Figure 4.16 concurrently trigger an \( ems^k \) and the \( ems^k \) from \( e \) arrives at \( c \) earlier, then \( c \) will not propagate the \( ems^k \) to \( f \).

In Section 4.5.1, we have defined a transition state for a non-emergency scenario to protect the handling of one scenario from being interrupted by other scenarios. Similarly, a component enters an Emergency Transition State (ETS) as it starts the handling of an emergency scenario:

**Definition 27. Emergency Transition State:** A component \( c_i \) is in an ETS for a scenario \( k \) within the interval \([t_1, t_2] \), where \( t_1 \) is the time when \( c_i \) starts to handle an \( ems^k \) in \( c_i.Q_{ems} \), and \( t_2 \) is the time when \( c_i \) has completed its mode switch for \( k \) (\( T^k_{c_i} = A \)) or has received an \( msc^k \) from all \( c_j \in SC^A_{c_i}(k) \) (\( T^k_{c_i} = B \)).

The \( ems^k \) is removed by \( c_i \) from \( c_i.Q_{ems} \) as \( c_i \) leaves the ETS for \( k \). Hereafter we use Normal Transition State (NTS) to indicate a transition state (Definition 19) as opposed to ETS. An MSS is not allowed to trigger any scenario in an NTS or ETS.

When the concurrent triggering of both emergency and non-emergency scenarios is considered, each component is supposed to periodically check its EMS, MSQ, and MSR queues. Reading from the EMS queue of a component should have higher priority than reading from its MSQ and MSR queues. Taking the EMS queue into account, we replace the MSR/MSQ queue checking rule (Definition 20) with the *pending scenario checking rule* below:

**Definition 28. Pending scenario checking rule:** If \( c_i \) is not in an NTS or ETS, it periodically checks its EMS queue, MSQ queue, and MSR queue until it identifies a primitive \( x \) that is immediately handled by \( c_i \), where
4.6 Emergency mode switch

- If $c_i.Q_{ems} \neq \emptyset$, then $x = c_i.Q_{ems}[1]$.
- If $(c_i.Q_{ems} = \emptyset) \land (c_i.Q_{msq} \neq \emptyset)$, then $x = c_i.Q_{msq}[1]$.
- If $(c_i.Q_{ems} = \emptyset) \land (c_i.Q_{msq} = \emptyset) \land (c_i.Q_{msr} \neq \emptyset)$ and $c_i.Q_{msr}[1]$ has not been propagated to $P_{c_i}$, then $x = c_i.Q_{msr}[1]$.

The pending scenario checking rule is implemented in Algorithm 24 which updates Algorithm 12. When a component is in an NTS or ETS, the corresponding boolean variable NTS or ETS will be set to true. The new function $HandleEMS(c_i)$, which will be further explained in Section 4.6.3, is called as $c_i$ starts to handle an EMS.

**Algorithm 24 Check_queues2(c_i)**

1: loop
2: if ($\neg$NTS) $\land$ ($\neg$ETS) then
3: if $c_i.Q_{ems} \neq \emptyset$ then
4: $HandleEMS(c_i)$;
5: else if $c_i.Q_{msq} \neq \emptyset$ then
6: $HandleMSQ(c_i)$;
7: else
8: if $(c_i.Q_{msr} \neq \emptyset) \land (\neg msrSent)$ then
9: $HandleMSR(c_i)$;
10: end if
11: end if
12: end if
13: end loop

The MSR/MSQ queue updating rule introduced as Definition 23 still works when emergency mode switch is taken into consideration. When a component applies the MSR/MSQ queue updating rule as it leaves an NTS, it never removes a pending EMS. When a component leaves an ETS, it will apply the same MSR/MSQ queue updating rule, removing invalid pending MSR and MSQ primitives in its MSR and MSQ queues. The validity of a pending MSR or a pending MSQ can still be judged by definitions 21 and 22.

However, a special issue must be addressed in the implementation of the MSR/MSQ queue updating rule. Consider a simple example in Figure 4.17 where three scenarios are concurrently triggered: a non-emergency scenario $k_1$ triggered by $a$, and an emergency scenario $k_2$ and another non-emergency scenario $k_3$ both triggered by $c$. The Type A components for each scenario are enclosed in the corresponding dotted loop. Shown in Figure 4.17, $k_1$ is
the first triggered scenario which leads to the reconfigurations of \( a, b, \) and \( d \). The initial propagation phase of \( k_1 \) is omitted to simplify the view. During the reconfiguration of \( b \), an \( ems^{k_2} \) arrives due to the triggering of \( k_2 \) by \( c \). Since \( b \) has a longer reconfiguration time than \( c \), the reconfiguration of \( b \) is still in progress by the time when \( b \) receives an \( msc^{k_2} \) from \( c \). After the emergency mode switch for \( k_2 \), \( c \) triggers \( k_3 \) at time \( t_0 \) by sending an \( msr^{k_3} \) to \( b \), which receives the \( msr^{k_3} \) before its reconfiguration completion.

As \( b \) receives an \( msc^{k_1} \) from \( d \) at \( t_1 \), \( b \) applies the MSR/MSQ queue updating rule. The pending \( msr^{k_3} \) in \( b \).\( Q_{msr} \) will not be removed because \( c \) is a Type B component for \( k_1 \). However, according to Algorithm 21, the boolean variable \( valid^{k_3} \) is reset to false at \( t_1 \). After \( t_1 \), \( b \) applies the scenario checking rule which suggests that the pending \( ems^{k_2} \) should be handled before the pending \( msr^{k_3} \). Figure 4.17 shows that \( b \) starts to handle the \( ems^{k_2} \) and leaves the ETS for \( k_2 \) at \( t_3 \). Component \( b \) applies the same MSR/MSQ queue updating rule at \( t_3 \). Since \( T_{c}^{k_2} = A \) and \( valid^{k_3} \) is false, according to Algorithm 21, \( b \) will remove the pending \( msr^{k_3} \) from \( b \).\( Q_{msr} \). This is undesired since \( c \) keeps running in the same mode within the interval \([t_0, t_3]\) and the \( msr^{k_3} \) is still valid. We prevent such a problem by extending Algorithm 21 to Algorithm 25, where the red lines make \( b \) set \( valid^{k_3} \) to true at \( t_3 \) because the \( ems^{k_2} \) in \( b \).\( Q_{ems} \) from \( c \) suggests that the pending \( msr^{k_3} \) from \( c \) is valid and must be reserved at \( t_3 \).

Figure 4.17: Demonstration of a pending MSR valid for two rounds
We prevent such a problem by extending Algorithm 21 to Algorithm 25, where

\begin{algorithm}
\caption{Update\_queues2(c_i, k)}
\begin{algorithmic}[1]
\State \textbf{if} $c_i \in \mathcal{P} \mathcal{C}$ \textbf{then}
\State \quad \textbf{for all} $msr_{c_i}^{k'} \in c_i.Q_{msr}$ \textbf{do}
\State \quad \quad dequeue($msr_{c_i}^{k'}, c_i.Q_{msr}$);
\State \quad \textbf{end for}
\State \textbf{else} ($c_i \in \mathcal{C} \mathcal{C}$)
\State \quad \textbf{if} $T_{c_i}^k = A$ \textbf{then}
\State \quad \quad \textbf{for all} $msr_{c_i}^{k'} \in c_i.Q_{msr}$ \textbf{do}
\State \quad \quad \quad dequeue($msr_{c_i}^{k'}, c_i.Q_{msr}$);
\State \quad \quad \textbf{end for}
\State \quad \textbf{end if}
\State \quad \textbf{for all} ($msr_{c_i}^{k'} \in c_i.Q_{msr}$) $\land$ ($c_j \in \mathcal{S} \mathcal{C}_{c_i}$) \textbf{do}
\State \quad \quad \textbf{if} ($\exists msr_{c_j}^{k''} \in c_i.Q_{ems}$) $\land$ ETS \textbf{then}
\State \quad \quad \quad \textbf{valid}^{k'} := true;
\State \quad \quad \textbf{end if}
\State \quad \textbf{end if}
\State \quad \textbf{if} ($T_{c_i}^k = A$) $\land$ ($\neg \text{valid}^{k'}$) \textbf{then}
\State \quad \quad dequeue($msr_{c_i}^{k'}, c_i.Q_{msr}$);
\State \quad \textbf{end if}
\State \textbf{end for}
\State \textbf{if} ($c_i = \text{Top}) \land (T_{c_i}^k = A$) \textbf{then}
\State \quad \textbf{for all} $msq_{c_i}^{k'} \in c_i.Q_{msq}$ \textbf{do}
\State \quad \quad dequeue($msq_{c_i}^{k'}, c_i.Q_{msq}$);
\State \quad \textbf{end for}
\State \textbf{end if}
\State \textbf{for all} \text{valid}^{k'} = true \textbf{do}
\State \quad \text{valid}^{k'} := false;
\State \textbf{end for}
\State \textbf{end if}
\State msrSent := false;
\end{algorithmic}
\end{algorithm}

Figure 4.17: Demonstration of a pending MSR
4.6.2 Issues due to the concurrent triggering of emergency and non-emergency scenarios

The EMSP protocol and the pending scenario checking rule in Section 4.6.1 provide dedicated handling for an emergency scenario, yet ignoring the interplay between an emergency scenario and other concurrently triggered non-emergency scenarios. After a comprehensive analysis of all the possible cases where an emergency scenario interleaves with other non-emergency scenarios, we have identified three major issues related to the concurrent triggering of both emergency and non-emergency scenarios:

**Issue 1:** When a component $c_i \in \mathcal{CC}$ receives an $ems^{k_2}$ from $c_j \in \mathcal{SC}_{c_i}$ after $c_i$ sends an $msr^{k_1}$ to $P_{c_i}$, the $ems^{k_2}$ may make the $msr^{k_2}$ invalid.

Issue 1 is illustrated by Figure 4.18, where $b$ receives an $ems^{k_2}$ ($T_{b}^{k_2} = A$) from $c$ after sending an $msr^{k_1}$ to $a$. This $msr^{k_1}$ may come from $c$, the sender of the $ems^{k_2}$, depicted in Figure 4.18(a), or another subcomponent $d$ depicted in Figure 4.18(b), or $c_i$ itself. Since $b$ is not in the NTS for $k_1$, according to the pending scenario checking rule, $b$ will handle the $ems^{k_2}$ and switch to the new mode, making the $msr^{k_1}$ previously sent to $a$ invalid.

**Issue 2:** When a component $c_i \in \mathcal{CC}$ receives an $ems^{k_2}$ immediately after having propagated an $msq^{k_1}$ to $\mathcal{SC}_{c_i}^A(k_1)$ and before receiving all the replies,
the handling of the $ems^k_2$ by $c_i$ will be blocked by $k_1$. If the $ems^k_2$ comes from $c_j \in SC_{c_i}$, the $ems^k_2$ may make the $msr^k_1$ from $c_i$ to $P_{c_i}$ and the $msq^k_1$ from $c_i$ to $SC_{c_i}(k_1)$ invalid.

Issue 2 is illustrated by Figure 4.19. In Figure 4.19(a), $b$ receives an $msr^k_1$ from $c$. As the MSDM of $k_1$, $b$ propagates an $msq^k_1$ to $c$ and $d$. Before $b$ receives the reply from $c$ or $d$, a downstream $ems^k_2$ from $a$ arrives at $b$ at time $t_0$. Since $b$ has entered the NTS for $k_1$ at $t_0$, it will not handle the $ems^k_2$ until it leaves the NTS for $k_1$. A worse situation is demonstrated in Figure 4.19(b) where $b$ receives an upstream $ems^k_2$ from $c$ at $t_0$. Note that the handling of the $msr^{k_0}$ is pended due to another scenario $k_1$, as $a$ propagates an $msq^k_1$ to $b$ instead of an $msq^{k_0}$. If $T_{b}^{k_2} = A$, the $msr^{k_0}$ from $b$ to $a$ will become invalid due to $k_2$. Moreover, $c$ will switch mode after sending the $ems^k_2$ to $b$. However, $b$ sends an $msq^k_1$ to $c$ assuming that $c$ runs in the old mode. Thus the $msq^{k_1}$ from $b$ to $c$ also becomes invalid due to $k_2$.

![Figure 4.19: Issue 2: An EMS blocked by the handling of a non-emergency scenario](image)

**Issue 3:** If a component $c_i \in CC$ has an $msq^k_1$ in $c_i.Q_{msq}$ while receiving an
$ems^{k_2}$ from $c_j \in SC_{c_i}$, the $ems^{k_2}$ may make the $msq^{k_1}$ invalid.

Issue 3 is illustrated by Figure 4.20, where $b$ receives an $msq^{k_1}$ from $a$ and an $ems^{k_2}$ from $c$ at the same time. Applying the pending scenario checking rule, $b$ will first handle the $ems^{k_2}$. If $T_b^{k_2} = A$, $b$ will switch mode for $k_2$. However, $a$ sends the $msq^{k_1}$ to $b$ when $b$ runs in the old mode. As a consequence, the emergency mode switch of $b$ for $k_2$ makes the $msq^{k_1}$ in $b.Q_{msq}$ invalid.

![Figure 4.20: Issue 3: An upstream EMS makes a pending MSQ invalid](image)

### 4.6.3 Solutions to the identified issues

The issues pinpointed in Section 4.6.2 pose extra challenge to the handling of concurrent emergency and non-emergency scenarios.

Concerning Issue 1, when an MSS $c_i$ triggers an emergency scenario $k$, $c_i$ must be aware if there is any pending $msr^{k'}$ in $c_i.Q_{msr}$ that has been sent from $c_i$ to $P_{c_i}$. If yes, the triggering of $k$ will make the $msr^{k'}$ previously sent from $c_i$ to $P_{c_i}$ invalid. Therefore, $c_i$ should let $P_{c_i}$ abort the handling of $k$ before $c_i$ starts to propagate an $ems^k$. We thus introduce a new upstream primitive **MSA** (Mode-Switch Abort). After $c_i$ sends an $msr^k$ to $P_{c_i}$, $c_i$ can send an $msa^k_{c_i}$ to $P_{c_i}$ which will abort the handling of $k$. The behavior of $c_i$ while triggering an emergency scenario $k$ is described by an emergency scenario triggering rule:

**Definition 29. Emergency scenario triggering rule:** When an MSS $c_i$ wants to trigger an emergency scenario $k$, before $c_i$ puts an $ems^k$ in $c_i.Q_{ems}$, if there is a pending $msr^{k'} = c_i.Q_{msr}[1]$ which has been sent to $P_{c_i}$, $c_i$ will send an $msa^{k'}_{c_i}$ to $P_{c_i}$.

Note that there is no need for the MSS $c_i$ to remove the $msr^{k'}$ from $c_i.Q_{msr}$ if $c_i$ needs to send an $msa^{k'}_{c_i}$ to $P_{c_i}$. Since $T_{c_i}^{k_1} = A$, $c_i$ will apply
4.6 Emergency mode switch

the MSR/MSQ queue updating rule after the emergency mode switch for $k$. If the $msr_k'$ becomes invalid due to $k$, it will be removed by the MSR/MSQ queue updating rule.

The emergency scenario triggering rule is implemented as a function $New_{scenario}3(c_i)$ in Algorithm 26, which replaces the function $New_{scenario}2(c_i)$ in Algorithm 9. The behavior of an MSS $c_i$ while triggering an emergency scenario is highlighted in red in Algorithm 26. A boolean variable $Emergency$ is used to denote if $c_i$ triggers an emergency scenario or non-emergency scenario.

Algorithm 26 $New_{scenario}3(c_i)$

1: loop
2: if $\neg NTS \land \neg ETS \land MS_{event\_detected}$ then
3:   Derive_new_mode($c_i$);
4:   $k = Assign_{scenario\_ID}(c_i)$;
5:   if $Emergency$ then
6:     if $(c_i \neq Top) \land msrSent$ then
7:       $msr_k' = c_i.Q_{msr}[1]$;
8:       Signal($c_i$, $p^{MS}$, $msa_{c_i}'$);
9:       $msrSent := false$;
10:   end if
11:   enqueue($ems_{c_i}^k$, $c_i.Q_{ems}$);
12: else
13:   if $c_i \neq Top$ then
14:     enqueue($msr_{c_i}^k$, $c_i.Q_{msr}$);
15:   else{$c_i = Top$}
16:     enqueue($msq_k^i$, $c_i.Q_{msq}$);
17:   end if
18: end if
19: end if
20: end loop

The emergency scenario triggering rule may give rise to an MSS sent from an MSS to its parent. The next question is: what should a component do upon receiving an MSS? Consider a component $c_i$ which receives an $msa_k^i$ from $c_j \in SC_{c_i}$. Component $c_i$ should take several aspects into account. First, if $c_i$ has propagated an $msr_k^i$ to $P_{c_i}$, it is also necessary to send an $msa_k^i$ to $P_{c_i}$. Second, if $c_i$ has propagated an $msq_k^i$ to $SC_{c_i}^A(k)$ but has not received all the replies yet, $c_i$ can propagate an $msd_k^i$ to $SC_{c_i}^A(k) \setminus \{c_j\}$ to abort the handling of $k$. Last but not least, $c_i$ should remove any possible $msq_k^i \in c_i.Q_{msq}$.
or \( msr_{c_j}^k \in c_i.Q_{msr} \). Figure 4.13 in Section 4.5.2 already shows that it is possible for \( c_i \) to have two \textsc{msr} primitives from \( c_j \in SC_{c_i} \). Figure 4.21 further demonstrates this phenomenon together with the arrival of an \textsc{msa}. At \( t_2 \), \( a \) receives an \( msi_{b}^{k_2} \) from a subcomponent \( b \) while \( a \) is reconfiguring itself based on \( k_1 \). Consequently, both \( msi_{b}^{k_1} \) and \( msi_{b}^{k_2} \) in \( a.Q_{msr} \) come from \( b \). At \( t_3 \), \( a \) receives an \( msa_{b}^{k_2} \) from \( b \), which suggests that \( a \) should remove the \( msi_{b}^{k_2} \) from \( a.Q_{msr} \). The other \( msi_{b}^{k_1} \) will be removed later at \( t_4 \) as \( a \) completes its mode switch for \( k_1 \).

Figure 4.21: A phenomenon when an MSR queue contains two \textsc{msr} primitives from the same subcomponent which subsequently sends an \textsc{msa}.

The behavior of a component upon receiving an \textsc{msa} is described by an \textit{MSA handling rule}:

**Definition 30. MSA handling rule:** Let \( c_i \) be a component which receives an \( msa_{c_j}^k \) from \( c_j \in SC_{c_i} \). Then,

- If \( c_i.Q_{msr}[1] = msi_{c_j}^k \) and \( c_i \) has sent an \( msi_{c_i}^k \) to \( P_{c_i} \), then \( c_i \) will send an \( msa_{c_i}^k \) to \( P_{c_i} \).
- If \( c_i.Q_{msq}[1] = msq^k \), then \( c_i \) will remove the \( msq^k \) from \( c_i.Q_{msq} \).
- If there is only one \( msi_{c_j}^k \) in \( c_i.Q_{msr} \), then \( c_i \) will remove the \( msi_{c_j}^k \) from \( c_i.Q_{msr} \). Otherwise, if \( c_i.Q_{msr} \) consists of an \( msi_{c_j}^{k_1} \) and an \( msi_{c_j}^{k_2} \),...
with \( m_{sr}^{k_1} \) closer to the head of \( c_i.Q_{msr} \), then \( k = k_2 \) and \( c_i \) will remove the \( m_{sr}^{k_2} \) from \( c_i.Q_{msr} \).

- If \( c_i \) has propagated an \( msq^k \) to \( SC_{c_i}^A(k) \) but has not received all the replies yet, then \( c_i \) will propagate an \( msd^k \) to \( SC_{c_i}^A(k) \) \( \backslash \{c_j\} \).

The MSA handling rule is demonstrated by Figure 4.22. A composite component \( b \), with \( a \) as its parent and \( c \) and \( d \) as its subcomponents, first forwards an \( m_{sr}^k \) from \( c \) to \( a \), and then receives an \( m_{sa}^k \) from \( c \) at \( t_0 \). In Figure 4.22(a), \( b \) has not received an \( msq^k \) from \( a \) at \( t_0 \). Hence \( b \) only needs to send an \( m_{sa}^k \) to \( a \) and remove the \( m_{sr}^k \) from \( b.Q_{msr} \). At a later time \( t_1 \) when \( b \) receives an \( msq^k \) from \( a \), the \( msq^k \) will be ignored by \( b \). Figure 4.22(b) differs from Figure 4.22(a) in the sense that \( b \) has propagated an \( msq^k \) to \( c \) and \( d \) at \( t_0 \) as it receives an \( m_{sa}^k \) from \( c \). By the MSA handling rule, \( b \) needs to send an \( m_{sa}^k \) to \( a \) and an \( msd^k \) to \( d \). Furthermore, the \( msq^k \) in \( b.Q_{msq} \) and the \( m_{sr}^k \) in \( b.Q_{msr} \) will be both removed.

![Figure 4.22: Demonstration of the MSA handling rule](Image)

The MSA handling rule is implemented as a function \( Handle_{MSA}(c_i, m_{sa}^k) \) in Algorithm 27, where \( c_j \in SC_{c_i} \). Algorithm 27 contains a new boolean variable \( wait4msok \) that is set to true when \( c_i \) has propagated an \( MSQ \) to \( SC_{c_i}^A(k) \) but has not received all the replies yet. This corresponds to Line 3 and Line 25 of the function \( Collect_response2(c_i, k) \) in Algorithm 28 which updates the function \( Collect_response(c_i, k) \) in...
Algorithm 2. Another shared boolean variable *abort* is introduced for mutual exclusion purpose. Once lines 16-19 of Algorithm 27 are executed, it is necessary to terminate the parallel execution of the function *Collect_response2*(c_i, k) called by *Handle_MSQ* and *Handle_MSR*. When *abort* is set to true (Line 16 of Algorithm 27), lines 7-8 of Algorithm 28 will be executed. As an additional requirement, whenever the function *Collect_response2*(c_i, k) is called by either *Handle_MSQ* or *Handle_MSR*, it should always be followed by the piece of code below:

1: *if* *abort* *then*
2: *abort* := *false*;
3: *return*;
4: *end if*

Line 3 of the piece of code above terminates the execution of *Handle_MSQ* or *Handle_MSR*.

**Algorithm 27** *Handle_MSA*(c_i ∈ CC, msa^k_j)

1: *if* msrSent ∧ c_i.Q_{msr}[1] = msr^k_j *then*
2:  Signal(c_i, p^{MS}, msa^k_j);
3:  msrSent := *false*;
4: *end if*
5: *if* c_i.Q_{msq}[1] = msq^k *then*
6:  dequeue(msq^k, c_i.Q_{msq});
7: *end if*
8: *if* ∃msr^k_j ∈ c_i.Q_{msr} *then*
9:  *if* ∃p < q s.t. ((c_i.Q_{msr}[p] = msr^{k_1}_j) ∧ (c_i.Q_{msr}[q] = msr^{k_2}_j)) *then*
10:  dequeue(msr^{k_2}_j, c_i.Q_{msr});
11:  *else*
12:  dequeue(msr^{k_2}_j, c_i.Q_{msr});
13:  *end if*
14: *end if*
15: *if* (TS_{c_i} = k) ∧ wait4msok *then*
16:  abort := true;
17:  ∀c_o ∈ SC^A_{c_i}(k) \ {c_j} : Signal(c_i, p^{MS}_{c_o}, msa^{k}_c_o);
18:  NTS := *false*;
19:  Resume(c_i, m_{c_i});
20: *end if*

Issue 2 reveals different challenges from downstream and upstream *EMS* primitives. A downstream *EMS* is relatively easier to deal with. When a com-
Algorithm 28 Collect_response2($c_i \in \mathcal{CC}, k$)

1: $n := 0$
2: $All\_ready := true$
3: $wait4msok := true$
4: if $SC^A_{c_i}(k) \neq \emptyset$ then
5: \hspace{1em} while $n < |SC^A_{c_i}(k)|$ do
6: \hspace{2em} if abort then
7: \hspace{3em} $wait4msok := false$
8: \hspace{3em} return
9: \hspace{1em} end if
10: if abort2 then
11: \hspace{1em} $All\_ready := false$
12: \hspace{1em} abort2 := false
13: \hspace{1em} $wait4msok := false$
14: \hspace{1em} return
15: end if
16: $Wait(c_i, p^MS_{in}, primitive)$
17: if $(primitive = msok^k_{c_j}) \land (c_j \in SC^A_{c_i}(k))$ then
18: \hspace{1em} $n := n + 1$
19: else $(primitive = msnok^k_{c_j}) \land (c_j \in SC^A_{c_i}(k))$
20: \hspace{1em} $n := n + 1$
21: \hspace{1em} $All\_ready := false$
22: end if
23: end while
24: end if
25: $wait4msok := false$
ponent \(c_i\) receives an \(\text{ems}^k\) from \(P_{c_i}\), there is no need for \(c_i\) to consider any \(\text{msr}^{k'}\) that has been sent to \(P_{c_i}\) and becomes invalid due to \(k\). If there exists such an \(\text{msr}^{k'}\) in the MSR queues of both \(c_i\) and \(P_{c_i}\), it will be removed as \(c_i\) and \(P_{c_i}\) apply the MSR/MSQ queue updating rule after the emergency mode switch for \(k\). What deserves extra care is that when \(c_i\) receives a downstream \(\text{ems}^k\), \(c_i\) may have propagated an \(\text{msq}^{k'}\) to \(\text{SC}_{c_i}(k')\) yet without receiving all the replies yet, as shown in Figure 4.19(a). Then \(c_i\) must be the MSDM of \(k'\). Otherwise, if \(c_i\) is not the MSDM of \(k'\), then \(P_{c_i}\) must be in the NTS for \(k'\) and will not be able to send the \(\text{ems}^k\) to \(c_i\). To facilitate the handling of the \(\text{ems}^k\), \(c_i\) can abort the handling of \(k'\) outright by propagating an \(\text{msd}^{k'}\) to \(\text{SC}_{c_i}(k')\) without waiting for all the replies.

When \(c_i\) receives an upstream \(\text{ems}^k\) from \(c_j \in \text{SC}_{c_i}\) after \(c_i\) propagates an \(\text{msq}^{k'}\) to \(\text{SC}_{c_i}(k')\) and before \(c_i\) receives all the replies, the handling of \(k'\) should also be aborted for the sake of the \(\text{ems}^k\). If \(c_i\) is the MSDM of \(k'\), \(c_i\) can immediately propagate an \(\text{msd}^{k'}\) to \(\text{SC}_{c_i}(k')\). If \(c_i\) is not the MSDM of \(k'\), \(c_i\) can immediately send an \(\text{msnok}^{k'}\) to \(P_{c_i}\). According to the MSP protocol (Definition 14), a single \(\text{msnok}^{k'}\) is sufficient to abort the handling of \(k'\). Moreover, \(c_i\) may have sent an \(\text{msr}^{k'}\) or \(\text{msr}^{k''}\) to \(P_{c_i}\). Taking both downstream and upstream \(\text{EMS}\) primitives into account, we propose an \(\text{EMS}\) receiving rule as the solution to Issue 2:

**Definition 31. EMS receiving rule:** Let \(c_i \in \text{CC}\) be a component which receives an \(\text{ems}^k\) from either \(P_{c_i}\) or \(c_j \in \text{SC}_{c_i}\). The \(\text{ems}^k\) will be put in \(c_i.Q_{\text{ems}}\). In addition,

- If the \(\text{ems}^k\) comes from \(P_{c_i}\) and \(c_i\) has propagated an \(\text{msr}^{k'}\) to \(\text{SC}_{c_i}(k')\), without receiving all the replies, then as the MSDM of \(k'\), \(c_i\) will propagate an \(\text{msq}^{k'}\) to \(\text{SC}_{c_i}(k')\), remove the \(\text{msr}^{k'}\) in \(c_i.Q_{\text{msr}}\), and leaves the NTS for \(k'\).

- If the \(\text{ems}^k\) comes from \(c_j \in \text{SC}_{c_i}\), \(c_i\) will act as follows: If \(c_i\) has sent an \(\text{msr}^{k'}\) to \(P_{c_i}\), then \(c_i\) will send an \(\text{msad}^{k'}\) to \(P_{c_i}\) when (1) \(c_i.Q_{\text{msq}}[1] \neq \text{msq}^{k'}\); or (2) \(c_i.Q_{\text{msq}}[1] = \text{msq}^{k'}\) and \(c_i\) is not in the NTS for \(k'\). Under Condition (2), \(c_i\) removes the \(\text{msq}^{k'}\) from \(c_i.Q_{\text{msq}}\). Moreover, if \(c_i\) has propagated an \(\text{msq}^{k'}\) to \(\text{SC}_{c_i}(k')\), without receiving all the replies, \(c_i\) will abort the handling of \(k'\) by either propagating an \(\text{msd}^{k'}\) to \(\text{SC}_{c_i}(k')\) (when \(c_i\) is the MSDM of \(k'\)) or sending an \(\text{msnok}^{k'}\) to \(P_{c_i}\) (when \(c_i\) is not the MSDM of \(k'\)).
The EMS receiving rule is demonstrated in Figure 4.23 where \( b \) is a composite component with \( a \) as the parent and \( c \) and \( d \) as the subcomponents. In Figure 4.23(a), \( b \) receives a downstream \( ems^{k_2} \) from \( a \) at time \( t_0 \), when \( b \) has propagated an \( msq^{k_1} \) to \( c \) and \( d \) but has not received the replies yet. Applying the EMS receiving rule, \( b \) immediately propagates an \( msd^{k_1} \) to \( c \) and \( d \). Then \( b \) will be able to handle the \( ems^{k_2} \). In contrast to Figure 4.23(a), \( b \) in Figure 4.23(b) receives an upstream \( ems^{k_2} \) from \( c \) at \( t_0 \). Since \( b \) is the MSDM of \( k_1 \), by Definition 31, \( b \) will propagate an \( msd^{k_1} \) to \( c \) and \( d \). In Figure 4.23(c), \( b \) receives an \( msq^{k_1} \) and an upstream \( ems^{k_2} \) at the same time. Since there is an \( msr^{k_1} \) in \( b.Q_{msr} \) and \( b \) has not entered the NTS for \( k_1 \), \( b \) aborts the handling of \( k_1 \) by sending an \( msq^{k_1} \) to \( a \) and removes the \( msq^{k_1} \) from \( b.Q_{msq} \).

Figure 4.23(d) is similar to Figure 4.23(b). However, since \( b \) is not the MSDM of \( k_1 \), \( b \) aborts the handling of \( k_1 \) by sending an \( msq^{k_1} \) to \( a \). At a later time \( t_1 \), when \( b \) receives an \( msd^{k_1} \) from \( a \), \( b \) will propagate the \( msd^{k_1} \) further to \( c \) and \( d \). After that \( b \) will handle the \( ems^{k_2} \) right away. Figure 4.23(e) slightly changes Figure 4.23(d) as \( b.Q_{msq}[1] \) in Figure 4.23(e) is \( msq^{k_0} \) rather than \( msq^{k_1} \). Hence, before \( b \) sends an \( msq^{k_0} \) to \( a \), an \( msq^{k_1} \) is first sent from \( b \) to \( a \) which will remove the \( msr^{k_1} \) from \( a.Q_{msr} \).

The EMS receiving rule is implemented as a function \( New_{EMS}(c_i,k) \) in Algorithm 29. Line 18 of this algorithm sets a boolean variable \( abort2 \) to true for synchronization purpose with functions \( Handle_MSQ(c_i) \) and \( Handle_{MSR}(c_i) \). When \( abort2 \) is true, lines 11-14 of Algorithm 28 will be executed, setting the boolean variable \( All_{ready} \) to false. Referring to algorithms 15-18, when \( All_{ready} \) is false and \( c_i \) is in an NTS for \( k \), \( c_i \) will propagate an \( msd^k \) to \( SC^A_{c_i}(k) \) (if \( c_i \) is the MSDM of \( k \)) or send an \( msq^{k_1} \) to \( P_{c_i} \) (if \( c_i \) is not the MSDM of \( k \)).

Integrating the MSA handling rule and EMS receiving rule into our MSRM, we replace the function \( Scenario_{reception}(c_i) \) in Algorithm 10 with another function \( Scenario_{reception2}(c_i) \) in Algorithm 30. The function \( Scenario_{reception2}(c_i) \) retains all the responsibilities of \( Scenario_{reception}(c_i) \) such as enqueueing an incoming \( MSR \) or \( MSQ \) or handling an \( MSD \) while \( c_i \) is not in an NTS or ETS. In addition, the handling of an \( EMS \) and an \( MSA \) is marked in red (lines 3-6 of Algorithm 30).

The EMS receiving rule alleviates the blocking of an \( EMS \) due to any ongoing non-emergency scenario, yet leaving Issue 3 unresolved. For instance, in Figure 4.20, when \( b \) applies the EMS receiving rule upon receiving the \( ems^{k_2} \) from \( c \), the \( msq^{k_1} \) from \( a \) will remain in \( b.Q_{msq} \). If \( T_{b}^{k_2} = B \), \( b \) can handle the \( ems^{k_2} \) first and then handle the \( msq^{k_1} \) after leaving the ETS for \( k_2 \). However, if \( T_{b}^{k_2} = A \), \( k_2 \) will invalidate \( k_1 \), since \( a \) sends the \( msq^{k_1} \) to \( b \) assuming that \( b \)
Algorithm 29 \( \text{New}_E\text{MS}(c_i \in CC, k) \)

1: \( \text{enqueue}(ems^k, c_i.Q_{ems}) \);
2: if \( ems^k \leftarrow c_j \in SC_{c_i} \) then
3: \( msr^{k'} := c_i.Q_{msr}[1] \);
4: if \( msrSent \) then
5: if \( c_i.Q_{msq}[1] \neq msq^{k'} \) then
6: \( \text{Signal}(c_i, p_{MS}^{'}, msd_{c_i}^{k'}) \);
7: \( msrSent := false \);
8: else
9: if \( (c_i.Q_{msq}[1] = msq^{k'}) \land (\neg NTS) \) then
10: \( \text{Signal}(c_i, p_{MS}^{'}, msd_{c_i}^{k'}) \);
11: \( msrSent := false \);
12: \( \text{dequeue}(msq^{k'}, c_i.Q_{msq}) \);
13: end if
14: end if
15: end if
16: if \( (TS_{c_i} = k') \land \text{wait4msok} \) then
17: \( Sc_{c_i} := false \);
18: \( \text{abort} := true \);
19: end if
20: else \( ems^k \leftarrow P_{c_i} \) \)
21: if \( (TS_{c_i} = k') \land \text{wait4msok} \) then
22: \( \text{abort} := true \);
23: \( \forall c_o \in SC_{c_i}^A(k') : \text{Signal}(c_i, p_{in}^{MS}, msd_{c_o}^{k'}) \);
24: \( \text{Clean_queues}(c_i, k') \);
25: \( NTS := false \);
26: \( \text{Resume}(c_i, m_{c_i}) \);
27: end if
28: end if
Algorithm 30 Scenario_reception2(c_i)

1:   loop
2:   Wait(c_i, p^{MS} \lor p^{MS}_in, primitive);
3:   if primitive = ems^k then
4:       New_EMS(c_i, k);
5:   else if (primitive = msa^k_{c_j}) \land (c_j \in SC_{c_i}) then
6:       Handle_MSA(c_i, msa^k_{c_j});
7:   else if (primitive = msr^k_{c_j}) \land (c_j \in SC_{c_i}) then
8:       if (c_i \in CC) \land mscSent_{c_j} then
9:          valid^k := true;
10:      end if
11:      enqueue(msr^k_{c_j}, c_i.Q_{msr});
12:   else if primitive = msq^k then
13:      enqueue(msq^k_{c_j}, c_i.Q_{msq});
14:   else (primitive = msd^k)
15:      if (\neg NTS) \land (\neg ETS) \land (primitive = msd^k) then
16:         if (c_i \in CC) \land (c_i.Q_{msr}[1] = msr^k_{c_j}) \land (c_j \in SC_{c_i}) then
17:            Signal(c_i, p^{MS}_in, msd^k_{c_j});
18:      end if
19:      Clean_queues(c_i, k);
20:   end if
21:   end if
22: end loop
is running in its old mode before the emergency mode switch for $k_2$. Therefore, it is necessary for $b$ to abort the handling of $k_1$ before it handles the $ems^{k_2}$. This is achieved by a preliminary EMS handling rule:

**Definition 32. Preliminary EMS handling rule:** Let $c_i \in \tilde{C}$ be a component which starts to handle an upstream $ems^k$. If $T_{c_i}^k = A$ and $\exists msq^{k'} \in c_i.Q_{msq}$,
c_i will send an msnok_k' to P_{c_i}. After that, when c_i receives an msd_k' from P_{c_i}, it will remove the msq_k' from c_i.Q_{msq}.

The preliminary EMS handling rule is demonstrated in Figure 4.24 where a composite component b receives an msq_k^1 from the parent a and an upstream ems_k^2 from a subcomponent c at t_0 simultaneously. In Figure 4.24(a), T_b^{k_2} = B and T_d^{k_2} = A. Then b can directly propagate an ems_k^2 to the other subcomponent d, following the EMSP protocol. By contrast, T_b^{k_2} = A in Figure 4.24(b), thus making b abort the handling of k_1 before propagating the ems_k^2.

![Figure 4.24: Demonstration of the preliminary EMS handling rule](image)

The preliminary EMS handling rule is implemented as a function Preliminary_EMS(c_i, k) in Algorithm 31. This function, together with the function EMSP(c_i, k) (Algorithm 23) that implements the EMSP protocol, is called by the function HandleEMS(c_i) in Algorithm 24. A detailed description of the function HandleEMS(c_i) is provided in Algorithm 32 which resembles the function Mode_switch2(c_i, k) in Algorithm 19, while fragments related to emergency mode switch are highlighted in red.

Next we use an example to demonstrate how our MSRM expedites an emergency mode switch concurrently triggered with non-emergency scenarios. Figure 4.25 shows the component hierarchy of a system, including a composite component b, its parent a, and three subcomponents c, d, and e. The other subcomponents of a are omitted to underline the behavior of b. This example consists of four concurrent scenarios: (1) k_0 triggered by a; (2) k_1 triggered by c; (3) k_2 triggered by d; and (4) k_3 triggered by e. The Type A components
Algorithm 31 Preliminary\_EMS($c_i \in \widehat{CC}, k$)

1: if $(T_{c_i}^k = A) \land (\exists msq^{k'} \in c_i.Q_{msq})$ then
2: \hspace{1em} abort2 := true;
3: \hspace{1em} Signal($c_i, p_{MS}^b, msnok_{c_i}^{k'}$);
4: \hspace{1em} Wait($c_i, p_{MS}^b, \text{primitive}$);
5: \hspace{1em} if primitive $= msd_{c_i}^{k'}$ then
6: \hspace{2em} dequeue(msq^{k'}, c_i, Q_{msq})
7: \hspace{1em} end if
8: \hspace{1em} end if

Algorithm 32 Handle\_EMS($c_i$)

1: \hspace{1em} ETS := true;
2: \hspace{1em} $ems^k := c_i.Q_{ems}[1]$;
3: \hspace{1em} if $ems^k \leftarrow c_j \in SC_{c_i}$ then
4: \hspace{2em} Preliminary\_EMS($c_i, k$);
5: \hspace{1em} end if
6: \hspace{1em} EMSP($c_i, k$);
7: \hspace{1em} MSC\_complete := true;
8: \hspace{1em} if ($c_i \in CC) \land (SC_{c_i}^A(k) \neq \emptyset)$ then
9: \hspace{2em} MSC\_complete := false;
10: \hspace{1em} end if
11: \hspace{1em} Activate Collect\_MSC2($c_i, k$);
12: \hspace{1em} if $T_{c_i}^k = A$ then
13: \hspace{2em} Reconfiguration($c_i, m_{c_i}, m_{c_i}^{new}$);
14: \hspace{1em} end if
15: \hspace{1em} while $\neg$MSC\_complete do
16: \hspace{2em} end while
17: \hspace{1em} if ($c_i \neq \text{Top}) \land (T_{c_i}^k = A)$ then
18: \hspace{2em} Signal($c_i, p_{MS}^b, msc_{c_i}^k$);
19: \hspace{1em} end if
20: \hspace{1em} Update\_queues2($c_i, k$);
21: \hspace{1em} dequeue($ems^k, c_i, Q_{ems}$);
22: \hspace{1em} ETS := false;
23: \hspace{1em} if $T_{c_i}^k = A$ then
24: \hspace{2em} Run($c_i, m_{c_i}^{new}$);
25: \hspace{2em} else ($T_{c_i}^k = B$)
26: \hspace{2em} Resume($c_i, m_{c_i}$);
27: \hspace{1em} end if
for each scenario are enclosed in a dotted loop in the corresponding color. Two mode-switch execution traces are compared in the figure, as \( k_3 \) in the left execution trace is a non-emergency scenario while \( k_3 \) in the right execution trace is an emergency scenario.

When \( b \) receives an \( msr^{k_3} \) or \( ems^{k_3} \) from \( e \) at time \( t_0 \), \( b \) is in the NTS for \( k_0 \), with an \( msr^{k_1} \) and an \( msr^{k_2} \) in \( b.Q_{msr} \). Due to the late arrival of the \( msr^{k_3} \), the handling of \( k_3 \) is substantially delayed by the other scenarios. After \( b \) completes the mode switch for \( k_0 \) at \( t_1 \), the \( msr^{k_1} \) in \( b.Q_{msr} \) is removed by the MSR/MSQ queue updating rule. Thence \( b \) starts to handle the \( msr^{k_2} \). The subsequent \( msnok^{k_2} \) from \( d \) makes \( b \) abort the handling of \( k_2 \) by propagating an \( msd^{k_2} \) to \( d \). At \( t_2 \), \( b \) starts the handling of the \( msr^{k_3} \). In contrast to the execution trace on the left side of Figure 4.25, the execution trace on the right side indicates that the handling of \( k_3 \) is much faster when \( k_3 \) is an emergency scenario. Upon receiving an \( ems^{k_3} \) from \( e \) at \( t_0 \), \( b \) applies the EMS receiving rule. First, \( b \) sends an \( msr^{k_1} \) to \( a \) to invalidate the \( msr^{k_1} \) previously sent to \( a \). Then \( b \) sends an \( msnok^{k_0} \) to \( a \) before receiving the \( msok^{k_0} \) from \( c \). After aborting \( k_0 \), \( b \) applies the EMSP protocol and performs the emergency mode switch for \( k_3 \). Apparently, the mode-switch completion time of the emergency scenario \( k_3 \) is accelerated using our MSRM.

### 4.6.4 The complete structure of the MSRM

Section 4.5.2 has summarized the implementation of our MSRM for concurrent non-emergency scenarios, identifying three major periodic tasks: \( New\_scenario2(c_i) \) (Algorithm 9), \( Scenario\_reception(c_i) \) (Algorithm 10), and \( Check\_queues(c_i) \) (Algorithm 12). Furthermore, Figure 4.14 depicts the algorithms directly or indirectly called by the task \( New\_scenario2(c_i) \) as well as the relation between these algorithms. Taking emergency scenarios into account, the MSRM of each component \( c_i \) is implemented by the following three tasks that replace the three aforementioned tasks:

1. \( New\_scenario3(c_i) \) (Algorithm 26): This task is used for an MSS \( c_i \) to trigger a new scenario which is either emergency or non-emergency. While triggering an emergency scenario, \( c_i \) applies the emergency scenario triggering rule (Definition 29).

2. \( Scenario\_reception2(c_i) \) (Algorithm 30): This task puts an incoming MSR, MSQ or EMS in the corresponding queues. Upon receiving an EMS, \( c_i \) applies the EMS receiving rule (Definition 31). This task is also responsible for handling an incoming MSD when \( c_i \) is not in an NTS or
ETS, and handling an **MSA** from a subcomponent by the MSA handling rule (Definition 30).

3. **Check queues2**($c_i$) (Algorithm 24): This task handles pending **EMS/MSQ/MSR** primitives by the pending scenario checking rule (Definition 28).

These tasks are periodically executed in parallel, however, a component runs **New_scenario3**($c_i$) only if it is an MSS. Figure 4.26 presents a comprehensive structure of the algorithms called by these tasks at all nested levels.

With reference to all the key issues of mode-switch handling addressed in this section, our MSRM embraces the following set of protocols and rules:

1. The MSP protocol (Definition 14)

2. The mode-switch dependency rule (Definition 15)

3. The MSQ delaying rule (Definition 17)
4.6 Emergency mode switch

Figure 4.26: Algorithm structure of the MSRM for the handling of concurrent emergency and non-emergency scenarios

4. The MSR/MSQ queue checking rule (Definition 20)
5. The MSR/MSQ queue updating rule (Definition 23)
6. The MSR enqueuing rule (Definition 24)
7. The MSQ enqueuing rule (Definition 25)
8. The EMSP protocol (Definition 26)
9. The pending scenario checking rule (Definition 28)
10. The emergency scenario triggering rule (Definition 29)

11. The MSA handling rule (Definition 30)

12. The EMS receiving rule (Definition 31)

13. The preliminary EMS handling rule (Definition 32)

Most elements in this list are included in the task Check_queues2(c_i) in Figure 4.26. The workflow of this task is depicted in Figure 4.27. When a component is not in an NTS or ETS, it schedules the handling of multiple pending scenarios as per the pending scenario checking rule, which replaces the MSR/MSQ queue checking rule. If a pending scenario exists, the scenario is propagated by the MSP protocol or the EMSP protocol, depending on whether the scenario is emergency or not. The MSP protocol guides the propagation of a non-emergency scenario, while the EMSP protocol guides the propagation of an emergency scenario. The propagation of an upstream emergency scenario is preceded the preliminary EMS handling rule. The triggering of an emergency scenario always leads to an emergency mode switch, whereas the triggering of a non-emergency scenario may trigger a mode switch or become aborted. If a mode switch (emergency or non-emergency) is triggered, the same mode-switch dependency rule which enforces bottom-up mode switch is applied so as to guarantee mode consistency. The mode-switch dependency rule is followed by the MSR/MSQ queue updating rule, which makes a component remove pending MSR and MSQ primitives invalidated by the mode switch of another scenario.

Among the remaining elements in the list, the emergency scenario triggering rule, which is implemented in the task New_scenario3(c_i) in Figure 4.26, suggests that an MSS should abort any pending non-emergency scenario before triggering an emergency scenario. The MSA handling rule and EMS receiving rule are addressed by the task Scenario_reception2(c_i) in Figure 4.26. Together with the preliminary EMS handling rule, they jointly overcome the problems stemming from the concurrent triggering of both emergency and non-emergency scenarios. Non-emergency scenarios are aborted whenever possible to advance an emergency mode switch. The MSQ delaying rule, optionally used by the MSP protocol, respects any possible ongoing atomic execution by delaying the MSQ propagation to certain components. Finally, the MSR enqueuing rule and MSQ enqueuing rule are introduced for enqueuing an MSR or an MSQ into the corresponding priority queues based on scenario priorities.

Despite the huge number of algorithms implementing our MSRM, the computation overhead of our MSRM for each component is insignificant.
Each component, its MSRM boils down to (1) sending the right primitives to the right recipients, which could be either the parent or the subcomponents, with the assistance of its local mode mapping; and (2) managing its EMS/MSQ/MSR queues. Section 4.5.3 states that for a component \( c_i \), the minimum required size of \( c_i.Q_{msr} \) is \( 2 \times |SC_{c_i}| + \sigma_{c_i} \), where \( \sigma_{c_i} \) is the maximum number of scenarios that \( c_i \) can trigger as an MSS while \( c_i \) runs in the same mode; and the minimum required size of \( c_i.Q_{msq} \) is \( \sigma_{c_i} \). According to our assumption that there is at most one emergency scenario in a system, the size of \( c_i.Q_{ems} \) is always 1. Therefore, the computation overhead of our MSRM for each component \( c_i \) scales with the number of its subcomponents and \( \sigma_{c_i} \). Since
4.6.5 Discussion

The handling of an emergency scenario in this section is rooted in an early contribution published in [53], which unfortunately contains a flaw in the EMS receiving rule. The flaw is revealed by the example in Figure 4.28 where a triggers a scenario \( k_1 \) by propagating an \( msq^{k_1} \) that is propagated downstream and stepwise to all the other components. After \( e \) propagates an \( msq^{k_1} \) to its subcomponents \( f \) and \( g \), an upstream \( ems^{k_2} \) arrives at \( e \) from \( f \). Since \( e \) has not received any \( msok^{k_1} \) or \( msnok^{k_1} \) from its subcomponents, \( e \) aborts the handling of \( k_1 \) by applying the EMS receiving rule. According to the EMS receiving rule in [53], \( e \) sends an \( msa^{k_1} \) to its parent \( b \) and an \( msd^{k_1} \) to \( f \) and \( g \). Nevertheless, it has been assumed that a component \( c_i \) may send an \( msa^{k} \) to \( P_{c_i} \) only if \( c_i \) has a pending \( msr^{k} \) in \( c_i.Q_{msr} \) that has been sent to \( P_{c_i} \). In this example, \( e \) is not supposed to send the \( msa^{k_1} \) because \( e \) has not sent any \( msr^{k_1} \) to \( b \). As a consequence, upon receiving the \( msa^{k_1} \), \( b \) applies the MSA handling rule, sending an \( msd^{k_1} \) to the other subcomponent \( d \). Since \( b \) has not sent any \( msr^{k_1} \) to \( a \), \( b \) will not send an \( msa^{k_1} \) to \( a \), which is still waiting for an \( msok^{k_1} \) or \( msnok^{k_1} \) from \( b \) and \( c \). If \( b \) or \( e \) is the MSDM of \( k_2 \), the \( ems^{k_2} \) will not reach \( a \), thus incurring a deadlock, as \( a \) is waiting for an \( msok^{k_1} \) or \( msnok^{k_1} \) from \( b \) that will never come.

This problem is eliminated in our MSRM revised in this section in the sense that \( e \) aborts the handling of \( k_1 \) by sending an \( msnok^{k_1} \) to \( b \). This \( msnok^{k_1} \) ensures that \( k_1 \) is aborted by all the involved components. Shown in Figure 4.28, an even better solution is to let \( b \) send an \( msa^{k_1} \) to \( a \), since sending an MSA can abort a scenario faster than sending an MSNOK. This only requires a slight revision of the MSA handling rule in [53]. However, verification becomes an intractable problem. The next chapter will present the verification of our MSRM, which is based on model checking and manual theorem proving. This new solution complicates our models and ends up with unacceptably long verification time. Hence, we in this thesis prohibit a component from sending an MSA if it has not sent an MSR to its parent. Our MSRM presented in this section is an efficient and verifiable solution, albeit not the optimal solution.
4.7 Summary

The mode-switch handling of multi-mode systems composed by multi-mode components is a challenging task. Since the mode switch of a component may imply the mode switches of some other components, a suitable runtime mechanism is demanded to coordinate the mode switches of different components at runtime. A centralized algorithm is not always feasible for component-based systems because component reuse often restricts the availability of global mode information. This chapter expounds an MSRM which is a distributed algorithm applied to all components. The MSRM of each component manipulates its mode-switch behavior and communication with its parent and subcomponents during a mode switch. A mode switch is initiated by the triggering of a (mode-switch) scenario from a component. With the aid of mode mapping, the MSRM is able to propagate a scenario to the affected components and guarantee the eventual mode consistency between these components after a mode switch. The MSRM respects atomic execution by deferring the propagation of a scenario to the set of components with ongoing atomic execution. Furthermore, the MSRM is capable of dealing with multiple scenarios concurrently triggered by different components. In particular, emergency scenarios, which are associated with emergency events that must be handled as soon as possible, are distinguished from non-emergency scenarios. When an emergency scenario is triggered together with other concurrent non-emergency scenarios, the MSRM endeavors to prioritize the execution of an emergency mode switch even at the sacrifice of aborting ongoing non-emergency scenarios.
Chapter 5

Verification of the mode-switch runtime mechanism

The MSRM presented in Chapter 4 plays a crucial role in the mode-switch handling of multi-mode systems composed by multi-mode components. The MSRM includes numerous protocols and rules to cope with diverse circumstances such as the propagation of emergency and non-emergency scenarios, the guarantee of mode consistency between components, atomic execution during a mode switch, and the concurrent triggering of emergency and non-emergency scenarios. In this chapter, we verify the correctness of the MSRM by a combination of model checking [26] and manual theorem proving. The initial verification targets the triggering of a single emergency or non-emergency scenario. Thereafter concurrently triggered non-emergency scenarios and concurrently triggered emergency and non-emergency scenarios are verified.

5.1 Overview of the verification

The verification of the MSRM is an arduous task due to the complex inter-component communication during a mode switch at runtime. We have already provided some preliminary discussions [52] by mathematical reasoning on the...
Chapter 5

Verification of the mode-switch runtime mechanism

The MSRM presented in Chapter 4 plays a crucial role in the mode-switch handling of multi-mode systems composed by multi-mode components. The MSRM includes numerous protocols and rules to cope with diverse circumstances such as the propagation of emergency and non-emergency scenarios, the guarantee of mode consistency between components, atomic execution during a mode switch, and the concurrent triggering of emergency and non-emergency scenarios. In this chapter, we verify the correctness of the MSRM by a combination of model checking [26] and manual theorem proving. The initial verification targets the triggering of a single emergency or non-emergency scenario. Thereafter concurrently triggered non-emergency scenarios and concurrently triggered emergency and non-emergency scenarios are verified.

5.1 Overview of the verification

The verification of the MSRM is an arduous task due to the complex inter-component communication during a mode switch at runtime. We have already provided some preliminary discussions [52] by mathematical reasoning on the
correctness of the MSP protocol and the mode-switch dependency rule. However, when the concurrent triggering of multiple scenarios is considered, the interplay between different components becomes much more complex, making verification an intractable problem for manual proofs such as mathematical reasoning. This motivates us to resort to model checking [26]. As a prevalent formal verification technique, model checking consists in the formal modeling of a system, the formulation of correctness properties, and the automatic verification of the formulated properties. The formal model is manually built via a specific model checking tool with certain abstraction. A property is formulated in some precise mathematical language, reflecting a particular aspect of the system specification, e.g., that an undesired situation never occurs.

The challenge of verifying the MSRM by model checking is how to build a general model with reasonable complexity that faithfully captures the aspects of interest in the modeled system. Model checking requires the modeling of a specific system, which is a concrete system with predefined parameters such as the number of components, the component hierarchy, mode mappings, and the specification of MSSs and scenarios. However, a successful verification of such a system may not imply that the MSRM works for all systems composed by multi-mode components. Furthermore, the notorious state explosion problem [25] of model checking restricts the complexity of a model. A complex model may result in unacceptably long verification time or may not be verifiable due to memory exhaustion.

Inspired by the verification technique adopted in [59], we model the MSRM of a single target component which exchanges primitives with parent and child stubs. The parent stub includes the set of components above the target component, simulating the interaction between the target component and its parent during a mode switch. Each child stub includes the set of components stemming from each subcomponent of the target component, simulating the interaction between the target component and the corresponding subcomponent.

Figure 5.1 illustrates the concept of parent and child stubs in an example. Suppose that $h$ in the component hierarchy is the target component. Then components $a$-$g$ will be considered as the parent stub of $h$. Since $j$ has three subcomponents $i$, $j$, and $k$, there are also three child stubs below $h$. Primitive components $i$ and $k$ are two separate child stubs, while another child stub includes $j$-$m$.

Depicted in Figure 5.2, the overall model structure comprises four elements: a target component $b$ running the MSRM, a parent stub $a$, and two child stubs $c$ and $d$. Since any component can be an MSS, $b$ may receive a scenario from the parent stub or from any child stub, or trigger a scenario by itself. For
Chapter 5. Verification of the mode-switch runtime mechanism

correctness of the MSP protocol and the mode-switch dependency rule. However, when the concurrent triggering of multiple scenarios is considered, the interplay between different components becomes much more complex, making verification an intractable problem for manual proofs such as mathematical reasoning. This motivates us to resort to model checking [26]. As a prevalent formal verification technique, model checking consists in the formal modeling of a system, the formulation of correctness properties, and the automatic verification of the formulated properties. The formal model is manually built via a specific model checking tool with certain abstraction. A property is formulated in some precise mathematical language, reflecting a particular aspect of the system specification, e.g., that an undesired situation never occurs.

The challenge of verifying the MSRM by model checking is how to build a general model with reasonable complexity that faithfully captures the aspects of interest in the modeled system. Model checking requires the modeling of a specific system, which is a concrete system with predefined parameters such as the number of components, the component hierarchy, mode mappings, and the specification of MSSs and scenarios. However, a successful verification of such a system may not imply that the MSRM works for all systems composed by multi-mode components. Furthermore, the notorious state explosion problem [25] of model checking restricts the complexity of a model. A complex model may result in unacceptably long verification time or may not be verifiable due to memory exhaustion.

Inspired by the verification technique adopted in [59], we model the MSRM of a single target component which exchanges primitives with parent and child stubs. The parent stub includes the set of components above the target component, simulating the interaction between the target component and its parent during a mode switch. Each child stub includes the set of components stemming from each subcomponent of the target component, simulating the interaction between the target component and the corresponding subcomponent. Figure 5.1 illustrates the concept of parent and child stubs in an example. Suppose that h in the component hierarchy is the target component. Then components a–g will be considered as the parent stub of h. Since j has three subcomponents i, j, and k, there are also three child stubs below h. Primitive components i and k are two separate child stubs, while another child stub includes j–m.

Depicted in Figure 5.2, the overall model structure comprises four elements: a target component b running the MSRM, a parent stub a, and two child stubs c and d. Since any component can be an MSS, b may receive a scenario from the parent stub or from any child stub, or trigger a scenario by itself. For instance, Figure 5.2 indicates that b can receive a scenario $k_0$ from a, trigger a $k_1$, or receive a $k_2$ and $k_3$ from c and d respectively. Each scenario can be either emergency or non-emergency, whereas the system should contain at most one emergency scenario according to the assumption of the MSRM. We allow the recurrent triggering of a scenario. Although in principle, each triggering of a scenario should have a unique scenario ID, the scenario ID in the model only indicates where the scenario comes from. For instance, $k_0$ always comes from the parent stub. However, the mode mapping of b is randomly generated every time it starts to handle a scenario. Thus any two recurrently triggered scenario with the same scenario ID in the model are treated as different scenarios.

The model structure in Figure 5.2 focuses on the MSRM of a composite
component which is not on top of the component hierarchy. In addition to this, there are two special cases that must not be overlooked: (1) the target component is \textit{Top}, on top of the component hierarchy; (2) the target component is a primitive component. For Case (1), the target component has no parent stub; for Case (2), the target component has no child stubs. Irrespective of the position of the target component in the component hierarchy, we formulate two key properties essential to the correctness of the MSRM:

1. Deadlock freeness: The MSRM is deadlock-free.
2. Eventual mode-switch completion: After triggering or receiving a scenario \( k \), a component eventually completes the handling of \( k \).

Aside from model checking, it is also necessary to generalize the verification results to an arbitrary system. Hence the verification of the MSRM is performed in two steps:

1. Building an abstract model with the structure in Figure 5.2 and verifying that both Property 1 (deadlock freeness) and Property 2 (eventual mode-switch completion) hold for the model.
2. Proving that the model faithfully captures the relevant behavior of an arbitrary complex finite system of components.

The second step aims to explain why a parent stub and two child stubs are general enough to represent an arbitrary system. This boils down to proving the following three assertions:

A1. The parent stub faithfully represents an arbitrary finite structure of components above the target component.
A2. A child stub faithfully represents a subcomponent with an arbitrary finite structure of enclosed components.
A3. Two child stubs faithfully represent an arbitrary number of child stubs.

### 5.2 The model checker UPPAAL

Our models are built using the model checker UPPAAL \cite{9, 78} that has been widely used to model, simulate, and verify real-time systems. Even though timing is not a major concern in the verification of the MSRM, UPPAAL is
Chapter 5. Verification of the mode-switch runtime mechanism

In addition to this, there are two special cases that must not be overlooked: (1) the target component is \textit{Top}, on top of the component hierarchy; (2) the target component is a primitive component. For Case (1), the target component has no parent stub; for Case (2), the target component has no child stubs. Irrespective of the position of the target component in the component hierarchy, we formulate two key properties essential to the correctness of the MSRM:

1. Deadlock freeness: The MSRM is deadlock-free.
2. Eventual mode-switch completion: After triggering or receiving a scenario \(k\), a component eventually completes the handling of \(k\).

Aside from model checking, it is also necessary to generalize the verification results to an arbitrary system. Hence the verification of the MSRM is performed in two steps:

1. Building an abstract model with the structure in Figure 5.2 and verifying that both Property 1 (deadlock freeness) and Property 2 (eventual mode-switch completion) hold for the model.
2. Proving that the model faithfully captures the relevant behavior of an arbitrary complex finite system of components.

The second step aims to explain why a parent stub and two child stubs are general enough to represent an arbitrary system. This boils down to proving the following three assertions:

A1. The parent stub faithfully represents an arbitrary finite structure of components above the target component.
A2. A child stub faithfully represents a subcomponent with an arbitrary finite structure of enclosed components.
A3. Two child stubs faithfully represent an arbitrary number of child stubs.

5.2 The model checker UPPAAL

Our models are built using the model checker UPPAAL \cite{9, 78} that has been widely used to model, simulate, and verify real-time systems. Even though timing is not a major concern in the verification of the MSRM, UPPAAL is still a convenient choice for us compared with other model checkers. In UPPAAL, a system is modeled as a network of timed automata synchronized with each other, each automaton graphically described by a number of states and transitions. A state is represented by a circle while an initial state is represented by double circles. A transition is represented by an arrow from one state to the same state or another state. A transition can be associated with a guard, a synchronization channel, and an action. A guard specifies a condition which must be satisfied to enable a transition. A synchronization channel is used to synchronize different UPPAAL automata. A transition of one automaton is synchronized with a transition of another automaton via a channel \(x\). The transition initiating the synchronization is marked with \(x!\) while the transition being synchronized is marked with \(x?\). If \(x\) is a broadcast channel, one transition can be synchronized with multiple transitions from different automata. Synchronized transitions are fired simultaneously. An action executes a user-specified behavior along with the corresponding transition.

Figure 5.3 shows the UPPAAL model of a lamp controlled by a user. The behavior of the lamp is modeled as a timed automaton (Figure 5.3(a)) with three states: the initial state \textit{Off} and two other states \textit{Low} with dim light and \textit{Bright} with bright light. The behavior of the user is modeled as an automaton (Figure 5.3(b)) with one state \textit{Idle}. The user switches on the lamp by pressing a button simulated as a channel \textit{press}, making the lamp switches from State \textit{Off} to State \textit{Low}. This transition contains an action \(y = 0\) which resets a clock value \(y\) to 0. The clock \(y\), which elapses at a certain rate, is used to specify timing constraints. When the lamp is in State \textit{Low}, if the user presses the button, two possible scenarios may occur depending on when the button is pressed. If the user presses the button twice quickly when the lamp is off so that the guard \(y < 5\) of an outgoing transition of State \textit{Low} is satisfied, the lamp will switch to \textit{Bright}. Otherwise, if \(y \geq 5\), the lamp will be switched off.

In UPPAAL, a property is formulated in the UPPAAL query language which is a subset of Timed Computation Tree Logic (TCTL) \cite{4}. The following lists three properties P1-P3 based on the lamp example in Figure 5.3:

**P1.** \(A[]\ not \ deadlock\): The model is deadlock-free.

**P2.** \(E<> \ Lamp.Bright\): It is possible that the lamp runs in State \textit{Bright}.

**P3.** \(Lamp.Low-->Lamp.Bright\): When the lamp is in State \textit{Low}, it will eventually reach State \textit{Bright}.

The notation \(A[]\) in P1 means that the expression after it must hold invariably. Otherwise, UPPAAL will automatically find an execution trace where the
expression fails to hold. In P2, $E<>$ means that it is possible for the model
to reach a state in which the expression after it holds. UPPAAL automatically
returns an execution trace satisfying this property. For two expressions $A$ and
$B$, $A\rightarrow B$ means that $B$ is eventually true if $A$ is true. If such a property is not
satisfied, UPPAAL will return an execution trace as a counter example. P1 and
P2 are both satisfied in the lamp example, whereas P3 is not. When the lamp is
in State *Low*, if the user never presses the button or presses the button slowly,
the lamp will never go to State *Bright*. This explains why P3 is not satisfied.

### 5.3 Verification of a non-emergency scenario

A component handles a single non-emergency scenario by the MSP protocol
and the mode-switch dependency rule of its MSRM. The MSQ delaying rule
may also be applied by the MSP protocol in the presence of atomic execution.
Nevertheless, our verification excludes the MSQ delaying rule which only de-
lays the propagation of an MSQ to certain components, without affecting the
verification result.

Since a scenario may be triggered by any component, the target component $b$
in Figure 5.2 should consider scenarios from the parent stub, any child stub,
or triggered by itself. Figure 5.4(a) shows three different cases in which a non-
emergency scenario arrives at $b$: (1) $b$ receives an $msq^{k_0}$ from the parent stub;
(2) $b$ issues an $msr^{k_1}$ itself; (3) $b$ receives an $msr^{k_2}$ from Child stub 1. Of
course, $b$ may also receive an $msr^{k_3}$ from Child stub 2. However, on account
of the symmetrical structure of two child stubs, it is sufficient to only consider
the scenario from one child stub. If the target component is *Top*, shown in
Figure 5.4(b), it only needs to consider a self-triggered $msr^{k_0}$ and an $msr^{k_1}$
from Child stub 1. If the target component is a primitive component, shown in Figure 5.4(c), it only needs to consider an $msq^k_0$ from the parent stub and a self-triggered $msr^{k_1}$.

Figure 5.4: The UPPAAL model structure assuming the triggering of a single non-emergency scenario

### 5.3.1 Modeling and verification in UPPAAL

We take the general case in Figure 5.4(a) as an example to model how the MSRM handles a non-emergency scenario. The UPPAAL model consists of four automata: *ParentStub*, *TargetComp*, *ChildStub1*, and *ChildStub2*, which correspond to the four entities, i.e. $a$, $b$, $c$, $d$ in Figure 5.4(a). Here *ChildStub1* and *ChildStub2* are instantiated from a common template *ChildStub*.

Figure 5.5 depicts *TargetComp* which implements the MSP protocol and the mode-switch dependency rule, marking its 40 transitions in red. The mode-switch dependency rule is implemented by transitions 38–40. The label $n:int[2,3]$ of Transition 38 is called a *Selection* in UPPAAL, implying that $n$ is an integer with a value of either 2 or 3. Combining $n:int[2,3]$ with $MSC[n]?$, it can be inferred that Transition 38 can be synchronized with two possible channels, either $MSC[2]$ or $MSC[3]$. In the model, we assign an ID 0-3 to $a$, $b$, $c$, $d$, respectively. Thus $MSC[2]$ equals $MSC[c]$, denoting an $MSC$ from $c$, Child stub 1. Similarly, $MSC[3]$ denotes an $MSC$ from the other child stub $d$. The number of received $MSC$ primitives is updated by a counter. When the counter is equal to the number of Type A components of the current scenario (denoted
as $A$\textit{childrenN}$, the target component $b$ will complete its mode switch by firing Transition 39 or 40, depending on if $b$ is the MSDM or not. If not, a boolean variable $MSDM$ is false, making $b$ send an $MSC$ to the parent stub $a$ by Transition 39. Note that before completing the mode switch, $b$ is also supposed to complete its reconfiguration if it is a Type A component. If the reconfiguration is not completed after $b$ has received all expected $MSC$ primitives, the firing of Transition 39 or 40 will be delayed until the reconfiguration is completed. Such a delay is not considered in the model because timing is not the focus of our verification.

The remaining part of the model is dedicated to the MSP protocol. The model contains committed states marked with a “C” in the circle. A committed state is instantaneous and atomic. An outgoing transition from a committed state must be fired immediately. For instance, when the target component $b$ receives an $msr^{k_2}$ from $c$ by Transition 1, its mode mapping is implemented by

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure55.png}
\caption{The UPPAAL model of a target component which handles a non-emergency scenario}
\end{figure}
a two-step atomic process using committed states. First, a function ModeMapping(n) determines if \( d \) is a Type A component for \( k_2 \). If \( n = 0 \), \( T_{d}^{k_2} = A \); if \( n = 1 \), \( T_{d}^{k_2} = B \). Subsequently, another function options(n) of Transition 2 determines if \( b \) is an MSDM. If yes, it may either approve (option==0) the \( msr^{k_2} \) by Transition 3 or reject (option==1) the \( msr^{k_2} \) by Transition 5. If no (option==2), \( b \) forwards the \( msr^{k_2} \) to \( a \) by Transition 4. The target component \( b \) triggers a scenario \( k_1 \) by Transition 6. After the triggering of \( k_1 \), a boolean variable pending becomes true until \( b \) completes the handling of \( k_1 \). The guard of Transition 6 contains a boolean variable \( ok[1] \) which determines whether \( k_1 \) can be triggered or not. There is also an \( ok[0] \) and \( ok[2] \) for \( k_0 \) and \( k_2 \). Among \( ok[0]-ok[2] \), only one of them is set to true to disable the concurrent triggering of multiple scenarios. The function ModeMapping2(n) of Transition 6 randomly returns four possible mode mapping results\(^1\) depending on the value of \( n \): (1) \( T_{c}^{k_1} = A \) and \( T_{d}^{k_1} = A \); (2) \( T_{c}^{k_1} = A \) and \( T_{d}^{k_1} = B \); (3) \( T_{c}^{k_1} = B \) and \( T_{d}^{k_1} = A \); and (4) \( T_{c}^{k_1} = B \) and \( T_{d}^{k_1} = B \). Also, Transition 6 always assigns 2 to option so that \( b \) can send an \( msr^{k_1} \) to \( a \) by Transition 5.

Transition 7 is fired when \( b \) receives an \( MSQ \) from \( a \). The \( MSQ \) may be an \( msq^{k_0} \) irrelevant to any \( MSR \) sent by \( b \), or an \( msq^{k_1} \) or \( msq^{k_2} \) as a consequence of the \( msr^{k_1} \) or \( msr^{k_2} \) sent from \( b \). The scenario ID of the \( MSQ \) is saved in a local variable \( localSID \). The target component \( b \) checks its current state by the function CheckState(n) of Transition 7. If \( b \) is ready to switch mode, either Transition 8 or 9 is fired. If \( b \) receives an \( msq^{k_0} \), the mode mapping for \( k_0 \) must be derived by the function ModeMappingMSQ(n) of Transition 8. Otherwise, if \( b \) receives an \( msq^{k_1} \) or \( msq^{k_2} \), the previous mode mapping result from the \( msr^{k_1} \) or \( msr^{k_2} \) is reused by the function reuseMM() of Transition 9. If \( b \) is not ready to switch mode, \( b \) will send an \( MSNOK \) to \( a \) by Transition 10. After that, \( b \) will receive an \( MSD \) from \( a \) by Transition 11. Upon receiving an \( msd^{k_0} \) or \( msd^{k_1} \), \( b \) can directly complete the handling of \( k_0 \) or \( k_1 \) by Transition 12. Upon receiving an \( msd^{k_2} \) though, \( b \) is obliged to propagate the \( msd^{k_2} \) further back to \( c \), the origin of \( k_2 \), by Transition 13. Transitions 12 and 13 may also follow Transition 14, when \( b \) directly receives an \( MSD \) from \( a \) after propagating an \( MSR \) upwards.

When the target component \( b \) is ready to propagate an \( msq^{k} \) in State \( MSQprop \) where \( k \) is \( k_0 \), \( k_1 \) or \( k_2 \), it first checks if \( SC_{d}^{A}(k) = \emptyset \). If yes, \( b \) directly sends an \( msok^{k} \) to \( a \) by Transition 18 or an \( msok^{k} \) to \( a \) by Transition 17 without propagating the \( msq^{k} \) further. Otherwise, the \( msq^{k} \) is sequentially

---

\(^1\)The mode mapping treats child stubs \( c \) and \( d \) as the subcomponents of \( b \), though strictly speaking, \( c \) and \( d \) can be a set of components.
propagated to components in \( SC^A_b(k) \) by transitions 15 and 16. After the propagation, \( b \) goes to State \textbf{OKCollection}, expecting an \( msok^k \) (Transition 23) or \( msnok^k \) (Transition 24) from its Type A subcomponents. If all the replies are \( msok^k \), a boolean variable allMSOK is true. Then \( b \) can start propagating an \( msi^k \) by transitions 27, 35 and 36 if \( b \) is the MSDM of \( k \), or send an \( msok^k \) to \( a \) by Transition 28 if \( b \) is not the MSDM. If \( b \) receives at least one \( msnok^k \), allMSOK is false. Hence \( b \) will propagate an \( msd^k \) by transitions 25, 30, 31 if \( b \) is the MSDM of \( k \), or send an \( msnok^k \) to \( a \) by Transition 26 if \( b \) is not the MSDM.

The UPPAAL model of the parent stub \( a \) is depicted in Figure 5.6. It is important to note that \( a \) is an interface rather than the parent of \( b \). As the parent stub, \( a \) functions as a black box that produces the expected output to \( b \) in response to any upstream primitive from \( b \). When \( a \) receives an \( msr^k \) from \( b \) by Transition 1, where \( k \) is either \( k_1 \) or \( k_2 \), if there are no concurrent scenarios, \( a \) will eventually send either an \( msq^k \) to \( b \) by Transition 3 or an \( msd^k \) to \( b \) by Transition 2. If the triggering of \( k_0 \) is allowed, \( ok[0] \) will be set to true so that \( a \) can send an \( msq^{k_0} \) to \( b \) by Transition 4. After sending the \( msq^{k_0} \), a local variable pending is set to true. When \( a \) receives an \( msr^{k_0} \) from \( b \) by Transition 10 or sends an \( msd^{k_0} \) to \( b \) by Transition 7 or 9, pending is reset to false by a function reset(). An interesting phenomenon is that if the parent stub \( a \) is replaced with \textit{Top}, the UPPAAL model in Figure 5.6 will remain the same. In other words, the parent stub is actually modeled in the same way as \textit{Top}.

![Figure 5.6: The UPPAAL model of a parent stub which handles a non-emergency scenario](image)
The UPPAAL model of a child stub, c or d, is depicted in Figure 5.7. A child stub functions as a black box that produces the expected output to b in response to any downstream primitive from b. When ok[2] is true, Child stub 1 (i.e., c) can trigger k2 by sending an msr\(^k2\) to b by Transition 1. Similar to the parent stub, a local boolean variable pending becomes true after the triggering of k2 and becomes reset to false either when c sends an msc\(^k2\) to b or when c receives an msd\(^k2\) from b. An interesting phenomenon is that if a child stub is replaced with a primitive component, the UPPAAL model in Figure 5.7 will remain the same. In other words, a child stub is modeled in the same way as a primitive component.

![Figure 5.7: The UPPAAL model of a child stub which handles a non-emergency scenario](image)

We formulate the following four properties essential to the correctness of the model:

**P1.** \(A[] \text{not deadlock} \): The model is deadlock-free.

**P2.** \(\text{ParentStub.pending} \rightarrow !\text{ParentStub.pending} \): After the parent stub a sends an msq\(^k0\) to b, \(k_0\) is eventually handled by a.

**P3.** \(\text{TargetComp.pending} \rightarrow !\text{TargetComp.pending} \): After the target component b triggers \(k_1\), b will eventually complete the handling of \(k_1\).

**P4.** \(\text{ChildStub1.pending} \rightarrow !\text{ChildStub1.pending} \): After Child stub 1 c sends an msr\(^k2\) to b, \(k_2\) is eventually handled by c.
Chapter 5. Verification of the mode-switch runtime mechanism

P1 in the list corresponds to Property 1 (deadlock freeness) formulated in Section 5.1. The other properties P2-P4 are jointly mapped to Property 2 (eventual mode-switch completion) formulated in Section 5.1. All the four properties are satisfied with verification time shorter than 0.2s².

In addition to the general case in which the target component has both parent and subcomponents, we have also modeled and verified the other two special cases in Figure 5.4(b) and Figure 5.4(c), where the target component is either Top or a primitive component. To avoid verbose and bulky presentation, the modeling and verification for the two special cases is omitted in the thesis. The complete UPPAAL models and verification results for all cases can be found in [47].

5.3.2 Generalization of the verification results

Since our UPPAAL model contains a parent stub and two child stubs, the verification results in Section 5.3.1 must be generalized. The crux is to prove that our UPPAAL model faithfully represents an arbitrary complex finite system of components while running the MSRM. In doing this, we first define the external/internal mode-switch behavior of a component:

**Definition 33. External mode-switch behavior:** The external mode-switch behavior, or simply external behavior of a component \( c_i \neq \text{Top} \), is the visible behavior of \( c_i \) from the perspective of \( P_{c_i} \) during a mode switch. The external behavior of \( c_i \) is represented by (1) the response of \( c_i \) after \( P_{c_i} \) sends a downstream primitive to \( c_i \); and (2) the capability of \( c_i \) to actively send any upstream primitive to \( P_{c_i} \) and the precondition for \( P_{c_i} \) to receive the primitive.

**Definition 34. Internal mode-switch behavior:** The internal mode-switch behavior, or simply internal behavior of a composite component \( c_i \), is the visible behavior of \( c_i \) from the perspective of each \( c_j \in SC_{c_i} \) during a mode switch. The internal behavior of \( c_i \) is represented by (1) the response of \( c_i \) after \( c_j \) sends an upstream primitive to \( c_i \); and (2) the capability of \( c_i \) to actively send any downstream primitive to \( c_j \) and the precondition for \( c_j \) to receive the primitive.

Note that Top has no visible external behavior while a primitive component has no visible internal behavior. Obviously, both the external and internal behaviors of a component are dependent on its MSRM. Our model implements

---

²The verification was performed on MacBook Pro, with 2.66GHz Intel Core 2 Duo CPU and 8GB 1067 MHz DDR3 memory.
the MSP protocol and the mode-switch dependency rule to handle a single non-emergency scenario. By Definition 14 and Definition 15 in Chapter 4, a component may send and receive 7 types of primitives, including 3 types of downstream primitives \textbf{MSQ}, \textbf{MSI} and \textbf{MSD}, and 4 types of upstream primitives \textbf{MSR}, \textbf{MSOK}, \textbf{MSNOK}, and \textbf{MSC}. Furthermore, we associate each component with a 	extit{composition level} and a 	extit{depth level}:

**Definition 35. Composition level:** Each component \(c_i\) is associated with a composition level denoted as \(l_{c_i}\). If \(c_i \in \mathcal{PC}\), then \(c_i\) has a composition level 0, denoted as \(l_{c_i} = 0\). If \(c_i \in \mathcal{CC}\), \(l_{c_i} = \max\{l_{c_j}\} + 1 | c_j \in \mathcal{SC}_{c_i}\).

**Definition 36. Depth level:** Each component \(c_i\) is associated with a depth level denoted as \(d_{c_i}\). If \(c_i = \text{Top}\), then \(d_{c_i} = 0\). For each \(c_p\) and \(c_q = \mathcal{P}_{c_p}\), \(d_{c_p} = d_{c_q} + 1\).

Based on definitions 35 and 36, we interpret the three assertions (A1-A3) stated in Section 5.1 as follows:

A1b. The internal behavior of the parent stub of our UPPAAL model faithfully represents the internal behavior of any composite component with arbitrary depth level.

A2b. The external behavior of each child stub of our UPPAAL model faithfully represents the external behavior of any component (excluding \textit{Top}) with arbitrary composition level.

A3b. For each composite component, two child stubs in our UPPAAL model faithfully represent an arbitrary number of child stubs.

We have stated that the UPPAL model of the parent stub in Figure 5.6 does not change if the parent stub is replaced with \textit{Top}, while the UPPAAL model of a child stub in Figure 5.7 does not change if the child stub is replaced with a primitive component. Based on this observation, A1b and A2b can be re-interpreted as A1c and A2c respectively:

A1c. The internal behavior of \textit{Top} faithfully represents the internal behavior of any composite component with arbitrary depth level.

A2c. The external behavior of a primitive component faithfully represents the external behavior of any component (excluding \textit{Top}) with arbitrary composition level.
A1c suggests that we must compare the internal behavior of Top with that of an arbitrary composite component. We first prove the internal behavior equivalence between Top and a composite component with depth level 1. By Definition 34, the comparison is based on two criteria: (1) the response upon receiving an upstream primitive from a subcomponent; and (2) the capability of actively sending any downstream primitive to a subcomponent as well as the precondition for the subcomponent to receive the primitive. Running the MSP protocol and the mode-switch dependency rule, a composite component may receive an msr, msok, msnok, or msc from a subcomponent, and may actively send an msq to a subcomponent. The transmission of all these primitives must be analyzed to prove the following lemma:

**Lemma 2.** Let \( c_i \) be a composite component with \( c_p = P_{c_i} \) and \( c_p = Top \). Then the internal behaviors of \( c_i \) and \( c_p \) are equivalent under the assumption of the triggering of a non-emergency scenario.

**Proof.** We first enumerate all the possible upstream primitives that can be sent from \( c_i \) to \( c_p \) and from each \( c_j \in SC_{c_i} \) to \( c_i \), and then compare the responses of \( c_p \) and \( c_i \):

1. **msr**: After \( c_i \) sends an msr to \( c_p \), since \( c_p = Top \), \( c_p \) must be the MSDM of \( k \). By the MSP protocol (Definition 14 and Algorithm 4), \( c_p \) will either propagate an msq to \( SC_{c_p}^{A} (k) \) (including \( c_i \)) or send an msd back to \( c_i \). After \( c_i \) receives an msr from \( c_j \), \( c_i \) may or may not be the MSDM of \( k \). If \( c_i \) is the MSDM, its response will be either an msq or an msd, which is the same as the response of \( c_p \). If \( c_i \) is not the MSDM, \( c_i \) will forward the msr to \( c_p \). Subsequently, if \( c_p \) sends an msq or msd to \( c_i \), \( c_i \) will also send an msq or msd to \( c_j \). Hence, the eventual response of \( c_i \) is the same as the response of \( c_p \) no matter whether \( c_i \) is the MSDM of \( k \) or not.

2. **msok**: By the MSP protocol, after \( c_i \) sends an msok to \( c_p \), \( c_p \) may send an msr to \( c_i \) (if \( c_i \) has received an msok from all components in \( SC_{c_p}^{A} (k) \)) or an msd to \( c_i \) (if \( c_i \) receives at least one msnok). Likewise, after \( c_i \) receives an msok from \( c_j \), the response of \( c_i \) will be the same as the response of \( c_p, \) if \( c_i \) is the MSDM of \( k \). If \( c_i \) is not the MSDM, \( c_i \) will send an msok or msnok to \( c_p \) after receiving the reply from all components in \( SC_{c_i}^{A} (k) \). Subsequently, \( c_i \) will receive an msi or an msd from \( c_p \). In either case, \( c_i \) will send the msi or msd.
to $c_j$, implying that the the response of $c_i$ is also the same as the response of $c_p$, even when $c_i$ is not the MSDM.

3. $msnok^k$: By the MSP protocol, after $c_i$ sends an $msnok^k$ to $c_p$, $c_p$ must send an $msd^k$ to $c_i$. Likewise, after $c_i$ receives an $msnok^k$ from $c_j$, its response is the same as the response of $c_p$ if $c_i$ is the MSDM of $k$. If $c_i$ is not the MSDM, $c_i$ will send an $msnok^k$ to $c_p$ which will send an $msd^k$ to $c_i$. Upon receiving the $msd^k$, $c_i$ will also send an $msd^k$ to $c_j$. Hence, no matter whether $c_i$ is the MSDM of $k$ or not, the eventual response of $c_i$ is always an $msd^k$, which is the same as the response of $c_p$.

4. $msc^k$: The mode-switch dependency rule (Definition 15 and Algorithm 6) suggests that neither $c_p$ nor $c_i$ is expected to respond to the $msc^k$ sender.

The enumeration above covers all possible cases and hence proves the internal behavior equivalence between $c_p$ and $c_i$ in terms of the response to an upstream primitive. In addition, both $c_p$ and $c_i$ are able to actively send an $msq^k$ to a subcomponent such that $k$ is not associated with any $msr^k$ from the subcomponent. This completes the proof of Lemma 2.

Lemma 2 further implies:

**Theorem 5.** For each component $c_i \in \tilde{CC}$, the internal behavior of $c_i$ is equivalent to that of $Top$ under the assumption of the triggering of a non-emergency scenario.

**Proof.** Let $c_p = P_{c_i}$. Theorem 5 can be proven by mathematical induction based on $d_{c_p}$.

**Basis:** $d_{c_p} = 0$, i.e. $c_p = Top$. By Lemma 2, the internal behaviors of $c_p$ and $c_i$ are equivalent. Theorem 5 follows.

**Inductive step:** Suppose that $d_{c_p} = n$ (where $n \in \mathbb{N}$) and the internal behavior of $c_p$ is equivalent to that of $Top$. Then we need to prove that the internal behavior of $c_i$ with $d_{c_i} = n + 1$ is also equivalent to that of $Top$. Since the internal behaviors of $c_p$ and $Top$ are equivalent, it is OK to replace $c_p$ with $Top$ without changing the internal behavior of $c_i$. Once $c_p$ is replaced with $Top$, $c_i$ will become a component with depth level 1. By Lemma 2, the internal behaviors of $Top$ and $c_i$ will be equivalent. Since such a replacement does not change the internal behavior of $c_i$, by induction, Theorem 5 also follows when $d_{c_i} = n + 1$.
Since the internal behaviors of $\textit{Top}$ and any composite component are equivalent, Theorem 5 satisfies $A1c$. In our UPPAAL model, the parent stub is modeled as $\textit{Top}$ from the perspective of the target component. Hence the internal behavior of the parent stub faithfully represents the internal behavior of any composite component, which satisfies $A1b$ as well.

The proof structure of $A2c$ resembles the proof structure of $A1c$. The first step is to compare the external behaviors of a primitive component and its parent. By Definition 33, the comparison is drawn by two criteria: (1) the response upon receiving a downstream primitive from the parent; and (2) the capability of actively sending any upstream primitive to the parent and the precondition for the parent to receive the primitive. The following lemma proves the external behavior equivalence between a primitive component and its parent:

**Lemma 3.** For $c_j \in \mathcal{PC}$ and $c_i = P_{c_j}$, the external behaviors of $c_i$ and $c_j$ are equivalent under the assumption of the triggering of a non-emergency scenario.

**Proof.** Let $c_p = P_{c_i}$. We first enumerate all the possible downstream primitives that can be sent from $c_i$ to $c_j$ and from $c_p$ to $c_i$:

1. $\text{msq}^k$: By the MSP protocol (Definition 14 and algorithms 3 and 4), after $c_i$ sends an $\text{msq}^k$ to $c_j$, $c_j$ will reply with either an $\text{msok}^k$ or an $\text{msnok}^k$. After $c_p$ sends an $\text{msq}^k$ to $c_i$, if $\mathcal{SC}^A_{c_i}(k) = \emptyset$, $c_i$ will reply with either an $\text{msok}^k$ or an $\text{msnok}^k$ without further propagating the $\text{msq}^k$ downwards. If $\mathcal{SC}^A_{c_i}(k) \neq \emptyset$, $c_i$ will propagate the $\text{msq}^k$ to $\mathcal{SC}^A_{c_i}(k)$. After that, $c_i$ will eventually send an $\text{msok}^k$ or an $\text{msnok}^k$ to $c_p$ after it has received all the replies from $\mathcal{SC}^A_{c_i}(k)$. Hence the responses of $c_i$ and $c_j$ upon receiving an $\text{msq}^k$ are the same.

2. $\text{msd}^k$: The MSP protocol suggests that neither $c_i$ nor $c_j$ is expected to respond to the $\text{msd}^k$ sender.

3. $\text{msi}^k$: By the mode-switch dependency rule (Definition 15 and Algorithm 6), after $c_i$ sends an $\text{msi}^k$ to $c_j$, $c_j$ will send an $\text{msc}^k$ to $c_i$ after $c_j$ completes its reconfiguration. After $c_p$ sends an $\text{msi}^k$ to $c_i$, if $\mathcal{SC}^A_{c_i}(k) = \emptyset$, $c_i$ will also send an $\text{msc}^k$ to $c_p$ after $c_i$ completes its reconfiguration. If $\mathcal{SC}^A_{c_i}(k) \neq \emptyset$, $c_i$ will propagate the $\text{msi}^k$ to $\mathcal{SC}^A_{c_i}(k)$. After that, $c_i$ will eventually send an $\text{msc}^k$ to $c_p$ after $c_i$ completes its reconfiguration and has received an $\text{msc}^k$ from all the components in $\mathcal{SC}^A_{c_i}(k)$. Hence the responses of $c_i$ and $c_j$ upon receiving an $\text{msi}^k$ are the same.
The enumeration above proves the external behavior equivalence between $c_i$ and $c_j$ in terms of the response to a downstream primitive. In addition, both $c_i$ and $c_j$ are able to actively send an $msr^k$ to the parent. This completes the proof of Lemma 3.

Lemma 3 further implies:

**Theorem 6.** For each component $c_i \in \widetilde{CC}$, the external behavior of $c_i$ is equivalent to that of a primitive component under the assumption of the triggering of a non-emergency scenario.

**Proof.** Let $c_p = P_{c_i}$. Theorem 6 can be proven by mathematical induction based on $l_{c_i}$:

**Basis:** $l_{c_i} = 1$. By Definition 35, for each $c_j \in SC_{c_i}$, $c_j$ must be a primitive component with $l_{c_j} = 0$. By Lemma 3, the external behaviors of $c_i$ and $c_j$ are equivalent. Theorem 6 follows.

**Inductive step:** Suppose that for any component $c_o$ with $l_{c_o} \leq n$ ($n \in \mathbb{N}$), the external behavior of $c_o$ is equivalent to that of a primitive component. Let $l_{c_i} = n$ and $l_{c_p} = n + 1$, i.e., $c_i$ has the highest composition level among $SC_{c_p}$. Then the external behavior of $c_i$ is equivalent to that of a primitive component. We need to prove that the external behavior of $c_p$ is also equivalent to that of a primitive component. Since for all $c_o \in SC_{c_p}$, $l_{c_o} \leq n$. The external behavior equivalence between $c_o$ and a primitive component suggests that it is OK to replace all $c_o \in SC_{c_p}$ with a primitive component without changing the external behavior of $c_p$. After the replacement, $l_{c_p} = 1$. According to the base case, the external behavior of $c_p$ is also equivalent to that of a primitive component. This completes the proof.

Since the external behaviors of any $c_i \in \widetilde{CC}$ and a primitive component are equivalent, Theorem 6 satisfies A2c. In our UPPAAL model, a child stub is modeled as a primitive component from the perspective of the target component. Hence the external behavior of each child stub faithfully represents the external behavior of any component (excluding Top) with arbitrary composition level, which satisfies A2b as well.

Finally, A3b is addressed by Theorem 7:

**Theorem 7.** Each component $c_i \in \widetilde{CC}$ with arbitrary number of subcomponents can correctly execute its MSRM under the assumption of the triggering of a non-emergency scenario.
Proof. Let $N = |SC_{c_i}|$. Theorem 7 can be proven by mathematical induction based on $N$:

Basis: $N = 2$. Since the target component in our UPPAAL model has two child stubs, Theorem 7 directly follows from our verification results in Section 5.3.1.

Inductive step: Suppose that $c_i$ can correctly execute its MSRM with $N = n$, while $SC_{c_i} = \{c_j^0, c_j^1, \ldots, c_j^{n-1}\}$ ($n \in \mathbb{N}, n \geq 2$). The goal is to prove that it also works if $N = n + 1$, as another $c_j^n$ is added to $SC_{c_i}$.

Since a non-emergency scenario $k$ has already been considered before $c_j^n$ is added, $c_i$ will not receive any scenario from $c_j^n$. Component $c_j^n$ can be either a Type A component or a Type B component for $k$.

If $T_{c_j^n}^k = B$, then $c_j^n \notin SC_{c_i}^A(k)$. Component $c_j^n$ will not exchange any primitive with other components, thus not affecting the original MSRM execution of $c_i$. If $T_{c_j^n}^k = A$, $c_i$ will add $c_j^n$ to $SC_{c_i}^A(k)$ so that $c_j^n$ can receive a primitive every time $c_i$ propagates a downstream primitive to $SC_{c_i}^A(k)$. Since the external behavior of $c_j^n$ is equivalent to that of a primitive component, $c_j^n$ is able to interact with $c_i$ in the same way as the other Type A subcomponents of $c_i$, i.e., whenever any $c_o \in SC_{c_i}^A(k)$ sends an $msok^k$, $msnok^k$, or $msc^k$ to $c_i$, $c_j^n$ will also send the same upstream primitive to $c_i$ at the right time.

Combining the basis and the inductive step, the proof of Theorem 7 is completed. \qed

5.4 Verification of an emergency scenario

A component handles a single emergency scenario by the EMSP protocol and the mode-switch dependency rule of its MSRM. Since a scenario may be triggered by any component, the target component $b$ in Figure 5.2 should consider scenarios from the parent stub, any child stub, or triggered by itself. Figure 5.8 depicts the model structure which is essentially the same as the structure in Figure 5.4. The only difference is that an emergency scenario $k$ is always propagated by an $ems^k$.

5.4.1 Modeling and verification in UPPAAL

We take the general case in Figure 5.8(a) as an example to model how the MSRM handles an emergency scenario. The UPPAAL model consists of our automata: ParentStub, TargetComp, ChildStub1, and ChildStub2, which corre-
5.4 Verification of an emergency scenario

Figure 5.8: The UPPAAL model structure assuming the triggering of a single emergency scenario

Figure 5.9 depicts TargetComp which implements the EMSP protocol and the mode-switch dependency rule. The mode-switch dependency rule is implemented by transitions 12, 13, 17-19. Transitions 12 and 13 are selected when the target component has no Type A subcomponents for an emergency scenario. Transitions 17-19 are exactly the same as the transitions 38-40 in Figure 5.5.

The remaining part of the model is dedicated to the EMSP protocol. The target component \( b \) receives an \( \text{ems}^{k_2} \) from Child stub 1 \( c \) by Transition 1, an \( \text{ems}^{k_0} \) from the parent stub \( a \) by Transition 3, or triggers an \( \text{ems}^{k_1} \) by Transition 2. Upon receiving an \( \text{ems}^{k_2} \) from \( c \), \( b \) reuses the function \text{ModeMapping}(n) \) in Figure 5.5 to determine if Child stub 2 \( d \) is a Type A component for \( k_2 \). Meanwhile, a boolean variable \text{fromC} \) is set to true, denoting that \( c \) is the sender of the \( \text{ems}^{k_2} \). When the boolean variable \text{ok[1]} \) is true, \( b \) can trigger an emergency scenario \( k_1 \) itself. After triggering \( k_1 \), \( b \) reuses the function \text{ModeMapping2}(n) \) in Figure 5.5 to determine if \( c \) or \( d \) is a Type A component for \( k_1 \). Meanwhile, \( b \) sets a boolean variable \text{pending} \) to true. The variable \text{pending} \) is reset to false until \( b \) completes the handling of \( k_1 \). Transitions 1 and 2 are immediately followed by Transition 4 which determines if \( b \) is the MSDM. If yes, Transition 5 will be selected. Otherwise, \( b \) should send an \text{EMS} \) to \( a \) by Transition 6. The propagation of an \text{EMS} \) from \( b \) to its subcomponents
is achieved by transitions 8, 9, 14, 15. If the EMS is $ems^{k_0}$ or $ems^{k_1}$, $b$ will propagate the EMS to its Type A subcomponents by transitions 14 and 15. Otherwise, if the EMS is $ems^{k_2}$, the boolean variable $fromC$ is true, making $b$ go for transitions 7-9 and never send an $ems^{k_2}$ to the $ems^{k_2}$ sender $c$.

Depicted in Figure 5.10(a), the parent stub $a$ is modeled as Top from the perspective of $b$. It may receive an $ems^{k_1}$ or $ems^{k_2}$ from $b$ by Transition 1. When a boolean variable $ok[0]$ is true, $a$ can trigger $k_0$, sending an $ems^{k_0}$ to $b$ by Transition 1 and setting a local boolean variable pending to true. The variable pending is reset to false when $a$ receives an $msc^{k_0}$ by Transition 2.

Figure 5.10(b) depicts the UPPAAL model of a child stub $c$ or $d$. A child stub is modeled as a primitive component from the perspective of $b$. A child stub receives an EMS from $b$ by Transition 3. When a boolean variable $ok[2]$ is true, $c$ can trigger $k_2$, sending an $ems^{k_2}$ to $b$ by Transition 1 and setting a local
5.4 Verification of an emergency scenario

We formulate the following four properties essential to the correctness of the model:

P1. *A[ ] not deadlock*: The model is deadlock-free.

P2. *ParentStub.pending→!ParentStub.pending*: After the parent stub *a* sends an *ems* to *b*, *k* is eventually handled by *a*.

P3. *TargetComp.pending→!TargetComp.pending*: After the target component *b* triggers *k*1, *b* will eventually complete the handling of *k*1.

P4. *ChildStub1.pending→!ChildStub1.pending*: After Child stub 1 *c* sends an *ems* to *b*, *k*2 is eventually handled by *c*.

These properties are exactly the same as P1-P4 formulated in Section 5.3.1. The verification was performed on the same hardware platform used for verifying the model in Section 5.3.1. All the four properties are satisfied with verification time shorter than 0.04s. We omit the other two special cases in Figure 5.8(b) and Figure 5.8(c), however, the complete UPPAAL models and verification results for all cases can be found in [47].
5.4.2 Generalization of the verification results

We generalize the verification results in Section 5.4.1 by proving the same three assertions A1b-A3b stated in Section 5.3.2. Again, A1b and A2b can be re-interpreted as A1c and A2c.

A1c manifests the internal behavior equivalence between Top and any composite component with arbitrary depth level. The first step is to prove the internal behavior equivalence between Top and its subcomponents:

Lemma 4. Let \( c_i \) be a composite component with \( c_p = P_{c_i} \) and \( c_p = Top \). Then the internal behaviors of \( c_i \) and \( c_p \) are equivalent under the assumption of the triggering of an emergency scenario.

Proof. Each component runs the EMSP protocol (Definition 26 and Algorithm 23) and the mode-switch dependency rule (Definition 15 and Algorithm 6) to handle a single emergency scenario. There are only two possible types of upstream primitives that can be sent from \( c_i \) to \( c_p \) and from each \( c_j \in SC_{c_i} \) to \( c_i \):

- \( ems^k \): The EMSP protocol suggests that neither \( c_p \) nor \( c_i \) is expected to respond to the \( ems^k \) sender.
- \( msc^k \): The mode-switch dependency rule suggests that neither \( c_p \) nor \( c_i \) is expected to respond to the \( msc^k \) sender.

Hence the internal behaviors of \( c_p \) and \( c_i \) are equivalent in terms of the response to an upstream primitive. In addition, both \( c_p \) and \( c_i \) are able to actively send a downstream \( ems^k \) to a subcomponent. Lemma 4 follows accordingly.

Lemma 4 further implies:

Theorem 8. For each component \( c_i \in \bar{CC} \), the internal behavior of \( c_i \) is equivalent to that of Top under the assumption of the triggering of an emergency scenario.

Proof. The proof for Theorem 5 is also valid for this theorem.

Theorem 8 satisfies A1c. Since the parent stub is modeled as Top from the perspective of the target component, A1b is satisfied as well.

A2c manifests the external behavior equivalence between any component \( c_i \in \bar{CC} \) and a primitive component. The first step is to compare the external behaviors of a primitive component and its parent by the following lemma:
5.4 Verification of an emergency scenario

Lemma 5. For $c_j \in \mathcal{PC}$ and $c_i = P_{c_j}$, the external behaviors of $c_i$ and $c_j$ are equivalent under the assumption of the triggering of an emergency scenario.

Proof. Let $c_p = P_{c_i}$. By the EMSP protocol (Definition 26 and Algorithm 23), the only downstream primitive that can be sent from $c_i$ to $c_j$ and from $c_p$ to $c_i$ is an $ems^k$. By the mode-switch dependency rule (Definition 15 and Algorithm 6), after $c_j$ receives an $ems^k$ from $c_i$, $c_j$ will send an $msc^k$ to $c_i$ after $c_j$ completes its reconfiguration. After $c_i$ receives an $ems^k$ from $c_p$, $c_i$ may or may not propagate the $ems^k$ further to its subcomponents, depending on if $SC_{c_i}^A(k) = \emptyset$. In either case, the mode-switch dependency rule suggests that $c_i$ will eventually send an $msc^k$ to $c_p$. Hence the responses of $c_i$ and $c_j$ upon receiving a downstream $ems^k$ are the same.

In addition, both $c_i$ and $c_j$ are able to actively send an $ems^k$ to the parent. This completes the proof of Lemma 5.

Lemma 5 further implies:

Theorem 9. For each component $c_i \in \tilde{\mathcal{CC}}$, the external behavior of $c_i$ is equivalent to that of a primitive component under the assumption of the triggering of an emergency scenario.

Proof. The proof for Theorem 6 is also valid for this theorem.

Theorem 9 satisfies A2c. Since a child stub is modeled as a primitive component from the perspective of the target component, A2b is satisfied as well. Finally, A3b is addressed by Theorem 10:

Theorem 10. Each component $c_i \in \tilde{\mathcal{CC}}$ with arbitrary number of subcomponents can correctly execute its MSRM under the assumption of the triggering of an emergency scenario.

Proof. Let $N = |SC_{c_i}|$. Theorem 10 can be proven by mathematical induction based on $N$:

Basis: $N = 2$. Since the target component in our UPPAAL model has two child stubs, Theorem 10 directly follows from our verification results in Section 5.4.1.

Inductive step: Suppose that $c_i$ can correctly execute its MSRM with $N = n$, while $SC_{c_i} = \{c^0, c^1, \ldots, c^{n-1}\}$ ($n \in \mathbb{N}, n \geq 2$). The goal is to prove that it also works if $N = n + 1$, as another $c^i$ is added to $SC_{c_i}$. We borrow the same reasoning in the inductive step of the proof for Theorem 7. After $c^i$ is added to $SC_{c_i}$, $c^i$ can be a Type A or Type B component for a triggered emergency.
scenario \( k \). If \( T_{c_i}^k = B \), \( c_j^n \) does not affect the original MSRM execution of \( c_i \). If \( T_{c_i}^k = A \), \( SC_{c_i}^A(k) \) will be updated. Since the external behavior of \( c_j^n \) is equivalent to that of a primitive component, \( c_j^n \) is able to interact with \( c_i \) in the same way as the other Type A subcomponents of \( c_i \).

Combining the basis and the inductive step, the proof of Theorem 10 is completed.

### 5.5 Verification of concurrent non-emergency scenarios

The handling of concurrent non-emergency scenarios requires the MSP protocol, the mode-switch dependency rule, the MSR/MSQ queue checking rule, and the MSR/MSQ queue updating rule. Moreover, Our MSRM uses supplementary rules such as the MSQ delaying rule for handling atomic execution and the MSR/MSQ enqueuing rule for priority MSR/MSQ queues. In Section 5.3 we have mentioned that the MSQ delaying rule does not affect the verification result, thus excluded from the verification. The MSR/MSQ enqueuing rule does not affect the verification result either because it simply extends the MSR and MSQ queues from FIFO queues to priority queues. Hence, the MSR/MSQ enqueuing rule is excluded from the verification for the same reason.

Depicted in Figure 5.11, the model structure resembles the structure in Figure 5.4. However, the target component must be prepared for the concurrent arrival of two MSR primitives from both child stubs. To achieve the highest degree of concurrency, all scenarios can be triggered an unlimited number of times whenever possible. Consequently, the target component \( b \) in Figure 5.11(a) may have an \( msq^{k_0} \) arriving at its MSQ queue in conjunction with the arrival of an \( msr^{k_1} \) from \( b \) itself, an \( msr^{k_2} \) from Child stub 1 \( c \), and an \( msr^{k_3} \) from Child stub 2 \( d \).

#### 5.5.1 Modeling and verification in UPPAAL

We take the general case in Figure 5.11(a) as an example to model how the MSRM handles concurrent non-emergency scenarios. The UPPAAL model consists of five automata \( newPrimitive, TargetComp, ParentStub, ChildStub1 \), and \( ChildStub2 \). While the last three automata correspond to the parent and child stubs, \( newPrimitive \) and \( TargetComp \) jointly implement the MSRM of the target component \( b \).
5.5 Verification of concurrent non-emergency scenarios

The handling of concurrent non-emergency scenarios requires the MSP protocol, the mode-switch dependency rule, the MSR/MSQ queue checking rule, and the MSR/MSQ queue updating rule. Moreover, our MSRM uses supplementary rules such as the MSQ delaying rule for handling atomic execution and the MSR/MSQ enqueuing rule for priority MSR/MSQ queues. In Section 5.3 we have mentioned that the MSQ delaying rule does not affect the verification result, thus excluded from the verification. The MSR/MSQ enqueuing rule does not affect the verification result either because it simply extends the MSR and MSQ queues from FIFO queues to priority queues. Hence, the MSR/MSQ enqueuing rule is excluded from the verification for the same reason.

Depicted in Figure 5.11, the model structure resembles the structure in Figure 5.4. However, the target component must be prepared for the concurrent arrival of two MSR primitives from both child stubs. To achieve the highest degree of concurrency, all scenarios can be triggered an unlimited number of times whenever possible. Consequently, the target component in Figure 5.11(a) may have an msqarrival at its MSQ queue in conjunction with the arrival of an msr from itself, an msr from Child stub 1, and an msr from Child stub 2.

5.5.1 Modeling and verification in UPPAAL

We take the general case in Figure 5.11(a) as an example to model how the MSRM handles concurrent non-emergency scenarios. The UPPAAL model consists of five automata newPrimitive, TargetComp, ParentStub, ChildStub1, and ChildStub2. While the last three automata correspond to the parent and child stubs, newPrimitive and TargetComp jointly implement the MSRM of the target component b.

Shown in Figure 5.12, newPrimitive implements algorithms 9 and 10 in Chapter 4. An incoming MSR or MSQ is enqueued by Transition 1 or 2. When the target component b triggers k, an msrk is put in its MSR queue by Transition 6. Meanwhile, a variable pending is set to true. When k is completely handled by b, pending will be reset to false. Transition 3 is fired when b has propagated an msrk to a and subsequently receives an msdk when b is not in a transition state. If k = k2 or k = k3, b is obliged to send an msdk to c or d by Transition 4.

Figure 5.11: The UPPAAL model structure assuming the concurrent triggering of non-emergency scenarios

Figure 5.12: The UPPAAL model for receiving concurrent non-emergency scenarios
Figure 5.13 depicts TargetComp which bears the same structure as the model in Figure 5.5. The key difference is the management of MSQ and MSR queues for storing concurrent scenarios. The number of elements in $b.Q_{msq}$ is denoted by $MSQqueueN$. As per the analysis in Section 4.5.3 of Chapter 4, the minimum required size of $b.Q_{msq}$ is 2. A a component $c_i$ with an $msq^k$ in $c_i.Q_{msq}$ may receive another $msq^{k'}$ from $P_{c_i}$ after $c_i$ receives an $msd^k$ from $P_{c_i}$ and before $c_i$ removes the $msq^k$ from $c_i.Q_{msr}$. After $P_{c_i}$ sends the $msd^k$ to $c_i$, $P_{c_i}$ will leave the transition state for $k$ and can send an $msq^{k'}$ to $c_i$ at any time. If the $msq^{k'}$ arrives at $c_i$ before $c_i$ removes the $msq^k$ from $c_i.Q_{msq}$, the number of elements in $c_i.Q_{msq}$ will be 2. Nevertheless, the handling of an $msd^k$ in our model is treated as an atomic process (see transitions 3-5 in Figure 5.12 and transitions 11-13 in Figure 5.13) so that the parent stub only sends a new $MSQ$ to $b$ after $b$ leaves a transition state. Hence it is sufficient to set $MSQqueueN$ to 1. This is a realistic simplification that reduces verification overhead without changing the MSRM. We use $MSQqueue$ to denote the scenario ID $k$ of any $msq^k$ in $b.Q_{msq}$.

Since the mode switch of a component based on one scenario may invalidate the pending primitives in its MSR queue, Definition 21 in Chapter 4 distinguishes valid and invalid MSR primitives. An invalid MSR will be removed from the MSR queue by the MSR/MSQ queue updating rule. Our model implements the MSR queue of $b$ by two queues represented as two-dimensional arrays $MSRqueue[3][2]$ and $MSRqueue2[3][2]$. By default, $b$ enqueues an incoming MSR in the first queue. However, if $b$ receives an MSR from $c$ or $d$ while $b$ is in State MSCCollection, the MSR will be put in the second queue, in which no primitives are removed by the MSR/MSQ queue updating rule. For each queue, $MSRqueue[i][0]$ or $MSRqueue2[i][0]$ denotes the sender of the $i$th element (starting from 0) in the queue, while $MSRqueue[i][1]$ or $MSRqueue2[i][1]$ denotes the scenario ID of the $i$th element. In the model, we assign an ID 0-3 to $a$-$d$, and assign an ID 0-3 to $k_0$-$k_3$. The length of both queues are denoted by $MSRqueueN$ and $MSRqueue2N$. Section 4.5.3 of Chapter 4 declares that the minimum required size of $b.Q_{msr}$ is $2 \times |SC_b| + \sigma_b$, where $\sigma_b$ is the maximum number of scenarios that $b$ can trigger while $b$ runs in the same mode. Since $b$ has two child stubs in the model, $|SC_b| = 2$. Besides, $\sigma_b$ is assumed to be 1. Hence the minimum required size of $b.Q_{msr}$ is 5. In the model, both $MSRqueueN$ and $MSRqueue2N$ are 3. This is sufficient as $MSRqueueN+MSRqueue2N = 6 > 5$.

The MSR/MSQ queue checking rule is reflected in the guards of transitions 1, 6, and 7. A boolean variable Transition$S$ is used to indicate whether $b$ is in a transition state or not. Initially, $b$ is not in any transition state and
Figure 5.13 depicts TargetComp which bears the same structure as the model in Figure 5.5. The key difference is the management of MSQ and MSR queues for storing concurrent scenarios. The number of elements in $b.Q_{msq}$ is denoted by $MSQ_{queue}$.

As per the analysis in Section 4.5.3 of Chapter 4, the minimum required size of $b.Q_{msq}$ is 2. A component $c_i$ with an $msq_{k}$ in $c_i.Q_{msq}$ may receive another $msq_{k}'$ from $P_{c_i}$ after $c_i$ receives an $msd_{k}$ from $P_{c_i}$ and before $c_i$ removes the $msq_{k}$ from $c_i.Q_{msr}$.

After $P_{c_i}$ sends the $msd_{k}$ to $c_i$, $P_{c_i}$ will leave the transition state for $k$ and can send an $msq_{k}$ at any time. If the $msq_{k}'$ arrives at $c_i$ before $c_i$ removes the $msq_{k}$ from $c_i.Q_{msq}$, the number of elements in $c_i.Q_{msq}$ will be 2. Nevertheless, the handling of an $msd_{k}$ in our model is treated as an atomic process (see transitions 3-5 in Figure 5.12 and transitions 11-13 in Figure 5.13) so that the parent stub only sends a new $MSQ$ to $b$ after $b$ leaves a transition state. Hence it is sufficient to set $MSQ_{queue}$ to 1. This is a realistic simplification that reduces verification overhead without changing the MSRM. We use $MSQ_{queue}$ to denote the scenario ID $k$ of any $msq_{k}$ in $b.Q_{msq}$.

Since the mode switch of a component based on one scenario may invalidate the pending primitives in its MSR queue, Definition 21 in Chapter 4 distinguishes valid and invalid MSR primitives. An invalid MSR will be removed from the MSR queue by the MSR/MSQ queue updating rule. Our model implements the MSR queue of $b$ by two queues represented as two-dimensional arrays $MSR_{queue}[3][2]$ and $MSR_{queue2}[3][2]$. By default, $b$ enqueues an incoming MSR in the first queue. However, if $b$ receives an MSR from $c$ or $d$ while $b$ is in State $MSCCollection$, the MSR will be put in the second queue, in which no primitives are removed by the MSR/MSQ queue updating rule. For each queue, $MSR_{queue}[i][0]$ or $MSR_{queue2}[i][0]$ denotes the sender of the $i$th element (starting from 0) in the queue, while $MSR_{queue}[i][1]$ or $MSR_{queue2}[i][1]$ denotes the scenario ID of the $i$th element. In the model, we assign an ID 0-3 to $a$-$d$, and assign an ID 0-3 to $k_0$-$k_3$. The length of both queues are denoted by $MSR_{queue}$ and $MSR_{queue2}$. Section 4.5.3 of Chapter 4 declares that the minimum required size of $b.Q_{msr}$ is $2 \times |SC_b| + \sigma_b$, where $\sigma_b$ is the maximum number of scenarios that $b$ can trigger while $b$ runs in the same mode. Since $b$ has two child stubs in the model, $|SC_b| = 2$. Besides, $\sigma_b$ is assumed to be 1. Hence the minimum required size of $b.Q_{msr}$ is 5.

In the model, both $MSR_{queue}$ and $MSR_{queue2}$ are 3. This is sufficient as $MSR_{queue} + MSR_{queue2} = 6 > 5$.

The MSR/MSQ queue checking rule is reflected in the guards of transitions 1, 6, and 7. A boolean variable $TransitionS$ is used to indicate whether $b$ is in a transition state or not. Initially, $b$ is not in any transition state and...
TransitionS is false. If MSQqueueN>0, b will handle the MSQ by Transition 7, setting TransitionS to true. A local variable tempID keeps track of the scenario ID of the scenario currently being handled. If b has sent the first MSR in b.Q_msr to the parent stub a, a boolean variable locked will evaluate to true. If MSQqueueN==0 and MSRqueueN>0, and locked is false, b will handle the first pending MSR in b.Q_msr. If the MSR comes from a child stub, Transition 1 is taken; if the MSR comes from b itself, Transition 6 is taken.

The MSR/MSQ queue updating rule is implemented by the function updateQueue() of Transition 38 and the function updateQueue2() of Transition 39. The difference between updateQueue() and updateQueue2() is that only updateQueue() can remove an invalid msr_k1 from b.Q_msr. The function updateQueue2() is called if b is the MSDM of the current scenario k, i.e. T_b^k = B, which implies that a pending msr_k1 from b itself is valid and should not be removed.

The parent stub a, depicted in Figure 5.14, is modeled as Top from the perspective of b. Compared with the parent stub in Figure 5.6, the parent stub in Figure 5.14 exhibits more flexible behaviors. Every time a receives an msr_k from b by Transition 1, a local boolean variable MSRpending becomes true, allowing a to handle the msr_k by Transition 2. Moreover, a is able to send an msq_ko to b by Transition 7 no matter whether MSRpending is true or false. A boolean variable newMSQOK with an initial value true enables the triggering of k_0. The value of newMSQOK also affects the response of a to an msr_k from b. By default, newMSQOK is true, making Transition 2 directly followed by Transition 3. For an msr_k from b, where k is k_1, k_2 or k_3, there are three possible decisions made by the function decide() of Transition 3: (1) a sends an msq_k to b by Transition 5; (2) a sends an msq_k to b by Transition 6; (3) a sends an msq_ko to b by Transition 5. After sending an msq_ko to b, a will set a local boolean variable MSQpending to true. The variable MSQpending is reset to false when k_0 is completely handled by a.

In principle, a is able to send an msq_ko to b as long as a is in State Init and MSQpending is false. However, since the MSQ queue of a component has a higher priority than its MSR queue according to the MSR/MSQ queue checking rule, if a keeps sending an msq_ko to b whenever possible, the msr_k2 from c and msr_k3 from d will never be handled by b, thus violating the eventual mode-switch completion property. To circumvent such a problem, we add a constraint on the triggering of k_0: If MSQpending is true due to an msr_k from b, a is allowed to send an msq_ko to b at most N consecutive times. When N is reached, newMSQOK will be set to false so that the msr_k from b will eventually be handled. After that, newMSQOK becomes true again. In our
model, \( N = 3 \). Yet \( N \) can be tuned to a different value if necessary.

Figure 5.15 depicts a child stub \( c \) or \( d \), which is modeled as a primitive component from the perspective of \( b \). Each child stub uses a local boolean variable \( MSRpending \) to indicate the existence of a pending scenario. Whenever \( c \) sends an \( msr^k \) to \( b \), \( c \) will set its \( MSRpending \) to true. The \( MSRpending \) is reset to false when \( c \) completes the handling of \( k \). Likewise, the \( MSRpending \) of \( d \) is set to true whenever \( d \) sends an \( msr^k \) to \( b \), and reset to false when \( d \) completes the handling of \( k \).

We formulate the following five properties essential to the correctness of the model:

**P1.** \( A[] \text{not deadlock} \): The model is deadlock-free.

**P2.** \( ParentStub.MSQpending \rightarrow \neg ParentStub.MSQpending \): After the parent stub \( a \) sends an \( msq \) to \( b \), \( k \) is eventually handled by \( a \).

**P3.** \( pending \rightarrow \neg pending \): After the target component \( b \) triggers \( k \), \( b \) will eventually complete the handling of \( k \).

**P4.** \( ChildStub1.MSRpending \rightarrow \neg ChildStub1.MSRpending \): After Child stub \( 1 \) \( c \) sends an \( msr^k \) to \( b \), \( k \) is eventually handled by \( c \).

**P5.** \( ChildStub2.MSRpending \rightarrow \neg ChildStub2.MSRpending \): After Child stub \( 2 \) \( d \) sends an \( msr^k \) to \( b \), \( k \) is eventually handled by \( d \).
Properties P1-P4 can be directly mapped to the properties P1-P4 in Section 5.3.1. P5 represents the eventual mode-switch completion of k3 from d. Although the verification time is always shorter than 0.2s when only one of these scenarios is considered, the concurrent triggering of k_i-k_3 complicates the inter-component communication remarkably, contributing to tremendous verification overhead. To meet the high demand on memory consumption for the verification, we ran the verification on an HP Z420 Workstation, with Intel(R) Xeon(R) CPU ES-1603 0 @ 2.8GHz, 4 cores, 64GB memory, and Linux operating system. In addition, based on the mode mapping within the target component b for k_1, we consider four different cases which can be verified separately: (1) T^{k_1}_c = T^{k_1}_d = A; (2) T^{k_1}_c = A, T^{k_1}_d = B; (3) T^{k_1}_c = B, T^{k_1}_d = A; (4) T^{k_1}_c = T^{k_1}_d = B. Cases (2) and (3) are symmetrical, therefore, Case (3) is ignored in the verification. All the five properties P1-P5 are satisfied for all cases. A summary of the verification time is reported in Table 5.1. The peak memory usage is 14.607GB, observed from the verification of P4 for Case (1). Currently, UPPAAL is unable to utilize multi-cores. Hence it is difficult to improve the verification time even on a workstation.

An important property which has not been explicitly verified is that there should be no overflow in either b.Q_{msr} or b.Q_{msq}. However, this property has been implicitly satisfied via the verification of P1-P5, since none of P1-P5 would be satisfied, should an overflow occur. We omit the other two special cases in Figure 5.11(b) and Figure 5.11(c), however, the complete UPPAAL
models and verification results for all cases can be found in [47].

5.5.2 Generalization of the verification results

We generalize the verification results in Section 5.5.1 by proving the same three assertions A1b-A3b stated in Section 5.3.2. Again, A1b and A2b can be re-interpreted as A1c and A2c. The first step is to prove the internal behavior equivalence between Top and its subcomponent:

**Lemma 6.** Let $c_i$ be a composite component with $c_p = P_{c_i}$ and $c_p = Top$. Then the internal behaviors of $c_i$ and $c_p$ are equivalent under the assumption of concurrent non-emergency scenarios.

**Proof.** Let $c_j \in SC_{c_i}$. We first compare $c_i$ and $c_p$ in terms of the capability of actively sending a downstream primitive to a subcomponent as well as the precondition for the subcomponent to receive the primitive. Both $c_p$ and $c_i$ can actively send a downstream \(msq^k\) to a subcomponent such that \(k\) is not associated with any \(msr^k\) from the subcomponent. Based on the assumption in Section 5.5.1 that the handling of an \(msd^k\) by a component is treated as an atomic process, the precondition for $c_i$ to receive such an $msq^k$ from $c_p$ is: (1) $c_i.Q_{msq} = \emptyset$; and (2) $c_i$ is not in a transition state, or $c_i$ is in a transition state for $k'$ while $T_{c_i}^{k'} = B$. The precondition for $c_j$ to receive such an $msq^k$ from $c_i$ is the same.

Next we compare the responses of $c_p$ and $c_i$ for all possible upstream primitives. By the MSP protocol (Definition 14 and algorithms 3 and 4) and the mode-switch dependency rule (Definition 15 and Algorithm 6), the possible
upstream primitives sent from \( c_i \) to \( c_p \) or from \( c_j \) to \( c_i \) include \( msr^k \), \( msok^k \), \( msnok^k \), and \( msc^k \). By Lemma 2, the responses of \( c_i \) and \( c_p \) are equivalent if there are no concurrent scenarios. When concurrent scenarios are considered, \( c_i \) and \( c_j \) may receive an \( msq^{k'} \) (\( k' \neq k \)) from the parent after sending an upstream \( msr^k \). The \( msq^{k'} \) is actually not a response. Instead, it is actively sent from the parent as a result of the concurrent triggering of \( k' \). Moreover, \( c_p \) and \( c_i \) may not respond to the \( msr^k \) if the \( msr^k \) is removed by the MSR/MSQ queue updating rule (Definition 23 and Algorithm 21). Hence, the responses of \( c_i \) and \( c_p \) to an \( msr^k \) are equivalent even with concurrent scenarios.

An \( msok^k \) or \( msnok^k \) from \( c_i \) to \( c_p \) or from \( c_j \) to \( c_i \) implies that the parent \( c_p \) or \( c_i \) must be in the transition state for \( k \). The responses of \( c_p \) and \( c_i \) will not be affected by concurrent scenarios, thus equivalent by Lemma 2. Upon receiving an \( msc^k \), neither \( c_p \) nor \( c_i \) is expected to respond to the \( msc^k \) sender.

Since \( c_p \) and \( c_i \) can both actively send an \( msq^k \) to a subcomponent which receives the \( msq^k \) with the same precondition, and they have the same response for all possible upstream primitives, Lemma 6 follows.

Lemma 6 further implies:

**Theorem 11.** For each component \( c_i \in \widetilde{CC} \), the internal behavior of \( c_i \) is equivalent to that of \( Top \) under the assumption of concurrent non-emergency scenarios.

**Proof.** The proof for Theorem 5 is also valid for this theorem. \( \Box \)

Theorem 11 satisfies A1c. Since the parent stub is modeled as \( Top \) from the perspective of the target component, A1b is satisfied as well.

The first step to prove A2c is to compare the external behaviors of a primitive component and its parent:

**Lemma 7.** For \( c_j \in PC \) and \( c_i = P_{c_j} \), the external behaviors of \( c_i \) and \( c_j \) are equivalent under the assumption of concurrent non-emergency scenarios.

**Proof.** Let \( c_p = P_{c_i} \). We first compare \( c_i \) and \( c_j \) in terms of the capability of actively sending an upstream primitive to a parent as well as the precondition for the parent to receive the primitive. Both \( c_i \) and \( c_j \) can actively send an upstream \( msr^k \) to a parent. The precondition for \( c_i \) to receive such an \( msr^k \) from \( c_j \) is: (1) There is no \( msr^{k'}_{c_j} \in c_i.Q_{msr} \) and \( c_i \) is not in a transition state; or (2) There is no \( msr^{k'}_{c_j} \in c_i.Q_{msr} \) and \( c_i \) is in a transition state for \( k'' \) while \( T_{c_j}^{k''} = B \); or (3) There is an \( msr^{k'}_{c_j} \in c_i.Q_{msr} \), while \( c_i \) has just received an
5.5 Verification of concurrent non-emergency scenarios 163

\[ \text{msc}^{k'} \] from \( c_j \) but has not left the transition state for \( k' \). The precondition for \( c_p \) to receive such an \( \text{msr}^k \) from \( c_i \) is the same.

Next we compare the responses of \( c_i \) and \( c_j \) for all possible downstream primitives. By the MSP protocol (Definition 14 and algorithms 4 and 5), the possible primitives sent from \( c_p \) to \( c_i \) or from \( c_i \) to \( c_j \) include \( \text{msq}^k \), \( \text{msd}^k \), and \( \text{msi}^k \). An \( \text{msq}^k \) or \( \text{msi}^k \) from \( c_p \) to \( c_i \) or from \( c_i \) to \( c_j \) implies that \( c_i \) or \( c_j \) as a receiver must be in the transition state for \( k \). The responses of \( c_i \) and \( c_j \) will not be affected by concurrent scenarios, thus equivalent by Lemma 7. Upon receiving an \( \text{msd}^k \), neither \( c_i \) nor \( c_j \) is expected to respond to the \( \text{msd}^k \) sender.

Since \( c_i \) and \( c_j \) can both actively send an \( \text{msr}^k \) to a parent which receives the \( \text{msr}^k \) with the same precondition, and they have the same response for all possible downstream primitives, Lemma 7 follows.

Lemma 7 further implies:

**Theorem 12.** For each component \( c_i \in \tilde{CC} \), the external behavior of \( c_i \) is equivalent to that of a primitive component under the assumption of concurrent non-emergency scenarios.

**Proof.** The proof for Theorem 6 is also valid for this theorem.

Theorem 12 satisfies A2c. Since a child stub is modeled as a primitive component from the perspective of the target component, A2b is satisfied as well. Finally, A3b is addressed by Theorem 13:

**Theorem 13.** Each component \( c_i \in \tilde{CC} \) with arbitrary number of subcomponents can correctly execute its MSRM under the assumption of concurrent non-emergency scenarios.

**Proof.** Let \( N = |\text{SC}_{c_i}| \). Theorem 13 can be proven by mathematical induction based on \( N \):

**Basis:** \( N = 2 \). Since the target component in our UPPAAL model has two child stubs, Theorem 13 directly follows from our verification results in Section 5.5.1.

**Inductive step:** Suppose that \( c_i \) can correctly execute its MSRM with \( N = n \), while \( \text{SC}_{c_i} = \{c_j^0, c_j^1, \ldots, c_j^{n-1}\} \) \((n \in \mathbb{N}, n \geq 2)\). The goal is to prove that it also works if \( N = n + 1 \), as another \( c_j^n \) is added to \( \text{SC}_{c_i} \). No matter whether concurrent scenarios exist or not, each primitive exchanged between \( c_i \) and \( \text{SC}_{c_i} \) is always associated with a particular scenario. Every time \( c_i \) sends an \( \text{msq}^k \), \( \text{msi}^k \) or \( \text{msd}^k \) to \( \text{SC}_{c_i}^A(k) \), if \( T^{c_i}_{c_j} = B \), \( c_j^n \) does not affect the original
MSRM execution of $c_i$. If $T_{c_j}^k = A$, since the external behavior of $c_j^n$ is equivalent to that of a primitive component, $c_j^n$ is able to interact with $c_i$ in the same way as the other Type A subcomponents of $c_i$.

Combining the basis and the inductive step, the proof of Theorem 13 is completed. \qed

Theorem 13 brings an opportunity to reduce the verification overhead. Suppose that the $msr^{k_1}$ in Figure 5.11(a) is not triggered by the target component $b$ itself but comes from another "virtual" child stub (Child stub 3). Moreover, $T_b^{k_1} = A$ always holds. Then the original external behavior of $b$ from the perspective of $a$ and the original internal behavior of $b$ from the perspective of $c$ and $d$ are both preserved, since $a$, $c$, $d$ cannot distinguish whether $k_1$ is triggered by $b$ or comes from Child stub 3. By Theorem 13, it is sufficient to only consider two child stubs in the model. Therefore, it is OK to disable the triggering of $k_1$ and ignore the property P3 ($pending \rightarrow lpending$). Without $k_1$, the verification times of P1, P2, P4, and P5 are 47.012s, 59.103s, 116.972s, and 120.251s, respectively. Compared with the verification results in Table 5.1, this is a dramatic improvement.

### 5.6 Verification of concurrent emergency and non-emergency scenarios

The concurrent triggering of both emergency and non-emergency scenarios requires a component to run a complete MSRM that includes all the protocols and rules listed in Section 4.6.4 of Chapter 4. Again, atomic execution and priority MSR/MSQ queues are not considered in the verification. Depicted in Figure 5.16, the model structure is similar to the model structure in Figure 5.11. Figure 5.16(a) represents the most general case, where the target component $b$ may have an $msq^{k_0}$ arriving at its MSQ queue in conjunction with the arrival of an $msr^{k_1}$ from Child stub 1 $c$ and an $msr^{k_2}$ from Child stub 2 $d$. No non-emergency scenario is triggered by $b$. In the last section we have explained that this is an acceptable tactic to reduce verification time. Furthermore, an emergency scenario $k_3$ can be triggered by any component. Our MSRM assumes that a system should contain at most one emergency scenario. If $k_3$ comes from the parent stub $a$, $b$ may receive a downstream $ems^{k_3}$. If $k_3$ comes from a child stub, $b$ may receive an upstream $ems^{k_3}$ from either $c$ or $d$. Concerning the symmetrical structure of two child stubs, we assume that the upstream
emergency scenario is triggered by the system itself but comes from another "virtual" child stub (Child stub 3). Moreover, a system should contain at most one emergency scenario. If a child stub, 
k, may have an msr of an msq completed. The concurrent triggering of both emergency and non-emergency scenarios results in an opportunity to reduce the verification overhead. Furthermore, an emergency scenario is implemented in newPrimitive, TargetComp, ParentStub, ChildStub1, and ChildStub2 that have been introduced in Section 5.5.1. The MSRM of the target component, 
b, is responsible for receiving and processing new primitives such as an msq from the parent stub 
a, an msr or an ems from Child stub 1c, and an msr from Child stub 2d. When 
b is not in an NTS or ETS, newPrimitive also handles an MSD from 
a. In addition, due to the triggering of 
k, 
b may receive an msd from 
c after the arrival of an msr from 
c.

Figure 5.16: The UPPAAL model structure assuming concurrent emergency and non-emergency scenarios

5.6 Verification of concurrent emergency and non-emergency scenarios

ems is always from 
c. The emergency scenario 
k3 may also be triggered by 
b itself, which may send an ems to its parent stub and child stubs.

5.6.1 Modeling and verification in UPPAAL

We take the general case in Figure 5.16(a) as an example to model how the MSRM handles concurrent emergency and non-emergency scenarios. In the beginning of this section we have identified three cases with respect to the origin of the emergency scenario 
k3: (1) an ems from the parent stub 
a; (2) an ems issued by the target component 
b itself; (3) an ems from Child stub 1c. Since a system contains at most one emergency scenario, these three cases can be modeled and verified separately. The models and verification results of all these three cases are available in [47]. Our focus in this section is on Case (3), which is the most representative case.

The UPPAAL model consists of the same set of automata newPrimitive, TargetComp, ParentStub, ChildStub1, and ChildStub2 that have been introduced in Section 5.5.1. The MSRM of the target component 
b is implemented in newPrimitive and TargetComp. Shown in Figure 5.17, newPrimitive is responsible for receiving and processing new primitives such as an msq from the parent stub 
a, an msr or an ems from Child stub 1c, and an msr from Child stub 2d. When 
b is not in an NTS or ETS, newPrimitive also handles an MSD from 
a. In addition, due to the triggering of 
k3, 
b may receive an msd from 
c after the arrival of an msr from 
c.
Compared with the newPrimitive in Figure 5.12, the new newPrimitive additionally takes care of an \( msa^{k_1} \) and an \( ems^{k_3} \) from \( c \). An \( msa^{k_1} \) from \( c \) is handled by the MSA handling rule which is implemented by transitions 7-12 of newPrimitive and Transition 19 of TargetComp shown in Figure 5.18. It is only possible for \( c \) to send an \( msa^{k_1} \) to \( b \) if \( c \) has previously sent an \( msr^{k_1} \) to \( b \). After an \( msa^{k_1} \) arrives at \( b \), if \( b \) has already sent the \( msr^{k_1} \) to \( a \), \( b \) should also send an \( msa^{k_1} \) to \( a \) by Transition 8. If there is an \( msr^{k_1} \in b.Q_{msr} \), the \( msr^{k_1} \) will be removed by the function \( resetQ() \) of Transition 10. The function \( resetQ() \) also checks if \( b \) is in another NTS, having propagated an \( msq^{k'} \) to \( SC^b_a(k') \) without receiving all the replies. If yes, a boolean variable \( sendMSD \) becomes true, enabling Transition 11 that initiates a synchronization with Transition 19 of TargetComp through a channel \( AbortMS \). Figure 5.18 shows that Transition 19 is followed by transitions 28-30, as \( b \) sends an \( msd^{k'} \) to \( SC^b_a(k') \).

An \( ems^{k_3} \) from \( c \) is handled by the EMS receiving rule which is implemented by transitions 13-17 of newPrimitive and Transition 21 of TargetComp. Upon receiving the \( ems^{k_3} \), \( b \) puts the \( ems^{k_3} \) in \( b.Q_{ems} \) by the function \( updateEMSBuf() \) of Transition 13. The function \( updateEMSBuf() \) sets a boolean
Figure 5.17: The UPPAAL model for receiving concurrent emergency and non-emergency scenarios

Compared with the newPrimitive in Figure 5.12, the newPrimitive additionally takes care of an msa\(^1\) and an ems\(^3\) from c. An msa\(^1\) from c is handled by the MSA handling rule which is implemented by transitions 7-12 of newPrimitive and Transition 19 of TargetComp shown in Figure 5.18. It is only possible for c to send an msa\(^1\) to b if c has previously sent an msr\(^1\) to b. After an msa\(^1\) arrives at b, if b has already sent the msr\(^1\) to a, b should also send an msa\(^1\) to a by Transition 8. If there is an msr\(^1\) ∈ b.Q\(^\text{msr}\), the msr\(^1\) will be removed by the function resetQ() of Transition 10. The function resetQ() also checks if b is in another NTS, having propagated an msq\(^{k'}\) to SC\(^\text{A}\) without receiving all the replies. If yes, a boolean variable sendMSD becomes true, enabling Transition 11 that initiates a synchronization with Transition 19 of TargetComp through a channel AbortMS.

Figure 5.18: The UPPAAL model of a target component which handles concurrent emergency and non-emergency scenarios

An ems\(^3\) from c is handled by the EMS receiving rule which is implemented by transitions 13-17 of newPrimitive and Transition 21 of TargetComp. Upon receiving the ems\(^3\), b puts the ems\(^3\) in b.Q\(^\text{ems}\) by the function updateEMSBuf() of Transition 13. The function updateEMSBuf() sets a boolean...
variable $EMSBuffering$ to true to indicate the existence of a pending emergency scenario in $b.Q_{ems}$. Another function $sendMSAorNot()$ of Transition 13 determines if $b$ needs to send an $msa^{k_1}$ or $msa^{k_2}$ to $a$. If yes, a boolean variable $sendMSA$ becomes true, enabling Transition 14. Finally, if $b$ is in an NTS for $k$, having propagated an $msq^k$ to $SC^A_k(k)$ without receiving all the replies, a boolean variable $SubTS$ becomes true so that Transition 17 initiates a synchronization with Transition 21 of $TargetComp$ through another channel $AbortMS2$. Then $b$ will abort the handling of $k$ to speed up the handling of the $ems^{k_3}$.

The handling of the emergency scenario $k_3$ is implemented by transitions 39-49 of $TargetComp$ in Figure 5.18. As $b$ starts to handle an $ems^{k_3}$ by Transition 39, a boolean variable $ETS$ becomes true, indicating that $b$ enters an ETS. The function $EmergencyMM()$ performs the mode mapping within $b$ for $k_3$. The mode mapping determines if $b$ and $d$ are Type A components for $k_3$. If $T_{b}^{k_3} = B$, $b$ will be the MSDM of $k_3$ and a boolean variable $eMSDM$ will be set to true. After mode mapping, $b$ applies the preliminary EMS handling rule by transitions 40-42. If $\exists msq^{k_0} \in b.Q_{msq}$ and $b$ is not the MSDM of $k_3$, $b$ will abort the handling of $k_0$ by sending an $msnok^{k_0}$ to $b$ by Transition 41. Transitions 44-47 implement the EMSP protocol. There is no need for $b$ to propagate an $ems^{k_3}$ back to the $EMS$ sender $c$. If $T_{d}^{k_3} = A$, a boolean variable $EMStod$ becomes true and $b$ will send an $ems^{k_3}$ to $d$ by Transition 45. If $T_{b}^{k_3} = A$, $eMSDM$ will be false and $b$ will send an $ems^{k_3}$ to $a$ by Transition 47. After the propagation of the $ems^{k_3}$, the mode-switch dependency rule is applied to complete the emergency mode switch.

The parent stub $a$, depicted in Figure 5.19, is modeled as $Top$ from the perspective of $b$. It retains all the states and transitions of the parent stub in Figure 5.14. Due to the emergency scenario $k_3$, $a$ must be prepared for the $msa^{k_1}$ and $ems^{k_3}$ from $b$. The arrival of an $ems^{k_3}$ corresponds to Transition 19, as $b$ enters the $Emergency$ state, in which $b$ waits for an $msc^{k_3}$ from $b$. The arrival of an $msa^{k_1}$ is relatively more unpredictable. After $a$ receives an $msr^{k_1}$ from $b$, the $msa^{k_1}$ may come at any time. Depending on the current state of $a$, an incoming $msa^{k_1}$ may lead to one of the transitions 8, 15, 18.

Figure 5.20 depicts a child stub $c$ or $d$, which is modeled as a primitive component from the perspective of $b$. Note that the first phase of the propagation of a non-emergency scenario from $b$ to both child stubs is abstracted away from the model for simplification purpose. Thereby the transmission of $msq^k$, $msok^k$, and $msnok^k$ between $b$ and both child stubs will be invisible in the model. This allows us to merge all the five states of the child stub in Figure 5.15 into a single state $Init$, as shown in Figure 5.20. Such a simplification is sensible since the abstracted interactions between $b$ and both child
5.6 Verification of concurrent emergency and non-emergency scenarios

stubs during a non-emergency mode switch have been verified in Section 5.5. Similarly, the MSC from each child stub to $b$ is abstract away, since the mode-switch dependency rule has already been verified. After Child stub 1 $c$ sends an $msr^{k_1}$ to $b$, a local boolean variable pending becomes true. After that $c$ may send an $msa^{k_1}$ to $b$ by Transition 2 at any time. Then $c$ may or may not send a subsequent $ems^{k_2}$ to $b$ depending on if $T_{c}^{k_3} = A$. A subsequent $ems^{k_3}$ following the $msa^{k_1}$ is sent to $b$ by Transition 4. If $c$ has not sent an $msr^{k_1}$ to $b$, $c$ can send an $ems^{k_3}$ to $b$ by Transition 7 at any time. The MSRM allows the recurrent triggering of an emergency scenario. However, since an emergency mode switch is a rare event, even if it can be triggered multiple times, the interval between two such events must be long enough for each component to complete its emergency mode switch. Then each triggering of $k_3$ is independent and it is sufficient to trigger $k_3$ only once in the model. This assumption considerably reduces verification overhead.

We formulate the following five properties essential to the correctness of the model:

P1. $A[f \not \text{ deadlock}]$: The model is deadlock-free.

P2. $ParentStub.pending \rightarrow !ParentStub.pending$: After the parent stub $a$ sends an $msq^{k_0}$ to $b$, $k_0$ is eventually handled by $a$.

P3. $ChildStub1.pending \rightarrow !ChildStub1.pending$: After Child stub $1$ $c$ sends an $msr^{k_1}$ to $b$, $k_1$ is eventually handled by $c$.
Figure 5.20: The UPPAAL model of a child stub which handles concurrent emergency and non-emergency scenarios

**P4.** ChildStub2.pending→!ChildStub2.pending: After Child stub 2 d sends an msr\(^k_2\) to b, \(k_2\) is eventually handled by d.

**P5.** EMSBuffering→!EMSBuffering: After b receives an ems\(^k_3\) from Child stub 1 c, b will eventually complete the handling of \(k_3\).

Properties P1-P4 have been verified in Section 5.5. P5 means that the emergency scenario is eventually handled by b after b has received an ems\(^k_3\) from c. Since \(k_3\) comes from c, \(T_c^{k_3} = A\). The mode mapping of b decides if b and d are Type A components for \(k_3\), leading to four different cases which can be verified separately: (1) \(T_b^{k_3} = T_d^{k_3} = A\); (2) \(T_b^{k_3} = A, T_d^{k_3} = B\); (3) \(T_b^{k_3} = B, T_d^{k_3} = A\); (4) \(T_b^{k_3} = T_d^{k_3} = B\). The verification was performed on the HP Z420 Workstation mentioned in Section 5.5.1. All the five properties P1-P5 are satisfied for all cases. A summary of the verification time is reported in Table 5.2. The peak memory usage is 17.208GB, observed from the verification of P4 for Case (2).

We omit the other two special cases in Figure 5.16(b) and Figure 5.16(c), however, the complete UPPAAL models and verification results for all cases can be found in [47].
5.6 Verification of concurrent emergency and non-emergency scenarios

<table>
<thead>
<tr>
<th>Mode mapping within $b$</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{b}^{k_{1}} = A$</td>
<td>500.324s</td>
<td>582.982s</td>
<td>598.52s</td>
<td>686.137s</td>
<td>439.983s</td>
</tr>
<tr>
<td>$T_{d}^{k_{1}} = A$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_{b}^{k_{1}} = A$</td>
<td>551.106s</td>
<td>689.889s</td>
<td>858.076s</td>
<td>1369.149s</td>
<td>491.271s</td>
</tr>
<tr>
<td>$T_{d}^{k_{1}} = B$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_{b}^{k_{1}} = B$</td>
<td>506.4s</td>
<td>579.912s</td>
<td>591.926s</td>
<td>655.323s</td>
<td>437.444s</td>
</tr>
<tr>
<td>$T_{d}^{k_{1}} = A$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_{d}^{k_{1}} = B$</td>
<td>592.111s</td>
<td>631.296s</td>
<td>661.763s</td>
<td>721.614s</td>
<td>478.149s</td>
</tr>
</tbody>
</table>

Table 5.2: A summary of the verification time for the concurrent triggering of one emergency scenario and three non-emergency scenarios

5.6.2 Generalization of the verification results

We generalize the verification results in Section 5.6.1 by proving the same three assertions A1b-A3b stated in Section 5.3.2. Again, A1b and A2b can be re-interpreted as A1c and A2c. The first step is to prove the internal behavior equivalence between $Top$ and its subcomponents:

**Lemma 8.** Let $c_i$ be a composite component with $c_p = P_{c_i}$ and $c_p = Top$. Then the internal behaviors of $c_i$ and $c_p$ are equivalent under the assumption of concurrent emergency and non-emergency scenarios.

**Proof.** Let $c_j \in SC_{c_i}$. We first compare $c_i$ and $c_p$ in terms of the capability of actively sending a downstream primitive to a subcomponent as well as the precondition for the subcomponent to receive the primitive. According to the proof for Lemma 6, both $c_p$ and $c_i$ can actively send a downstream $msq^k$ to a subcomponent which receives the $msq^k$ with the same precondition. If an emergency scenario is triggered by $c_p$, $c_i$ may receive an $ems^k$ from $c_p$ when (1) $c_i.Q_{msq} = \emptyset$ and $c_i.Q_{ems} = \emptyset$; and (2) $c_i$ is not in an NTS or ETS, or $c_i$ is in an NTS for $k$ while $T_{c_i}^k = B$. The same precondition can be observed from an $ems^k$ actively sent from $c_i$ to $c_j$. Moreover, When $c_p$ applies the MSA handling rule (Definition 30 and Algorithm 27), $c_p$ may actively send an $msd^k$ to $c_i$ after sending an $msq^k$ to $c_i$. When $c_p$ or $c_i$ applies the MSA handling rule, $c_i$ may also actively send an $msd^k$ to $c_j$ after sending an $msq^k$ to $c_j$.

Next we compare the responses of $c_p$ and $c_i$ for all possible upstream prim-
Chapter 5. Verification of the mode-switch runtime mechanism

...itives. We know from the proof for Lemma 6 that \(c_p\) and \(c_i\) have the same response for all upstream primitives related to a non-emergency scenario. When an emergency mode switch is considered, a component may send an \(msa^k\) or \(ems^k\) to its parent. The MSA handling rule and the EMSP protocol (Definition 26 and Algorithm 23) suggest that neither \(c_p\) nor \(c_i\) is expected to respond to an \(msa^k\) or \(ems^k\) sender.

Since \(c_p\) and \(c_i\) can both actively send an \(ems^k\), an \(msq^k\), a subsequent \(msd^k\) after an \(msq^k\) to a subcomponent which receives these primitives with the same preconditions, and they have the same response for all possible upstream primitives, Lemma 8 follows.

Lemma 8 further implies:

**Theorem 14.** For each component \(c_i \in \tilde{CC}\), the internal behavior of \(c_i\) is equivalent to that of \(Top\) under the assumption of concurrent emergency and non-emergency scenarios.

*Proof.* The proof for Theorem 5 is also valid for this theorem. \(\Box\)

Theorem 14 satisfies A1c. Since the parent stub is modeled as \(Top\) from the perspective of the target component, A1b is satisfied as well.

The first step to prove A2c is to compare the external behaviors of a primitive component and its parent:

**Lemma 9.** For \(c_j \in PC\) and \(c_i = P_{c_j}\), the external behaviors of \(c_i\) and \(c_j\) are equivalent under the assumption of concurrent emergency and non-emergency scenarios.

*Proof.* Let \(c_p = P_{c_i}\). We first compare \(c_i\) and \(c_j\) in terms of the capability of actively sending an upstream primitive to a parent as well as the precondition for the parent to receive the primitive. We know from the proof for Lemma 7 that both \(c_i\) and \(c_j\) may actively send an \(msr^k\) to the parent that receives the \(msr^k\) with the same precondition. When an emergency mode switch is considered, both \(c_i\) and \(c_j\) may actively send an \(msa^k\) or \(ems^k\) to the parent. The precondition for \(c_i\) to receive an \(msa^k\) from \(c_j\) is: (1) \(c_i\) has received an \(msr^k\) from \(c_j\) at an earlier time \(t_1\); and (2) \(c_i\) has not sent an \(MSQ\) to \(c_j\) after \(t_1\). If \(c_j\) has sent such an \(MSQ\), \(c_i\) has not received the reply from \(c_j\); and (3) There is no \textbf{EMS} transmission between \(c_i\) and \(c_j\) before \(c_i\) receives the \(msa^k\). The \(msa^k\) from \(c_j\) to \(c_i\) results from the emergency scenario triggering rule (Definition 29 and Algorithm 26) applied by \(c_j\). The precondition for \(c_p\) to receive such an \(msa^k\) from \(c_i\) is the same, whereas the \(msa^k\) may be sent when \(c_i\)
 applies either the emergency scenario triggering rule or the MSA handling rule (Definition 30 and Algorithm 27).

Apart from the $msa^k$, both $c_i$ and $c_j$ may actively send an $ems^{k'}$ to the parent by the EMSP protocol (Definition 26 and Algorithm 23). The precondition for $c_i$ to receive an $msr^k$ from $c_j$ mentioned in the proof for Lemma 7 still holds for $c_i$ to receive an $ems^{k'}$ from $c_j$. In addition, $c_i, Q_{ems}$ must be empty before it can receive an $ems^{k'}$. The precondition for $c_p$ to receive an $ems^{k'}$ from $c_i$ is the same. What deserves extra attention is an upstream $ems^{k'}$ that follows an $msa^k$. Since $c_j$ sends an $msa^k$ to $c_j$ by applying the emergency scenario triggering rule when $c_j$ triggers an emergency scenario $k'$ as the MSS, $c_j$ must send a subsequent $ems^{k'}$ to $c_i$. By contrast, after $c_i$ sends an $msa^k$ to $c_p$ after receiving an $msa^k$ from $c_j$ and applying the MSA handling rule, $c_i$ does not always send a subsequent $ems^{k'}$ to $c_p$. If $T_{c_i}^{k'} = B$, $c_i$ will not propagate the $ems^{k'}$ to $c_p$. Apparently, if a component is able to handle an $ems^{k'}$ that follows an $msa^k$, it causes no problem if the $ems^{k'}$ does not come, which means that the external behaviors of $c_i$ and $c_j$ can be considered to be equivalent from this perspective.

We know from the proof for Lemma 7 that $c_i$ and $c_j$ have the same responses for all downstream primitives related to a non-emergency scenario. When an emergency mode switch is considered, $c_i$ and $c_j$ may receive an $ems^k$ from the parent and they both reply with an $msc^k$. Hence they have the same response for all possible downstream primitives.

The reasoning above proves the lemma. \hfill \Box

Lemma 9 further implies:

**Theorem 15.** For each component $c_i \in \tilde{CC}$, the external behavior of $c_i$ is equivalent to that of a primitive component under the assumption of concurrent emergency and non-emergency scenarios.

**Proof.** The proof for Theorem 6 is also valid for this theorem. \hfill \Box

Theorem 15 satisfies A2c. In the child stub described in Figure 5.20 is modeled as a primitive component from the perspective of the target component, though the child stub also considers the case when the child stub is not a primitive component and an $msa^k$ is not always followed by an $ems^{k'}$. Thus A2b is satisfied as well. Finally, A3b is addressed by Theorem 16:

**Theorem 16.** Each component $c_i \in \tilde{CC}$ with arbitrary number of subcomponents can correctly execute its MSRM under the assumption of concurrent emergency and non-emergency scenarios.
Proof. Let $N = |SC_{c_i}|$. Theorem 16 can be proven by mathematical induction based on $N$:

**Basis:** $N = 2$. Since the target component in our UPPAAL model has two child stubs, Theorem 16 directly follows from our verification results in Section 5.6.1.

**Inductive step:** Suppose that $c_i$ can correctly execute its MSRM with $N = n$, while $SC_{c_i} = \{c_j^0, c_j^1, \ldots, c_j^{n-1}\}$ ($n \in \mathbb{N}, n \geq 2$). The goal is to prove that it also works if $N = n + 1$, as another $c_j^n$ is added to $SC_{c_i}$. Since an emergency mode switch has already been considered before adding $c_j^n$ and the system contains only one emergency scenario, $c_j^n$ will not bring additional emergency scenarios to the system. Therefore, the proof for Theorem 13 also holds here.

Combining the basis and the inductive step, the proof of Theorem 16 is completed.

5.7 Summary

This chapter is dedicated to the verification of the MSRM presented in Chapter 4. The verification is conducted progressively through four different cases, starting from the triggering of a single emergency or non-emergency scenario, which is followed by the concurrent triggering of non-emergency scenarios, to the concurrent triggering of both emergency and non-emergency scenarios. For each case, the verification consists of two steps. The first step is model checking in the model checker UPPAAL, where the MSRM is implemented in a target component with one parent stub and two child stubs. The parent stub simulates the communication between the target component and its parent while a child stub simulates the communication between the target component and a subcomponent. Our verification results indicate that the UPPAAL model satisfies key properties such as deadlock freeness and eventual mode-switch completion. The second step aims to generalize the UPPAAL verification results to an arbitrary system by manual theorem proving. The nub is to justify why one parent stub and two child stubs can faithfully represent an arbitrary system. The verification in this chapter proves the correctness of our MSRM for all the four cases.
Chapter 6

Mode-switch timing analysis

To the best of our knowledge, almost all multi-mode systems are also real-time systems, whose correctness depends not only on the correctness of the computing result, but also on the time when the result is delivered [21]. A classic example of real-time systems is the airbag of a car. When a collision leads to the inflation of an airbag, the inflation must be triggered at the proper time, neither too early nor too late. Therefore, the mode switch of a multi-mode system must fulfill both functional and timing requirements. Chapter 5 has verified the functional correctness of our MSRM. What is of equal importance is to meet the timing constraints such as the maximum tolerated mode-switch time. In this chapter, we present a mode-switch timing analysis for our MSRM to compute the worst-case mode-switch time, i.e., the interval between the triggering of a scenario and the mode-switch completion based on this scenario. The timing analysis assumes the triggering of a single non-emergency scenario with atomic execution. A more advanced timing analysis, which supports the concurrent triggering of both emergency and non-emergency scenarios, is out of the scope of this thesis. In addition to the mode-switch timing analysis, we present a model-checking based approach to derive the worst-case atomic execution time of an Atomic Execution Group (AEG).

6.1 Overview

According to Chapter 4, a scenario is triggered by an MSS and propagated to Type A components. The propagation of a non-emergency scenario is guided
by the MSP protocol of the MSRM. The MSP protocol first identifies the MSDM of the scenario and then the MSDM initiates a 2-phase propagation. In Phase 1, the MSDM checks if all Type A components are ready to switch mode. In Phase 2, the MSDM triggers the mode switch if all Type A components are ready, or aborts the mode switch otherwise. After a mode switch is triggered by the MSDM, the mode switch is performed for all Type A components and completed bottom-up in accordance with the mode-switch dependency rule. Atomic execution is handled by applying the MSQ delaying rule in Phase 1.

From the moment a scenario is triggered to the mode-switch completion of this scenario, we divide a complete mode-switch process into three non-overlapping and continuous phases:

- **Phase 1:** Identifying the MSDM of a scenario. Phase 1 starts when an MSS triggers a non-emergency scenario $k$ by sending an $msr^k$ to the parent. When the $msr^k$ is propagated to a component $c_i$ which is identified as the MSDM of $k$, Phase 1 is completed. Phase 1 is skipped when the MSS is $Top$.

- **Phase 2:** Collecting the current states of Type A components. Phase 2 is equivalent to Phase 1 of the MSP protocol, starting when $c_i$, the MSDM of $k$, propagates a $msq^k$ to $SC^A_{c_i}(k)$ and ending when $c_i$ has received all the replies from $SC^A_{c_i}(k)$. In order to trigger a mode switch, all the replies must be $msok^k$.

- **Phase 3:** Mode-switch execution. Phase 3 combines Phase 2 of the MSP protocol with the reconfigurations of Type A components, starting when $c_i$ propagates an $msi^k$ to $SC^A_{c_i}(k)$ and ending when $c_i$ leaves the NTS for $k$.

These three phases are demonstrated in an example in Figure 6.1, which is based on Figure 4.9 in Chapter 4.3. A scenario $k$ is triggered by an MSS $e$, with $a$ as the MSDM, and $a, c, d, e$ as the Type A components. Phase 1 is represented by the upstream transmission of the $msr^k$ from $e$ to $c$, and then from $c$ to the MSDM $a$. Phase 2 starts when $a$ issues an $msq^k$ and ends when $a$ has received an $msok^k$ from both $c$ and $d$. Phase 3 starts when $a$ issues an $msi^k$ and ends when $a$ completes its mode switch.

The timing analysis of each phase can be performed separately and the total mode-switch time is the summation of the time spans of all phases. The rejection of a scenario will be excluded from our timing analysis because no mode
switch is performed. However, it would be necessary to analyze the time spent on rejected scenarios when concurrent scenarios are taken into account, since even a rejected scenario may delay the handling of another pending scenario. In this thesis, we only focus on the timing analysis of a single non-emergency scenario, for which the mode-switch time depends on the transmission time of each type of primitive, the computation time of the MSRM, the reconfiguration time of each Type A component, and the atomic execution time of each AEG. The MSRM computation time of each component varies at different stages of its mode switch. For instance, it takes time to refer to the local mode mapping of a composite component or check the current state upon receiving an \texttt{MSQ}. The key timing factors and their notations used in our timing analysis are listed as follows:

- \( c_i \cdot t_{msr} \), \( c_i \cdot t_{msq} \), \( c_i \cdot t_{ok} \), \( c_i \cdot t_{nok} \), \( c_i \cdot t_{msi} \), \( c_i \cdot t_{msd} \), \( c_i \cdot t_{msc} \): the transmission time of each type of primitive between \( c_i \) and \( SC_{c_i} \). To simplify the timing analysis, we assume that the transmission time of the same type of primitive from \( c_i \) to its different subcomponents is the same.
• \( c_i.msdt \): the Mode-Switch Detecting Time (MSDT) of an MSS \( c_i \) when it triggers a scenario, i.e. the time required to trigger a scenario after \( c_i \) detects a mode-switch event. The time is spent in analyzing the mode-switch event and deriving the new mode of \( c_i \).

• \( c_i.mmt \): the Mode Mapping Time (MMT) of \( c_i \in CC \), i.e. the time required to obtain the mode mapping results from the local mode mapping of \( c_i \).

• \( c_i.sct \): the State-Checking Time (SCT) of \( c_i \), i.e. the time required to check if the current state of \( c_i \) allows a mode switch.

• \( AE_{G_i} \): the atomic execution time of an AEG \( G_i \).

• \( c_i.qrt \): the Query Response Time (QRT) of \( c_i \), i.e. the time required to respond to an \( msq^k \) with an \( msok^k \) or \( msnok^k \).

• \( c_i.rct \): the ReConfiguration Time (RCT) of \( c_i \) from one mode to another mode.

• \( c_i.mst \): the Mode-Switch Time (MST) of \( c_i \), i.e. the time required to complete the mode switch of \( c_i \) after \( c_i \) starts reconfiguration.

All the timing factors are for the worst case and can be mode-dependent or scenario-dependent. For instance, the reconfiguration time of \( c_i \) may vary according to its current mode and the new mode. Parallel mode-switch execution among different components is permitted in our timing analysis. We also assume that the values of all these timing factors except \( AE_{G_i} \) are already provided. The calculation of these timing factors is a separate research problem with many other additional concerns such as hardware platform, scheduling policy, and resource sharing.

Next, our mode-switch timing analysis will be presented phase by phase.

6.2 Phase 1—Identifying the MSDM of a scenario

Phase 1 only exists when the MSS is not \( Top \). Let \( \alpha \) be the MSS of a scenario \( k \) and \( \beta \) be the MSDM of \( k \). The only activity in Phase 1 is the upstream and stepwise propagation of an \( msr^k \) from \( \alpha \) to \( \beta \) through \( C_{\alpha}^\beta \), the set of components forwarding the \( msr^k \) between \( \alpha \) and \( \beta \).

Figure 6.2 illustrates Phase 1 by an example, where \( d \) is the MSS of a scenario \( k \) and \( a \) is the MSDM of \( k \), with \( C_d^a = \{b, c\} \). When \( d \) detects a
mode-switch event, it spends $d.msdt$ time units determining to trigger $k$ by sending an $msr^k$ to its parent $c$. After $c.t_{msr}$, the $msr^k$ arrives at $c$, which first spends $c.mmt$ on mode mapping and then spends $c.sct$ checking its current state. In this example, since the current state of $c$ allows the mode switch for $k$, $c$ forwards the $msr^k$ further to its parent $b$. The behaviors of $b$ and $a$ together with their timing factors displayed in Figure 6.2 can be explained by the same token. Let $T_1$ denote the duration of Phase 1. In Figure 6.2, $T_1$ can be easily calculated as

$$T_1 = d.msdt + c.t_{msr} + c.mmt + c.sct + b.t_{msr} + b.mmt + b.sct + a.t_{msr} + a.mmt + a.sct$$  \hspace{30mm} (6.1)$$

Generalizing (6.1),

$$T_1 = \alpha.msdt + \sum_{c_i \in C_{\alpha} \cup \{\beta\}} (c_i.t_{msr} + c_i.mmt + c_i.sct)$$  \hspace{30mm} (6.2)$$
6.3 Phase 2—Collecting the current states of Type A components

Phase 2 is featured by the downstream and stepwise propagation of an $msq^k$ originating from the MSDM, followed by the upstream transmission of an $msok^k$ replied by each Type A component. Atomic execution is also handled in this phase. The timing analysis in Phase 2 is essentially realized by the composition of the QRT of each Type A component.

Figure 6.3 illustrates the behavior of a composite component in Phase 2 by an example, where a composite component $a$ receives an $msq^k$ from its parent and propagates the $msq^k$ to its Type A subcomponents $b$, $c$, and $e$. Among the subcomponents of $a$, there is an AEG $G_1$ including $b$ and $c$. Upon receiving the $msq^k$ from the parent, $a$ suspends its current execution and spends $a.sct$ time units checking its state. If the current state of $a$ does not allow the mode switch, $a$ will send an $msnok^k$ back to its parent, leading to the rejection of $k$. In this example, $a$ is ready to switch mode. Thereafter $a$ spends $a.mmt$ deriving its Type A subcomponents based on its mode mapping. However, if $a$ has a pending $msr^k$ in its MSR queue that has been sent to its parent, $a$ must have obtained the mode mapping results in Phase 1. Under such a condition, $a.mmt = 0$ in Phase 2.

![Figure 6.3: Mode-switch timing analysis—Phase 2](image)

Due to the atomic execution of $G_1$, the $msq^k$ from $a$ to $b$ and $c$ is delayed
by $AE_{\mathcal{G}_1}$ (represented by frameless grey bars), which is the worst-case atomic execution time of $\mathcal{G}_1$. The atomic execution time of $\mathcal{G}_1$ may vary depending on when $a$ starts to check the execution status of $\mathcal{G}_1$. In the best case, there is no ongoing atomic execution in $\mathcal{G}_1$ when $a$ decides to propagate the $msq^k$. Then the $msq^k$ to $b$ and $c$ will not be delayed. In the worst case though, the atomic execution of $\mathcal{G}_1$ just starts when $a$ decides to propagate the $msq^k$, thus delaying the $msq^k$ to $\mathcal{G}_1$ by $AE_{\mathcal{G}_1}$.

When $b$, $c$ and $e$ receive the $msq^k$, they will reply with an $msok^k$ or an $msnok^k$ after some time, denoted by $b.qrt$, $c.qrt$, and $e.qrt$, respectively. It can be observed that the QRT of $a$ relies on the QRT of $b$, $c$ and $e$. More precisely, $a.qrt$ is determined by $a.sct$, $a.mmt$ and the longest response from $b$, $c$ and $e$:

$$a.qrt = a.sct + a.mmt + \max\{AE_{\mathcal{G}_1} + a.t_{msq} + b.qrt + a.t_{ok}, AE_{\mathcal{G}_1} + a.t_{msq} + c.qrt + a.t_{ok}, a.t_{msq} + e.qrt + a.t_{ok}\}$$

(6.3) where $a.mmt = 0$ if $a.Q_{msr}[1] = msr^k$.

In general, for each $c_i \in \mathcal{P}C$ which receives an $msq^k$, $c_i$ is able to reply with an $msok^k$ or $msnok^k$ in response to an $msq^k$ right after checking its state, implying

$$c_i.qrt = c_i.sct$$

(6.4)

For each component $c_i \in \mathcal{C}C$ such as $a$ in Figure 6.3, after $c_i$ receives an $msq^k$, its QRT depends on the QRT of $SC_{c_i}^A(k)$ and if there is any AEG among $SC_{c_i}$. Let $\mathcal{G}_{c_i}$ be the set of AEGs among $SC_{c_i}$ in its current mode (see also Definition 17 in Section 4.4). If $\mathcal{G}_{c_i} = \emptyset$,

$$c_i.qrt = c_i.sct + c_i.mmt + \max\{c_i.t_{msq} + c_o.qrt + c_i.t_{ok}\}$$

(6.5) where $c_i.mmt = 0$ if $c_i \in \mathcal{C}C \cap \{C_{\beta, \alpha}^A\}$.

If $\mathcal{G}_{c_i} \neq \emptyset$, then let $\mathcal{G}_{c_i} = \{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n\} \ (n \in \mathbb{N})$ with each $\mathcal{G}_l \ (l = [1, n])$ being an AEG. Also, let $\mathcal{H}_{c_i}$ be the set of components in $SC_{c_i}$ without atomic execution. We partition $SC_{c_i}^A(k)$ into two disjunct groups: the atomic group and the non-atomic group. In addition to (6.5), a delay $AE_{\mathcal{G}_l}$ is introduced to each $c_o \in SC_{c_i}^A(k) \cap \mathcal{G}_l$:
\( c_i.qrt = c_i.sct + c_i.mmt + \max\{ c_i.t_{msq} + c_j.qrt + c_i.t_{ok}, AE_{\vec{G}_i} + c_i.t_{msq} + c_o.qrt + c_i.t_{ok} \} \)

\[ \text{where } c_i.mmt = 0 \text{ if } c_i \in CC \cap \{ C^\beta_{\alpha} \cup \alpha \}. \]

By composing the QRT of each Type A component for \( k \) in the bottom-up fashion, the QRT of \( SC_{A^\beta} \) can be obtained. Since \( \beta \) has checked its current state and obtained its mode mapping results in Phase 1, \( \beta.sct = \beta.mmt = 0 \) in Phase 2. Let \( T_2 \) denote the duration of Phase 2. If \( \tilde{G}_\beta = \emptyset \), then

\[ T_2 = \max_{c_o \in SC_{A^\beta} \cap \tilde{G}_\beta} \{ \beta.t_{msq} + c_o.qrt + \beta.t_{ok} \} \quad (6.7) \]

If \( \tilde{G}_\beta \neq \emptyset \), then

\[ T_2 = \max_{c_j \in SC_{A^\beta} \cap \tilde{H}_\beta, c_o \in SC_{A^\beta} \cap \tilde{G}_\beta} \{ \beta.t_{msq} + c_j.qrt + \beta.t_{ok}, AE_{\vec{G}_i} + \beta.t_{msq} + c_o.qrt + \beta.t_{ok} \} \]

\[ \text{if } \tilde{G}_\beta \neq \emptyset \]  

\[ (6.8) \]

6.4 Phase 3—Mode-switch execution

The timing analysis in Phase 3 resembles Phase 2. The MSDM \( \beta \) propagates an \( msi^k \) which follows the propagation trace of the \( msq^k \) in Phase 2. The \( msc^k \) sent by each Type A component after reconfiguration is comparable to the \( msok^k \) sent in reply to an \( msq^k \). Both mode mapping and atomic execution can be ignored in Phase 3. Whenever a component propagates an \( msi^k \), it can reuse the mode mapping results obtained in Phase 1 or Phase 2. The propagation of an \( msi^k \) is free from the obstruction of atomic execution which has already been handled in Phase 2.

Figure 6.4 illustrates the behavior of a composite component in Phase 3 based on the same example in Figure 6.3. After receiving an \( msi^k \), \( a \) propagates the \( msi^k \) to its Type A subcomponents \( b, c, \) and \( e \), and then starts its reconfiguration. The mode switches of \( b, c, \) and \( e \) take \( b.mst, c.mst, \) and \( e.mst \).
6.4 Phase 3—Mode-switch execution

The timing analysis in Phase 3 resembles Phase 2. The MSDM \( \beta \) propagates an \( msq_k \) which follows the propagation trace of the \( msq_k \) in Phase 2. The \( msc_k \) sent by each Type A component after reconfiguration is comparable to the \( msok_k \) sent in reply to an \( msq_k \). Both mode mapping and atomic execution can be ignored in Phase 3. Whenever a component propagates an \( msi_k \), it can reuse the mode mapping results obtained in Phase 1 or Phase 2. The propagation of an \( msi_k \) is free from the obstruction of atomic execution which has already been handled in Phase 2.

Figure 6.4 illustrates the behavior of a composite component in Phase 3 based on the same example in Figure 6.3. After receiving an \( msi_k \), \( a \) propagates the \( msi_k \) to its Type A subcomponents \( b, c, \) and \( e \), and then starts its reconfiguration. The mode switches of \( b, c, \) and \( e \) take \( b.mst \), \( c.mst \), and \( e.mst \) time units, respectively. The mode-switch time of \( a \) is actually the longest path among four branches: the reconfiguration of \( a \) and the time required to receive an \( msc_k \) from each Type A subcomponent. More precisely,

\[
a.mst = \max\{ a.rc, a.t_{msi} + b.mst + a.t_{msc}, a.mst + a.t_{msc}, \\
a.t_{msi} + e.mst + a.t_{msc}\}
\]  

(6.9)

![Figure 6.4: Mode-switch timing analysis—Phase 3]

Generalizing (6.9), for each \( c_i \in PC \) which receives an \( msi^k \),

\[
c_i.mst = c_i.rc
\]  

(6.10)

For each \( c_i \in CC \) which receives an \( msi^k \), the mode-switch time of \( c_i \) depends on the mode-switch times of \( SC^A_{c_i}(k) \):

\[
c_i.mst = \max_{c_o \in SC^A_{c_i}(k)} \{ c_i.rc, c_i.t_{msi} + c_o.mst + c_i.t_{msc} \}
\]  

(6.11)
By composing the mode-switch time of each Type A component in the bottom-up fashion, the mode-switch time of each component in \( SC_A^k(k) \) can be obtained. Let \( T_3 \) denote the duration of Phase 3. The calculation of \( T_3 \) depends on two conditions: (1) If \( T^k_\beta = B \), \( \beta \) will not reconfigure itself; (2) If \( T^k_\beta = A \), \( \beta \) must be Top and will reconfigure itself. If \( T^k_\beta = B \),

\[
T_3 = \max_{c_o \in SC^k_\beta(k)} \{ \beta.msi + c_o.mst + \beta.msc \} \tag{6.12}
\]

If \( T^k_\beta = A \),

\[
T_3 = \max_{c_o \in SC^k_\beta(k)} \{ \beta.rct, \beta.msi + c_o.mst + \beta.msc \} \tag{6.13}
\]

Let \( T \) denote the complete mode-switch time of a scenario. Since the three identified phases are sequential without mutual interference, \( T \) is simply the summation of the time spans of all three phases:

\[
T = T_1 + T_2 + T_3 \tag{6.14}
\]

### 6.5 Demonstration of the timing analysis

In this section we demonstrate our mode-switch timing analysis by the example in Figure 4.4 in Section 4.2 of Chapter 4. An MSS \( e \) triggers a non-emergency scenario \( k \), with \( a \) as the MSDM of \( k \) and \( a, c, d, e \) as the Type A components. The complete mode-switch process for \( k \) has been depicted in Figure 6.1. For timing analysis purpose, we introduce various timing factors assigned in Table 6.1, including the MSDT of \( e \), the MMT of composite components \( a \) and \( c \), and the SCT and RCT of each Type A component. The transmission time of a primitive is constantly 1 to simplify the demonstration. Moreover, \( c \) is an AEG with the worst-case atomic execution time \( AE_c = 6 \). The atomic execution within \( c \) derives from its subcomponents, as \( e \) contributes to 4 and \( f \) contributes to 2 in the worst case. A lucid illustration of the complete mode-switch process with all timing factors is provided in Figure 6.5 which highlights the values of these timing factors and indicates that the total mode-switch time is 36.

Our mode-switch timing analysis is performed by dint of the computation of the time spans from Phase 1 to Phase 3. The time span in Phase 1, \( T_1 \), can be calculated with ease. Since \( \alpha = e, \beta = a, \) and \( C^a_e = \{ c \} \), by (6.2),
### 6.5 Demonstration of the timing analysis

<table>
<thead>
<tr>
<th>Component</th>
<th>MSDT</th>
<th>MMT</th>
<th>SCT</th>
<th>RCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>–</td>
<td>3</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>$c$</td>
<td>–</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>$d$</td>
<td>–</td>
<td>–</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>$e$</td>
<td>3</td>
<td>–</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 6.1: Timing factors assignment

\[
T_1 = e.msd + (c.msr + c.mmt + c.sct) + (a.tsr + a.mmt + a.sct) \\
= 3 + (1 + 2 + 1) + (1 + 3 + 2) = 13
\]  

(6.15)

In order to calculate $T_2$ in Phase 2, the first step is to derive the QRT of all Type A primitive components. By (6.4),

\[
d.qrt = d.sct = 3
\]  

(6.16)

\[
e.qrt = e.sct = 3
\]  

(6.17)

Given $e.qrt$, the QRT of $c$ can be worked out accordingly. Since $c$ has been identified as an AEG by $a$, there is no need for $c$ to consider atomic execution among its subcomponents. By (6.5),

\[
c.qrt = c.sct + c.mmt + (c.tsq + e.qrt + c.tok)
\]  

(6.18)

Note that $c.mmt = 0$ in (6.18) according to the annotation of (6.5). The reason is that $C_e^a = \{c\}$, which means that $c$ has obtained its mode mapping results based on the $msr^k$ in Phase 1. Hence the mode mapping results can be reused in Phase 2. Then,

\[
c.qrt = 1 + 0 + (1 + 3 + 1) = 6
\]  

(6.19)

Both $c.qrt$ and $d.qrt$ are used for calculating $T_2$ by (6.8). The AEG $c$ can be specified as $\bar{G}_a = \{G_1\}$ and $G_1 = \{c\}$, with $AE_{\bar{G}_1} = 6$. Meanwhile, $\mathcal{H}_a = \{b, d\}$ and $SC_a^A(k) = \{c, d\}$. This gives $SC_a^A(k) \cap \mathcal{H}_a = \{d\}$ and $SC_a^A(k) \cap G_1 = \{c\}$. Integrating these results into (6.8), we have
\[ T_2 = \max\{a.t_{msq} + d.qrt + a.t_{ok}, AE_G + a.t_{msq} + c.qrt + a.t_{ok}\} \]
\[ = \max\{1 + 3 + 1, 6 + 1 + 6 + 1\} = 14 \] (6.20)

The calculation of \( T_3 \) in Phase 3 starts with the mode-switch time (MST).
of Type A primitive components by (6.10):

\[ d.mst = d.rct = 7 \]  
\[ e.mst = e.rct = 3 \]

Once \( e.mst \) is known, the MST of \( c \) can be computed by (6.11):

\[ c.mst = \max\{c.rct, c.t_{msi} + e.mst + c.t_{msc}\} = \max\{4, 1 + 3 + 1\} = 5 \]  

(6.23)

Based on \( c.mst \) and \( d.mst \), \( T_3 \) is computed by (6.13):

\[
T_3 = \max\{a.rct, a.t_{msi} + c.mst + a.t_{msc}, a.t_{msi} + d.mst + a.t_{msc}\} \\
= \max\{8, 1 + 5 + 1, 1 + 7 + 1\} = 9
\]

(6.24)

Finally, applying (6.14), the total mode-switch time \( T \) is:

\[ T = T_1 + T_2 + T_3 = 13 + 14 + 9 = 36 \]

(6.25)

This result is fully consistent with the mode-switch time indicated in Figure 6.5.

### 6.6 Deriving atomic execution time

Among the timing factors defined in Chapter 6.1, \( AE_{G_i} \) is a special parameter which depends on the execution time of each component in the AEG \( G_i \), the connections of the components in \( G_i \), and the data flow in \( G_i \). As a supplement to our mode-switch timing analysis, we propose a model-checking approach to derive \( AE_{G_i} \). The basic idea is to model the behavior of an AEG \( G_i \) as timed automata in the model checker UPPAAL [9, 78] that is used for the verification of our MSRM, and then derive \( AE_{G_i} \) by verifying certain properties.

#### 6.6.1 The AEG model

The modeling of an AEG requires a clear specification of its execution semantics. Stated in Section 4.4 of Chapter 4, we focus on the atomic execution in
pipe-and-filter multi-mode systems, whose execution semantics is similar to the pure data flow component described in [79]. Each AEG of such a system can also be considered as a smaller system with the same execution semantics:

- A primitive component has data going through all its input and output ports. Input data has to be available at all input ports before processing can start, and output data must be sent via all output ports. Any new data arriving at an input port is first queued in a corresponding input buffer. The data at the head of the buffer is immediately processed when the component completes its current data processing.

- A composite component does not buffer its input data. It forwards any input data to its subcomponents and forwards any output data from its subcomponents via its output port(s).

We also make the following restrictions on an AEG composed by components with the execution semantics above:

- An AEG $G_i$ should have only one input port. The MSQ delaying rule (Definition 17 in Chapter 4) states that the parent of $G_i$ must freeze the input of $G_i$ before checking its execution status. If $G_i$ has multiple inputs, the freezing of its inputs must be performed at a proper instant such that the completion of the ongoing atomic execution in $G_i$ is guaranteed. The handling of this is outside the scope of the thesis.

- There is no cyclic connection, i.e., feedback loop, in an AEG.

- Data transmission between different components within an AEG is instantaneous. An AEG is most likely to reside in the same physical (sub)system, therefore, compared with component execution time, data transmission time should be marginal.

As an example of such an AEG fulfilling these restrictions, Figure 6.6 depicts an AEG $G_i$ with one input $in$ and three outputs $out1$, $out2$, and $out3$. This AEG encloses six primitive components $a$-$f$, whose ports are marked in red in Figure 6.6. Some of these components may belong to a composite component which is also included in $G_i$. All composite components within $G_i$ are excluded here because they contribute nothing to $AE_{G_i}$ as per the execution semantics assumed by us. Deactivated components in $G_i$ are also ignored for the same reason. We assume that the data processing time of each component $c_i$ in $G_i$ is bounded by a timing interval $[C_{c_i}^{min}, C_{c_i}^{max}]$, and that the input data
rate of the AEG is within the interval \([R_{min}, R_{max}]\). To ensure that \(AE_{G_i}\) is bounded and that our calculations terminate, we enforce a maximum number of data elements in \(G_i\). Depending on the input data rate and data processing times of different components, this bound may or may not be reached. In fact, the bound could be used as a modeling artifact, but could also be a mechanism in a real system. The timing factors and their values in \(G_i\) are as follows:

- Input data rate \(R = [7, 8]\).
- Data processing time \(C\) of components \(a-f\): \(C_a \in (4, 5]\), \(C_b \in [7, 8]\), \(C_c \in (6, 7]\), \(C_d \in (5, 6]\), \(C_e = 5\), \(C_f \in (7, 8]\).
- Maximum number of data elements in \(G_i\) \(N = 5\).

![Figure 6.6: An AEG \(G_i\) with multiple components](image)

### 6.6.2 UPPAAL modeling

Using UPPAAL, we first model the execution and mode-switch behavior of \(G_i\) and then derive \(AE_{G_i}\) via property verification. No matter how complex an AEG \(G_i\) is, we can always divide its UPPAAL model into four parts:

1. **Data source**: generates input data at a flexible rate.
2. **AEG**: receives data from **Data source**, processes the data, and deposits the results at its output port(s). Furthermore, it ensures that the number of data elements \(n\) in the AEG is within the bound \(N\). **Data source**
is turned off when \( n = N \). If no mode-switch scenario arrives during the atomic execution of \( G_i \), Data source is turned on again when \( n \) decreases. When \( G_i \) receives an MSQ, Data source will be turned off to simulate the freezing of the input of \( G_i \) by its parent. After that, \( \text{AE}_G \) boils down to the maximal data processing time to reach \( n = 0 \).

3. **Data forwarder**: forwards data between components without the sender knowing the identity of the receiver. This simplifies the modeling, since when some connections are changed, or a component is removed or added, only Data forwarder needs to be updated.

4. **Primitive components**: modeled by a UPPAAL automaton for each of them. Though the number of components can be arbitrary, a parameterized generic model that is instantiated for every component can be used.

Depicted in Figure 6.7(a), the UPPAAL model of Data source is a timed automaton consisting of only a single state Running. The status of Data source is reflected in a boolean variable DSstatus which is set to true when Data source is turned on, and set to false when Data source is turned off. The timing constraint, i.e. the input data rate \( R \in [7, 8] \), is represented by the boolean expression \( x \geq 7 \) as the guard of transitions 1 and 2, together with the invariant \( x \leq 8 \) of State Running, where \( x \) is a local clock. When the timing constraint is satisfied and DSstatus is true, Data source will send new data to \( G_i \) via the channel newData.

![Figure 6.7: UPPAAL models: Data source and data forwarder](image)

Another UPPAAL model depicted in Figure 6.7(b) describes the behavior of Data forwarder. Whenever a component among a-f has new data produced
at an output port, Transition 1 of the model in Figure 6.7(b) will be fired, as **Data forwarder** receives the data via the channel *dataOut*. Meanwhile, a function *findNext()* finds the connection between this output port and the input port of the receiver. Subsequently, **Data forwarder** forwards the data to the correct port via the channel *dataIn* in Transition 2.

The UPPAAL model of $G_i$ in Figure 6.6 is presented in Figure 6.8. The initial state of $G_i$ is *Waiting*, where no data is within $G_i$, thus without ongoing atomic execution. When new data enters $G_i$ by Transition 1, $G_i$ will forward this data to its enclosed components through the *dataOut* channel (Transition 2 synchronized with Transition 1 in Figure 6.7(b)) and enters State *Processing*, where at least one data element is in $G_i$. The number of data elements within $G_i$ is counted by *dataCounter*. In the function *shutDS()* of Transition 2, $G_i$ compares *dataCounter* with the threshold $N$. If the threshold is reached, **Data source** will be turned off in *shutDS()* which sets *DSstatus* to false. When *dataCounter* decreases and no *MSQ* arrives, **Data source** is turned on by the function *dataNControl()* of Transition 5. An incoming *MSQ* is modeled by the channel *MSQ?* of Transition 4. The AEG $G_i$ stays in State *Processing* until new data arrives (Transition 6 in Figure 6.8) from **Data source** which is turned on by Transition 5, or until there is no data being processed within $G_i$, which is indicated by Transition 3. The atomic execution time of $G_i$ corresponds to the time that elapses in State *Processing*. It can be observed that a clock $z$ is reset to 0 by Transition 2. Hence the worst-case atomic execution time $AE_{G_i}$ is essentially the maximal possible value of $z$ in State *Processing*.

![Figure 6.8: UPPAAL model: AEG $G_i$](image)

The primitive components in $G_i$ can be categorized into four types: (1) components with single input and output; (2) components with single input...
and multiple outputs; (3) components with multiple inputs and single output; (4) components with multiple inputs and outputs. Figure 6.9 illustrates the UPPAAL model of Component $f$, which has multiple inputs and outputs. Component $f$ receives data through the channel $dataIn$ and sends output data through the channel $dataOut$, with both channels synchronized with Data forwarder. In the initial state nonProcessing, $f$ recognizes new data by the guard $target == fi1 || target == fi2$ of Transition 1 where $fi1$ and $fi2$, shown in Figure 6.6, are its input ports. When all input buffers are non-empty, a boolean variable $readyToProcess$ is set to true and $f$ will switch to State Processing by an urgent channel $Go$ of Transition 2. Data is processed by the function $processData()$, representing the mode-specific behavior of a primitive component. The invariant $x <= 8$ of State Processing and the guard $x > 7$ of Transition 4 jointly define the interval of its data processing time, i.e., $C_f \in (7, 8]$. After processing the data, $f$ immediately sends its output data via all its output ports. This is modeled by the sequential and atomic output data generation from its output ports (transitions 5 and 6). A variable $outputCounter$ records how many output ports have sent out the data. The atomicity of sending the output data is ensured by two committed states Temp1 and Temp2. After sending all the output data, $f$ goes back to State nonProcessing by Transition 7 and checks its buffer status again.

Figure 6.9: UPPAAL model: Component $f$

Component $f$ belongs to Type 4 due to its multiple input ports and output
ports. Similarly, components of the other three types can be modeled with the same pattern. Figures 6.10-6.12 illustrate the UPPAAL models of $a$ (Type 2), $b$ (Type 1) and $e$ (Type 3), respectively. Despite the variation of some parameters, these components share the same model structure. If a component has only one output port, the model can be simplified by removing State $Temp2$ and outputCounter. The complete UPPAAL models of the example in Figure 6.6 is available in [50].

![Figure 6.10: UPPAAL model: Component $a$](image)

### 6.6.3 Verification

Based on the UPPAAL model described in the previous subsection, some interesting results including $AE_{G_i}$ can be obtained from the verification of the following properties formulated in the UPPAAL query language:

**P1.** $A[]$ not deadlock: the model is deadlock free.

**P2.** $\text{sup}(AEG.\text{Processing}):AEG.z$: returns the maximal value of the clock $z$ of AEG in State Processing, viz., $AE_{G_i}$. The operator $sup$ is used to find the maximal value of a variable or clock.

**P3.** $E<> AEG.\text{Processing} \&\& AEG.z == AE_{G_i}$: there exists an execution trace in which the clock $z$ reaches $AE_{G_i}$ when AEG is in State Processing.
In addition to $AE_{G_i}$, UPPAAL has a "Diagnostic Trace" function which automatically returns the worst-case execution trace leading to $AE_{G_i}$.

Figure 6.11: UPPAAL model: Component $b$

Figure 6.12: UPPAAL model: Component $e$
6.6 Deriving atomic execution time

**P4.** \textit{sup:}\texttt{dataCounter}: returns the maximal number of data items that can be simultaneously processed in $G_i$. If $N$ is only a modeling artifact, then for the validity of the calculated $AE_{G_i}$, this value must be less than $N$. In other cases, validity requires a mechanism in the deployed system that keeps $n$ within the bound $N$.

**P5.** \textit{sup:}\texttt{Component.bufferN[Index]}: returns the maximal number of elements in an input buffer of a component.

Our verification result is positive in the sense that all these properties have been satisfied. When $R \in [7, 8]$, according to the verification result of the property P2, the worst-case atomic execution time $AE_{G_i}$ is 40. Moreover, the verification of P4 shows that the maximal number of data items in $G_i$ is 5, meaning that the threshold $N = 5$ can be reached for $R \in [7, 8]$.

The verification time of these properties is heavily impacted by $R$ and $C_{c_i}$ ($c_i$ is a primitive component activated in $G_i$) which contribute to variations in the number and length of executions to find the worst-case scenario. We repeated the verification of the same set of properties for different data rates. The most important results for P1-P4 are summarized in Table 6.2, which includes the verification times of P1-P3, $AE_{G_i}$, and the maximal number of data elements in $G_i$. Apparently, the verification overhead grows with the increase of the input data rate, which potentially induces more buffered data in $G_i$.

<table>
<thead>
<tr>
<th>Property/Value</th>
<th>$R \in [6, 8]$</th>
<th>$R \in [7, 8]$</th>
<th>$R \in [8, 10]$</th>
<th>$R \in [10, 12]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>No deadlock (P1)</td>
<td>28.64s</td>
<td>5.617s</td>
<td>0.139s</td>
<td>0.108s</td>
</tr>
<tr>
<td>Deriving $AE_{G_i}$ (P2)</td>
<td>45.667s</td>
<td>4.36s</td>
<td>0.1s</td>
<td>0.069s</td>
</tr>
<tr>
<td>$AE_{G_i}$ (P2)</td>
<td>40</td>
<td>40</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Worst-case scenario (P3)</td>
<td>36.716s</td>
<td>4.576s</td>
<td>0.013s</td>
<td>0.016s</td>
</tr>
<tr>
<td>Maximal $n$ (P4)</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 6.2: Verification results for different data rates

An interesting subsidiary result is that we can use the last property P5 to obtain the maximal buffer usage (i.e. required buffer sizes) for the input buffers of each component (see Table 6.3). For instance, the maximal usage of the

\footnote{The verification was performed on MacBook Pro, with 2.66GHz Intel Core 2 Duo CPU and 8GB 1067 MHz DDR3 memory.}
buffer associated with the port $ei2$ in Figure 6.6 is 1 when $R \in [10, 12]$, 2 when $R \in [8, 10]$, and 4 when $R \in [6, 8]$ or $R \in [7, 8]$.

<table>
<thead>
<tr>
<th>Buffer index</th>
<th>$R \in [6, 8]$</th>
<th>$R \in [7, 8]$</th>
<th>$R \in [8, 10]$</th>
<th>$R \in [10, 12]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ai1$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$bi1$</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$ci1$</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$di1$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$ei1$</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$ei2$</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$ei3$</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$fi1$</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$fi2$</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.3: Maximal buffer usage for different data rates

**6.6.4 Generalization**

Apart from the input data rate $R$ of an AEG $G_i$, the verification time of our UPPAAL models also depends on the number of activated primitive components in $G_i$, the connections between these components, the maximum number of data elements allowed in $G_i$, and the execution time of each component in $G_i$. Regardless of the verification time, the way that we model an AEG does not change. Although we have only demonstrated how to derive $AE_{G_i}$ in a simple example, our UPPAAL models are generic. We conjecture that for any AEG with the same execution semantics as the example in Figure 6.6, we are able to make transformation rules, based on which the corresponding UPPAAL models can be automatically generated. Since the verification is based on generating and exploring the global state space of our models, it is subject to state explosion as the complexity of an AEG grows. However, we do not expect this to be a limitation in practice, since the complexity of a typical AEG should be rather low.

**6.7 Summary**

Multi-mode systems are expected to meet certain temporal requirements while performing a mode switch. In this chapter, a mode-switch timing analysis is
6.7 Summary

provided for the MSRM of our MSL, assuming the triggering of a single non-
emergency scenario with atomic execution. The purpose of the timing analysis
is to calculate the worst-case mode-switch time of a component-based system
running our MSRM. We divide the entire mode-switch process into three non-
overlapping and continuous phases: (1) the identification of the MSDM of a
scenario; (2) the request for the current states of Type A components; and (3)
mode-switch execution. The time spent in each phase is computed independ-
dently. The total mode-switch time is obtained by summing the time spans
of all three phases. Phase 2 may be prolonged by the atomic execution of an
Atomic Execution Group (AEG). We propose a model-checking approach to
derive the worst-case atomic execution time of an AEG. The behavior of an
AEG is modeled using the model checking tool UPPAAL. Based on the UP-
PAAL model, the verification of certain properties not only tells us the worst-
case atomic execution time of an AEG, but also finds a possible execution trace
leading to the worst-case scenario. In addition, our approach is able to analyze
the buffer usage of each input port of a component.
Chapter 7

Mode switch for the ProCom component model

Our MSL is mainly dedicated to reuse of multi-mode software components and mode-switch handling, thus independent of the choice of component models [80]. However, a concrete component model, with rules for constructing individual components and rules for composing components, is necessary to enable actual development of multi-mode systems. In lieu of the proposal of a new concrete component model tailored to MSL, a more efficient alternative is to integrate MSL into an existing component model. There are a multitude of component models [31, 67] targeting various application domains. In this chapter, we show how MSL can be integrated into ProCom [110], a component model for distributed real-time embedded systems, with a particular focus on vehicular, automation and telecommunication applications. The original ProCom component model has no support for multi-mode systems. Despite such limitation, with only slight extension of ProCom, we are able to introduce reusable multi-mode components in ProCom and run the MSRM of MSL for the collaborative mode-switch handling of ProCom components. Our approach is labor-saving in the sense that it is amenable to the automatic generation of mode-related elements in line with a given mode specification.
Chapter 7

Mode switch for the ProCom component model

Our MSL is mainly dedicated to reuse of multi-mode software components and mode-switch handling, thus independent of the choice of component models [80]. However, a concrete component model, with rules for constructing individual components and rules for composing components, is necessary to enable actual development of multi-mode systems. In lieu of the proposal of a new concrete component model tailored to MSL, a more efficient alternative is to integrate MSL into an existing component model. There are a multitude of component models [31, 67] targeting various application domains. In this chapter, we show how MSL can be integrated into ProCom [110], a component model for distributed real-time embedded systems, with a particular focus on vehicular, automation and telecommunication applications. The original ProCom component model has no support for multi-mode systems. Despite such limitation, with only slight extension of ProCom, we are able to introduce reusable multi-mode components in ProCom and run the MSRM of MSL for the collaborative mode-switch handling of ProCom components. Our approach is labor-saving in the sense that it is amenable to the automatic generation of mode-related elements in line with a given mode specification.
7.1 The ProCom component model

Taking the advantage of both CBSE [30, 115] and Model-Driven Engineering (MDE) [13], ProCom allows the modeling of a system using reusable software components at different nested levels. Automatic code generation from the system model is supported. The system model also enables early analysis and verification before the system is fully implemented. Compared with other component models, the most distinctive feature of ProCom is its two-layer structure: ProSave—the lower layer, and ProSys—the higher layer. With divergent concerns, the two layers are complementary to each other, enabling the modeling of a system at different levels of granularity. The ProSave layer models subsystems allocated to a single physical node, while the ProSys layer models distributed subsystems deployed on separate nodes.

7.1.1 The ProSave layer

The ProSave layer adopts the pipes and filters architectural style [45] and has explicit separation between control flow and data flow. Control flow controls the activation of components while data flow transmits data from one component to another. A component belonging to this layer is called a ProSave component. A ProSave component provides one or more services, each of which realizes a particular functionality. Each service has a single input port group, and one or multiple output port groups. A port group consists of a trigger port and one or more data ports, with the trigger port dedicated to control flow and the data ports dedicated to data flow.

The initial state of each service belonging to a ProSave component is passive. The service can receive trigger signals and data to its input ports, yet without execution which requires external activation. A service becomes active when a trigger signal arrives at its input trigger port. An activated service strictly follows a 3-phase atomic execution pattern: (1) Phase 1: reads data from its input data port(s); (2) Phase 2: performs computation based on the input data; (3) Phase 3: produces the computation results at its output data port(s) and then activates the corresponding output trigger port(s). After the 3-phase execution, a service becomes passive again. Different services of the same ProSave component are activated independently with concurrent execution. However, it is possible to share data among these services.

Figure 7.1 depicts a ProSave component with two services $S_1$ and $S_2$. Service $S_1$ has an input port group (consisting of an input trigger port and an input data port) and an output port group (consisting of an output trigger port and
two output data ports). Service $S_2$ has an input port group and two output port groups, with each port group consisting of a trigger port and a data port.

![ProSave component](image)

Figure 7.1: A ProSave component

The communication between ProSave components is achieved by establishing a control flow via their trigger ports or a data flow via their data ports. Both control flow and data flow may exist from one ProSave component to another ProSave component. As an output trigger port of a ProSave component $c_i$ is connected to an input trigger port of another ProSave component $c_j$, a control flow from $c_i$ to $c_j$ is established. Likewise, as an output data port of $c_i$ is connected to an input port of $c_j$, a data flow from $c_i$ to $c_j$ is established. Both trigger ports and data ports must conform to this one-to-one mapping, i.e., an output port of $c_i$ must be connected to a single input port of $c_j$ of the same type (trigger port or data port). Notwithstanding, ProCom includes a couple of connectors for more advanced connections such as one-to-many mapping. Figure 7.2 lists the most typical connectors:

- **Control Or**: It has multiple input trigger ports and one output trigger port. Its output trigger port is activated when any one of its input trigger ports is activated.

- **Control Join**: It has multiple input trigger ports and one output trigger port. Its output trigger port is activated only when all its input trigger ports are activated. A simplified graphical notation of a Control Join connector is a small circle.

- **Control Fork**: It has one input trigger port and multiple output trigger ports. When its input trigger port is activated, all its output trigger ports will be activated. A simplified graphical notation of a Control Fork connector is a thick dot.
- **Data Or**: It has multiple input data ports and one output data port. The data arriving at any one of its input data ports is forwarded to its output data port.

- **Data Fork**: It has one input data port and multiple output data ports. The data arriving at its input data port will be duplicated and produced at all its output data ports. Just like Control Fork, it can also be represented by a thick dot graphically.

- **Selection**: It has an input trigger port, at least one input data port, and multiple output trigger ports. When its input trigger port is activated, it will activate exactly one of its output trigger ports according to the data written to its input data port(s).

![Figure 7.2: Typical connectors in ProSave](image)

Figure 7.2: Typical connectors in ProSave

Figure 7.3 illustrates four ProSave components connected via three connectors. The output trigger port of *a* is connected to a Selection connector. Depending on the data from the first output data port of *a*, Selection may activate *b* or *c*. The connector Control Or allows *d* to be activated by either *b* or *c*. Similarly, the connector Data Or forwards the output data of *b* or *c* to *d*.

![Figure 7.3: The connections between ProSave components](image)
The ProSave layer supports the hierarchical composition of ProSave components. The ports of a composite ProSave component \( c_i \) is internally delegated to the ports of its subcomponents. An input trigger/data port of \( c_i \) is delegated to an input trigger/data port of one of its subcomponents; an output trigger/data port of \( c_i \) is delegated to an output trigger/data port of one of its subcomponents.

### 7.1.2 The ProSys layer

The ProSys layer, which models distributed subsystems on top of the ProSave layer, differs from the ProSave layer in several aspects. The concept of service in ProSave does not apply to the ProSys layer. A component belonging to this layer is called a ProSys component. A ProSys component has a number of input and output message ports which replace the port groups of a ProSave component. Unlike passive ProSave components, each ProSys component is active, as it has its own threads whose execution requires no external activation. Hence, it is possible to have concurrent execution in different ProSys components.

Figure 7.4 depicts a ProSys component with one input message port and two output message ports. The communication between ProSys components is realized by asynchronous message passing. A message is sent from an output message port of a ProSys component, to be received at an input message port of another ProSys component through a message channel. A message channel can be associated with multiple input and output message ports, achieving many-to-many communication. No connectors are used in the ProSys layer. Figure 7.5 shows the communication between a ProSys component \( a \) and two other ProSys components \( b \) and \( c \) through a message channel. Every time \( a \) sends a message from its output message port, the message will be received by both \( b \) and \( c \).

![Figure 7.4: A ProSys component](image)

The hierarchical composition of ProSys components is akin to the ProSave
7.1.3 The integration of ProSave and ProSys

The integration of ProSys and ProSave is realized by building a ProSys component with ProSave components. In order to map the pipes and filters architecture in ProSave to message passing in ProSys, a message port is internally treated as a pair of a trigger port and a data port. When a message arrives at an input message port of a ProSys component composed by ProSave components, the message data will be forwarded to the internal data port of this message port and the corresponding internal trigger port will be activated. Conversely, when the internal trigger port of an output message port is activated, the data of the corresponding internal data port will be sent out from the message port as an output message.

Figure 7.6 demonstrates the integration of ProSave and ProSys with a ProSys component $c_i$ composed by three ProSave components $c_{ij}^1$, $c_{ij}^2$, and $c_{ij}^3$. Apparently, all the message ports of $c_i$ are internally treated as a port group with a trigger port and a data port. In addition, a special connector *Clock* can be used for the periodic activation of ProSave components composing a ProSys component. The example in Figure 7.6 contains two *Clock* connectors with different activation periods, one for the periodic triggering of the service $S_2$ of $c_{ij}^1$ at a frequency of 50Hz, the other for the periodic triggering of an output message port of $c_i$ at a frequency of 10Hz.
7.2 Multi-mode ProCom components

The original ProCom component model is devoid of mode specification. As the first step of integrating MSL in ProCom, we define multi-mode ProCom components without additional extension while conforming to our mode-aware component model. Since ProCom distinguishes ProSave and ProSys, multi-mode ProSave and ProSys components will be introduced separately.

In the ProSave layer, a dedicated service $S_{mode}$ is attached to each multi-mode ProSave component for its mode-switch handling. In this way, we guarantee a clear separation between the mode-switch handling and the functional behavior of each component. The service $S_{mode}$ includes the mode specification of a component, the configuration for each mode, mode mapping and the MSRM. Furthermore, $S_{mode}$ also has dedicated mode-switch ports that correspond to $p_{in}^{MS}$ and $p_{in}^{MS}$ defined in the mode-aware component model. The service $S_{mode}$ consists of an input port group and an output port group. The input port group comprises an input trigger port $p_{i}^{ms}$ and an input data port $p_{i}^{ms}$, while the output port group comprises an output trigger port $p_{o}^{ms}$ and an output data port $p_{o}^{ms}$. Figure 7.7(a) shows a multi-mode ProSave component $c_i$ with two services. The mode switch of $c_i$ is handled by the lower service $S_{mode}$, where the dedicated mode-switch ports are highlighted in purple.

In the ProSys layer, no concept of service exists. Hence we use a dedicated internal thread to handle the mode switch of a multi-mode ProSys component. This dedicated thread is separated from the functional behavior of the component. Shown in Figure 7.7(b), a multi-mode ProSys component has an input message port $p_{i}^{ms}$ and an output message port $p_{o}^{ms}$ as its dedicated mode-switch ports. Both ports are highlighted in purple in the figure.
7.3 Handling mode switch in ProCom

The previous section states that the mode switch of a multi-mode ProSave component is handled by a dedicated service $S_{mode}$ while the mode switch of a multi-mode ProSys component is handled by a dedicated internal thread. According to MSL, the mode switch of each multi-mode component is handled by its MSRM. Now the question is: how can the MSRM be implemented in the dedicated service $S_{mode}$ and the dedicated internal thread? For primitive multi-mode ProCom components, its MSRM is simply implemented by code; for composite multi-mode ProCom components though, our solution is to implement the MSRM in a single dedicated subcomponent or a pair of dedicated subcomponents. To simplify the presentation, the rest of this chapter uses ProSave or ProSys components to imply multi-mode ProCom components unless stated otherwise.

7.3.1 Handling mode switch in ProSave

Since a composite ProSave component is only a composition of other ProSave components, there is no concrete functional behavior in a composite ProSave component. A reasonable strategy to handle its mode switch is to implement its MSRM in a dedicated subcomponent.

For a composite component $c_i$, which is either a ProSave component or a ProSys component composed by ProSave components, we introduce a pair of primitive ProSave components: $MSL^A_{c_i}$ and $MSL^B_{c_i}$ as dedicated subcomponents of $c_i$ for its mode-switch handling. The pair of $MSL^A_{c_i}$ and $MSL^B_{c_i}$ jointly

![Figure 7.7: Multi-mode ProCom components](image-url)
interact with the $S_{\text{mode}}$ service of each subcomponent of $c_i$.

Let $c_i,p$ denote the port $p$ of component $c_i$. Also, let $SC_{c_i} = \{c_j^1, c_j^2, \ldots, c_j^n\}$ ($n \in \mathbb{N}$) denote the set of subcomponents of $c_i$, excluding $MSL_{c_i}^A$ and $MSL_{c_i}^B$. Figure 7.8 describes the ports of $MSL_{c_i}^A$ and $MSL_{c_i}^B$, both of which are synchronized with each other via their synchronization ports $p_{i}^{\text{sync}}$ and $p_{o}^{\text{sync}}$. Component $MSL_{c_i}^A$ has a single service with an input port group and an output port group. Apart from the synchronization ports $p_{i}^{\text{sync}}$ and $p_{o}^{\text{sync}}$, these port groups consist of the following ports:

- $p_{i}^{t}$: an input trigger port whose activation makes $MSL_{c_i}^A$ active.
- $p_{i}^{msx}$: an input data port for receiving a downstream primitive from the parent of $c_i$.
- $p_{o}^{t}$: an output trigger port activated after $MSL_{c_i}^A$ completes its current instance of execution.
- $P_{o}^{msx} = \{p_{o}^1, p_{o}^2, \ldots, p_{o}^n\}$ ($n = |SC_{c_i}|$): a set of output data ports for sending a downstream primitive to $SC_{c_i}$.
- $p_{o}^{s}$: an output data port indicating the current mode of $c_i$.

![Figure 7.8: The port definition of $MSL_{c_i}^A$ and $MSL_{c_i}^B$](image)

Note that by abuse of notation, the $i$ of $p_{i}^{t}$, $p_{i}^{msx}$ and $p_{i}^{\text{sync}}$ represents that they are the input ports of $MSL_{c_i}^A$, without no connection with the $i$ of $c_i$.

Similarly, apart from the synchronization ports $p_{i}^{\text{sync}}$ and $p_{o}^{\text{sync}}$, $MSL_{c_i}^B$ also has the following ports:

- $p_{i}^{t}$: an input trigger port whose activation makes $MSL_{c_i}^B$ active.
Chapter 7. Mode switch for the ProCom component model

- $\mathcal{P}_i^{msx} = \{p_i^1, p_i^2, \ldots, p_i^n\} \ (n = |\mathcal{SC}_c|)$: a set of input data ports for receiving an upstream primitive from $\mathcal{SC}_c$.

- $p_i^o$: an output trigger port activated after $\text{MSL}_{c_i}$ completes its current instance of execution.

- $p_i^{o\text{msx}}$: an output data port for sending an upstream primitive to the parent of $c_i$.

The connections around $\text{MSL}_{c_i}^A$ and $\text{MSL}_{c_i}^B$ are illustrated in Figure 7.9. The ports associated with services rather than $S_{\text{mode}}$ of both $c_i$ and $\mathcal{SC}_c$ have been omitted to simplify the view. The pair of $\text{MSL}_{c_i}^A$ and $\text{MSL}_{c_i}^B$ are connected to both $c_i$ and $\mathcal{SC}_c$ via their dedicated mode-switch ports. Their connection with $c_i$ is represented by (1) the control flows from $c_i.p_i^{m\text{st}}$ to $\text{MSL}_{c_i}^A.p_i^l$ and from $\text{MSL}_{c_i}^B.p_i^l$ to $c_i.p_i^{m\text{st}}$, and (2) the data flows from $c_i.p_i^{m\text{s}}$ to $\text{MSL}_{c_i}^A.p_i^{m\text{sx}}$ and from $\text{MSL}_{c_i}^B.p_i^{m\text{sx}}$ to $c_i.p_i^{m\text{s}}$. Their connection with $\mathcal{SC}_c$ is represented by (1) the control flows from $\text{MSL}_{c_i}^A.p_i^o$ to each $e_j^k.p_j^{m\text{st}} \ (k \in [1, n])$ via a Control Fork connector (the thick dot in Figure 7.9) and from each $e_j^k.p_j^{m\text{st}}$ to $\text{MSL}_{c_i}^B.p_i^l$ via a Control Or connector, and (2) the data flows from $\text{MSL}_{c_i}^A.p_i^o$ to each $e_j^k.p_i^{m\text{s}}$ and from each $e_j^k.p_i^{m\text{s}}$ to $\text{MSL}_{c_i}^B.p_i^k$. Moreover, there are two data flows between $\text{MSL}_{c_i}^A$ and $\text{MSL}_{c_i}^B$, one from $\text{MSL}_{c_i}^A.p_i^{s\text{ync}}$ to $\text{MSL}_{c_i}^B.p_i^{s\text{ync}}$ and the other from $\text{MSL}_{c_i}^B.p_i^{s\text{ync}}$ to $\text{MSL}_{c_i}^A.p_i^{s\text{ync}}$, for synchronizing updated mode information.

Based on the connections around the pair of $\text{MSL}_{c_i}^A$ and $\text{MSL}_{c_i}^B$, a mode-related control flow is established within $c_i$: $c_i \rightarrow \text{MSL}_{c_i}^A \rightarrow \mathcal{SC}_c \rightarrow \text{MSL}_{c_i}^B \rightarrow c_i$. Thanks to the Control Or connector between $\mathcal{SC}_c$ and $\text{MSL}_{c_i}^B$, any subcomponent of $c_i$ is able to activate $\text{MSL}_{c_i}^B$ so that $\text{MSL}_{c_i}^B$ may receive an upstream primitive from any subcomponent of $c_i$.

This connection pattern in Figure 7.9 is repeated within all composite ProSave components. For instance, $\forall c_j^k \in \mathcal{SC}_c \ (k \in [1, n])$ and $c_j^k \in \mathcal{CC}$, the internal connections of $c_j^k$ will exhibit the same internal connection pattern as that of $c_i$. Such a connection pattern enables the transmission of both downstream and upstream primitives between a parent and its subcomponents. For instance, a downstream primitive from $c_i$ to $c_j^k$ can be transmitted by the control flow from $\text{MSL}_{c_i}^A.p_i^o$ to $c_j^k.p_j^{m\text{st}}$ together with the data flow from $\text{MSL}_{c_i}^A.p_i^k$ to $c_j^k.p_i^{m\text{s}}$. Once $c_j^k$ receives the primitive, its dedicated subcomponent $\text{MSL}_{c_j}$ is responsible for propagating the primitive further to the Type A subcomponents of $c_j^k$. Conversely, an upstream primitive from $c_j^k$ to $c_i$ can be transmitted
by the control flow from $c_i^k \cdot p_o^{ms}$ to $MSL^B_{c_i} \cdot p_i^l$ together with the data flow from $c_i^k \cdot p_o^{ms}$ to $MSL^B_{c_i} \cdot p_i^k$. If $c_i$ needs to forward the primitive further to its parent $c_l$, $MSL^B_{c_i}$ can send the primitive to $c_i \cdot p_o^{ms}$ and activate $c_i \cdot p_o^{ms}$ such that the primitive is sent out from $c_i \cdot p_o^{ms}$ to $MSL^B_{c_i}$.

Since ProSave components are passive and require external activation, a Clock connector is placed at the top ProSave level, periodically triggering the mode-related control flow through all ProSave components. As primitive ProSave components, $MSL^A_{c_i}$ and $MSL^B_{c_i}$ handle the mode switch of $c_i$ only when they are active, while their activation period is determined by the frequency of the Clock connector.

The reason why we use two such components to implement the MSRM of $c_i$ is attributed to the rigorous execution semantics in the ProSave layer, which prohibits mutual triggering between two neighboring ProSave components. If a single component, e.g. $MSL_{c_i}$, is used instead of $MSL^A_{c_i}$ and $MSL^B_{c_i}$, the mutual triggering between $MSL_{c_i}$ and $SC_{c_i}$ will violate the execution semantics.

Figure 7.9: The connections around $MSL^A_{c_i}$ and $MSL^B_{c_i}$
of ProSave.

To demonstrate how our MSRM is implemented by the pair of $MSL^A_i$ and $MSL^B_i$, algorithms 33 and 34 jointly handle an emergency (mode-switch) scenario $k$ arriving at $c_i \in \mathcal{C}$, where $\mathcal{S}c_{c_i} = \{c_{i1}^j, c_{i2}^j, \ldots, c_{in}^j\} \ (n \in \mathbb{N})$. We assume that there is no concurrent triggering of other non-emergency scenarios to simplify the demonstration. Another assumption is that $c_i$ is not the MSS of $k$. Then $c_i$ can handle an $ems^k$ by applying the EMSP protocol (Definition 26 in Chapter 4) and the mode-switch dependency rule (Definition 15 in Chapter 4).

Since all the ports in Algorithm 33 belong to $MSL^A_i$, each port $MSL^A_i,p$ is simplified as $p$ in Algorithm 33. The same simplification is applied to Algorithm 34.

When $c_i$ receives an $ems^k$ from $P_{c_i}$, an $ems^k(m_{c_{i}^new})$ arrives at $MSL^A_{c_i}.p_i^{\text{msx}}$, triggering the execution of lines 3-14 in Algorithm 33 during the next activation of $MSL^A_{c_i}.p_i^t$. Lines 6-8 in Algorithm 33 is executed as $c_i$ further propagates the $ems^k$ to $\mathcal{S}c_{c_i}^A(k)$. However, each service of a ProSave component is obliged to deposit data to all its output data ports. For each $MSL^A_{c_i}.p_i^d$ where $c_{i}^d \in \mathcal{S}c_{c_i} \setminus \mathcal{S}c_{c_i}^A(k)$, dummy data can be deposited, denoted as $p_i^d := \emptyset$ in Line 10 of Algorithm 33. Shown in Line 13 of Algorithm 33, an $ems^k(m_{c_{i}^new})$ is written at $MSL^A_{c_i}.p_o^{\text{sync}}$ to synchronize $MSL^A_{c_i}$ and $MSL^B_{c_i}$. During the next activation of $MSL^B_{c_i}.p_i^t$, lines 14-25 of Algorithm 34 will be executed. $MSL^B_{c_i}$ executes the same reconfiguration as $MSL^A_{c_i}$. If $\mathcal{S}c_{c_i}^A(k) = \emptyset$, the mode switch of $c_i$ will be completed, as $c_i$ sends an $msc^k$ to $P_{c_i}$, which corresponds to Line 18 of Algorithm 34. In addition, Line 19 of Algorithm 34 writes an $msc^k$ at $MSL^B_{c_i}.p_o^{\text{sync}}$ so that $MSL^A_{c_i}$ also confirms the mode-switch completion of $c_i$ through the execution of lines 29-39 of Algorithm 33 during its next activation, updating the mode of $c_i$ at $MSL^A_{c_i}.p_o^d$. If $\mathcal{S}c_{c_i}^A(k) \neq \emptyset$, $MSL^B_{c_i}$ will receive an $msc^k$ from $\mathcal{S}c_{c_i}^A(k)$ which is handled by lines 26-38 of Algorithm 34. A variable $n$ is used to keep track of the number of $msc^k$ primitives that have been received.

When $c_i$ receives an $ems^k$ from $c_{j}^i \in \mathcal{S}c_{c_i}$, lines 3-13 of Algorithm 34 is first executed. Line 11 of Algorithm 34 writes an $ems^k(m_{c_{j}^i}, m_{c_{i}^new})$ at $MSL^B_{c_i}.p_o^{\text{sync}}$ such that lines 15-28 of Algorithm 33 is executed during the next activation of $MSL^A_{c_i}$. Note that both algorithms 33 and 34 betray slight deviation from our mode-switch dependency rule. According to the mode-switch dependency rule which allows the concurrent reconfiguration of different components, $c_i$ is supposed to start reconfiguration after the propagation of the
Algorithm 33 \( EMS_{MSL}^A(c_i \in \hat{C}) \)

1: loop
2: if \( p^l_t \) then
3: if \( p^msx_t = ems^k(m^{new}_{c_i}) \) then
4: Mode\_mapping\((c_i, m^{new}_{c_i})\);
5: Reconfiguration\((c_i, m_{c_i}, m^{new}_{c_i})\);
6: for all \( c'_{j} \in SC_{c_i}(k) \) do
7: \( p^l_o := ems^k(m^{new}_{c'_{j}}); \)
8: end for
9: for all \( c'_{j} \in SC_{c_i} \setminus SC_{c_i}(k) \) do
10: \( p^l_o := \emptyset; \)
11: end for
12: \( p^s_o := m_{c_i}; \)
13: \( p^{sync}_o := ems^k(m^{new}_{c_i}); \)
14: end if
15: if \( (p^{sync}_t = ems^k(m^{new}_{c'_j}, m^{new}_{c'_j})) \land (c'_{j} \in SC_{c_i}) \) then
16: Mode\_mapping\((c_i, c'_j, m_{c'_j}, m^{new}_{c'_j})\);
17: if \( T_{c_i}^k = A \) then
18: Reconfiguration\((c_i, m_{c_i}, m^{new}_{c_i})\);
19: end if
20: for all \( c_{i}^{q} \in SC_{c_i}(k) \setminus \{c'_{j}\} \) do
21: \( p^q_o := ems^k(m^{new}_{c_{i}^{q}}); \)
22: end for
23: for all \( c'_{j} \in SC_{c_i} \setminus (SC_{c_i}(k) \setminus \{c'_{j}\}) \) do
24: \( p^l_o := \emptyset; \)
25: end for
26: \( p^s_o := m_{c_i}; \)
27: \( p^{sync}_o := \emptyset; \)
28: end if
29: if \( p^{sync}_t = m\_sync^k \) then
30: for all \( c_{i}^{l} \in SC_{c_i} \) do
31: \( p^l_o := \emptyset; \)
32: end for
33: if \( T_{c_i}^k = A \) then
34: \( p^s_o := m_{c_i}^{new}; \)
35: else
36: \( p^s_o := m_{c_i}; \)
37: end if
38: \( p^{sync}_o := \emptyset; \)
39: end if
40: \( p^l_o := true; \)
41: end if
42: end loop
Chapter 7. Mode switch for the ProCom component model

Algorithm 34 $EMS_{MSL}^{B}(c_i \in \mathring{C})$

1: loop
2: if $p_i^1$ then
3: if $\exists p_i^j = ems^k(m_{c_j}^l, m_{c_j}^{new})$ then
4: Mode_mapping($c_i$, $c_j^l$, $m_{c_j}^l$, $m_{c_j}^{new}$);
5: if $T_{c_i} = A$ then
6: Reconfiguration($c_i$, $m_{c_i}^l$, $m_{c_i}^{new}$);
7: $p_o^{msx} := ems^k(m_{c_i}^l, m_{c_i}^{new})$;
8: else
9: $p_o^{msx} := \emptyset$;
10: end if
11: $p_o^{sync} := ems^k(m_{c_i}^l, m_{c_i}^{new})$;
12: $n := 0$;
13: end if
14: if $p_i^{sync} = ems^k(m_{c_i}^{new})$ then
15: Mode_mapping($c_i$, $m_{c_i}^{new}$);
16: Reconfiguration($c_i$, $m_{c_i}^l$, $m_{c_i}^{new}$);
17: if $SC_{c_i}^A (k) = \emptyset$ then
18: $p_o^{msx} := msc^k$;
19: $p_o^{sync} := msc^k$;
20: else
21: $n := 0$;
22: $p_o^{msx} := \emptyset$;
23: $p_o^{sync} := \emptyset$;
24: end if
25: end if
26: if $\exists p_i^l = msc^k$ then
27: $n := n + 1$;
28: if $n = |SC_{c_i}^A (k)|$ then
29: $p_o^{sync} := msc^k$;
30: $p_o^{msx} := \emptyset$;
31: if $T_{c_i} = A$ then
32: $p_o^{msx} := msc^k$;
33: end if
34: else
35: $p_o^{msx} := \emptyset$;
36: $p_o^{sync} := \emptyset$;
37: end if
38: end if
39: $p_o^i := true$;
40: end if
41: end loop
However, the mode-switch handling of all ProSave components share the same control flow originally triggered by a single Clock connector shown in Figure 7.9, thus prohibiting the concurrent reconfiguration of different ProSave components. Therefore, the propagation of an $emsk^k$ from $c_i$ to $SC_{c_i}(k)$ is delayed after the reconfiguration of $MSL^A_{c_i}$ or $MSL^B_{c_i}$. This deviation only prolongs mode-switch time, yet without affecting the correctness of algorithms 33 and 34.

A remaining special issue should be clarified here concerning algorithms 33 and 34. As far as we know, an input data port of a ProSave component retains the current data until the data is overwritten by a new input data. This may lead to the recurrent execution of the same fragment of both algorithms. For instance, after $c_i$ receives an $emsk^k$ from $P_{c_i}$, $p^m_{msx} = emsk^k(m_{c_i}^{new})$ (Line 3 of Algorithm 33) will hold until $MSL^A_{c_i}, p^m_{msx}$ is overwritten. As a consequence, lines 3-14 of Algorithm 33 is executed every time $MSL^A_{c_i}$ is activated. Such an undesirable behavior can be prevented by comparing the data of $MSL^A_{c_i}, p^m_{msx}$ between two activations of $MSL^A_{c_i}$ in succession. Lines 3-14 of Algorithm 33 is only executed when the current data at $MSL^A_{c_i}, p^m_{msx}$ is $emsk^k(m_{c_i}^{new})$, yet different from the previous data at $MSL^A_{c_i}, p^m_{msx}$. We skip such mechanism to avoid unnecessary complication of both algorithms.

### 7.3.2 Handling mode switch in ProSys

The mode-switch handling in the ProSys layer is similar to that in ProSave. For a composite ProSys component $c_i$, we introduce a dedicated subcomponent of $c_i$: $MSL_{c_i}$, which is a primitive ProSys component and plays an equal role as the pair of $MSL^A_{c_i}$ and $MSL^B_{c_i}$ in ProSave. However, message passing between ProSys components is more flexible than the pipes and filters communication in ProSave. Two ProSys components can send messages to each other, therefore, a single subcomponent $MSL_{c_i}$ is sufficient to handle the mode switch of $c_i$.

Let $SC_{c_i} = \{c^1_j, c^2_j, \ldots, c^n_j\}$ ($n \in \mathbb{N}, n = |SC_{c_i}|$) denote the set of subcomponents of $c_i$, excluding $MSL_{c_i}$. Figure 7.10 describes the ports of $MSL_{c_i}$:

- $p^m_{msx}$: an input message port for receiving a downstream primitive from $P_{c_i}$.

- $P_i = \{p_1^i, p_2^i, \ldots, p^n_i\}$: a set of input message ports for receiving an upstream primitive from $SC_{c_i}$.

- $p^o_0$: an output message port indicating the current mode of $c_i$. 


- \( P_o = \{p^1_o, p^2_o, \ldots, p^n_o\} \): a set of output message ports for sending a downstream primitive to \( SC_c_i \).

- \( p^{msx}_o \): an output message port for sending an upstream primitive to \( P_c_i \).

![Diagram of MSLc_i](image)

**Figure 7.10:** The port definition of \( MSLc_i \)

The connections around \( MSLc_i \) are illustrated in Figure 7.11, where the ports not pertinent to the mode switches of \( c_i \) and \( SC_c_i \) have been omitted to simplify the view. Each connection between an output message port and an input message port is established through a message channel. Component \( MSLc_i \) has direct communication with both \( c_i \) and \( SC_c_i \). On the one hand, \( MSLc_i, p^{msx}_i \) is connected to \( c_i.p^{ms}_i \) and \( MSLc_i, p^{msx}_o \) is connected to \( c_i.p^{ms}_o \). On the other hand, \( MSLc_i, p^{ms}_o \) (\( k \in [1, n] \)) is connected to \( c^k_i.p^{ms}_j \) and \( c^k_i.p^{msx}_i \) is connected to \( MSLc_i, p^k_i \). The execution of \( MSLc_i \) requires no external activation by any Clock connector because ProSys components are inherently active. The Control Or connector in ProSave is removed for the same reason. This connection pattern, which allows the transmission of both downstream and upstream primitives, is repeated for all composite ProSys components.

Similar to algorithms 33 and 34 for the pair of \( MSL^A_{c_i} \) and \( MSL^B_{c_i} \), Algorithm 35 describes the implementation of the EMSP protocol and the mode-switch dependency rule in \( MSLc_i \) for the handling of an \( ems^k \) from either \( P_c_i \) or a subcomponent of \( c_i \), where \( c_i \in \bar{C} \). Since concurrent execution is supported in ProSys, \( MSLc_i \) starts the reconfiguration of \( c_i \) after the propagation of the \( ems^k \), strictly following the mode-switch dependency rule.

As far as we know, a ProSys component has message queues for each input message port. An input message is removed from its input message queue after it is consumed by \( MSLc_i \). Hence the recurrent execution problem of \( MSL^A_{c_i} \) and \( MSL^B_{c_i} \) does not arise in Algorithm 35.
7.3 Handling mode switch in ProCom

Algorithm 35 $EMS_{MSL}_{c_i}(c_i \in \hat{C})$

1: loop
2: if $p^m_{i \text{ms}} = \text{ems}^k(m^\text{new}_{c_i})$ then
3:   Mode_mapping($c_i, m^\text{new}_{c_i}$);
4:   for all $c_j \in SC^A_{c_i}(k)$ do
5:     $p^o_{i} := \text{ems}^k(m^\text{new}_{c_j})$;
6:   end for
7:   Reconfiguration($c_i, m_{c_i}, m^\text{new}_{c_i}$);
8: if $SC^A_{c_i}(k) = \emptyset$ then
9:   $p^m_{i \text{ms}} := \text{ems}^k(m_{c_i}, m^\text{new}_{c_i})$;
10: $p^o_{i} := m^\text{new}_{c_i}$;
11: end if
12: $n := 0$;
13: end if
14: if $\exists p^l_{i} = \text{ems}^k(m^\text{new}_{c_j}, m^\text{new}_{c_j})$ then
15:   Mode_mapping($c_i, c_j, m^\text{new}_{c_j}, m^\text{new}_{c_j}$);
16:   for all $c^q_j \in SC^A_{c_i}(k) \setminus \{c^l_j\}$ do
17:     $p^o_{i} := \text{ems}^k(m^\text{new}_{c_j})$;
18: end for
19: if $T^k_{c_i} = A$ then
20:   Reconfiguration($c_i, m_{c_i}, m^\text{new}_{c_i}$);
21: end if
22: $n := 0$;
23: end if
24: if $\exists p^l_{i} = \text{ms}^k$ then
25:   $n := n + 1$;
26: if $n = |SC^A_{c_i}(k)|$ then
27:   $p^m_{i \text{ms}} := \text{ems}^k(m_{c_i}, m^\text{new}_{c_i})$;
28: $p^o_{i} := m^\text{new}_{c_i}$;
29: end if
30: end if
31: end loop
7.4 Managing the variability of component connections

Our mode-aware component model presented in Chapter 2 states that a multi-mode component can have separate configurations for different modes. Since a component may be activated for some modes and deactivated for some other modes, the configuration of a multi-mode composite component for a given mode is influenced by the activated subcomponents and their connections. Hence the activated subcomponents and inner component connections of a multi-mode composite component may be changed during a mode switch. Ideally, the inner component connections of a composite component should be separately defined at design time and changed to each other during a mode switch at runtime. However, it is a tricky problem in ProCom to define multiple configurations which can be dynamically switched at runtime. In order to manage the variability of component connections in different modes in ProCom, we provide a solution which can automatically generate a complete view of inner component connections of each composite ProCom component for all modes.
7.4 Managing the variability of component connections

its modes, based on its inner component connections separately defined for each mode. Depending on the current mode of a composite component, the activated subcomponents and corresponding inner component connections are selected.

7.4.1 Managing the variability of component connections in ProSave

Consider a multi-mode composite component $c_i$ which is a ProSave component, or a ProSys component composed by ProSave components. The inner component connections of $c_i$ may be changed when $c_i$ switches mode. The basic idea of managing the variability of inner component connections of $c_i$ is to package each $c_j^k \in SC_{c_i}$ ($k \in [1, n]$, $n = |SC_{c_i}|$) with additional connectors. Each connector integrates all the possible incoming or outgoing connections of a specific port for all modes and can select the correct connection based on the current mode of $c_i$.

Let $M_{c_i} = \{m_{c_i}^1, m_{c_i}^2, \ldots, m_{c_i}^q\}$ ($q \in \mathbb{N}$ and $q \geq 1$) be the set of supported modes of $c_i$. Suppose that the inner component connections of $c_i$ for all the modes of $c_i$ have been well-defined. Besides, for a ProSave component $c$, the following sets of ports are defined:

- $\mathcal{P}_i^t$: the set of input trigger ports excluding $c.p_i^{m_{c_i}}$.
- $\mathcal{P}_i^d$: the set of input data ports excluding $c.p_i^{m_{c_i}}$.
- $\mathcal{P}_o^t$: the set of output trigger ports excluding $c.p_o^{m_{c_i}}$.
- $\mathcal{P}_o^d$: the set of output data ports excluding $c.p_o^{m_{c_i}}$.

With the intention to merge the inner component connections of $c_i$ into a complete view, we automatically generate connectors within $c_i$ based on the following rules:

- For each $p$ where $p \in c_j^k.\mathcal{P}_i^t$ or $p \in c_i.\mathcal{P}_o^t$, a Control Or connector $A$ is generated, with a set of input trigger ports $\mathcal{P}_i = \{p_i^{t1}, p_i^{t2}, \ldots, p_i^{tq}\}$ ($q = |M_{c_i}|$) and an output trigger port $p_o^t$. The incoming connection to $A.p_i^{tl}$ follows the pre-defined connection for mode $m_{c_i}^l$. The output trigger port $A.p_o^t$ is directly connected to $p$.

- For each $p$ where $p \in c_j^k.\mathcal{P}_i^d$ or $p \in c_i.\mathcal{P}_o^d$, a Data Or connector $B$ is generated, with a set of input data ports $\mathcal{P}_i = \{p_i^{d1}, p_i^{d2}, \ldots, p_i^{dq}\}$ ($q = |M_{c_i}|$) and an output data port $p_o^d$. The incoming connection to $A.p_i^{dl}$ follows the pre-defined connection for mode $m_{c_i}^l$. The output data port $A.p_o^d$ is directly connected to $p$.
and an output data port \( p^d_o \). The incoming connection to \( B.p^{dl}_i \) (\( l \in [1, q] \)) follows the pre-defined connection for mode \( m^l_{c_i} \). The output data port \( B.p^d_o \) is directly connected to \( p \).

- For each \( p \) where \( p \in c_j^k.P^d_o \) or \( p \in c_i.P^d_i \), a Selection connector \( C \) is generated, with an input trigger port \( p^t_i \), an input data port \( p^s_i \) and a set of output trigger ports \( \mathcal{P}_o = \{p^t_{o1}, p^t_{o2}, \ldots, p^t_{oq}\} \) (\( q = |\mathcal{M}_{c_i}| \)). The input trigger port \( C.p^t_i \) is directly connected to \( p \). The input data port \( C.p^s_i \) is connected to MSIA\(_{c_i}.p^s_o \) (see Section 7.3.1). The outgoing connection from \( C.p^o_l \) (\( l \in [1, q] \)) follows the pre-defined connection for mode \( m^l_{c_i} \) according to the data from \( C.p^s_i \): If the data returns \( m^l_{c_i} (l \in [1, q]) \), then \( C.p^o_l \) will be triggered.

- For each \( p \) where \( p \in c_j^k.P^d_o \) or \( p \in c_i.P^d_i \), a Data Selection connector \( D \) is generated, with an input data port \( p^d_i \), and another input data port \( p^s_i \) and a set of output data ports \( \mathcal{P}_o = \{p^d_{o1}, p^d_{o2}, \ldots, p^d_{oq}\} \) (\( q = |\mathcal{M}_{c_i}| \)). The input data port \( D.p^d_i \) is directly connected to \( p \). The input data port \( D.p^s_i \) is connected to MSIA\(_{c_i}.p^s_o \). The outgoing connection from \( D.p^o_l \) (\( l \in [1, q] \)) follows the pre-defined connection for mode \( m^l_{c_i} \) according to the data from \( D.p^s_i \): If the data returns \( m^l_{c_i} (l \in [1, q]) \), the data from \( D.p^o_l \) will be forwarded exactly to \( D.p^{dl}_l \).

When the rules above are applied to a ProSys component \( c_i \) composed by ProSave components, each message port of \( c_i \) will be considered as a port group consisting of a trigger port and a data port to be internally connected to the generated connectors \( A-D \). Among these four generated connectors, Data Selection does not exist in the current ProCom component model. Nonetheless, it can be easily developed because its execution semantics is fairly similar to Selection. This is the only extension of ProCom required by our approach. The above connector generation rules are illustrated in Figure 7.12 where generated connectors are externally connected to \( c_j^k \in SC_{c_i} \). Additional connectors internally connected to \( c_i \) can also be generated accordingly (further illustrated in Section 7.5).

### 7.4.2 Managing the variability of component connections in ProSys

The central idea of managing the variability of ProSys component connections is rather comparable to the connector generation rules in the ProSave layer. A straightforward conversion is to substitute the four generated connectors \( A-D \)
with primitive ProSys components that simulate the behaviors of these connectors. That said, since an input message port can receive messages from multiple senders, there is no need to generate ProSys components playing the role of Control Or or Data Or. The only ProSys component that needs to be generated is a Selection ProSys component which functions as both connectors Selection and Data Selection.

Let $c_i$ be a composite ProSys component composed by ProSys components, with the set of supported modes $\mathcal{M}_{c_i} = \{m_{c_i}^1, m_{c_i}^2, \ldots, m_{c_i}^q\}$ ($q \in \mathbb{N}$ and $q \geq 1$) and the set of subcomponents $\mathcal{SC}_{c_i} = \{c_{i1}^1, c_{i2}^1, \ldots, c_{in}^n\}$ ($n = |\mathcal{SC}_{c_i}|$). Suppose that the inner component connections of $c_i$ for each mode $m_{c_i}^l$ ($l \in [1, q]$) have been well-defined. For a ProSys component $c$, let $\mathcal{P}_i$ be the set of input message ports excluding $c.p_i^{ms}$ and let $\mathcal{P}_o$ be the set of output message ports excluding $c.p_o^{ms}$. Then for each $p$ where $p \in c_j^i.\mathcal{P}_o$ ($k \in [1, n]$) or $p \in c_i.\mathcal{P}_i$, a primitive ProSys component, called Selection and denoted as $E$, is generated, with two input message ports $p_i$ and $p_s$ and a set of output message ports $\mathcal{P} = \{p_o^1, p_o^2, \ldots, p_o^q\}$ ($q = |\mathcal{M}_{c_i}|$). The port $E.p_i$ is directly connected to $p$ while $E.p_s$ is connected to $\text{MSL}_{c_i}.p_o^s$ (see Section 7.3.2). The outgoing connection from $E.p_o^i$ ($l \in [1, q]$) follows the pre-defined connection for mode $m_{c_i}^l$ according to the data from $E.p_s$. If the data returns $m_{c_i}^l$ ($l \in [1, q]$), the message sent to $E.p_i$ will be forwarded to $E.p_o^l$. Figure 7.13 illustrates $c_j^k \in \mathcal{SC}_{c_i}$ externally connected to the generated Selection component $E$. Additionally, a Selection component can also be generated and internally connected to $c_i$ (further illustrated in Section 7.5).
7.5 Demonstrating development of multi-mode systems in ProCom

In this section, we use the example in Figure 1.2 in Chapter 1 to demonstrate how a multi-mode system can be developed by the use of multi-mode components in ProCom.

7.5.1 System description

Consider a multi-mode system to be developed in ProCom, with its component hierarchy given in Figure 1.2. The basic mode mappings of composite components $a$ and $c$ are presented in tables 3.2 and 3.3 in Chapter 3. The inner component connections of $a$ and $c$ for different modes are described in Figure 1.2 in Chapter 1.

A specific requirement in ProCom is that $d$ and $e$ are ProSave components while the others are ProSys components. As a ProSys component composed by ProSys components, $a$ has two possible inner component connections illustrated in Figure 7.14, which matches the connections in Figure 1.2. By contrast, $c$ is a ProSys component composed by ProSave components. Since the connections in Figure 3.1 only address data flow, there are multiple design choices with respect to the inner component connections of $c$ in ProCom to match the connections in Figure 1.2, depending on the control flow specification. Figure 7.15 illustrates the inner component connections of $c$ with a possible control flow specification that matches the connections in Figure 1.2.

The first step to develop such a multi-mode system in ProCom is to define each multi-mode ProSave and ProSys components. Figure 7.16 shows the hierarchy of all multi-mode ProCom components. The dedicated mode-switch ports of each component are marked in purple. Furthermore, as multi-mode ProSave components, $d$ and $e$ also have a dedicated service $S_{mode}$. 
7.5 Demonstrating development of multi-mode systems in ProCom

In this section, we use the example in Figure 1.2 in Chapter 1 to demonstrate how a multi-mode system can be developed by the use of multi-mode components in ProCom.

7.5.1 System description
Consider a multi-mode system to be developed in ProCom, with its component hierarchy given in Figure 1.2. The basic mode mappings of composite components $a$ and $c$ are presented in tables 3.2 and 3.3 in Chapter 3. The inner component connections of $a$ and $c$ for different modes are described in Figure 1.2 in Chapter 1.

A specific requirement in ProCom is that $d$ and $e$ are ProSave components while the others are ProSys components. As a ProSys component composed by ProSys components, $a$ has two possible inner component connections illustrated in Figure 7.14, which matches the connections in Figure 1.2. By contrast, $c$ is a ProSys component composed by ProSave components. Since the connections in Figure 3.1 only address data flow, there are multiple design choices with respect to the inner component connections of $c$ in ProCom to match the connections in Figure 1.2, depending on the control flow specification. Figure 7.15 illustrates the inner component connections of $c$ with a possible control flow specification that matches the connections in Figure 1.2.

The first step to develop such a multi-mode system in ProCom is to define each multi-mode ProSave and ProSys components. Figure 7.16 shows the hierarchy of all multi-mode ProCom components. The dedicated mode-switch ports of each component are marked in purple. Furthermore, as multi-mode ProSave components, $d$ and $e$ also have a dedicated service $S_{mode}$.

7.5.2 Mode-switch handling
In this example, $b$, $d$, $e$ and $f$ are primitive components, whose mode-switch handling can be directly implemented by code. As composite components, $a$
and $c$ must use additional subcomponents to handle their mode switches.

Component $c$ is a ProSys component composed by ProSave components, implying that its mode switch should be handled by a pair of dedicated subcomponents $MSL^A_c$ and $MSL^B_c$, both of which are primitive ProSave components that can be automatically generated. Figure 7.17 presents $MSL^A_c$ and $MSL^B_c$ and their ports, which strictly conform to the port definitions of $MSL^A_{c_i}$ and $MSL^B_{c_i}$ in Section 7.3.1. The ports $p^e_o$ and $p^f_o$ are grouped as $MSL^A_{c_i}.P^{msx}_o$; the ports $p^e_i$ and $p^f_i$ are grouped as $MSL^B_{c_i}.P^{msx}_i$, where $e$ and $f$ are the subcomponents of $c$. The MSRM of $c$ is implemented in the pair of $MSL^A_c$ and $MSL^B_c$. Moreover, the connections around $MSL^A_c$ and $MSL^B_c$ are presented in Figure 7.19.

![Figure 7.17: The port definition of $MSL^A_c$ and $MSL^B_c$](image)

Component $a$ is a ProSys component composed by ProSys components, implying that its mode switch should be handled by a single dedicated subcomponent $MSL_a$ which is a primitive ProSys component that can be automatically generated. Figure 7.18 presents $MSL_a$ and its ports, which strictly conform to the port definition of $MSL_{c_i}$ in Section 7.3.2. The ports $p^b_o$, $p^c_i$, and $p^d_i$ are grouped as $MSL_a.P^{msx}_i$; the ports $p^b_o$, $p^c_o$, and $p^d_o$ are grouped as $MSL_o.P^{msx}_o$, where $b$, $c$, and $d$ are the subcomponents of $a$. The MSRM of $a$ is implemented in $MSL_a$. Moreover, the connections around $MSL_a$ are presented in Figure 7.20.

### 7.5.3 Managing component connections for different modes

The inner component connections of $a$ and $c$ separately defined for their modes have been provided in figures 7.14 and 7.15. The key to manage the variability of inner component connections in ProCom is to merge the inner component connections of each composite component into a complete view for all its
7.5 Demonstrating development of multi-mode systems in ProCom

Figure 7.18: The port definition of $MSL_a$

modes. Such a complete view can be automatically generated by adhering to the principles introduced in Section 7.4.

The complete view of the inner component connections of $c$ is presented in Figure 7.19, automatically generated based on Figure 7.15. Automated generated elements are highlighted in different colors. The green color highlights the elements for handling the mode switch of $c$, while the purple regions highlight the elements for managing the variability of the inner component connections of $c$. The mode switch of $c$ is handled by the pair of $MSL^A_c$ and $MSL^B_c$ introduced in Section 7.5.2. Furthermore, a Clock and a Control Or connector are generated in line with the connection pattern in Figure 7.9. Placed at the top ProSave level, the Clock periodically activates the control flow through the $S_{mode}$ services of all ProSave components.

The light purple region covers 15 generated connectors including all the four types of connectors Control Or, Data Or, Selection and Data Selection introduced in Section 7.4.1. These connectors are either externally connected to the subcomponents of $c$ or internally connected to $c$. Each port of $e$ and $f$ not belonging to their $S_{mode}$ services is externally connected to a connector. Besides, each message port of $c$ not dedicated to mode switch is treated as a port group with a trigger port and a data port. The connector generation rule in Section 7.4.1 regards each input port of $c$ as an output port of a subcomponent and regards each output port of $c$ as an input port of a subcomponent. Hence there are six generated connectors internally connected to three message ports of $c$. All these connectors have three input or output ports because $c$ can run in three modes: $m^1_c$, $m^2_c$ and $m^3_c$. Each Selection or Data Selection has an input data port marked in red. This port is connected to $MSL^A_c.p^o_c$ which tells the current mode of $c$ so that the correct outgoing connection is selected.

A complete view of the inner component connections of $a$ is presented
Figure 7.19: The complete view of inner component connections of \( c \).

225

Demonstrating development of multi-mode systems in ProCom in Figure 7.20, automatically generated based on Figure 7.14. Highlighted in green, a dedicated subcomponent \( MSL_a \) described in Section 7.5.2 is generated for handling the mode switch of \( a \). The light purple region covers six generated Selection ProSys components to merge the inner component connections of \( a \) for all its modes. Section 7.4.2 specifies that each output message port of the subcomponents of \( a \) and each input message port of \( a \) should be connected to a generated Selection component, provided that they are not dedicated mode-switch ports. Each Selection component has two output message ports because \( a \) can run in two modes: \( m_1 \) and \( m_2 \). Meanwhile, each Selection component also has a particular input message port marked in red. This port is connected to \( MSL_a.p \) which tells the current mode of \( a \) such that the correct outgoing connection is selected.

As a final remark, the generated complete views of component connections in figures 7.19 and 7.20 appear to be rather complex even for a simple example, making development of complex multi-mode systems questionable in ProCom. However, with the support of at least two justifiable arguments, we do not deem this a threat to the feasibility of our approach. First, a complete view of component connections can be automatically generated from the separate specification of component connections for each mode by the system designer, while the inner component connections of a composite component for a particular mode are much less complex. Second, it is possible to apply significant simplification to the generated complete views. An implicit assumption made in Section 7.4 for merging the inner component connections of a composite component \( c_i \) in ProCom is that each port of the subcomponent of \( c_i \) not dedicated to mode switch has a unique external connection for each mode of \( c_i \), while each port of \( c_i \) not dedicated to mode switch has a unique internal connection for each mode of \( c_i \). On many occasions though, some inner connections of \( c_i \) may remain the same for different modes of \( c_i \). For instance, it can be observed from Figure 7.15 that the outgoing connection of \( f \) is never changed regardless of the current mode of \( c \). Then the four generated connectors between \( f \) and the second output port of \( c \) in Figure 7.19 can be removed without altering the inner component connections of \( c \). If such optimization is employed for all composite ProSave and ProSys components, the entire system model in ProCom is likely to be substantially simplified.
in Figure 7.20, automatically generated based on Figure 7.14. Highlighted in green, a dedicated subcomponent $MSL_a$ described in Section 7.5.2 is generated for handling the mode switch of $a$. The light purple region covers six generated Selection ProSys components to merge the inner component connections of $a$ for all its modes. Section 7.4.2 specifies that each output message port of the subcomponents of $a$ and each input message port of $a$ should be connected to a generated Selection component, provided that they are not dedicated mode-switch ports. Each Selection component has two output message ports because $a$ can run in two modes: $m_1^a$ and $m_2^a$. Meanwhile, each Selection component also has a particular input message port marked in red. This port is connected to $MSL_a.p^s_o$ which tells the current mode of $a$ such that the correct outgoing connection is selected.

As a final remark, the generated complete views of component connections in figures 7.19 and 7.20 appear to be rather complex even for a simple example, making development of complex multi-mode systems questionable in ProCom. However, with the support of at least two justifiable arguments, we do not deem this a threat to the feasibility of our approach. First, a complete view of component connections can be automatically generated from the separate specification of component connections for each mode by the system designer, while the inner component connections of a composite component for a particular mode are much less complex. Second, it is possible to apply significant simplification to the generated complete views. An implicit assumption made in Section 7.4 for merging the inner component connections of a composite component $c_i$ in ProCom is that each port of the subcomponent of $c_i$ not dedicated to mode switch has a unique external connection for each mode of $c_i$, while each port of $c_i$ not dedicated to mode switch has a unique internal connection for each mode of $c_i$. On many occasions though, some inner connections of $c_i$ may remain the same for different modes of $c_i$. For instance, it can be observed from Figure 7.15 that the outgoing connection of $f$ is never changed regardless of the current mode of $c$. Then the four generated connectors between $f$ and the second output port of $c$ in Figure 7.19 can be removed without altering the inner component connections of $c$. If such optimization is employed for all composite ProSave and ProSys components, the entire system model in ProCom is likely to be substantially simplified.
As a theoretical framework, our MSL requires a concrete component model to fulfill its implementation for practical usage. MSL boasts great potential to be integrated into many existing component models, upgrading them with the support of multi-mode systems. This chapter integrates MSL in the ProCom component model which in its original form does not incorporate support for development of multi-mode applications. Taking both ProSave and ProSys layers of ProCom into account, our approach yields a slightly extended ProCom component model that not only allows reuse of multi-mode components but also is able to run our MSRM to handle mode switch at runtime.

Our first attempt is to define multi-mode ProCom components in compliance with our mode-aware component model. Dedicated mode-switch ports are added to both ProSave and ProSys components. Moreover, a dedicated service is reserved for each ProSave component to separate its mode-switch behavior from its original functionalities. For primitive components, our MSRM can be directly implemented by code in both ProSave and ProSys. For composite components, the MSRM is implemented by additional dedicated subcomponents. If a composite component is a ProSave component, or a ProSys component composed by ProSave components, its mode switch is handled by a pair of dedicated subcomponents synchronized with each other. If a composite component is a ProSys component composed by ProSys components, its mode switch is handled by a single dedicated subcomponent. Finally, we merge the inner component connections of a composite component for different modes into a complete view to manage the variability of component connections in a multi-mode context. Additional elements such as connectors in ProSave and ProSys components are automatically generated so that component connections defined for a particular mode are properly selected.
7.6 Summary

As a theoretical framework, our MSL requires a concrete component model to fulfill its implementation for practical usage. MSL boasts great potential to be integrated into many existing component models, upgrading them with the support of multi-mode systems. This chapter integrates MSL in the ProCom component model which in its original form does not incorporate support for development of multi-mode applications. Taking both ProSave and ProSys layers of ProCom into account, our approach yields a slightly extended ProCom component model that not only allows reuse of multi-mode components but also is able to run our MSRM to handle mode switch at runtime.

Our first attempt is to define multi-mode ProCom components in compliance with our mode-aware component model. Dedicated mode-switch ports are added to both ProSave and ProSys components. Moreover, a dedicated service is reserved for each ProSave component to separate its mode-switch behavior from its original functionalities. For primitive components, our MSRM can be directly implemented by code in both ProSave and ProSys. For composite components, the MSRM is implemented by additional dedicated subcomponents. If a composite component is a ProSave component, or a ProSys component composed by ProSave components, its mode switch is handled by a pair of dedicated subcomponents synchronized with each other. If a composite component is a ProSys component composed by ProSys components, its mode switch is handled by a single dedicated subcomponent. Finally, we merge the inner component connections of a composite component for different modes into a complete view to manage the variability of component connections in a multi-mode context. Additional elements such as connectors in ProSave and Selection ProSys components are automatically generated so that component connections defined for a particular mode are properly selected.
Chapter 8

Improving mode-switch efficiency

Reuse of multi-mode software components brings more flexible design choices for software development of multi-mode components while preserving the benefits of CBSE. The MSRM of our MSL has the power to cope with highly complex scenarios during a mode switch, even if no component knows the global mode information. The distributed nature of MSRM makes it possible to substitute, add, and remove components on the fly without reconfiguring the entire system. In spite of these advantages, the runtime mode-switch efficiency arises as a practical issue for MSL. Both the computation overhead of the MSRM and the inter-component communication are prone to exert negative influence upon mode-switch efficiency. The computation overhead of the MSRM from the perspective of an individual component is rather low because the only task of the MSRM is to determine how a component exchange primitives with its parent and subcomponents. Nevertheless, from the system's perspective, the overall computation overhead of the MSRM is in proportion to the number of components on account of the concurrent execution of the MSRM in each component. The inter-component communication during a mode switch, characterized by the stepwise transmission of primitives, may incur long mode-switch time.

Of course, runtime computation overhead and communication is an inevitable price to pay for the MSRM as a distributed mechanism, which is necessary when software components are deployed on distributed hardware nodes or no global mode information is available. However, for systems where all the software components are deployed onto the same physical hardware platform...
Chapter 8

Improving mode-switch efficiency

Reuse of multi-mode software components brings more flexible design choices for software development of multi-mode components while preserving the benefits of CBSE. The MSRM of our MSL has the power to cope with highly complex scenarios during a mode switch, even if no component knows the global mode information. The distributed nature of MSRM makes it possible to substitute, add, and remove components on the fly without reconfiguring the entire system. In spite of these advantages, the runtime mode-switch efficiency arises as a practical issue for MSL. Both the computation overhead of the MSRM and the inter-component communication are prone to exert negative influence upon mode-switch efficiency. The computation overhead of the MSRM from the perspective of an individual component is rather low because the only task of the MSRM is to determine how a component exchange primitives with its parent and subcomponents. Nevertheless, from the system’s perspective, the overall computation overhead of the MSRM is in proportion to the number of components on account of the concurrent execution of the MSRM in each component. The inter-component communication during a mode switch, characterized by the stepwise transmission of primitives, may incur long mode-switch time.

Of course, runtime computation overhead and communication is an inevitable price to pay for the MSRM as a distributed mechanism, which is necessary when software components are deployed on distributed hardware nodes or no global mode information is available. However, for systems where all the software components are deployed onto the same physical hardware platform
and the mode information of all components is globally accessible, a more efficient scheme is centralized mode-switch handling by a global mode-switch manager. Combining the flexibility and multi-mode component reuse of MSL at design time with the runtime efficiency of global mode-switch management, this chapter presents a mode transformation technique that improves the mode-switch efficiency of MSL by transforming component modes to system modes to be handled by a global mode-switch manager. After mode transformation, there will be no need for each component to run its MSRM and exchange primitives with other components during a mode switch. A mode-switch process is facilitated, as a mode switch becomes a direct transition between two system modes.

8.1 Mode transformation

The purpose of mode transformation is to replace the distributed mode-switch handling of MSL with a centralized solution which yields better run-time performance. Mode transformation is conducted once the multi-mode component architecture of a system is completed. Illustrated in Figure 8.1, mode transformation transfers the responsibility of mode-switch handling from the MSRM of each component to a single mode-switch manager of the system. As a result of mode transformation, each multi-mode component turns into a single-mode component without MSRM and mode mapping, freed from all mode-related activities.

Figure 8.1 also reveals the internal overall structure of the mode-switch manager. When a scenario is triggered, it is first stored in an input buffer of the mode-switch manager. The mode-switch manager periodically checks the input buffer. If no mode switch is in progress, the mode-switch manager will perform a mode switch based on the first pending scenario in the input buffer. The mode-switch manager can apply appropriate arbitration mechanisms and mode-switch protocols [105] to handle each scenario, ensuring that a mode switch does not violate any functional and timing requirements. These protocols are assisted by a mode transition graph that guides the mode-switch manager to switch to the right mode for each scenario. Designing or selecting the suitable mode-switch protocols is out of the scope of the thesis. Instead, we delve into the construction of the mode transition graph.

Shown in Figure 8.2, our mode transformation process includes two sequential steps. First, given the mode mappings of all composite components, we construct an intermediate representation, a Mode Combination Tree (MCT)
Chapter 8. Improving mode-switch efficiency

As the mode information of all components is globally accessible, a more efficient scheme is centralized mode-switch handling by a global mode-switch manager. Combining the flexibility and multi-mode component reuse of MSL at design time with the runtime efficiency of global mode-switch management, this chapter presents a mode transformation technique that improves the mode-switch efficiency of MSL by transforming component modes to system modes to be handled by a global mode-switch manager. After mode transformation, there will be no need for each component to run its MSRM and exchange primitives with other components during a mode switch. A mode-switch process is facilitated, as a mode switch becomes a direct transition between two system modes.

8.1 Mode transformation

The purpose of mode transformation is to replace the distributed mode-switch handling of MSL with a centralized solution which yields better runtime performance. Mode transformation is conducted once the multi-mode component architecture of a system is completed. Illustrated in Figure 8.1, mode transformation transfers the responsibility of mode-switch handling from the MSRM of each component to a single mode-switch manager of the system. As a result of mode transformation, each multi-mode component turns into a single-mode component without MSRM and mode mapping, freed from all mode-related activities.

Figure 8.1: The overview of mode transformation

where all the possible system modes are identified. In the second step, taking the specification of all mode-switch scenarios in the system as the input, we add all the possible transitions between the identified system modes to construct the mode transition graph.

Figure 8.2: The mode transformation process

In the following two sections, we shed light on the two transformation steps separately.
8.2 Construction of the mode combination tree

The aim of constructing the MCT is to identify all the system modes. Let \( M_{c_i} \) denote the set of supported modes of \( c_i \) and \( D \) denote the current mode of a deactivated component. Then we define system modes as follows:

**Definition 37. System modes based on component modes:** For a system composed by a set of components \( C = \{c_1, c_2, \cdots, c_n\} \) \((n \in \mathbb{N})\), the set of system modes is defined as \( M_s \subseteq \bigtimes_{i \in [1, n]} M_{c_i} \cup \{D\} \). Each system mode \( m \in M_s \) is a mode combination of all components.

By Definition 37, each system mode \( m = (m_{c_1}, m_{c_2}, \cdots, m_{c_n}) \), where \( m_{c_i} \in M_{c_i} \cup \{D\} \) for \( i = [1, n] \). In order to more explicitly indicate the relationship between \( c_i \) and \( m_{c_i} \), we shall hereafter use an alternative expression to represent a system mode: \( m = \{(c_i, m_{c_i})|i \in [1, n]\} \), where \( m_{c_i} \in M_{c_i} \cup \{D\} \). Using the same formalism, an MCT is defined as follows:

**Definition 38. Mode Combination Tree:** An MCT is a tree with a set of nodes \( N = \{N_0, N_1, \cdots, N_n\} \) \((n \in \mathbb{N})\), where \( N_0 = \emptyset \) is the root node, and each other node \( N_i = \{(c_j, m_{c_j})|j \in [1, k], k \in \mathbb{N}\} \) \((i \in [1, n]\)\), where for all \( j \), \( m_{c_j} \in M_{c_j} \cup \{D\} \) and all \( c_j \) have the same depth level.

Definition 38 implies that each node of an MCT, except the root node, provides a mode combination of components with the same depth level (Definition 36 in Chapter 5). A typical outlook of MCT is displayed in Figure 8.5 of Section 8.4, where the construction of the MCT will be further explained.

Before the formal description of the MCT construction process, we need to introduce a number of additional notations and concepts. For each \( c_i \in \mathbb{C} \), a valid Local Mode Combination (LMC) of \( c_i \) is defined as follows:

**Definition 39. Valid Local Mode Combination:** For \( c_i \in \mathbb{C} \) with \( \mathbb{S}c_{c_i} = \{c_j^1, \cdots, c_j^n\} \) \((n \in \mathbb{N})\), we call the set \( \mathbb{V}_{c_i} = \{(c_i, m_{c_i}), (c_j^1, m_{c_j}^1), \cdots, (c_j^n, m_{c_j}^n)\} \) a valid LMC of \( c_i \), if (1) \( m_{c_i} \in M_{c_i} \cup \{D\} \) and \( \forall k \in [1, n], m_{c_j}^k \in M_{c_j} \cup \{D\} \); and (2) \( m_{c_i} \) and all \( m_{c_j}^k \) \((k \in [1, n]\) can be simultaneously executed by the corresponding components, i.e. conforming to the mode mapping of \( c_i \).

Note that each element in \( \mathbb{V}_{c_i} \) is a pair \((x, y)\) where \( x \in \mathbb{S}c_{c_i} \cup \{c_i\} \) and \( y \in M_x \cup \{D\} \). For instance, the mode mapping of \( c \) in Table 3.3 of Chapter 3 implies four valid LMCs of
8.2 Construction of the mode combination tree

Definition 39. Valid Local Mode Combination: 
other node modes is defined as introduce a number of additional notations and concepts. For each of Section 8.4, where the construction of the MCT will be further explained. 

Definition 36 in Chapter 5). A typical outlook of MCT is displayed in Figure 8.5 

c denote the set of supported modes of The aim of constructing the MCT is to identify all the system modes. Let c i denote deactivated component. Then we define system modes as follows: 

System modes based on component modes: 

Definition 37. 

c i can be simultaneously executed by the corresponding components, i.e. con-

Definition 38 implies that each node of an MCT, except the root node, 

In order to more explicitly indicate the For the sake of lucidity, let us clarify the ± operator using a small example. Suppose \( W_1 = \{V_1, V_2\} \) where \( V_1 = \{(a, m^1_a), (b, m^1_b)\} \) and \( V_2 = \{\}

**Theorem 17.** \( W_{c_i,m_{c_i}} \) property: For \( c_i \in CC \) with \( SC_{c_i} = \{c^1_{j_1}, \ldots, c^n_{j_n}\} \) \( n \in \mathbb{N} \), \( |W_{c_i,D}| = 1 \) and \( W_{c_i,D} = \{V_{c_i,D}\} \) where \( V_{c_i,D} \equiv \{(c^1_{j_1}, D), \ldots, (c^n_{j_n}, D)\} \).

**Proof.** When a composite component \( c_i \) is deactivated, all its enclosed components must also be deactivated. Hence, if the current mode of \( c_i \) is \( D \), then the current modes of its subcomponents must also be \( D \). \( V_{c_i,D} = \{(c^1_{j_1}, D), \ldots, (c^n_{j_n}, D)\} \) is the only valid LMC of \( c_i \) for \( D \), implying \( |W_{c_i,D}| = 1 \). Theorem 17 follows. \( \square \)

Next we introduce an important operator for combining different valid LMCs:

**Definition 41.** Valid LMC operation: Consider two set of valid LMCs \( W_1 = \{V_1, V_2, \ldots, V_m\} \) and \( W_2 = \{V_{k+1}, V_{k+2}, \ldots, V_{k+n}\} \), where \( m, n, k \in \mathbb{N} \). Let \( \oplus \) be an operator such that \( W_1 \oplus W_2 = \{V_i \cup V_{k+j} | i \in [1, m], j \in [1, n]\} \). In addition, for each \( l \in \mathbb{N} \), \( W_1 \oplus W_2 \oplus \cdots \oplus W_l \) can be represented as \( \bigoplus_{o \in [1,l]} W_o \). For the sake of lucidity, let us clarify the \( \oplus \) operator using a small example. Suppose \( W_1 = \{V_1, V_2\} \) where \( V_1 = \{(a, m^1_a), (b, m^1_b)\} \) and \( V_2 = \{\}

\[(a, m_2^a), (b, m_2^b)\}; and \(W_2 = \{V_3, V_4\}\) where \(V_3 = \{(c, m_3^1), (d, m_4^1)\}\) and \(V_4 = \{(e, m_2^e), (d, m_3^3)\}\). Then, 

\[
W_1 \oplus W_2 = \{V_1 \cup V_3, V_1 \cup V_4, V_2 \cup V_3, V_2 \cup V_4\} = \{(a, m_1^a), (b, m_1^b), (c, m_1^c), (d, m_1^d)\}, \\
\{(a, m_2^a), (b, m_1^b), (c, m_2^c), (d, m_2^d)\}, \\
\{(a, m_2^a), (b, m_2^b), (c, m_1^c), (d, m_1^d)\}, \\
\{(a, m_2^a), (b, m_2^b), (c, m_2^c), (d, m_2^d)\}\}

Given the component hierarchy of a system and the mode mappings of all composite components in the system, the MCT of the system can be constructed by creating nodes top-down from the root node. For each node \(N\) of an MCT, let \(d_N\) be its depth level, and \(\lambda_N\) be the number of new nodes created from this node. We use \(N_i \succ N_j\) to denote that a new node \(N_i\) is created from an old node \(N_j\). Moreover, let \(\mathcal{M}_\text{Top} = \{m_1^T, m_2^T, \ldots, m_{|\mathcal{M}_\text{Top}|}^T\}\) be the set of supported modes of \(\text{Top}\). The MCT is constructed by the following steps:

1. From \(N_0\), create \(\lambda_{N_0} = |\mathcal{M}_\text{Top}|\) new nodes, such that for each new node \(N_i \succ N_0, N_i = \{(\text{Top}, m_i^T)\} (i \in [1, |\mathcal{M}_\text{Top}|])\).

2. From each \(N_i = \{(\text{Top}, m_i^T)\} (i \in [1, |\mathcal{M}_\text{Top}|])\), create \(\lambda_{N_i} = |W_{\text{Top}, m_i^T}|\) new nodes, such that for each \(N'' \succ N_i, N'' \in W_{\text{Top}, m_i^T}\). Moreover, if \(\lambda_{N_i} > 1\), then for each \(N'', N'' \succ N_i\), we have \(N'' \neq N''\).

3. For each node \(N = \{(c_1, m_{c_1}), (c_2, m_{c_2}), \ldots, (c_n, m_{c_n})\} (n \in \mathbb{N})\) with \(d_N \geq 2\), if \(\forall i \in [1, n], c_i \in \mathcal{PC}\), then \(N\) is marked as a leaf node and no new node is created from \(N\). Otherwise, if \(\exists i \in [1, n]\) such that \(c_i \in \mathcal{CC}\), then create \(\lambda_N = \prod_{i \in [1, n]} |W_{c_i, m_{c_i}}|\) new nodes, such that for each \(N' \succ N, N' \in \bigoplus_{i \in [1, n], c_i \in \mathcal{CC}} W_{c_i, m_{c_i}}\). Moreover, if \(\lambda_N > 1\), then for each \(N', N'' \succ N\), we have \(N' \neq N''\).

4. Repeat Step 3 until all branches of the MCT have reached the leaf node.

The MCT construction process is implemented as Algorithm 36, which is a recursive function constructMCT\((\mathcal{N}, d_N)\) that has two input parameters: \(\mathcal{N}\) is the node currently being explored and \(d_N\) is the depth level of \(\mathcal{N}\). Initially,
$\mathcal{N} = \emptyset$ and $d_N = 0$. We assume that $\text{Top}$ must have subcomponents. Otherwise, $\text{Top}$ itself will be the entire system and mode transformation will be meaningless. Moreover, for each component $c_i$ running in mode $m$, we assume that $\mathcal{W}_{c_i,m}$ is an indexed set such that $\mathcal{W}_{c_i,m}[j]$ represents the $j$th element of $\mathcal{W}_{c_i,m}$.

**Algorithm 36** $\text{constructMCT}(\mathcal{N}, d_N)$

1: if $d_N = 0$ then
2: \quad $\lambda_N := |\mathcal{M}_\text{Top}|$;
3: \quad for $i$ from 1 to $\lambda_N$ do
4: \quad \quad $\mathcal{N}_i := \{(\text{Top}, m_{\lambda})\}$;
5: \quad \quad $\text{constructMCT}(\mathcal{N}_i, 1)$;
6: \quad end for
7: end if
8: if $d_N = 1$ then
9: \quad $\{(\text{Top}, m)\} := \mathcal{N}$;
10: \quad $\text{Derive } \mathcal{W}_{\text{Top},m}$;
11: \quad $\lambda_N := |\mathcal{W}_{\text{Top},m}|$;
12: \quad for $i$ from 1 to $\lambda_N$ do
13: \quad \quad $\text{constructMCT}(\mathcal{W}_{\text{Top},m}[i], 2)$;
14: \quad end for
15: end if
16: if $d_N \geq 2$ then
17: \quad $\{(c_1, m_{c_1}), (c_2, m_{c_2}), \ldots, (c_n, m_{c_n})\} := \mathcal{N}$;
18: \quad if $\forall i \in [1, n] : c_i \in \mathcal{P} \mathcal{C}$ then
19: \quad \quad return ;
20: \quad else
21: \quad \quad $\text{Derive } \mathcal{W} := \bigoplus_{i\in[1,n], \ c_i\in\mathcal{C}} \mathcal{W}_{c_i,m_{c_i}}$;
22: \quad \quad $\lambda_N := \prod_{i\in[1,n], \ c_i\in\mathcal{C}} |\mathcal{W}_{c_i,m_{c_i}}|$;
23: \quad \quad for $i$ from 1 to $\lambda_N$ do
24: \quad \quad \quad $\text{constructMCT}(\mathcal{W}[i], d_N + 1)$;
25: \quad \quad end for
26: \quad end if
27: end if

The complexity of Algorithm 36 depends on the structure of the component hierarchy, the number of modes of each component, and all the mode mappings. Since all these contributing factors are bounded, this algorithm will terminate with a bounded number of steps. Actually, the computation overhead
of the algorithm is not a major concern for run-time performance, inasmuch as the MCT is constructed at design time.

An interesting property can be observed from an MCT, formulated as the following theorem:

**Theorem 18.** A non-root node \( N \) of an MCT is represented by a valid mode combination of components with depth level \( d_N - 1 \).

**Proof.** Since \( N \) cannot be the root node, \( d_N \geq 1 \). When \( d_N = 1 \), \( N = \{(Top, m) | m \in \mathcal{M}_{Top}\} \) and \( d_{Top} = d_N - 1 = 0 \). When \( d_N = 2 \), according to Step 2 of the MCT construction procedure, \( N \in \mathcal{W}_{Top, m} \) (\( m \in \mathcal{M}_{Top}\)). The definition of \( \mathcal{W} \) implies that \( N \) is represented by a valid LMC of the subcomponents of \( Top \), while each subcomponent of \( Top \) has a depth level \( d_N - 1 = 1 \). Note that for \( d_N \geq 3 \), \( N \) is created by the same rule (Step 3). Hence Theorem 18 can be proved by induction for \( d_N \geq 3 \). Suppose \( N = \{(c_1, m_{c_1}), \ldots, (c_n, m_{c_n})\} \) (\( n \in \mathbb{N} \)).

**Basis:** When \( d_N = 3 \), according to Step 3 of the MCT construction procedure, the definition of \( \mathcal{W} \), and the definition of the operator \( \bigoplus \), \( N \in \bigoplus_{d_{c_i}=1}^{d_N} \mathcal{W}_{c_i, m_{c_i}} \).

Then for all \( (c_j, m_{c_j}) \in N \), \( d_{c_j} = d_N - 1 = 2 \).

**Inductive step:** Suppose that when \( d_N = k \), for all \( (c_j, m_{c_j}) \in N \), we have \( d_{c_j} = k - 1 \). Then we need to prove that for another node \( N' = \{(c'_1, m_{c'_1}), \ldots, (c'_o, m_{c'_o})\} \) (\( o \in \mathbb{N} \)) with \( d_{N'} = k + 1 \), we have \( d_{c'_i} = k \) (\( i \in [1, o] \)) for all \( (c'_i, m_{c'_i}) \in N' \). The node \( N' \) can be created either from \( N \) or from another node \( N'' \) with \( d_{N''} = d_N \). Since \( N \) and \( N'' \) include the same set of components with depth level \( k - 1 \), we always have \( N' \in \bigoplus_{d_{c_j}=k-1}^{d_N} \mathcal{W}_{c_j, m_{c_j}} \).

By the definition of \( \mathcal{W} \) and the operator \( \bigoplus \), all \( c'_i \) included in \( N' \) have a depth level equal to \( k \), i.e. \( d_{N'} - 1 \).

The induction above together with our analysis for \( d_N = 1 \) and \( d_N = 2 \) proves Theorem 18.

Theorem 18 further implies the following corollary:

**Corollary 1.** All leaf nodes of an MCT have the same depth level.

**Proof.** Let \( N_1 \) and \( N_2 \) be two leaf nodes of the same MCT. By Theorem 18, both \( N_1 \) and \( N_2 \) include the same set of primitive components with the maximum depth level of the component hierarchy. Let \( d \) be the depth level of these primitive components. Then \( d = d_{N_1} - 1 = d_{N_2} - 1 \). Therefore, \( d_{N_1} = d_{N_2} \), thus proving the corollary.
Once the MCT is constructed, the system modes can be derived as a path from the root node to a leaf node of the MCT. Let $\mathcal{N}^k$ be the set of nodes of an MCT with depth level $k$. Then,

**Theorem 19.** Given an MCT, a system mode $m$ is represented by a valid mode combination $\bigcup_{i=0}^{\delta} \mathcal{N}_i$ where $\mathcal{N}_i \in \mathcal{N}^i$, and $\delta$ is the maximum depth level of the MCT, i.e. $\mathcal{N}_\delta$ is a leaf node. The total number of system modes is equal to the total number of leaf nodes of the MCT.

**Proof.** Let $\mathcal{C} = \{c_1, c_2, \cdots, c_n\}$ ($n \in \mathbb{N}$) be the set of components of a system. First we prove that $\bigcup_{i=0}^{\delta} \mathcal{N}_i$ includes the modes of all components. Then we prove that each component is included in $m$ only once.

By Theorem 18, each $\mathcal{N} \neq \mathcal{N}_0$ with $d_{\mathcal{N}} = k$ is represented by a valid mode combination of components with depth level $k - 1$. Since the maximum depth level of the MCT is $\delta$, the maximum depth level of a component in the system is $\delta - 1$. Each path of the MCT is a union $\bigcup_{i=0}^{\delta} \mathcal{N}_i$ that includes the mode combination of components with depth levels from 0 to $\delta - 1$. Since the depth level of any component is within $[0, \delta - 1]$, the mode of the component must be included in $\bigcup_{i=0}^{\delta} \mathcal{N}_i$, i.e., a path of the MCT.

Next consider any two pairs $(c_i, m_{c_i}) \in m$ and $(c_j, m_{c_j}) \in m$ ($i, j \in [1, n]$). If $d_{c_i} \neq d_{c_j}$, then apparently $c_i \neq c_j$. If $d_{c_i} = d_{c_j} = k$, then $(c_i, m_{c_i})$ and $(c_j, m_{c_j})$ must be both included in any node $\mathcal{N} \in \mathcal{N}^{k+1}$ by Theorem 18. Components $c_i$ and $c_j$ may or may not share the same parent. If their parents are different, then apparently $c_i \neq c_j$. If they share the same parent, by Definition 40, we also have $c_i \neq c_j$. Therefore, $c_i \neq c_j$ always holds regardless of the depth levels of $c_i$ and $c_j$. Stated otherwise, each component is included in $m$ only once.

Hence, $m = \bigcup_{i=0}^{\delta} \mathcal{N}_i$ represents a system mode. Furthermore, we know that each path of an MCT includes a leaf node. Since all leaf nodes are different, all paths are also different. The number of paths, i.e. the number of system modes, equals the number of leaf nodes.

Among the system modes, the initial system mode can be recognized based on the specification of the initial modes of all components.
8.3 Deriving the mode transition graph

Once the system modes are identified from the constructed MCT, the next step is to derive the (system) mode transition graph on top of these system modes. For a system with a set of identified system modes \( \mathcal{M} = \{m_1, m_2, \ldots, m_n\} \) \((n \in \mathbb{N})\), a mode switch is a transition from \( m_{\text{old}} \) to \( m_{\text{new}} \), where \( m_{\text{old}}, m_{\text{new}} \in \mathcal{M} \) and \( m_{\text{old}} \neq m_{\text{new}} \). A mode transition graph contains all the possible transitions between these system modes and associates each transition with the corresponding scenario. Here we provide the formal definition of a mode transition graph:

**Definition 42. Mode transition graph:** A mode transition graph is a tuple:

\[
< \mathcal{S}, s^0, \mathcal{K}, \mathcal{T} >
\]

where \( \mathcal{S} \) is a set of states, with each state \( s \in \mathcal{S} \) corresponding to a system mode; \( s^0 \in \mathcal{S} \) is the initial state, corresponding to the initial system mode; \( \mathcal{K} \) is a set of scenarios specified for the system; \( \mathcal{T} \subseteq \mathcal{S} \times \mathcal{K} \times \mathcal{S} \) is a set of state transitions. Each state transition signifies a possible mode switch for a specific scenario.

Each state of a mode transition graph can be graphically represented as a location with a circle, with the initial state being marked by a double circle. Each state transition triggered by scenario \( k \in \mathcal{K} \) from \( s_1 \) to \( s_2 \) \((s_1, s_2 \in \mathcal{S})\) is denoted as \( s_1 \xrightarrow{k} s_2 \) and graphically represented by an arrow starting from \( s_1 \) and ending in \( s_2 \) with the label \( k \). A graphical illustration of mode transition graph can be found in Figure 8.10 of Section 8.4. Note that it is also possible to associate each transition of the graph with a guard represented by an application-dependent boolean expression that has been introduced to a parent MMA in Chapter 3. Such a guard is not the focus of mode transformation, thus omitted in Definition 42.

The key issue of deriving the mode transition graph is to identify the system modes \( m_{\text{old}} \) and \( m_{\text{new}} \) for each scenario \( k \) for which \( m_{\text{old}} \xrightarrow{k} m_{\text{new}} \) is possible. Consider a scenario \( c_i : m_{c_i}^1 \rightarrow m_{c_i}^2 \). The only condition satisfying the triggering of \( k \) is that the MSS \( c_i \) is currently running in \( m_{c_i}^1 \). For each \( k, m_{\text{old}} \) can be easily identified as long as \((c_i, m_{c_i}^1) \in m_{\text{old}} \). Note that more than one system mode could be identified as \( m_{\text{old}} \). Depending on the current system mode, a scenario may enable different transitions.

In contrast to \( m_{\text{old}} \), only one system mode can be the \( m_{\text{new}} \) for each scenario \( k \). The identification of \( m_{\text{new}} \) for \( k \) is more difficult because it depends
not only on $m_{c_i}^2$, but also on the target modes of the other components. We identify the $m_{\text{new}}$ for each scenario with the assistance of a Component Target Mode (CTM) table:

**Definition 43. Component Target Mode table:** A CTM table is a table with $n_1$ rows and $n_2$ columns, where $n_1$ is the number of components of a system and $n_2$ is the number of scenarios. Each row is associated with a component, each column is associated with a scenario, and each cell contains the target mode $m_{c_i}$ of the corresponding component $c_i$ for the corresponding scenario $k$. If $c_i$ is a Type B component for $k$, the cell contains $X$, indicating that $m_{c_i}$ is independent of $k$.

An example of a CTM table can be found above the mode transition graph in Figure 8.10 of Section 8.4. A CTM table can be automatically constructed offline based on the scenario specification and the mode mapping of each composite component. Let $m_{c_i}^k$ be the target mode of $c_i$ for $k$ in a CTM table. Taking advantage of the CTM table, the $m_{\text{new}}$ for each scenario $k$ can be identified as follows: For each system mode $m = \{(c_i, m_{c_i}) | i \in [1, n], n \in \mathbb{N}\}$, if $\forall i$ where $m_{c_i}^k \neq X$ in the CTM table (i.e., $T_{c_i}^k = A$), we have $m_{c_i} = m_{c_i}^k$, then $m$ is the $m_{\text{new}}$ for $k$. Algorithm 37 describes the process of building the mode transition graph, with a search space of $O(|\mathcal{M}| \cdot |\mathcal{K}|)$.

**Algorithm 37** $\text{constructMTG}(\mathcal{C}, \mathcal{M}, \mathcal{K})$

1: $\mathcal{C} = \{c_1, \cdots, c_o\}$ ($o \in \mathbb{N}$); \{The set of all components\}
2: $\mathcal{M} = \{m_1, \cdots, m_n\}$ ($n \in \mathbb{N}$); \{The set of identified system modes\}
3: $\mathcal{K} = \{k_1, \cdots, k_l\}$ ($l \in \mathbb{N}$); \{The set of all scenarios\}
4: for all $k_i \in \mathcal{K}$ where $k_i \in [1, l]$ and $k_i = c : m_1^c \rightarrow m_2^c$ do
5: \hspace{0.5cm} if $\exists m_j \in \mathcal{M}$ s.t. ($\forall c_p \in \mathcal{C}$ and $T_{c_i}^{k_i} = A$, $(c_p, m_{c_i}^k) \in m_j$) then
6: \hspace{1cm} $m_{\text{new}} = m_j$;
7: \hspace{0.5cm} for all $m_j \in \mathcal{M}$ do
8: \hspace{1cm} if $(c, m_1^c) \in m_j$ then
9: \hspace{1.5cm} addTransition($m_j, m_{\text{new}}, k_i$); \{Add a transition from $m_j$ to $m_{\text{new}}$ labeled with $k_i$\}
10: \hspace{1cm} end if
11: \hspace{0.5cm} end for
12: \hspace{0.5cm} end if
13: end for

The mode transition graph, presented in Figure 8.1, helps the global mode-switch manager keep track of the current system mode and makes the system switch to the right target mode when a scenario is triggered.
8.4 Mode transformation example

To make our mode transformation technique more comprehensible, let us demonstrate mode transformation in the example in Figure 8.3, which changes the primitive component \( d \) in Figure 1.2 of Chapter 1 into a composite component with two subcomponents \( g \) and \( h \). The mode mapping tables of composite components \( a, c, \) and \( d \) are declared in Figure 8.4, which retains the mode mappings of \( a \) and \( c \) presented in tables 3.2 and 3.3 and adds another mode mapping table for \( d \).

![Figure 8.3: The component hierarchy of an example](image)

![Figure 8.4: The mode mapping tables of \( a, c, d \)](image)

The first step of mode transformation is to construct an MCT with reference to the component hierarchy in Figure 8.3 and mode mappings in Figure 8.4. Mode mapping tables do not specify the new modes of Type A components for a scenario, however, they are sufficient to construct the MCT. The construction of the MCT exactly follows the procedures described in Section 8.2:

1. From the root node \( N_0 = \emptyset \), create \( \lambda_{N_0} = |M_a| = 2 \) nodes: \( N_1 = \)
2. From $N_1$, create $\lambda_{N_1} = |W_{a,m_1^1}|$ nodes. According to Figure 8.4(a), there are in total two valid LMCs of $a$ when $a$ runs in $m_1^1$. Hence, $\lambda_{N_1} = 2$. Let the two new nodes be $N_3$ and $N_4$, then $N_3 = \{(b, m_2^1), (c, m_2^1), (d, D)\}$ and $N_4 = \{(b, m_2^1), (c, m_2^2), (d, D)\}$. The same procedure is applied to $N_2$, i.e. by creating $\lambda_{N_2} = |W_{a,m_2^1}|$ nodes from $N_2$. Figure 8.4(a) indicates that there is only one valid LMC of $a$ when $a$ runs in $m_2^1$. Therefore, $\lambda_{N_2} = 1$ and the new node $N_5 = \{(b, m_2^1), (c, m_2^2), (d, m_2^1)\}$.

3. Now there are three nodes with depth level 2: $N_3$, $N_4$, and $N_5$. Let’s first look at $N_3 = \{(b, m_2^1), (c, m_2^1), (d, D)\}$. Among, $b$, $c$ and $d$, there are two composite components: $c$ and $d$. Hence $N_3$ is not a leaf node and $\lambda_{N_3} = |W_{c,m_2^1}| * |W_{d,D}|$ new nodes are supposed to be created. Figure 8.4(b) implies that $|W_{c,m_2^1}| = 1$ and $W_{c,m_2^1} = \{(e, m_2^1), (f, m_2^1)\}$. Meanwhile, Theorem 17 implies that $|W_{d,D}| = 1$ and $W_{d,D} = \{(g, D), (h, D)\}$. Hence, $\lambda_{N_3} = 1$. In addition, $W_{c,m_2^1} \oplus W_{d,D} = \{(e, m_2^1), (f, m_2^1), (g, D), (h, D)\}$. Let $N_6$ be the new node created from $N_3$. Since $N_6 \in W_{c,m_2^1} \oplus W_{d,D}$, $N_6 = \{(e, m_2^1), (f, m_2^1), (g, D), (h, D)\}$.

4. Repeat Step 3 for $N_4$. Create $\lambda_{N_4} = |W_{c,m_2^2}| * |W_{d,D}|$ nodes from $N_4$. Figure 8.4(b) implies that $|W_{c,m_2^2}| = 1$ and $W_{c,m_2^2} = \{(e, m_2^1), (f, m_2^2)\}$. By Theorem 17, $|W_{d,D}| = 1$ and $W_{d,D} = \{(g, D), (h, D)\}$. Hence, $\lambda_{N_4} = 1$ and $W_{c,m_2^2} \oplus W_{d,D} = \{(e, m_2^1), (f, m_2^1), (g, D), (h, D)\}$. Let $N_7$ be the new node created from $N_4$. Then we have $N_7 = \{(e, D), (f, m_2^1), (g, D), (h, D)\}$.

5. Repeat Step 3 for $N_5$. We need to create $\lambda_{N_5} = |W_{c,m_2^1}| * |W_{d,m_2^1}|$ new nodes from $N_5$. Figure 8.4(b) implies that $|W_{c,m_2^2}| = 2$ and $W_{c,m_2^2} = \{(e, m_2^3), (f, m_2^1)\}, \{(e, m_2^3), (f, m_2^2)\}$. Figure 8.4(c) implies that $|W_{d,m_2^1}| = 2$ and $W_{d,m_2^1} = \{(g, m_2^1), (h, m_2^1)\}, \{(g, m_2^1), (h, m_2^2)\}, \{(g, m_2^2), (h, m_2^2)\}$. Hence, $\lambda_{N_5} = 4$ and

$$W_{c,m_2^3} \oplus W_{d,m_2^3} = \{(e, m_2^1), (f, m_2^1), (g, m_2^1), (h, m_2^1)\},$$

$$\{(e, m_2^3), (f, m_2^1), (g, m_2^1), (h, m_2^1)\},$$

$$\{(e, m_2^3), (f, m_2^1), (g, m_2^2), (h, m_2^2)\},$$

$$\{(e, m_2^3), (f, m_2^2), (g, m_2^2), (h, m_2^2)\}$$
Let $\mathcal{N}_6, \mathcal{N}_9, \mathcal{N}_{10}, \mathcal{N}_{11}$ be the four new nodes created from $\mathcal{N}_5$. Then

$$\mathcal{N}_8 = \{(e, m^2_e), (f, m^2_f), (g, m^1_g), (h, m^1_h)\}$$
$$\mathcal{N}_9 = \{(e, m^3_e), (f, m^1_f), (g, m^1_g), (h, m^1_h)\}$$
$$\mathcal{N}_{10} = \{(e, m^2_e), (f, m^1_f), (g, m^2_g), (h, m^2_h)\}$$
$$\mathcal{N}_{11} = \{(e, m^3_e), (f, m^1_f), (g, m^2_g), (h, m^2_h)\}$$

6. Explore new nodes from the nodes with depth level 3, including $\mathcal{N}_6$–$\mathcal{N}_{11}$. Since all these nodes represent a mode combination of $e, f, g, h$, all of which are primitive components, $\mathcal{N}_6$–$\mathcal{N}_{11}$ are all identified as leaf nodes, thus terminating the MCT construction.

The constructed MCT is presented in Figure 8.5. The MCT consists of 12 nodes, including one root node $\mathcal{N}_0$, two nodes with depth level 1, three nodes with depth level 2, and 6 nodes with depth level 3, i.e.

$$\mathcal{N}^0 = \{\mathcal{N}_0\}$$
$$\mathcal{N}^1 = \{\mathcal{N}_1, \mathcal{N}_2\}$$
$$\mathcal{N}^2 = \{\mathcal{N}_3, \mathcal{N}_4, \mathcal{N}_5\}$$
$$\mathcal{N}^3 = \{\mathcal{N}_6, \mathcal{N}_7, \mathcal{N}_8, \mathcal{N}_9, \mathcal{N}_{10}, \mathcal{N}_{11}\}$$

By Theorem 19, six system modes $m_1$–$m_6$ are identified as different paths of the MCT:
8.4 Mode transformation example

\[ m_1 = N_0 \cup N_1 \cup N_3 \cup N_6 \]
\[ = \left\{ (a, m_1^0), (b, m_1^6), (c, m_1^1), (d, D), (e, m_1^e), (f, m_1^f), (g, D), (h, D) \right\} \]
\[ m_2 = N_0 \cup N_1 \cup N_4 \cup N_7 \]
\[ = \left\{ (a, m_1^0), (b, m_1^6), (c, m_1^3), (d, D), (e, D), (f, m_1^f), (g, D), (h, D) \right\} \]
\[ m_3 = N_0 \cup N_2 \cup N_5 \cup N_8 \]
\[ = \left\{ (a, m_2^0), (b, m_2^3), (c, m_2^2), (d, m_2^1), (e, m_2^e), (f, m_1^f), (g, m_2^g), (h, m_2^h) \right\} \]
\[ m_4 = N_0 \cup N_2 \cup N_5 \cup N_9 \]
\[ = \left\{ (a, m_2^0), (b, m_2^3), (c, m_2^2), (d, m_2^1), (e, m_2^e), (f, m_1^f), (g, m_2^g), (h, m_2^h) \right\} \]
\[ m_5 = N_0 \cup N_2 \cup N_5 \cup N_{10} \]
\[ = \left\{ (a, m_2^0), (b, m_2^3), (c, m_2^2), (d, m_2^1), (e, m_2^e), (f, m_1^f), (g, m_2^g), (h, m_2^h) \right\} \]
\[ m_6 = N_0 \cup N_2 \cup N_5 \cup N_{11} \]
\[ = \left\{ (a, m_2^0), (b, m_2^3), (c, m_2^2), (d, m_2^1), (e, m_2^e), (f, m_1^f), (g, m_2^g), (h, m_2^h) \right\} \]
The initial system mode can be inferred from the initial mode of each component. In this example, we assume that the initial modes of \( a \) and \( c \) are \( m_1^a \) and \( m_1^c \), respectively. Under this assumption, \( m_1 \) is the initial system mode. Each system mode has a unique global configuration that is characterized by factors such as activated components and their connections at all nested levels, or the mode-specific behaviors of certain primitive components. Figure 8.6 illustrates a possible set of global configurations for the six system modes. To simplify the view, composite components \( c \) and \( d \) are removed in Figure 8.6. Similar to Figure 1.2, deactivated components are dimmed while black and grey colors represent different mode-specific behaviors.

![Figure 8.6: The global configurations of different system modes](image)

After the identification of system modes, the next step is to derive the mode transition graph. A key intermediate step is to construct the CTM table. This requires both the specification of all scenarios and the complete mode mappings of all composite components. Figure 8.10 indicates that the system consists of six scenarios from \( k_1 \) to \( k_6 \). As mentioned in Chapter 3, the complete mode mapping of a composite component \( c_i \) is described by a set of internally synchronized Mode Mapping Automata (MMAs), including a parent MMA denoted as \( MMA_{c_i}^p \), and a number of child MMAs. For each \( c_j \in SC_{c_i} \), there is a corresponding child MMA denoted as \( MMA_{c_j}^c \) in the set of MMAs of \( c_i \). Figures 8.7-8.9 depict the set of MMAs of \( a \), \( c \), and \( d \), respectively, covering all six scenarios.
With the mode mappings of \(a, c, d\) and the specification of six scenarios as the input, a CTM table is generated and shown in Figure 8.10. As the last step of mode transformation, the MCT in Figure 8.5 and the CTM table are combined to derive the mode transition graph depicted in Figure 8.10. The mode transition graph can be either manually derived or automatically generated by means of a thorough analysis of the possible transitions between different system modes for each scenario.

There are two major tasks involved in the construction of the mode transition graph. First, the identified system modes from the MCT are interpreted as states of the mode transition graph. In this example, the mode transition graph contains six states, with the initial system mode \(m_1\) as its initial state. Second, state transitions are added based on the CTM table. For instance, since the triggering condition of \(k_1\) is that \(a\) must run in \(m_{1a}\), we need to identify system modes which include \((a, m_{1a})\). In this case, both \(m_1\) and \(m_2\) meet the triggering condition of \(k_1\). Hence \(k_1\) can lead to an outgoing transition from either \(m_1\) or \(m_2\). The target system mode for \(k_1\) is identified by comparing each system...
mode with the target modes of all Type A components for \( k_1 \) listed in the CTM table. Apparently, \( m_3 \) is the only one target mode for \( k_1 \). Therefore, \( k_1 \) can enable two transitions in the mode transition graph, either from \( m_1 \) to \( m_3 \) or from \( m_2 \) to \( m_3 \). The mode transition graph is completed by repeating the same logic for all scenarios.

In order to demonstrate how mode transformation enhances runtime mode-switch efficiency, we compare the mode-switch time for \( k_1 \) before and after mode transformation. Figure 8.10 implies that the system can run in either \( m_1 \) or \( m_2 \) when \( k_1 \) is triggered. We assume \( m_1 \) is the current mode upon the triggering of \( k_1 \). We also assume that \( k_1 \) is an emergency scenario to minimize the

---

**Figure 8.8:** The set of MMAs of \( c \)
246 Chapter 8. Improving mode-switch efficiency

Figure 8.8: The set of MMAs of \( c \) mode with the target modes of all Type A components for \( k_1 \) listed in the CTM table. Apparently, \( m_3 \) is the only one target mode for \( k_1 \). Therefore, \( k_1 \) can enable two transitions in the mode transition graph, either from \( m_1 \) to \( m_3 \) or from \( m_2 \) to \( m_3 \). The mode transition graph is completed by repeating the same logic for all scenarios.

In order to demonstrate how mode transformation enhances runtime mode-switch efficiency, we compare the mode-switch time for \( k_1 \) before and after mode transformation. Figure 8.10 implies that the system can run in either \( m_1 \) or \( m_2 \) when \( k_1 \) is triggered. We assume \( m_1 \) is the current mode upon the triggering of \( k_1 \). We also assume that \( k_1 \) is an emergency scenario to minimize the mode-switch time before mode transformation. In Figure 8.11, the mode-switch process for \( k_1 \) before transformation is depicted on the left side of the timeline. After mode transformation, shown on the right side of the timeline, the mode switch is directly performed by changing the system global configuration from \( m_1 \) to \( m_3 \) without inter-component communication. The system reconfiguration time is assumed to be the maximum reconfiguration time of these components. It is self-evident that the mode-switch time is significantly reduced after mode transformation. This improvement will become even more conspicuous if \( k_1 \) is not an emergency scenario. In fact, when the depth level of component hierarchy is increased, or when each component runs a more complex MSRM, the mode-switch time before mode transformation stretches much more than the mode-switch time after mode transformation does.

8.5 Mode transformation verification

Shown in Figure 8.2 at the beginning of this chapter, mode transformation is conducted in two sequential steps: the construction of the MCT and then the derivation of the mode transition graph. The correctness of these two steps can be verified separately. First, we need to guarantee that the correct set of system modes is identified from the MCT:

**Theorem 20.** The system modes identified from the MCT are correct and complete, and each identified system mode is unique.

**Proof.** The identification of correct system modes is ensured by correct mode
8.5 Mode transformation verification

Let $m_1$ and $m_2$ be two different system modes. By Theorem 19, $m_1$ and $m_2$ correspond to two different paths of the same MCT. Then there must exist a node $N$ where the two paths diverge. Let $N_1$ and $N_2$ be the two nodes created from $N$ such that the path of $m_1$ follows $N_1$ and the path of $m_2$ follows $N_2$. Then $N_1$ and $N_2$ must be different because all new nodes created from $N$ must be different. Therefore, $N_1 \subseteq m_1$ and $N_1 \subsetneq m_2$ while $N_2 \subseteq m_2$ and $N_2 \subsetneq m_1$, implying that $m_1$ and $m_2$ are unique.

Next we need to verify the correctness of the mode transition graph, formulated as follows:

**Theorem 21.** The transitions of a mode transition graph are correct and complete.

**Proof.** We divide the proof into two parts. First we prove the correctness of the transitions of a mode transition graph. Then we prove the completeness of the transitions of a mode transition graph. Both parts can be proved by contradiction. Let $M = \{m_1, \ldots, m_n\} (n \in \mathbb{N})$ be the set of identified system modes, where $m_i, m_j \in M$ and $k = \text{co}: m_1 \text{co} \rightarrow m_2 \text{co}$.

To prove the correctness, suppose a mode transition graph contains a wrong transition $L = m_i \rightarrow m_j$ (i.e. a transition from $m_i$ to $m_j$ for $k$) where either $m_i$ fails to satisfy the triggering condition of $k$ or $m_j$ is the wrong target system mode for $k$. By Algorithm 37, we know that $(\text{co}, m_1\text{co}) \in m_i$. Hence $m_i$ must satisfy the triggering condition of $k$. Moreover, $m_j$ must be the only system mode that matches the CTM table. The correctness of the identified system modes is guaranteed by Theorem 20. The construction of the CTM table is independent of our mode transformation technique. Instead, it only depends on mode mapping. We assume that the mode mappings of all composite components are correctly specified, thus ensuring the correctness of the CTM table, which further proves that $m_j$ is the right target system mode for $k$. This con-

---

Figure 8.10: Deriving the mode transition graph

Figure 8.11: Mode transformation facilitates a mode switch
mappings. Assuming correct mode mappings, the correct \( W_{c_i,m_{c_i}} \) can be derived for each \( c_i \) running in \( m_{c_i} \) so that new nodes can be correctly created from each node.

The completeness of system modes depends on the completeness of the nodes of an MCT. Whenever we create new nodes from a node \( \mathcal{N} = \{(c_i,m_{c_i})|i \in [1,n], n \in \mathbb{N}\} \), we consider all the possible valid LMCs of each \( c_i \) for \( m_{c_i} \) and all the possible combinations of their valid LMCs. Hence an MCT has a complete set of nodes which further guarantee a complete set of identified system modes.

Let \( m_1 \) and \( m_2 \) be two different system modes. By Theorem 19, \( m_1 \) and \( m_2 \) correspond to two different paths of the same MCT. Then there must exist a node \( \mathcal{N} \) where the two paths diverge. Let \( \mathcal{N}_1 \) and \( \mathcal{N}_2 \) be the two nodes created from \( \mathcal{N} \) such that the path of \( m_1 \) follows \( \mathcal{N}_1 \) and the path of \( m_2 \) follows \( \mathcal{N}_2 \). Then \( \mathcal{N}_1 \) and \( \mathcal{N}_2 \) must be different because all new nodes created from \( \mathcal{N} \) must be different. Therefore, \( \mathcal{N}_1 \subseteq m_1 \) and \( \mathcal{N}_1 \not\subseteq m_2 \) while \( \mathcal{N}_2 \subseteq m_2 \) and \( \mathcal{N}_2 \not\subseteq m_1 \), implying that \( m_1 \) and \( m_2 \) are unique.

Next we need to verify the correctness of the mode transition graph, formulated as follows:

**Theorem 21.** The transitions of a mode transition graph are correct and complete.

**Proof.** We divide the proof into two parts. First we prove the correctness of the transitions of a mode transition graph. Then we prove the completeness of the transitions of a mode transition graph. Both parts can be proved by contradiction. Let \( \mathcal{M} = \{m_1, \cdots, m_n\} \ (n \in \mathbb{N}) \) be the set of identified system modes, where \( m_i, m_j \in \mathcal{M} \) and \( k = c_o : m_{c_o}^1 \rightarrow m_{c_o}^2 \).

To prove the correctness, suppose a mode transition graph contains a wrong transition \( L = m_i \xrightarrow{k} m_j \) (i.e. a transition from \( m_i \) to \( m_j \) for \( k \)) where either \( m_i \) fails to satisfy the triggering condition of \( k \) or \( m_j \) is the wrong target system mode for \( k \). By Algorithm 37, we know that \( (c_o,m_{c_o}^1) \in m_i \). Hence \( m_i \) must satisfy the triggering condition of \( k \). Moreover, \( m_j \) must be the only system mode that matches the CTM table. The correctness of the identified system modes is guaranteed by Theorem 20. The construction of the CTM table is independent of our mode transformation technique. Instead, it only depends on mode mapping. We assume that the mode mappings of all composite components are correctly specified, thus ensuring the correctness of the CTM table, which further proves that \( m_j \) is the right target system mode for \( k \). This con-
tradicts the assumption that $L$ is a wrong transition. Therefore, the correctness of the transitions of a mode transition graph follows.

To prove the completeness, suppose a mode transition graph misses a transition $L = m_i \xrightarrow{k} m_j$ which is supposed to be included. First, $m_i$ must satisfy the triggering condition of $k$, implying $(c_o, m_{c_o}^1) \in m_i$. Algorithm 37 ensures that all system modes satisfying this condition can be identified as the mode with an outgoing transition labeled with $k$. Moreover, from the proof of the correctness above, we know that there is only one target system mode for $k$ which can be correctly identified. Hence $m_j$ must be the target system mode for $k$, with an incoming transition labeled with $k$. Since both $m_i$ and $m_j$ can be identified in association with $k$, it is impossible for a mode transition graph to miss the transition $L$. Therefore, the completeness of a mode transition graph follows.

\[ \square \]

8.6 Discussion

This section begins with a discussion on the industrial value of our mode transformation technique, followed by some practical issues which need to be considered during mode transformation.

8.6.1 Industrial value

Our mode transformation technique adds significant potential industrial value to MSL since it allows more efficient reuse of multi-mode components. Apart from the ProCom component model described in Chapter 7, we have investigated the usability of MSL for Rubus [57], an industrial component model developed by Arcticus Systems\footnote{http://www.arcticus-systems.com/} for software development of ground vehicles. In Rubus, a system running on an Electronic Control Unit (ECU) can support multiple modes while each mode is associated with a unique global configuration represented by factors such as activated components and activated component connections. A mode switch is guided by a transition diagram similar to our mode transition graph. Therefore, mode in Rubus is treated in the same way as our mode-switch handling after mode transformation.

Rubus does not support multi-mode components, however, MSL is able to improve the design time flexibility of Rubus by means of reuse of multi-mode components. For each ECU, mode transformation can derive the system
modes, the global configurations for each mode, and the mode transition graph, all of which are consistent with the current mode-switch handling of Rubus.

In addition, we believe that MSL is also beneficial to some other component-based frameworks with mode-switch support such as AUTOSAR (AUTomotive Open System ARchitecture) [43], a standard prevalent in the design of vehicular systems. A mode switch in AUTOSAR is handled by a mode manager allocated to each ECU. This mode manager is comparable to our mode-switch manager after mode transformation.

8.6.2 Merging system modes

Using mode transformation, the number of identified system modes is sensitive to the number of modes of each single component and the mode mapping of each composite component. As a consequence, mode transformation may end up with a huge number of system modes. Nonetheless, it only makes sense to distinguish one mode from another mode when the system behaviors are noticeably different in these modes. Depending on the application, it is more efficient to merge several modes with similar global configurations into one mode. For instance, in Figure 8.6, \( m_5 \) and \( m_6 \) can be merged since the only difference between their global configurations is the behavior of \( e \), which can be simply distinguished by an "IF...ELSE..." expression. Following this principle, the number of system modes will be dramatically reduced. The criteria for merging system modes are application-dependent and out of the scope of the thesis.

8.6.3 Partial mode transformation

Mode transformation does not have to be applied to an entire system. Sometimes it might be recommended or necessary to conduct mode transformation partially, i.e. within a composite component instead of the top component. There are at least two motivating reasons for partial mode transformation. Firstly, when a system consists of a third-party composite component whose internal information is unavailable, it will be impossible to derive the system modes represented by the modes of all components. A compromised solution would be to treat each third-party composite component as a primitive component for mode transformation. Secondly, for a distributed system, it is a natural choice to apply separate mode transformation for each node. For instance, Figure 8.12 deploys the components \( b, c, \) and \( d \) in Figure 8.3 into three distributed
nodes (ECUs). Instead of global mode transformation for the entire system, it is preferable to perform local mode transformation for \( c \) and \( d \) separately.

Figure 8.12: Partial mode transformation for distributed nodes

### 8.6.4 Implementing mode switch after mode transformation

After mode transformation, a mode switch is carried out by changing the global configuration of the system. Depending on the system requirement, such a reconfiguration can be implemented in different ways. For instance, the mode-switch manager can store all the global configurations so that it can simply change the global configuration when a mode switch is triggered. Thereby each component will be completely unaware of modes.

Alternatively, each component can maintain its own local configurations for all its modes. To start a mode switch, the mode-switch manager can broadcast a set of primitives to all the Type A components which then change their own local configurations. After reconfiguration, each component sends a primitive to the mode-switch manager. A mode switch is completed when the mode-switch manager receives the primitive from all the Type A components. In this way, there is no need for the mode-switch manager to store global configurations, yet at the sacrifice of two additional communication steps (from the mode-switch manager to all Type A components which then reply to the mode-switch manager after reconfiguration). Notwithstanding, this communication overhead is rather limited compared with the inter-component communication overhead before mode transformation. The implementation choices of mode switch after mode transformation is out of the scope of the thesis, yet left for future work.
8.6.5 The concurrent triggering of multiple scenarios

Mode transformation makes it easier to handle concurrently triggered scenarios. Without mode transformation, concurrent scenarios must be cooperatively handled by the MSRM of each component described in Chapter 4. The MSR/MSQ/EMS queues of all components are resource-consuming. After mode transformation, all triggered scenarios are directly sent to the global mode-switch manager. Shown in Figure 8.1, the mode-switch manager is equipped with an input buffer to queue multiple scenarios to be handled one after another. In case of concurrent scenarios with different criticality levels, higher priorities can be assigned to more critical scenarios which should be distinguished by the input buffer.

What deserves extra attention is that the handling of one scenario may invalidate the subsequent scenario(s). For instance, suppose the system in Figure 8.10 is running in mode $m_1$. In the interim $k_1$ is triggered, immediately followed by the concurrent triggering of $k_3$. The mode transition graph in Figure 8.10 suggests that the system will run in $m_3$ after the mode switch for $k_1$, whereas there is no outgoing transition from $m_3$ for $k_3$. The implication is that the mode switch for $k_1$ makes $k_3$ invalid. Hence $k_3$ should be removed from the buffer without being handled.

Alternatively, the concurrent triggering of scenarios can be prohibited by enforcing specific constraints in the mode-switch manager.

8.7 Summary

The runtime mode-switch efficiency of MSL is negatively impacted by the runtime computation overhead of MSRM and inter-component communication. This chapter proposes a mode transformation technique for more efficient handling of multi-mode components. When software components are deployed on the same hardware platform and the mode information of each component is globally accessible, mode transformation is able to transform the distributed mode-switch handling based on the MSRM into a centralized mechanism. Mode transformation is then performed in two sequential steps. The first step is to construct an MCT based on the component hierarchy and the mode mappings of all composite components. The MCT caters to the identification of system modes, each system mode identified as a path of the MCT. The second step derives a mode transition graph by adding mode transitions between the identified system modes, given the specification of all mode-switch scenarios.
After mode transformation, a mode switch is handled by a single global mode-switch manager which replaces the MSRM of each component. Guided by the mode transition graph, the mode-switch manager switches mode by directly changing the global configuration of the current system mode to the global configuration of the new system mode without inter-component communication, thereby enhancing mode-switch efficiency at runtime. Since the inter-component communication overhead typically increases as the number of components or the complexity of MSRM grows, the benefit of mode transformation becomes more noteworthy for more complex systems.

We have demonstrated our mode transformation technique by an example. Finally, we have revealed the potential industrial value of mode transformation and discussed several practical issues towards the implementation and use of mode transformation.
Chapter 9

MCORE: the Multi-mode COmponent Reuse Environment

Our MSL framework has been implemented in a prototype tool named MCORE (the Multi-mode COmponent Reuse Environment) [86, 88, 96] which serves as a platform for development of multi-mode systems by reusing multi-mode components. MCORE was developed as a JointJS [71] plugin. JointJS is a modern HTML 5 JavaScript library for visualization and interaction with diagrams and graphs. In MCORE, a user can develop a multi-mode component from scratch, save it in a library for future reuse, or reuse an existing component in the library.

Compared with other component-based development tools, a distinguishing feature of MCORE is reuse of multi-mode software components. Both the modes and configurations of each multi-mode component can be manually specified in accordance with our mode-aware component model introduced in Chapter 2. Figure 9.1 shows the workspace of MCORE, where Component a in Figure 1.2 is modeled as the composition of three other multi-mode components b, c, and d. On top of the workspace is a navigation bar with a New Component button on the left and a menu on the right. The New Component button is used to create new components. The menu provides a list of functions such as saving a component in the library and exporting the system model as a set of xml files. In addition, navigation information is displayed
in the middle of the navigation bar. For instance, Home/a in Figure 9.1 means that the current view in the workspace is within Component a. On the right side of the workspace, there is a configuration panel used to specify the modes and configurations of each component. The configuration panel allows a user to define component ports, mode-dependent or mode-independent properties, and the values of different properties. For instance, Figure 9.1 indicates that a has a property WCET (Worst-Case Execution Time) equal to 50 for mode $m^2_a$. At the bottom of the configuration panel, all saved components are listed in a library for further reuse.

![Figure 9.1: The workspace of MCore](image)

In MCore, the supported modes of a component is represented by tags located at its top right corner. For instance, $m^2_a$ in Figure 9.1 is the currently selected mode of a, represented by the white tab. Hence the inner component connections of a while a runs in mode $m^2_a$ is displayed in Figure 9.1. Meanwhile, the white tabs of b, c, d in the same figure indicate their current modes when a runs in $m^2_a$, based on the mode mapping of a. The mode mapping of a is manually specified by clicking an Edit button next to Home/a in the navigation bar. Once the Edit button is clicked, the Edit button will be switched to a Done button, waiting for the user to specify mode mapping. A user can map the modes of a to the modes of its subcomponents by clicking the corresponding mode tags. Each mode of a can be mapped to multiple modes of a subcomponent. If a subcomponent has no modes mapped to a, it is considered to be deactivated. Figure 9.1 indicates that $m^2_a$ is mapped to $m^2_b$ of b, $m^2_c$ of c, and $m^1_d$ of d. Mode mapping is completed when the Done button is clicked.
Note that MCORE is not a stand-alone tool for development of multi-mode systems. Instead, it functions as a preprocessor for Rubus ICE [114], which is an IDE for the Rubus component model [57] developed by Arcticus Systems in Sweden. Rubus ICE is a commercial tool in use at several companies, including Volvo Construction Equipment¹, BAE Systems², Elektroengine³, Haldex⁴, Hoerbiger⁵, and Knorr-Bremse⁶. As an industrial component model, Rubus brings component-based development to vehicular systems. Rubus supports multi-mode systems, however, modes can only be specified at system level and reuse of multi-mode components is not supported. This limitation can be complemented by MCORE as a preprocessor. Depicted in Figure 9.2, A system is initially modeled in MCORE in which multi-mode components are reused. The system model is then transformed to a centralized mode management by the mode transformation technique presented in Chapter 8. After mode transformation, the system architecture and mode specification will be in compliance with the mode management of the Rubus component model. The mode transformation technique has been implemented in Javascript [86] and is ready to be integrated in MCORE. The link between MCORE and Rubus ICE is the xml files exported from MCORE to Rubus ICE for further analysis, test and code generation.

MCORE is the result of a four-month cooperative work by Daniele Orlando, Francesco Miscia, and Simone Di Marco who used to be involved in a project under the supervision by us. Thus in total, 12 man-months’ effort has been invested to the implementation of MSL in MCORE. However, there are still several remaining challenges to be surmounted towards the consummation of MCORE:

- Implementing MMAs. Even though MCORE provides a user-friendly way for mode mapping specification, it does not cover all the mode mapping rules described by our MMA. A typical example is: when \( a \) in Figure 9.1 switches from \( m^2_a \) to \( m^1_a \), what are the new modes of its subcomponents? New features are required to allow the complete specification of mode mappings.

- MSRM specification for each component. The MSRM is indispensa-

¹http://www.volvoce.com/
²http://www.baesystems.com/
³http://www.electroengine.com/
⁴http://www.haldex.com/
⁵http://www.hoerbiger.com/
Integrating the mode transformation technique in MCORE. The current implementation of the mode transformation technique is separate from MCORE. The integration of mode transformation is an important step to obtain a system model with centralized mode management in MCORE.

- Consistent system modeling between MCORE and Rubus. For instance, Rubus distinguishes control flow through trigger ports from data flow through data ports, while a control flow can be either event-triggered or time-triggered. These features must also be reflected in MCORE.

- Exporting xml files to Rubus ICE. This requires that MCORE should follow the xml format of Rubus ICE. Alternatively, some existing techniques, such as XSLT (Extensible Stylesheet Language Transformations) [24], provide opportunities for the transformation between different xml files.

As a final remark, the potential value of MCORE is not only limited to its role as a preprocessor for Rubus ICE. The MSRM of MSL plays no role in Rubus ICE because the MSRMs of all components are removed and replaced with a single global mode-switch manager after mode transformation. Nevertheless, there are circumstances where mode transformation is undesired or
not viable. Then it will be imperative to retain the MSRM of each multi-mode component for distributed mode-switch handling. To put this feature into practice, MCORE needs to become independent of Rubus ICE. An ambitious goal would be to make MCORE develop into a stand-alone IDE. Alternatively, we may associate MCORE with some other framework supporting multi-mode components.
This chapter serves as a preliminary evaluation of the usability of our MSL, adopting the main principles of MSL in two proof-of-concept case studies: an Adaptive Cruise Control (ACC) system [2], and a healthcare monitoring system. Although we have not reached as far as the real implementation of such systems, we are still able to show that it is viable to use MSL to develop a non-trivial system with reusable multi-mode components. In the following sections, these two case studies will be presented separately alongside a thorough explanation of multi-mode component specification, mode mapping, runtime mode-switch handling, mode-switch timing analysis, and mode transformation.

10.1 An Adaptive Cruise Control system

An ACC system is able to maintain the speed of a vehicle as well as its distance to a preceding vehicle. Our design of the ACC system is heavily influenced by the ACC system built by SaveCCM components [58], which is also a proof-of-concept implementation, yet with strong industrial support. We reconstruct the ACC system through reuse of multi-mode components while retaining the original component architecture described in [58].

10.1.1 System description

An ACC system is typically used as a subsystem of a car to automatically maintain both the desired speed and the safe distance to the vehicle ahead.
Chapter 10

Case studies

This chapter serves as a preliminary evaluation of the usability of our MSL, adopting the main principles of MSL in two proof-of-concept case studies: an Adaptive Cruise Control (ACC) system [2], and a healthcare monitoring system. Although we have not reached as far as the real implementation of such systems, we are still able to show that it is viable to use MSL to develop a non-trivial system with reusable multi-mode components. In the following sections, these two case studies will be presented separately alongside a thorough explanation of multi-mode component specification, mode mapping, runtime mode-switch handling, mode-switch timing analysis, and mode transformation.

10.1 An Adaptive Cruise Control system

An ACC system is able to maintain the speed of a vehicle as well as its distance to a preceding vehicle. Our design of the ACC system is heavily influenced by the ACC system built by SaveCCM components [58], which is also a proof-of-concept implementation, yet with strong industrial support. We reconstruct the ACC system through reuse of multi-mode components while retaining the original component architecture described in [58].

10.1.1 System description

An ACC system is typically used as a subsystem of a car to automatically maintain both the desired speed and the safe distance to the vehicle ahead.
The desired speed can be obtained by information from the driver or by speed-limit regulations, e.g., from road signs or road map information together with the GPS. The presence and distance of a preceding vehicle can be detected by a radar or laser sensor mounted at the front of the car. The detected distance must be sufficiently far to avoid rear-end collisions. When the distance is changed due to the speed discrepancy of the preceding vehicle and the car where the ACC system resides, the ACC system will automatically accelerate or decelerate to keep the distance as per a pre-defined value without the driver’s interference. In addition, when the preceding vehicle suddenly brakes or an obstacle abruptly appears in front, a brake-assist function will be activated for a more aggressive braking in such extreme situations.

The component hierarchy of the ACC system is shown in Figure 10.1, including the following components:

- Speed Limit (SL): derives the desired speed.
- Object Recognition (OR): detects the presence of a preceding vehicle and calculates the relative speed and the distance to the preceding vehicle.
- ACC Controller (ACC): maintains both the desired speed and distance. As the core of the ACC system, it consists of two subcomponents: Distance Controller (DC) and Speed Controller (SC).
- Brake Assist (BA): assists the driver to brake in extreme situations. The braking force depends on the relative speed and the distance to the preceding vehicle.
- HMI Output (HMI): displays information related to the ACC system to the driver.
- Speed Controller (SC): controls the vehicle speed.
- Distance Controller (DC): provides distance information to SC.

To simplify the presentation, the top component (ACC system) will be called "Top" while the others will be called as per their above abbreviations. Furthermore, Top can run in three different modes:

- CC mode: the traditional cruise control mode. In this mode, components OR and BA are deactivated and ACC equals a traditional CC Controller, i.e., it maintains the desired speed regardless of obstacles and distance
The desired speed can be obtained by information from the driver or by speed-limit regulations, e.g., from road signs or road map information together with the GPS. The presence and distance of a preceding vehicle can be detected by a radar or laser sensor mounted at the front of the car. The detected distance must be sufficiently far to avoid rear-end collisions. When the distance is changed due to the speed discrepancy of the preceding vehicle and the car where the ACC system resides, the ACC system will automatically accelerate or decelerate to keep the distance as per a pre-defined value without the driver's interference. In addition, when the preceding vehicle suddenly brakes or an obstacle abruptly appears in front, a brake-assist function will be activated for a more aggressive braking in such extreme situations.

The component hierarchy of the ACC system is shown in Figure 10.1, including the following components:

- **Speed Limit (SL):** derives the desired speed.
- **Object Recognition (OR):** detects the presence of a preceding vehicle and calculates the relative speed and the distance to the preceding vehicle.
- **ACC Controller (ACC):** maintains both the desired speed and distance. As the core of the ACC system, it consists of two subcomponents: **Distance Controller (DC)** and **Speed Controller (SC).**
- **Brake Assist (BA):** assists the driver to brake in extreme situations. The braking force depends on the relative speed and the distance to the preceding vehicle.
- **HMI Output (HMI):** displays information related to the ACC system to the driver.
- **Speed Controller (SC):** controls the vehicle speed.
- **Distance Controller (DC):** provides distance information to SC.

To simplify the presentation, the top component (ACC system) will be called "Top" while the others will be called as per their above abbreviations. Furthermore, Top can run in three different modes:

- **CC mode:** the traditional cruise control mode. In this mode, components OR and BA are deactivated and ACC equals a traditional CC Controller, i.e., it maintains the desired speed regardless of obstacles and distance information. The inner component connections of Top in this mode are depicted in Figure 10.2(a). Top can be manually controlled via two input ports: "Pedals" and "CC/ACC switch". The former is used to regulate the vehicle speed while the latter is used to manually switch between CC mode and ACC mode.

- **ACC mode:** the adaptive cruise control mode, in which both the desired speed and distance are maintained. Compared with CC mode, this mode activates OR, however, BA is still deactivated. The inner component connections of Top in this mode are depicted in Figure 10.2(b).

- **EMERGENCY mode:** a mode in which BA is activated to help the driver with the braking process. The inner component connections of Top in this mode are depicted in Figure 10.2(c).

Component ACC supports the same modes as Top: **Cc, Acc** and **Emergency** (lowercase letters distinguish them from the modes supported by Top). The inner component connections of ACC are depicted in Figure 10.3 which also marks the port names of ACC and its subcomponents. Component DC is deactivated when ACC is in **Cc** mode, and activated when ACC is in **Acc** or **Emergency** mode. As indicated by white, black and grey colors, SC has three different mode-specific behaviors which correspond to its supported modes **Basic, Advanced, Brake.**
10.1.2 Component specification

Let $\alpha$, $\beta$ and $\gamma$ denote the mode-specific behaviors of SC in modes Basic, Advanced and Brake, respectively. Suppose that SC has a mode-dependent property: WCET (denoted as $\text{wcet}$) such that $SC.\text{wcet} = 50$ when SC runs in Basic; $SC.\text{wcet} = 75$ when SC runs in Advanced; and $SC.\text{wcet} = 100$ when SC runs in Brake. Also suppose SC has a mode-independent property: memory consumption (denoted as $\text{mem}$) which is always 40 irrespective of mode. Abiding by the formal specification of a multi-mode primitive component in Section 2.2 of Chapter 2, we define SC by the following tuple:

$$
\langle SC.\text{P}, SC.\text{p}\text{MS}, SC.\tilde{M}, SC.\text{B}, SC.\text{MI}, SC.\text{MD}, SC.\text{MSRM}, SC.\text{MB}, SC.\text{MP}\rangle
$$

where $SC.\text{MSRM}$ represents the MSRM of SC, and $SC.\text{P} = \{SC.\text{p}_0\text{in}, SC.\text{p}_1\text{in}, SC.\text{p}_2\text{in}, SC.\text{p}_3\text{in}, SC.\text{p}_0\text{out}, SC.\text{p}_1\text{out}, SC.\text{p}_2\text{out}\}$. $SC.\text{B} = \{\alpha, \beta, \gamma\}$, $SC.\text{MI} = \{\text{mem} = 40\}$, $SC.\text{MD} = \{\text{wcet}\}$, $SC.\text{MB} = \{\text{Basic} \rightarrow \alpha, \text{Advanced} \rightarrow \beta, \text{Brake} \rightarrow \gamma\}$, and $SC.\text{MP} = \{(\text{Basic}, \text{wcet}) \rightarrow 50, (\text{Advanced}, \text{wcet}) \rightarrow 75, (\text{Brake}, \text{wcet}) \rightarrow 100\}$. $SC.\tilde{M} = \langle SC.\text{M}, SC.\text{m}_0, SC.\text{m}\rangle$.
10.1 An Adaptive Cruise Control system

Figure 10.2: The ACC system in different modes

10.1.2 Component specification

Let $\alpha$, $\beta$, and $\gamma$ denote the mode-specific behaviors of SC in modes Basic, Advanced, and Brake, respectively. Suppose that SC has a mode-dependent property: WCET (denoted as $\text{SC.wcet}$) such that $\text{SC.wcet} = 50$ when SC runs in Basic; $\text{SC.wcet} = 75$ when SC runs in Advanced; and $\text{SC.wcet} = 100$ when SC runs in Brake. Also suppose SC has a mode-independent property: memory consumption (denoted as $\text{mem}$) which is always 40 irrespective of mode.

Abiding by the formal specification of a multi-mode primitive component in Section 2.2 of Chapter 2, we define SC by the following tuple:

$$< \text{SC.P}, \text{SC.p}^{MS}, \text{SC.}\tilde{M}, \text{SC.B}, \text{SC.MI}, \text{SC.MD}, \text{SC.MSRM}, \text{SC.MB}, \text{SC.MP} >$$

where $\text{SC.MSRM}$ represents the MSRM of SC, and

$$\text{SC.P} = \{\text{SC.p}^{0}_{in}, \text{SC.p}^{1}_{in}, \text{SC.p}^{2}_{in}, \text{SC.p}^{3}_{in}, \text{SC.p}^{0}_{out}, \text{SC.p}^{1}_{out}, \text{SC.p}^{2}_{out} \}$$

$$\text{SC.B} = \{\alpha, \beta, \gamma\}$$

$$\text{SC.MI} = \{\text{mem} = 40\}$$

$$\text{SC.MD} = \{\text{wcet}\}$$

$$\text{SC.MB} = \{\text{Basic} \rightarrow \alpha, \text{Advanced} \rightarrow \beta, \text{Brake} \rightarrow \gamma\}$$

$$\text{SC.MP} = \{((\text{Basic, wcet}) \rightarrow 50, (\text{Advanced, wcet}) \rightarrow 75, (\text{Brake, wcet}) \rightarrow 100\}$$

$$\text{SC.}\tilde{M} = < \text{SC.M}, \text{SC.m}^{0}, \text{SC.m} >$$

Figure 10.3: The ACC Controller in different modes
where

$$SC.\mathcal{M} = \{\text{Basic}, \text{Advanced}, \text{Brake}\}$$
$$SC.m^0 = \text{Basic}$$

Furthermore, ACC has a mode-dependent property: CPU consumption (denoted as $cpu$) such that $ACC.cpu = 60$ when ACC runs in mode $Cc$; $ACC.cpu = 120$ when ACC runs in $Acc$; and $ACC.cpu = 150$ when ACC runs in $Emergency$. ACC also has a mode-independent property: activation period (denoted as $T$) such that $ACC.T = 200$ irrespective of mode. Abiding by the formal specification of a multi-mode composite component in Section 2.3 of Chapter 2, we define ACC by the following tuple:

$$<ACC.\mathcal{P}, ACC.p^{MS}_{in}, ACC.p^{MS}_{out}, ACC.\mathcal{M}, ACC.SC, ACC.CN, ACC.MI, ACC.MD, ACC.MSRM, ACC.ASC, ACC.ACN, ACC.MP>$$

where $ACC.MSRM$ represents the MSRM of ACC, and

$$ACC.\mathcal{P} = \{ACC.p^0_{in}, ACC.p^1_{in}, ACC.p^2_{in}, ACC.p^3_{in}, ACC.p^4_{in}, ACC.p^0_{out}, ACC.p^1_{out}, ACC.p^2_{out}\}$$
$$ACC.SC = \{SC, DC\}$$
$$ACC.CN = \{(ACC.p^0_{in}, SC.p^0_{in}), (ACC.p^1_{in}, SC.p^1_{in}), (ACC.p^2_{in}, SC.p^2_{in}), (ACC.p^3_{in}, SC.p^3_{in}), (ACC.p^4_{in}, SC.p^4_{in}), (SC.p^0_{out}, ACC.p^0_{out}), (SC.p^1_{out}, ACC.p^1_{out}), (SC.p^2_{out}, ACC.p^2_{out}), (DC.p^0_{out}, SC.p^0_{in}), (DC.p^1_{out}, SC.p^1_{in}), (ACC.p^0_{in}, DC.p^0_{in}), (ACC.p^1_{in}, DC.p^1_{in})\}$$
$$ACC.MI = \{T = 200\}$$
$$ACC.MD = \{cpu\}$$
$$ACC.ASC = \{Cc \rightarrow \{SC\}, Acc \rightarrow \{SC, DC\}, Emergency \rightarrow \{SC, DC\}\}$$
\[
ACC.ACN = \{ Cc \rightarrow \{(ACC.p_{in}^{0}, SC.p_{in}^{0}), (ACC.p_{in}^{1}, SC.p_{in}^{1})\},
\]
\[
(ACC.p_{in}^{2}, SC.p_{in}^{2}), (SC.p_{out}^{0}, ACC.p_{out}^{0}),
\]
\[
(SC.p_{out}^{0}, ACC.p_{out}^{1}), (SC.p_{out}^{1}, ACC.p_{out}^{2})\},
\]
\[
Acc \rightarrow \{(ACC.p_{in}^{0}, SC.p_{in}^{0}), (ACC.p_{in}^{1}, SC.p_{in}^{1})\},
\]
\[
(ACC.p_{in}^{2}, SC.p_{in}^{2}), (SC.p_{out}^{0}, ACC.p_{out}^{0}),
\]
\[
(SC.p_{out}^{0}, ACC.p_{out}^{1}), (SC.p_{out}^{1}, ACC.p_{out}^{2})\},
\]
\[
(SC.p_{out}^{2}, DC.p_{in}^{2}), (DC.p_{out}^{0}, SC.p_{in}^{3}),
\]
\[
(ACC.p_{in}^{3}, DC.p_{in}^{0}), (ACC.p_{in}^{4}, DC.p_{in}^{1})\},
\]
\[
Emergency \rightarrow \{(ACC.p_{in}^{1}, SC.p_{in}^{1}), (ACC.p_{in}^{2}, SC.p_{in}^{2})\},
\]
\[
(SC.p_{out}^{0}, ACC.p_{out}^{0}), (SC.p_{out}^{0}, ACC.p_{out}^{1}),
\]
\[
(SC.p_{out}^{0}, ACC.p_{out}^{2}), (SC.p_{out}^{2}, DC.p_{in}^{2}),
\]
\[
(DC.p_{out}^{0}, SC.p_{in}^{3}), (ACC.p_{in}^{3}, DC.p_{in}^{0}),
\]
\[
(ACC.p_{in}^{4}, DC.p_{in}^{1})\}
\]
\[
ACC.MP = \{(Cc, cpu) \rightarrow 60, (Acc, cpu) \rightarrow 120,
\]
\[
(Emergency, cpu) \rightarrow 150\}
\]
\[
ACC.\tilde{M} = < ACC.\tilde{M}, ACC.m^{0}, ACC.m, ACC.M_{SC}, ACC.m_{SC}^{0},
\]
\[
ACC.m_{SC}, ACC.MM >
\]

where \(ACC.MM\) represents the mode mapping of ACC by a set of MMAs. Let \(ON\_DC\) be the running mode of DC when ACC is in \(Acc\) and \(Emergency\), then

\[
ACC.M = \{Cc, Acc, Emergency\}
\]
\[
ACC.m^{0} = Cc
\]
\[
ACC.M_{SC} = \{SC \rightarrow \{Basic, Advanced, Brake\}, DC \rightarrow \{ON\_DC\}\}
\]
\[
ACC.m_{SC}^{0} = \{SC \rightarrow Basic, DC \rightarrow Deactivated\}
\]

### 10.1.3 Mode mapping specification

The ACC system consists of two composite components: Top and ACC. Their mode mappings are initially confined by tables 10.1 and 10.2. Note that SL, OR, HMI, BA and DC are all single-mode components. SL and HMI are always activated, while OR, BA and DC can be deactivated under certain conditions. The modes of Component SC, i.e. Basic, Advanced and Brake are mapped to \(Cc, Acc\) and \(Emergency\) of ACC.

The system has two MSSs. One is Top which can request to switch between \(CC\) and \(ACC\) instructed by the driver. Let \(Top : CC \rightarrow ACC\) be a scenario \(k_{1}\) and \(Top : ACC \rightarrow CC\) be a scenario \(k_{2}\). When Top is in mode \(EMERGENCY\), it can only switch to \(CC\) manually. Let \(Top : EMERGENCY \rightarrow CC\) be a scenario \(k_{3}\). The other MSS is ACC which can automatically request to switch
Component | Modes  
|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|
| ACC System (Top)    | CC                  | ACC                | EMERGENCY          | Speed Limit (SL)    | ON_SL               | Object Recognition (OR) | Deactivated  
| HMI                 | ON_HMI             | ACC                | Emergency          | ACC Controller (ACC) | Cc                  | Acc                 | Emergency           | Brake Assist (BA)  | Deactivated          | ON_BA               |

Table 10.1: The mode mapping table of the ACC system

| Component | Modes  
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC Controller (ACC)</td>
<td>Cc</td>
<td>Acc</td>
<td>Emergency</td>
<td>Distance Controller (DC)</td>
</tr>
</tbody>
</table>

Table 10.2: The mode mapping table of the ACC Controller

between Acc and Emergency. Let ACC : Acc → Emergency be a scenario \( k_4 \) and ACC : Emergency → Acc be a scenario \( k_5 \). When ACC is in mode Acc, if the distance to the preceding vehicle or an obstacle in front is closer than a threshold and the relative speed exceeds an upper bound, ACC will request to switch to Emergency. Similarly, when both the distance and relative speed return to normal after a successful braking process, ACC will request to switch back to Acc.

Incorporating the five scenarios triggered by both MSSs into the basic mode mappings in tables 10.1 and 10.2, we describe the complete mode mappings of Top and ACC as MMAs. Figure 10.4 presents the parent MMA of Top while Figure 10.5 presents the child MMAs associated with the subcomponents of Top. At one level down, figures 10.6 and 10.7 illustrate the parent MMA of ACC and the child MMAs of DC and SC, which jointly define the mode mapping of ACC. All MMAs of the ACC system comply with the formal semantics formulated in Section 3.2 of Chapter 3.
Component Modes

ACC System (Top)
- CC
- ACC EMERGENCY
- Speed Limit (SL)
- Object Recognition (OR)

ACC Controller (ACC)
- Cc
- ACC Emergency
- Brake Assist (BA)
- Deactivated

Table 10.1: The mode mapping table of the ACC system

Distance Controller (DC)
- Deactivated

Speed Controller (SC)
- Basic Advanced Brake

Table 10.2: The mode mapping table of the ACC Controller

Table between Acc and Emergency. Let ACC: Acc → Emergency be a scenario and ACC: Emergency → Acc be a scenario.

When ACC is in mode Acc, if the distance to the preceding vehicle or an obstacle in front is closer than a threshold and the relative speed exceeds an upper bound, ACC will request to switch to Emergency. Similarly, when both the distance and relative speed return to normal after a successful braking process, ACC will request to switch back to Acc.

Incorporating the five scenarios triggered by both MSSs into the basic mode mappings in tables 10.1 and 10.2, we describe the complete mode mappings of Top and ACC as MMAs. Figure 10.4 presents the parent MMA of Top while Figure 10.5 presents the child MMAs associated with the subcomponents of Top. At one level down, figures 10.6 and 10.7 illustrate the parent MMA of ACC and the child MMAs of DC and SC, which jointly define the mode mapping of ACC. All MMAs of the ACC system comply with the formal semantics formulated in Section 3.2 of Chapter 3.

10.1.4 Mode switch at runtime

Each component of the ACC system runs its own MSRM to cooperatively handle mode switch at runtime. To demonstrate how a mode switch is handled
in the ACC system, consider the two scenarios $k_4$ and $k_5$ triggered by ACC. Scenario $k_4$ is triggered when ACC in mode $Acc$ detects an emergency event.
that entails an emergency brake. Hence $k_4$ should be treated as an emergency scenario. The emergency mode switch for $k_4$ is depicted in Figure 10.8. ACC triggers $k_4$ by issuing an $ems^{k_4}$ which is propagated to all the Type A components according to the EMSP protocol (Definition 26 in Chapter 4). The mode mappings of Top and ACC specified in figures 10.4-10.7 suggest that Top, ACC, BA, and SC are the Type A components for $k_4$ while the others (SL, OR, HMI, and DC) are Type B components. After the propagation of the $ems^{k_4}$, each Type A switches mode by the mode-switch dependency rule (Definition 15 of Chapter 4).

![Figure 10.8: The mode switch for an emergency scenario $k_4$](image)

Once a hazard is successfully averted as the ACC system switches to the EMERGENCY mode due to the triggering of $k_4$, ACC will automatically request to switch from mode Emergency to Acc by triggering $k_5$. The mode mappings in figures 10.4-10.7 indicate that $k_4$ and $k_5$ have the same set of Type A components. However, $k_5$ is a non-emergency scenario which should be propagated by the MSP protocol (Definition 14 in Chapter 4) instead. The mode-switch process based on $k_5$ is depicted in Figure 10.9. Top is identified as the MSDM. Moreover, the $msq^{k_5}$ from ACC to SC is delayed by the ongoing atomic execution in SC, as ACC applies the MSQ delaying rule (Definition 17 in Chapter 4). After the propagation of $k_5$, each Type A component follows the same mode-switch dependency rule to switch mode.

### 10.1.5 Mode-switch timing analysis

In this section we perform a mode-switch timing analysis for the ACC system. As an example, we apply the timing analysis in Chapter 6 to the mode-switch process for $k_5$ depicted in Figure 10.9. Reusing the notations in Chapter 6,
Figure 10.9: The mode switch for a non-emergency scenario $k_5$

ACC is the MSS $\alpha$ for $k_5$, with Top as the MSDM $\beta$, and $\mathcal{C}_\beta^\alpha = \emptyset$ (i.e., there are no other components between Top and ACC). Let SC be an AEG $\mathcal{G}_1$ with atomic execution. We assume that the transmission time of each primitive is 1 and $AE_{\mathcal{G}_1}$ is 2. In addition, Table 10.3 lists the other key timing factors of each Type A component for $k_5$, including Top, ACC, BA, and SC. All the timing factors in Table 10.3 plus $AE_{\mathcal{G}_1}$ are addressed in the illustration of our timing analysis in Figure 10.10 which refines Figure 10.9 with timing annotations.

<table>
<thead>
<tr>
<th>Component</th>
<th>MSDT</th>
<th>MMT</th>
<th>SCT</th>
<th>RCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top</td>
<td>–</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>ACC</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>BA</td>
<td>–</td>
<td>–</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>SC</td>
<td>–</td>
<td>–</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 10.3: Mode-switch timing factors of the ACC system

With reference to the timing analysis in Chapter 6, the complete mode-switch process for a non-emergency scenario without the interference from
other scenarios is divided into three non-overlapping and continuous phases. Shown in Figure 10.10, Phase 1 starts when the MSS ACC triggers $k_5$ and ends when the MSDM Top is ready to propagate an $msq_{k_5}$. Referring to (6.2), the time $T_1$ spent in Phase 1 is

$$T_1 = ACC.msd + (Top.t_{msr} + Top.mmt + Top.sct) = 2 + (1 + 3 + 3) = 9$$

(10.1)

Phase 2 starts when Top starts to propagate an $msq_{k_5}$ and ends when Top is ready to propagate an $msi_{k_5}$. In order to derive the time spent in Phase 2, $T_2$, the QRT of Type A primitive components, including BA and SC, should be
calculated by (6.4):

\[
BA.qrt = BA.sct = 3 \quad (10.2)
\]

\[
SC.qrt = SC.sct = 3 \quad (10.3)
\]

Then \(SC.qrt\) can be used to derive \(ACC.qrt\) using (6.6):

\[
ACC.qrt = ACC.sct + ACC.mmt + (AE_{q_1} + ACC.t_{msq} + SC.qrt + ACC.t_{ok})
= 2 + 2 + (2 + 1 + 3 + 1) = 11 \quad (10.4)
\]

Using the results from (10.2) and (10.4) and with the help of (6.7), we get

\[
T_2 = \max \{Top.t_{msq} + ACC.qrt + Top.t_{ok}, Top.t_{msq} + BA.qrt + Top.t_{ok}\} = \max \{1 + 11 + 1, 1 + 3 + 1\} = 13 \quad (10.5)
\]

Phase 3 starts when \(Top\) propagates an \(msi_{k_5}\) and ends when \(Top\) completes its mode switch for \(k_5\). The calculation of \(T_3\) starts with the MST of BA and SC (see (6.10)):

\[
BA.mst = BA.rct = 2 \quad (10.6)
\]

\[
SC.mst = SC.rct = 4 \quad (10.7)
\]

Then the MST of ACC can be obtained using (6.11):

\[
ACC.mst = \max \{ACC.rct, ACC.t_{msi} + SC.mst + ACC.t_{msc}\}
= \max \{5, 1 + 4 + 1\} = 6 \quad (10.8)
\]

Using the results from (10.6) and (10.8) and with the help of (6.13), we get

\[
T_3 = \max \{Top.rct, Top.t_{msi} + ACC.mst + Top.t_{msc}, Top.t_{msi} + BA.mst + Top.t_{msc}\} = \max \{6, 1 + 6 + 1, 1 + 2 + 1\} = 8 \quad (10.9)
\]

Finally, by (6.14), the complete mode-switch time of the ACC system for \(k_5\) is
\[ T = T_1 + T_2 + T_3 = 9 + 13 + 8 = 30 \quad (10.10) \]

This result is consistent with Figure 10.10 which also indicates that the total mode-switch time is 30.

### 10.1.6 Discussion

The ACC system has been previously studied by several researchers. What is most interesting is to compare our MSL with the ACC system developed with SaveCCM components [2]. Despite the resemblance between the component structures of the ACC system in the thesis and in [2], SaveCCM handles mode switch in a completely different way from MSL.

In SaveCCM, a special connector called "switch" is used to select the right outgoing components in different modes. This idea is widely adopted to deal with diverse structures in a component-based system. Nonetheless, it is rather rudimentary in that it lacks a systematic logic and cannot properly handle the mode correlation between different components. The "switch" connector must be manually implemented wherever a connection change is required. As a consequence, it becomes an arduous task for complex component connections. Many "switch" connectors are subject to manual revision even in case of any slight change in the component hierarchy or component connections.

In contrast to SaveCCM, MSL is more advanced and flexible owing to reuse of multi-mode components, distributed mode-switch handling, and mode transformation opportunities. Although the ACC system presented in this chapter is composed by a limited number of components, MSL is highly scalable and suitable for developing complex systems. The scalability of MSL is reflected in the mode mapping mechanism and the MSRM, both of which are distributed in the component hierarchy.

### 10.2 A healthcare monitoring system

The healthcare monitoring system that we will consider monitors the health condition of a patient by sending video and audio data from the monitoring room to a local or remote health centre. An alarm is triggered when the system detects a dangerous situation for the patient, such as falling or suffering from a heart attack, or when the patient presses an emergency button.
10.2.1 System description

The healthcare monitoring system studied in this thesis consists of two subsystems: Data Acquisition Subsystem and Monitoring Subsystem. The Data Acquisition Subsystem uses cameras and microphones to collect video and audio data from a ward or a private home. The data is encoded, encrypted, and transmitted to the Monitoring Subsystem over a reliable network. The data arriving at the Monitoring Subsystem is decrypted, decoded, and reported to the health center. Video and audio data are decoded separately. The video is displayed on a screen, while the audio is played by a speaker. The Monitoring Subsystem also includes an alarm which is triggered when the patient encounters a dangerous situation.

Our focus is on the Monitoring Subsystem (MoS), which is a multi-mode system composed by multi-mode components. The component structure of the system is illustrated in Figure 10.11 where its component hierarchy is presented on the left and component connections are depicted on the right. The system is represented by Component MoS at the top level, composed by three subcomponents: DaD for data decryption, the multimedia decoder MuD, and EvA for event analysis. The system can run in two different modes: Regular monitoring mode (denoted as $Rm$) and Attention mode (denoted as $Att$).

The system runs in mode $Rm$ if nothing special happens to the patient. This mode has very low requirement on video resolution and there is no need to transmit audio data. Hence it is assumed that only video data is transmitted and a fast video encoding/decoding algorithm is used to provide medium video quality which can be maintained by low CPU and bandwidth usage. Shown in Figure 10.11, the video data from the network is first decrypted by DaD and then decoded by MuD, which sends the decoded video data to the display in the health center. Represented by the grey color, EvA is deactivated when MoS runs in $Rm$. Meanwhile, DaD runs in a Regular mode $R1$, while MuD runs in a Regular decoding mode $Rd$. The internal structure of MuD for mode $Rd$ is also depicted in Figure 10.11. MuD consists of three subcomponents: VAE for video/audio extraction, a video decoder ViD and an audio decoder AuD. While MuD runs in $Rd$, ViD is its only activated subcomponent running in a Regular video decoding mode $Rvd$, since MuD receives only decrypted video data.

When an incident is detected by the Data Acquisition Subsystem, both subsystems will switch to an Attention mode $Att$. This mode intends to raise the attention of the health center. Both video and audio data are expected to be transmitted. Moreover, the video resolution should be higher, thus requiring a different encoding/decoding algorithm. Component EvA is activated, running
10.2 A healthcare monitoring system

The healthcare monitoring system studied in this thesis consists of two subsystems: Data Acquisition Subsystem and Monitoring Subsystem. The Data Acquisition Subsystem uses cameras and microphones to collect video and audio data from a ward or a private home. The data is encoded, encrypted, and transmitted to the Monitoring Subsystem over a reliable network. The data arriving at the Monitoring Subsystem is decrypted, decoded, and reported to the health center. Video and audio data are decoded separately. The video is displayed on a screen, while the audio is played by a speaker. The Monitoring Subsystem also includes an alarm which is triggered when the patient encounters a dangerous situation.

Our focus is on the Monitoring Subsystem (MoS), which is a multi-mode system composed by multi-mode components. The component structure of the system is illustrated in Figure 10.11 where its component hierarchy is presented on the left and component connections are depicted on the right. The system is represented by Component MoS at the top level, composed by three subcomponents: DaD for data decryption, the multimedia decoder MuD, and EvA for event analysis. The system can run in two different modes: Regular monitoring mode (denoted as $R_m$) and Attention mode (denoted as $Att$).

The system runs in mode $R_m$ if nothing special happens to the patient. This mode has very low requirement on video resolution and there is no need to transmit audio data. Hence it is assumed that only video data is transmitted and a fast video encoding/decoding algorithm is used to provide medium video quality which can be maintained by low CPU and bandwidth usage. Shown in Figure 10.11, the video data from the network is first decrypted by DaD and then decoded by MuD, which sends the decoded video data to the display in the health center. Represented by the grey color, EvA is deactivated when MoS runs in $R_m$. Meanwhile, DaD runs in a Regular mode $R_1$, while MuD runs in a Regular decoding mode $R_d$. The internal structure of MuD for mode $R_d$ is also depicted in Figure 10.11. MuD consists of three subcomponents: VAE for video/audio extraction, a video decoder ViD and an audio decoder AuD. While MuD runs in $R_d$, ViD is its only activated subcomponent running in a Regular video decoding mode $R_{vd}$, since MuD receives only decrypted video data.

When an incident is detected by the Data Acquisition Subsystem, both subsystems will switch to an Attention mode $Att$. This mode intends to raise the attention of the health center. Both video and audio data are expected to be transmitted. Moreover, the video resolution should be higher, thus requiring a different encoding/decoding algorithm. Component EvA is activated, running in a Regular mode $R_2$, to analyze the detected event and may trigger an alarm when necessary. Due to the growing video data size and the additional audio data, the network load will be substantially increased. Depending on the current network condition, MuD may run in an Enhanced decoding mode $Ed$ or a Degraded QoS mode $Dq$. Under satisfying network conditions, MuD runs in $Ed$, with all its subcomponents activated. The subcomponent VAE runs in a Regular mode $R_3$, separating decrypted video and audio data and sending them to ViD and AuD, respectively. To produce higher quality video, the video decoder ViD runs in an Enhanced video decoding mode $Evd$ by applying a different video decoding algorithm. This is represented by the grey color of ViD in Figure 10.11. The audio decoder AuD runs in a Regular audio decoding mode $Rad$. However, if the network condition deteriorates to a certain level, it may become unsuitable to transmit both video and audio data. A possible strategy is to skip the transmission of audio data to maintain the video quality which could be more important. As a result, MuD switches to the mode $Dq$, in which AuD is deactivated to skip audio decoding. When the network condition returns to normal, MuD will automatically switch back to $Ed$. Note that VAE...
only receives video data while MuD runs in $Dq$. This implies that VAE can be deactivated in the same way as AuD. However, VAE remains activated when MuD runs in $Dq$ to avoid frequent activation and deactivation. Without audio data, VAE simply forwards the video data to ViD.

### 10.2.2 Component specification

To demonstrate the formal specification of multi-mode components, let’s formally define the primitive component ViD and the composite component MuD in the example illustrated in Figure 10.11. Figure 10.12 displays the inner component connections of MuD which runs in the mode $Ed$, with the port names of MuD and its subcomponents marked in red. Let $\alpha$ and $\beta$ denote the video decoding algorithms applied by ViD for its modes $Rvd$ and $Evd$, respectively. Suppose that ViD has a mode-independent property, memory consumption (denoted by $mem$) that is the same for both modes, e.g. $mem = 10$, and ViD also has a mode-dependent property, CPU consumption (denoted by $cpu$) that is different for different modes, e.g. $cpu = 5$ for $Rvd$ and $cpu = 8$ for $Evd$. Then as a primitive component, ViD can be formally defined by the tuple,

\[
< \text{ViD.P}, \text{ViD.p}^\text{MS}, \text{ViD.}\tilde{M}, \text{ViD.M}, \text{ViD.MM}, \text{ViD.MD}, \text{ViD.MSRM}, \text{ViD.MB}, \text{ViD.MP} >
\]

where $\text{ViD.MSRM}$ represents the MSRM of ViD, and

\[
\begin{align*}
\text{ViD.P} & = \{ vid.p^0_{in}, vid.p^0_{out} \} \\
\text{ViD.B} & = \{ \alpha, \beta \} \\
\text{ViD.MI} & = \{ mem = 10 \} \\
\text{ViD.MD} & = \{ cpu \} \\
\text{ViD.MB} & = \{ Rvd \rightarrow \alpha, Evd \rightarrow \beta \} \\
\text{ViD.MP} & = \{ (Rvd, cpu) \rightarrow 5, (Evd, cpu) \rightarrow 10 \} \\
\text{ViD.}\tilde{M} & = < \text{ViD.M}, \text{ViD.m}^0, \text{ViD.m} >
\end{align*}
\]

where

\[
\begin{align*}
\text{ViD.M} & = \{ Rvd, Evd \} \\
\text{ViD.m}^0 & = Rvd
\end{align*}
\]

Now suppose the composite component MuD has WCET (denoted as $wcet$) as $\text{MuD.MD}$ such that $wcet = 50$ for mode $Rd$, $wcet = 100$ for mode $Ed$ and $wcet = 75$ for mode $Dq$. Also, MuD has a mode-independent property, i.e.
its activation period $T = 100$ for all modes. As a composite component, MuD can be formally defined by the tuple,

$$<\text{MuD}.\mathcal{P}, \text{MuD}.p^\text{MS}, \text{MuD}.p^\text{MS}_\text{in}, \text{MuD}.\tilde{N}, \text{MuD}.\mathcal{S}, \text{MuD}.\mathcal{C}, \text{MuD}.\mathcal{N}, \text{MuD}.\mathcal{M}, \text{MuD}.\mathcal{D}, \text{MuD}.\text{MSRM}, \text{MuD}.\text{ASC}, \text{MuD}.\mathcal{ACN}, \text{MuD}.\text{MP}>$$

where $\text{MuD}.\text{MSRM}$ represents the MSRM of $\text{MuD}$, and

$$\text{MuD}.\mathcal{P} = \{\text{mud}.p^0_\text{in}, \text{mud}.p^0_\text{out}, \text{mud}.p^1_\text{out}\}$$
$$\text{MuD}.\mathcal{SC} = \{\text{VAE}, \text{ViD}, \text{AuD}\}$$
$$\text{MuD}.\mathcal{CN} = \{(\text{mud}.p^0_\text{in}, \text{vae}.p^0_\text{in}), (\text{vae}.p^0_\text{out}, \text{vid}.p^0_\text{in}), (\text{vae}.p^1_\text{out}, \text{aud}.p^0_\text{in}), (\text{vid}.p^0_\text{out}, \text{mud}.p^0_\text{out}), (\text{aud}.p^0_\text{out}, \text{mud}.p^1_\text{out})\}$$
$$\text{MuD}.\mathcal{MI} = \{T = 100\}$$
$$\text{MuD}.\mathcal{MD} = \{\text{wcet}\}$$
$$\text{MuD}.\text{ASC} = \{\text{Rd} \rightarrow \{\text{ViD}\}, \text{Ed} \rightarrow \{\text{VAE, ViD, AuD}\}, \text{Dq} \rightarrow \{\text{VAE, ViD}\}\}$$
$$\text{MuD}.\text{ACN} = \{\text{Rd} \rightarrow \{(\text{mud}.p^0_\text{in}, \text{vid}.p^0_\text{in}), (\text{vid}.p^0_\text{out}, \text{mud}.p^0_\text{out})\}, \text{Ed} \rightarrow \{(\text{mud}.p^0_\text{in}, \text{vae}.p^0_\text{in}), (\text{vae}.p^0_\text{out}, \text{vid}.p^0_\text{in}), (\text{vid}.p^0_\text{out}, \text{mud}.p^0_\text{out}), (\text{aud}.p^0_\text{out}, \text{mud}.p^1_\text{out})\}, \text{Dq} \rightarrow \{(\text{mud}.p^0_\text{in}, \text{vae}.p^0_\text{in}), (\text{vae}.p^0_\text{out}, \text{vid}.p^0_\text{in}), (\text{vid}.p^0_\text{out}, \text{mud}.p^0_\text{out})\}\\}$$
$$\text{MuD}.\text{MP} = \{(\text{Rd, wcet}) \rightarrow 50, (\text{Ed, wcet}) \rightarrow 100, (\text{Dq, wcet}) \rightarrow 75\}$$
$$\text{MuD}.\tilde{N} = <\text{MuD}.\mathcal{M}, \text{MuD}.m^0, \text{MuD}.m, \text{MuD}.m_S^\mathcal{S}, \text{MuD}.m^0_{S^\mathcal{S}}, \text{MuD}.m_S^\mathcal{S}, \text{MuD}.\mathcal{MM}>$$

where $\text{MuD}.\mathcal{MM}$ is the mode mapping of $\text{MuD}$, and

$$\text{MuD}.\mathcal{M} = \{\text{Rd, Ed, Dq}\}$$
$$\text{MuD}.m^0 = \text{Rd}$$
$$\text{MuD}.m_S^\mathcal{S} = \{\text{VAE} \rightarrow \{\text{R3}\}, \text{ViD} \rightarrow \{\text{Rvd, Evd}\}, \text{AuD} \rightarrow \{\text{Rad}\}\}$$
$$\text{MuD}.m^0_{S^\mathcal{S}} = \{\text{VAE} \rightarrow \text{Deactivated, ViD} \rightarrow \text{Rvd, AuD} \rightarrow \text{Deactivated}\}$$
10.2.3 Mode mapping specification

Based on the system description in Section 10.2.1, Tables 10.4 and 10.5 present the basic mode mappings of composite components MoS and MuD of the Monitoring Subsystem.

<table>
<thead>
<tr>
<th>Component</th>
<th>Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MoS</td>
<td>Rm</td>
</tr>
<tr>
<td>DaD</td>
<td>R1</td>
</tr>
<tr>
<td>MuD</td>
<td>Rd</td>
</tr>
<tr>
<td>EvA</td>
<td>Deactivated</td>
</tr>
</tbody>
</table>

Table 10.4: The mode mapping table of MoS

<table>
<thead>
<tr>
<th>Component</th>
<th>Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MuD</td>
<td>Rd</td>
</tr>
<tr>
<td>VAE</td>
<td>Deactivated</td>
</tr>
<tr>
<td>ViD</td>
<td>Rvd</td>
</tr>
<tr>
<td>AuD</td>
<td>Deactivated</td>
</tr>
</tbody>
</table>

Table 10.5: The mode mapping table of MuD

The Monitoring Subsystem consists of two MSSs: MoS and MuD. When a dangerous situation of the patient is automatically detected or when the patient manually presses an emergency button, MoS will trigger a scenario $k_1$, requesting to switch from mode $Rm$ to mode $Att$. While running in $Att$, MoS may also trigger another scenario $k_2$ to switch from $Att$ back to $Rm$, as the health centre staff start to react and dismiss the alarm. Furthermore, when MoS runs in $Att$, Table 10.4 indicates that MuD can run in either $Ed$ or $Dq$. As the other MSS, MuD can trigger a scenario $k_3$, requesting to switch from $Ed$ to $Dq$ when the network bandwidth falls below a certain threshold. MuD can also trigger another scenario $k_4$ to switch from $Dq$ back to $Eq$ when the network bandwidth returns to normal.

Taking all the four scenarios triggered by MoS and MuD into account, figures 10.13 and 10.14 display the set of MMAs of MoS while figures 10.15 and 10.16 display the set of MMAs of MuD. Note that the parent MMA of MoS in Figure 10.13 uses boolean expressions $guard_1$ and $guard_2$ to distinguish
the new mode of MuD when MoS triggers $k_1$. If $guard 1$ evaluates to true, MuD will switch to mode $Ed$; if $guard 2$ evaluates to true, MuD will switch to mode $Dq$. For simplification purpose, we assume in this section that $guard 1$ is always true while $guard 2$ is always false.

Figure 10.13: The parent MMA of MoS

In addition, Figure 10.17 further demonstrates the MMA composition for MoS based on the MMAs in figures 10.13 and 10.14. After MMA composition, the mode mapping of MoS becomes a single automaton with three states $s_1$, $s_2$, and $s_3$, where $s_1 = (Rm, R1, Rd, D)$, $s_2 = (Att, R1, Ed, R2)$, and $s_3 = (Att, R1, Dq, R2)$.

### 10.2.4 Mode switch at runtime

Suppose that $k_1$ triggered by MoS is an emergency scenario, while all the other scenarios are non-emergency scenarios. When MoS runs in mode $Rm$, the only possible scenario is $k_1$ triggered by MoS. When MoS runs in mode $Att$ though, the triggering of $k_2$ by MoS may coincide with the concurrent triggering of $k_3$ or $k_4$ by MuD. To demonstrate the handling of concurrent scenarios, suppose $k_2$ and $k_3$ are triggered simultaneously. Then MoS may receive an $msr^{k_3}$ from MuD in the MSR queue before or after it triggers $k_2$ and puts an $msq^{k_2}$ in the MSQ queue. The arrival sequence of the $msq^{k_2}$ and $msr^{k_3}$ affects the subsequent handling of $k_2$ and $k_3$. For instance, Figure 10.18 illustrates the mode switches for $k_2$ and $k_3$ while $k_3$ is handled first. Both scenarios are handled by the MSP protocol and mode-switch dependency rule, assuming the mode mapping specified in Section 10.2.3. During the mode switch for $k_3$, the $msq^{k_2}$ is temporarily buffered in the MSQ queue of MoS. At time $t_0$, MoS completes the mode switch for $k_3$ and starts to handle $k_2$. On the contrary, if $k_2$ is first handled instead, the $msr^{k_3}$ from MuD is temporarily buffered in the MSR queue of MoS during the mode switch of $k_2$. However, since MuD is a
Figure 10.14: The child MMAs of MoS

Figure 10.15: The parent MMA of MuD
Type A component for $k_2$, by the MSR/MSQ queue updating rule, the $msr^{k_3}$ from MuD becomes invalid due to $k_2$. Hence both MoS and MuD will discard the $msr^{k_3}$ and no mode switch is performed for $k_3$. 
10.2.5 Mode transformation

Suppose that all the components of the Monitoring Subsystem are to be deployed on a single physical platform, with the mode information of all components globally accessible. Their mode transformation can be applied to improve runtime mode-switch efficiency. According to Chapter 8, mode transformation is conducted in two sequential steps both of which can be automated and performed at design time. The first step is to construct an MCT based on the mode mapping of each composite component. Figure 10.19 presents the MCT constructed from the mode mapping tables 10.4 and 10.5. The MCT consists of 9 nodes $N_0$–$N_8$ with four depth levels. Represented by the respective paths of the MCT, the three identified system modes are:

$m_1 = N_0 \cup N_1 \cup N_3 \cup N_6$

$m_2 = N_0 \cup N_2 \cup N_4 \cup N_7$

$m_3 = N_0 \cup N_2 \cup N_5 \cup N_8$

Since we assume that the Monitoring Subsystem initially runs in mode $R_m$, $m_1$ is the initial system mode after mode transformation. The global configurations of these system modes are depicted in Figure 10.20.

As the second step of mode transformation, a mode transition graph is derived by adding transitions to the system modes identified in the first step. Shown in Figure 10.21, a CTM table is first obtained as per the specification of all scenarios $k_1$-$k_4$ and the mode mappings of MoS and MuD described by the MMAs in figures 10.13-10.16. From the CTM table, the target system mode after a mode switch can be derived for each scenario. The four scenarios result in 5 transitions of the mode transition graph. Mode transformation improves the runtime mode-switch efficiency of MSL in the sense that a direct transition between two system modes is faster than the stepwise inter-component communication, e.g., in Figure 10.18. Moreover, there is no need for each component to run its MSRM, which reduces runtime overhead.

10.3 Summary

In this chapter, the major principles of MSL are adopted for the design of two multi-mode systems composed by multi-mode components: an ACC system and a healthcare monitoring system.
ponents globally accessible. Their mode transformation can be applied to improve runtime mode-switch efficiency. According to Chapter 8, mode transformation is conducted in two sequential steps both of which can be automated and performed at design time. The first step is to construct an MCT based on the mode mapping of each composite component. Figure 10.19 presents the MCT constructed from the mode mapping tables 10.4 and 10.5. The MCT consists of 9 nodes $N_0-N_8$ with four depth levels. Represented by the respective paths of the MCT, the three identified system modes are:

$$m_1 = N_0 \cup N_1 \cup N_3 \cup N_6$$
$$m_2 = N_0 \cup N_2 \cup N_4 \cup N_7$$
$$m_3 = N_0 \cup N_2 \cup N_5 \cup N_8$$

Since we assume that the Monitoring Subsystem initially runs in mode $Rm$, $m_1$ is the initial system mode after mode transformation. The global configurations of these system modes are depicted in Figure 10.20.

As the second step of mode transformation, a mode transition graph is derived by adding transitions to the system modes identified in the first step. Shown in Figure 10.21, a CTM table is first obtained as per the specification of all scenarios $k_1-k_4$ and the mode mappings of MoS and MuD described by the MMAs in figures 10.13-10.16. From the CTM table, the target system mode after a mode switch can be derived for each scenario. The four scenarios result in 5 transitions of the mode transition graph. Mode transformation improves the runtime mode-switch efficiency of MSL in the sense that a direct transition between two system modes is faster than the stepwise inter-component communication, e.g., in Figure 10.18. Moreover, there is no need for each component to run its MSRM, which reduces runtime overhead.

10.3 Summary

In this chapter, the major principles of MSL are adopted for the design of two multi-mode systems composed by multi-mode components: an ACC system and a healthcare monitoring system.
The ACC system can be manually switched between the traditional cruise control mode and adaptive cruise control mode. The former is featured by the automatic maintenance of a constant speed, while the latter is featured by the automatic maintenance of both speed and the distance to a preceding vehicle. Moreover, the ACC system can automatically switch to an emergency mode, assisting the driver to brake when an emergency situation is encountered.

The healthcare monitoring system, or more exactly its Monitoring Subsystem, initially runs in Regular monitoring mode and can be automatically or manually switched to Attention mode when the patient being monitored is in need of emergency care. The Attention mode is featured by video with higher resolution and additional audio data transmitted to the health centre. However, audio data is discarded under unsatisfying network conditions.

All the multi-mode components of both case studies are consistent with the mode-aware component model. The mode mappings of their composite components are specified by MMAs in accordance with the mode mapping mechanism. For the first case study, we have demonstrated how emergency and non-emergency mode switches are performed by the MSRM running in each component. Moreover, we have provided a mode-switch timing analysis for a non-emergency scenario in the ACC system. For the second case study,
The ACC system can be manually switched between the traditional cruise control mode and adaptive cruise control mode. The former is featured by the automatic maintenance of a constant speed, while the latter is featured by the automatic maintenance of both speed and the distance to a preceding vehicle. Moreover, the ACC system can automatically switch to an emergency mode, assisting the driver to brake when an emergency situation is encountered.

The healthcare monitoring system, or more exactly its Monitoring Subsystem, initially runs in Regular monitoring mode and can be automatically or manually switched to Attention mode when the patient being monitored is in need of emergency care. The Attention mode is featured by video with higher resolution and additional audio data transmitted to the health centre. However, audio data is discarded under unsatisfying network conditions.

All the multi-mode components of both case studies are consistent with the mode-aware component model. The mode mappings of their composite components are specified by MMAs in accordance with the mode mapping mechanism. For the first case study, we have demonstrated how emergency and non-emergency mode switches are performed by the MSRM running in each component. Moreover, we have provided a mode-switch timing analysis for a non-emergency scenario in the ACC system. For the second case study, we have demonstrated MMA composition, the handling of concurrent non-emergency scenarios, and mode transformation.
Figure 10.21: Deriving the mode transition graph of the monitoring subsystem
Chapter 11
Related work

Mode switch, which is also called "mode change", has been widely studied in various research areas, such as CBSE, programming languages and real-time systems. Since mode switch is essentially the change from one configuration to another configuration, we have also investigated numerous references germane to dynamic reconfiguration. Due to the lack of standard terminology and the limitation of our survey, we might be unaware of certain pertinent publications presenting relevant work. Nevertheless, we believe that the most related and influential works are covered in this chapter. Related works are grouped into four categories: design and modeling methods, software component models, programming languages, Dynamic Software Product Line (DSPL), and real-time systems.

11.1 Design and modeling methods

This section summaries a number of techniques for the component-based design and modeling of multi-mode or adaptive systems.

11.1.1 MechatronicUML

MechatronicUML [8] is a modeling language for mechatronic systems with a focus on development of distributed systems and dynamic reconfiguration. Heinzemmann and Becker [59, 60] have extended MechatronicUML to allow the dynamic reconfiguration of MechatronicUML components. Similar to MSL,
**MECHATRONICUML** separates component reconfiguration from its functional behavior. Each **MECHATRONICUML** component has a bidirectional reconfiguration port which resembles the dedicated mode-switch ports specified in our mode-aware component model. The reconfiguration of a **MECHATRONICUML** component is locally managed and executed by two dedicated subcomponents: the Manager and Executor. The Manager of a component is connected to both the component and its subcomponents via their reconfiguration ports. Moreover, the Manager communicates with the Executor via another reconfiguration port. The Manager controls the reconfiguration process, e.g., reconfiguration propagation and decision to make reconfiguration. The reconfiguration decision from the Manager is executed by the Executor which encapsulates a set of predefined reconfiguration rules. When a component triggers a reconfiguration, reconfiguration messages will be propagated stepwise throughout the component hierarchy. The reconfiguration propagation follows a protocol akin to the MSP protocol of MSL. To a large extent, the initial development of our MSP protocol was independent of **MECHATRONICUML**, though we did get inspiration from **MECHATRONICUML** in defining the first propagation phase of a non-emergency scenario: an idea originating from the 2-phase commit protocol for distributed databases [12]. **MECHATRONICUML** requires that reconfiguration should be performed bottom-up; this is congruous with our mode-switch dependency rule. The Manager and Executor jointly take the role of our MSRM. Moreover, the MSRM is much more advanced than the reconfiguration protocol of **MECHATRONICUML**, which provides no solutions to atomic execution, the concurrent triggering of multiple reconfigurations, and emergency events. **MECHATRONICUML** also considers real-time properties during reconfiguration. Compared with the preliminary analysis of the reconfiguration time in **MECHATRONICUML**, our mode-switch timing analysis is more comprehensive.

The major difference between **MECHATRONICUML** and MSL is that **MECHATRONICUML** focuses on component reconfiguration without addressing mode. **MECHATRONICUML** does not pre-define component configurations at design time. Instead, reconfiguration rules are statically specified for each component, allowing more flexible reconfiguration such as adding a new subcomponent or removing an existing subcomponent at runtime. In contrast to **MECHATRONICUML**, MSL emphasizes mode-switch predictability by assuming that all component modes, the corresponding configurations, and mode mappings are specified at design time. In addition, **MECHATRONICUML** provides the formal modeling of reconfiguration based on component story diagrams [117], which is a formal specification of structural transformations. MSL focuses more on the mode-switch coordination of different components rather than the
11.1 Design and modeling methods

mode switch of an individual component.

11.1.2 The oracle-based approach for mode switch using property networks

Pop et al. propose an oracle-based approach [103] to reuse of multi-mode components. Multi-mode components are informally defined in [103], however, the informal definition is consistent with our mode-aware component model. The basic idea of the oracle-based approach is to abstract the behavior of each component into a local property network, constructed by a set of properties and property functions defining the transitions between different properties. A property represents a particular feature of a component and can be either functional or non-functional. The mode of each component is modeled as a property dependent on other property values. The local property networks of different components at various levels can be interconnected and jointly become a global property network. A mode switch is initiated by a component with a value change in one of its properties. Such a value change is propagated throughout the property network, potentially leading to the value change of some properties of the other components in the network. The propagation is stabilized when the value change of all affected properties is completed. After the update of the global property network, the new component modes are derived in a top-down fashion.

The required time to update the property network is in proportion to the complexity of the network. In order to achieve fast and predictable network update time, a finite-state machine called Oracle is constructed at design time, with an analysis of all possible mode-switch events, the propagation of each event, and the possible mode switch of each component. Each state in Oracle represents a particular value combination of all properties in the network. Oracle excludes all intermediate states when the property network is not stabilized. Thereby each stable state in Oracle corresponds to an architecture variant of the system which resembles the system mode after mode transformation in MSL. The property network update can always be completed by a single transition between two stable states in Oracle. In an extended report [97], Outlý et al. introduce a Hierarchical Mode Automaton (HMA), which represents a system configuration by the combination of all component modes. This is similar to the mode transition graph described in Chapter 8 of the thesis. The difference is that HMA keeps the hierarchical structure of component modes while the hierarchical structure is flattened in a mode transition graph. Unlike the explicit mode mapping in MSL, HMA does not explicitly address how the mode of a
composite component is mapped to the modes of its subcomponents. Instead, mode mapping is implicitly established as all component modes are derived from the property network.

The oracle-based approach handles mode switch in a centralized fashion, as Oracle provides the global mode information. There is no local mode-switch manager in each component that can be compared with our MSRM. Hence distributed mode-switch handling is out of the scope of the oracle-based approach.

### 11.1.3 Input-output blocks for modeling multi-mode systems

Weimer and Krogh [120] propose a set of input-output blocks to build multi-mode systems. These blocks are categorized into five categories: (1) Basic System Block (BSB): a primitive component; (2) Composite System Block (CSB): a composite component; (3) Mode Block (MB): a parallel grouping of system blocks, one for each mode; (4) Supervisor Block (SB): a block telling one or more MBs to switch to the desired mode; (5) Hybrid System Block (HSB): a block consisting of one SB and a set of MBs controlled by the SB. An MB can include one or more BSBs, CSBs, and HSBs. The usage of these blocks are demonstrated by a steer-by-wire system, where the driver and the steering controller can run in different modes. The proposed approach is implemented in Simulink [87]. Simulation results are reported based on the steer-by-wire system.

Similar to MSL, this block-based approach allows each component to support multiple modes. However, a key difference is that this approach uses completely different components (defined separately within an MB) for different modes, whereas in MSL it is possible to share some components and connections in different modes. Hence MSL is more suitable for component reuse. Moreover, this approach does not provide composition rules for component modes which function as our mode mapping mechanism, or systematic runtime mechanisms for handling mode switch.

### 11.1.4 The modeling of multi-mode systems in MARTE

Quadri et al. [104] add a mode concept to Gaspard2 [44], a model-driven engineering framework for System-on-Chip co-design. Gaspard2 uses the UML MARTE profile [94] for modeling real-time embedded systems. A multi-mode component is represented by a Macro Component which consists of a state graph component and one or more mode switch components. A mode switch
component, which plays the same role as the MB in the block-based approach introduced in the previous subsection, contains the configurations for all modes and can switch mode based on the specified mode from the corresponding state graph component. A state graph component models the behavior of a component/system as a finite state machine. Each state graph component is responsible for including the mode-switch triggering conditions and sending the mode-switch request to the expected mode switch component. Compared with MSL, this approach has almost the same limitations as the block-based approach, i.e., without mechanisms for mode mapping and mode-switch handling.

11.1.5 Component-based coordination of autonomic managers

Delaval et al.[34] has proposed a component-based design of self-adaptive computing systems. Compared with the traditional approach of using a single autonomic manager to govern the online adaptation of a monolithic adaptive system, the component-based approach fosters reuse and composition of self-adaptive components with autonomic managers. The autonomic manager of a composite component, which acts as a dedicated subcomponent, is responsible for the coordination of the autonomic managers of its subcomponents. Coordination policies are manually specified at each level to satisfy the system requirement. For instance, an autonomic manager may not trigger an action when another autonomic manager is in a particular state.

The modular and hierarchical coordination of autonomic managers conduces to reusability and scalability. A self-adaptive component can be reused in various contexts without changing its autonomic manager. Distributed and coordinated autonomic managers are preferable to a single autonomic manager for large complex systems.

The autonomic manager of a component is comparable to the MSRM of our MSL, as both control the adaptive behavior of an individual component and can be hierarchically organized as a result of component composition. However, no general coordination patterns have been discussed for autonomic managers, since the specification of coordination policies is application-dependent. This is in contrast with our MSL, which incorporates a set of protocols and rules in its MSRM for the coordination of the mode switches of different components at runtime.
11.1.6 Top-down mode specification using Event-B

Dotti et al. [35] provide the formal specification of modes and mode transitions for fault-tolerant systems using the formalism Event-B [1], a formal method for system-level modeling and analysis. Each mode is associated with an assumption and a guarantee. An assumption is a predicate over the current system state, while a guarantee is a relation over the current and next states of the system. The concept of hierarchical modes in Modechart [73] (Modechart will be discussed later in this chapter) is adopted for the top-down mode specification, viz. mode refinement. Initially, top-level modes and mode transitions are specified to meet a given system requirement. Then it is possible to refine each mode with sub-modes and transitions between them at a lower level. The same refinement can also be applied to sub-modes. Mode refinement makes it easier for system developers to map requirements to models and to trace requirements. Despite the particular assumption of fault-tolerant systems, the proposed approach seems to be applicable to other types of systems as well.

11.2 Component models supporting mode switch or dynamic reconfiguration

Numerous software component models [31, 67] have been proposed for the component-based development of a broad spectrum of systems. However, very few of them take mode switch or dynamic reconfiguration into account. An overview of those component models is presented here, including BlueArX, COMDES-II, Koala, SaveCCM, MyCCM-HI, Rubus, AUTOSAR, and Fractal, followed by a brief discussion on the limitations of these component models in comparison with MSL.

11.2.1 BlueArX

BlueArX [75] is a component model developed by Bosch ¹ particularly for automotive applications with constrained resources. BlueArX supports multi-mode applications built by BlueArX software components. Mode is regarded as a type of semantic context information. Different modes imply different scheduling or different control strategies. The mode definition is integrated in the component specification and can be imported or exported from component ports.

¹http://www.bosch.com/
11.2 Component models supporting mode switch or dynamic reconfiguration

Bosch has come up with a heuristics approach to determine the supported modes of different components. The heuristics provides reasonable mode candidates for each component by exploring various control conditions that dramatically influence system behavior and performance. This approach has been implemented in a tool called XGen. The mode of a BlueArX component is not explicitly defined until a multi-mode system is completely built.

The major purpose of defining modes for the BlueArX component model is more precise prediction of system properties such as WCET, however, we have not been able to find how a mode switch is handled by BlueArX.

11.2.2 COMDES-II

COMDES-II (COMponent-based design of software for Distributed Embedded Systems-version II) [74] employs a hierarchical model to specify a system architecture. A multi-mode component is treated as a pair of a state machine Function Block (FB) and a modal FB. The original objective for jointly using state machine FBs and modal FBs is to specify the system sequential behavior (i.e. control flow) to ensure deterministic state transitions. A state machine FB contains the state transition rules of a component and the corresponding modal FB executes the control actions associated with the current state. In a similar way, a state machine can define the mode transition graph of a component and a modal FB includes the component configurations in different modes. A state machine FB can trigger a mode switch based on its input. After receiving the mode-switch command from the state machine FB, the modal FB will switch to the configuration in the new mode. COMDES-II allows the composition of multi-mode components so that a pair of state machine FB and modal FB can be included in another modal FB.

11.2.3 Koala and SaveCCM

Koala [95], a component model developed by Philips\(^2\), is used to manage software complexity in consumer electronics. Koala does not explicitly deal with mode switch. Instead, it proposes some primitive but practical solutions to efficiently handle architectural diversity. First, it introduces diversity interfaces for components with multiple configurations. A component can change its configuration based on the input values from a diversity interface. In addition, Koala uses a special construct, the switch, to realize the structural diversity of a component. A switch is able to divert an incoming data flow to different outgoing

\(^2\)http://www.philips.com/
branches according to different conditions evaluated at runtime. Both diversity interface and switch can be effectively implemented and they are easy to manipulate. However, a major disadvantage is that the system functionality is polluted by the diversity management.

The SaveCCM component model [58] is designed specifically for the vehicular domain, with focus on predictability and analysability. Regarding mode switch, SaveCCM adopts the same methodology as Koala, i.e., by using a dedicated component or connector switch with the same semantics as the switch in Koala. Similar to Koala, SaveCCM integrates structural diversity management in the functional behaviors of each component, without clear separation between functional behaviors and mode-switch handling.

11.2.4 MyCCM-HI

MyCCM-HI (Make Your Component-Container Model-High Integrity) [15] is a component framework for critical, distributed, real-time and embedded software. Software components and their behaviors are described by an input architecture description language COAL (Component-Oriented Architecture Language), which can enumerate different operational modes of the system and of each component. Each multi-mode component has a dedicated mode automaton that implements its mode-switch mechanism. Mode automaton components can interact with their sibling components or components at adjacent levels during a mode switch. This establishes the foundation for the mode-switch propagation throughout different parts of the system. However, no runtime mechanisms for the mode-switch coordination between different components have been reported in MyCCM-HI.

11.2.5 Rubus

Rubus [57] is an industrial component model jointly developed by Arcticus systems\(^3\) and Mälardalen University targeting embedded control systems for ground vehicles. In Rubus, mode is essentially a system-level concept for each Electronic Control Unit (ECU). In each mode, there is a system-wide static configuration of components. Guided by a global mode transition diagram, a system mode switch corresponds to the switch between different system configurations, with each configuration represented by a state in the diagram. Since mode-switch handling is delegated to the system, each individual

\(^3\)http://www.arcticus-systems.com/
Chapter 11. Related work

Branches according to different conditions evaluated at runtime. Both diversity interface and switch can be effectively implemented and they are easy to manipulate. However, a major disadvantage is that the system functionality is polluted by the diversity management.

The SaveCCM component model [58] is designed specifically for the vehicular domain, with focus on predictability and analysability. Regarding mode switch, SaveCCM adopts the same methodology as Koala, i.e., by using a dedicated component or connector switch with the same semantics as the switch in Koala. Similar to Koala, SaveCCM integrates structural diversity management in the functional behaviors of each component, without clear separation between functional behaviors and mode-switch handling.

11.2.4 MyCCM-HI

MyCCM-HI (Make Your Component-Container Model-High Integrity) [15] is a component framework for critical, distributed, real-time and embedded software. Software components and their behaviors are described by an input architecture description language COAL (Component-Oriented Architecture Language), which can enumerate different operational modes of the system and of each component. Each multi-mode component has a dedicated mode automaton that implements its mode-switch mechanism. Mode automaton components can interact with their sibling components or components at adjacent levels during a mode switch. This establishes the foundation for the mode-switch propagation throughout different parts of the system. However, no runtime mechanisms for the mode-switch coordination between different components have been reported in MyCCM-HI.

11.2.5 Rubus

Rubus [57] is an industrial component model jointly developed by Arcticus systems3 and Mälardalen University targeting embedded control systems for ground vehicles. In Rubus, mode is essentially a system-level concept for each Electronic Control Unit (ECU). In each mode, there is a system-wide static configuration of components. Guided by a global mode transition diagram, a system mode switch corresponds to the switch between different system configurations, with each configuration represented by a state in the diagram. Since mode-switch handling is delegated to the system, each individual Rubus component has no sense of mode. In general, the mode management in Rubus is in line with the centralized mode-switch handling of MSL after mode transformation.

11.2.6 AUTOSAR

AUTOSAR [43], AUTomotive Open System ARchitecture, is an open industry standard for the automotive software architecture between suppliers and manufacturers. AUTOSAR separates software from hardware and facilitates reuse of software components for different vehicle platforms. Featured by a layered structure, AUTOSAR abstracts the hardware platform by a Basic Software layer over the ECU hardware layer. The Basic Software layer contains Basic Software (BSW) Modules which provide fundamental services such as communication, operating system, and ECU abstraction for higher-level applications. Above the Basic Software layer, there is a Runtime Environment (RTE) which abstracts the communication between AUTOSAR Software Components (SW-Cs) on top.

The system on each ECU can run in multiple modes statically specified at design time. Two parties are involved for mode management [29] in AUTOSAR: mode manager and mode user, both of which can be either a SW-C or a BSW Module. An application mode manager communicates with one or multiple application mode users. Likewise, a basic software mode manager communicates with one or multiple basic software mode users. A mode switch can be requested by either a mode manager or a mode user. However, the execution of a mode switch is always decided by a mode manager. AUTOSAR supports both local and global mode switches. A local mode switch is performed within a single ECU, whereas a global mode switch impacts the systems running on a set of networked ECUs belonging to the same automotive system.

Although AUTOSAR allows reuse and composition of SW-Cs, both mode managers and mode users are global entities for an ECU. Compared with the centralized mode management of AUTOSAR, the distributed mode-switch handling of MSL is more scalable and flexible.

11.2.7 Fractal

Fractal [19] is a hierarchical and reflective component model that has been adopted by various programming languages for developing adaptive systems.
Each Fractal component has a container which includes a set of local controllers to control the runtime reconfiguration of the component. Such a container is called a membrane. A reconfiguration can be structural, such as adding or removing subcomponents, or binding/unbinding of component interfaces, or behavioral, such as modifying the life cycle of a component. Fractal is reflective on account of its introspection and intercession abilities, which allow a system to monitor its execution at runtime and automatically reconfigure itself when necessary.

We anticipate that the reconfiguration of a Fractal component may affect the configurations of some other Fractal components in the same system. Fractal has been extended [10] to cover distributed reconfiguration of components by introducing a reconfiguration controller in the membrane of each component. However, the coordination between multiple component reconfigurations requires the controller to run a unanimous distributed algorithm which is reminiscent of the MSRM of MSL. Such an algorithm is still missing in Fractal.

11.2.8 Limitations

Among these component models, only COMDES-II, MyCCM-HI, and Fractal support the mode switch or dynamic reconfiguration of individual components. None of them considers the composition of component modes or the coordination of the mode switches of different components.

11.3 Languages supporting mode switch

Mode switch has also been covered by some programming and specification languages, such as Ada, AADL, Giotto, TDL, Darwin, Modechart and mode-automata. This section includes an overview of these languages and points out their limitations compared with MSL.

11.3.1 Ada

Ada [28] is a prominent object-oriented programming language suitable for the implementation of embedded and real-time systems. An Ada real-time framework [121] has been extended [106] to support operating modes and mode changes in real-time systems. Sáez et al.[107] have managed to specify the mode manager by means of a behavioral UML Finite State Machine (FSM)
and automatically generate Ada codes from the FSM specification. The development process, including mode specification, code generation, and schedulability analysis, is demonstrated by an industrial robotic system.

### 11.3 Languages supporting mode switch

#### 11.3.2 AADL

The Architecture Analysis & Design Language (AADL) [38], is a modeling language that supports analyses of a system’s architecture with respect to performance-critical properties at software component level. AADL represents modes as states within a state machine abstraction, where each state corresponds to a distinct mode and the transition between different states represents a mode switch. For each component, a mode switch is triggered by a predefined mode-switch event arriving at its input event port(s). A component can also spontaneously trigger a mode switch from its output event port(s). In each mode, the running components and their connections are strictly defined. Furthermore, mode-specific properties of a component can be defined to distinguish its alternative behaviors in different modes. For instance, a component may have different worst-case execution times in different modes.

#### 11.3.3 Giotto and TDL

Giotto [63] is a time-triggered language for implementing real-time embedded systems. In Giotto, each mode defines the periodic invocation of a fixed set of tasks. While running in a mode, a system periodically checks an exit condition with a tunable switch frequency. The exit condition is expressed by a boolean expression. A mode switch is performed when the exit condition evaluates to true. Giotto assumes instantaneous mode switch in the sense that a mode switch must be completed within the "current tick", thus implying that the mode-switch time is always 0. If a mode switch is triggered in the presence of ongoing execution of a task, it is allowed to switch mode if the invocation period of the task is unchanged after the mode switch. Then the task must complete its remaining execution in the old mode before its next invocation in the new mode.

TDL [116] is a time-triggered language derivative of Giotto, yet with more convenient syntax and improved programming tools. It preserves the fundamental part of Giotto for mode-switch handling. However, mode declaration is enhanced in TDL. In addition, TDL forbids a mode switch performed during the execution of a task.
11.3.4 Darwin

Hirsch et al. [65] introduce modes to software architectures. Mode is regarded as a new element of architectural descriptions. They also incorporate the notion of mode to an existing Architecture Description Language: Darwin [83]. To be aware of modes, Darwin is extended by adding a mode attribute to each component indicating its current mode. The modes of a Darwin composite component is directly related to the modes of its subcomponents. Yet no mode mapping solutions are provided in [65]. The variation of the software architecture and different components in different modes is illustrated by an automotive case study, where a mode switch can imply the change of the component running status, functionality and connections. It is still unclear in [65] how a mode switch is handled at runtime.

11.3.5 Modechart

Modechart [73] is a specification language for real-time systems. The semantics of Modechart is based on Real Time Logic (RTL) [72]. Modechart focuses on the specification of absolute timing properties as well as modes and mode transitions. Modes can be hierarchical in Modechart. There are primitive or compound modes that are classified in the same way as primitive and composite components. Moreover, both serial and parallel mode relations are defined. Compared to the mode in MSL, the mode concept in Modechart has a broader meaning, since sometimes even concurrent running threads are considered as parallel modes. Modechart assumes instantaneous mode switch and does not aim for component-based systems.

11.3.6 Mode-automata

Mode-automata [84, 85] is a programming model proposed as an extension of the synchronous language Lustre [23]. Devoted to the description of running modes of reactive systems, mode-automata allows a collection of execution states to be considered as a mode, and the complete behavior of a complex system is a sequence of modes. A program can be projected onto a given mode so that the behavior restricted to this mode can be obtained. Moreover, parallel and hierarchical compositions of mode-automata are also supported. This enables a hierarchical mode structure just like Modechart. Mode-automata not only enhances the program readability of Lustre, but also improves the quality of the generated code. The hierarchical mode structure of mode-automata
 resembles the hierarchical component structure in a component-based system. However, the mode decomposition of mode-automata is based on the system behavior rather than reusable components. Mode-automata has been integrated in a graphical development environment SCADE [77].

11.3.7 Limitations

Most of the aforementioned languages only deal with system-level mode switch. AADL and Darwin take multi-mode components into consideration. The mode switch of a component in AADL is relatively independent of the modes of other components. Darwin is aware of the mode mapping between components, yet providing no mode mapping mechanisms. Neither AADL nor Darwin addresses the mode-switch coordination of different components at runtime.

11.4 Dynamic software product line

DSPL [22, 46, 64] is an emerging technique for developing adaptive systems. DSPL originates from conventional Software Product Lines (SPLs) [27] which have been successful in producing a family of software systems while being adopted commercially due to cost reduction and software quality improvement. Different systems configured from the same SPL share certain features, whereas the SPL uses variation points to distinguish the unique features of each system. A system is configured by selecting and binding the most appropriate variation points at design time to meet specific system requirements.

In addition to SPL, DSPL allows the binding of variation points at runtime so that a system can dynamically change configurations on the fly to accommodate to the changing environment. The ability of binding variation points at design time is still preserved by DSPL. When all variation points are bound at design time, DSPL will be equal to SPL. When all variation points are bound at runtime, DSPL will contribute to a single adaptive system.

The configuration space of a DSPL is typically predefined at design time. This is fairly analogous to multi-mode systems, as each configuration can be considered as a mode and dynamic reconfiguration is comparable to mode switch. The assumption of static configuration space limits the adaptivity of DSPL, however, it makes a system more reliable and predictable compared with adaptive systems with unbounded configuration space. In recent years there has been a growing interest in DSPLs with dynamic variation
points [16, 61] which potentially lead to new system configurations identified at runtime. The validity of each new configuration must be examined online before reconfiguration is executed.

To the best of our knowledge, DSPL only considers global system configurations. Hence reuse of adaptive software components is not supported in DSPL. MSL provides an opportunity to reuse multi-mode components in DSPL.

### 11.5 Mutli-mode real-time systems

The research on mode switch in the real-time systems domain [20] has been conducted for decades. There are miscellaneous topics studying multi-mode real-time systems and mode switch, while topics that have been investigated most are the design of mode-switch protocols and schedulability analysis considering a mode switch. The mode of a multi-mode real-time system is typically represented by a set of running tasks, and a mode switch amounts to the suspension of tasks running in the old mode but not in the new mode, and the activation of tasks running in the new mode but not in the old mode. A task may keep running both before and after a mode switch, which may change some parameters of the task, such as execution time or period. An extensive survey and classification of some classic mode-switch protocols can be found in [105].

In general, there are five types of tasks: (1) old-mode aborted task that runs in the old mode and aborts its execution when a mode switch is triggered; (2) old-mode completed task that runs in the old mode and needs to complete its current execution before switching mode; (3) changed task that runs in both the old and new modes but changes its parameter in the new mode; (4) wholly new-mode task that only runs in the new mode and is activated after a mode switch; (5) unchanged task whose execution is unaffected by a mode switch. Apart from these five types of tasks, one can additionally imagine tasks that only run during the transition from the old mode to the new mode. Depending on the treatment of unchanged tasks, two types of mode-switch protocols are identified: (1) protocols with periodicity which do not affect the execution of unchanged tasks during a mode switch; (2) protocols without periodicity which change the activation of unchanged tasks during a mode switch. Most existing mode-switch protocols belong to the second type, however, the first type has been proposed in recent years [91] [92]. With respect to the co-execution permission of old and new mode tasks, there are (1) synchronous protocols...
where new mode tasks are never released until all old mode tasks complete their last activation in the old mode; and (2) asynchronous protocols where old and new mode tasks are allowed to be executed at the same time during a mode switch. Synchronous protocols [6, 105, 118] do not execute new mode tasks until the execution of all old mode tasks is completed, thus requiring no schedulability analysis during a mode switch. Nevertheless, synchronous protocols are notorious for long mode-switch time which is often intolerable to meet the timing constraints of real-time systems. Thus more efforts have been paid on asynchronous protocols, which enable swift mode switch but require additional schedulability analysis during a mode switch, as the execution of new mode tasks together with the remaining execution of old mode tasks may lead to a temporary overload.

A mode-switch protocol is highly dependent on many contributing factors such as the task model, the scheduling policy, and the hardware platform. Different combinations of these factors call for different mode-switch protocols. Mode-switch protocols were originally developed [111, 119] under fixed-priority preemptive scheduling for uniprocessor systems. Subsequent efforts have been invested by others [98, 105] to reduce mode-switch time and improve schedulability during a mode switch. Holenderski et al.[66] develop a mode-switch protocol for memory constrained systems under fixed-priority scheduling with limited preemption. Upon receiving a mode-switch request, a task can abort its execution at pre-defined termination points to achieve a swift mode switch with graceful service interruption. Huang and Chen [68] provide a schedulability analysis for multi-mode tasks under fixed-priority scheduling, allowing each task to have its own modes. Aside from mode-switch protocols for fixed-priority scheduling policies, there are also protocols targeting dynamic scheduling policies such as Earliest Deadline First (EDF) [5, 113] or offline scheduling [40]. A common but restrictive assumption in multi-mode real-time systems is independent and periodic tasks. Ekberg et al.[36] have presented a graph-based task model which is able to express more complex task instance arrival patterns for multiple modes. They also provide the EDF schedulability analysis for the expressive task model. Many other protocols and schedulability analysis have been proposed to support multiprocessor platforms [81, 90, 92, 93, 122], distributed systems [37, 62, 91], and hierarchical scheduling [39, 69, 70, 109] where a real-time system is partitioned into subsystems with temporal isolation. In addition, Phan et al. use multi-mode Real-Time Calculus [100] and a multi-mode automaton model [99] to model and analyze multi-mode systems. They also introduce an interface-based technique for compositional analysis of multi-mode systems [101], as well as a semantic
framework for the specification and analysis of mode-switch protocols [102].

Multi-mode systems have strong relation to mixed-criticality systems [7, 20] which are gaining more and more attention nowadays. In a mixed-criticality system, a task may run at different criticality levels with varying execution times. The higher criticality level, the longer execution time. The criticality change of a task at runtime can be considered as a mode switch modifying its execution time.

The study of mode switch in real-time systems is rather complementary to our MSL. On the one hand, MSL focuses on reuse of multi-mode components and mode-switch handling at architectural level. On the other hand, the design of mode-switch protocols and schedulability analysis focus on meeting timing constraints at implementation level. It would be an interesting future research direction to incorporate existing research results of real-time systems into MSL to achieve better mode-switch timing analysis.
Chapter 12

Conclusions and future work

Software development of multi-mode systems is challenging in several aspects. The growing software complexity is often a direct ramification of the diversified functionalities and dynamic mode-switch behaviors of multi-mode systems. In this thesis we have developed a component-based framework, Mode-Switch Logic (MSL), for development of multi-mode systems. MSL supports reuse of multi-mode components which run a uniform distributed mechanism for the cooperative handling of mode switches at runtime. The distributed mechanism is scalable and requires no global mode information, and is thus applicable to large complex systems. Moreover, MSL includes a mode transformation technique for transforming the distributed mechanism into centralized mode-switch handling when appropriate, thereby improving runtime mode-switch efficiency. As the closure of the thesis, this chapter summarizes the contributions presented in previous chapters and sketches possible future research directions.

12.1 Summary and contributions

All the contributions in the thesis have been motivated by our research goal, which is decomposed into four research questions stated in Chapter 1, with each research question addressed by at least one contribution. These research questions are revisited as follows:

Research Question (RQ) 1: What distinctive features and what techniques should a multi-mode component support in order to ensure flexible and efficient
reuse?

This research question is supported by three contributions: the mode-aware component model in Chapter 2, the mode mapping mechanism in Chapter 3, and the mode transformation technique in Chapter 8.

The mode-aware component model provides the formal definition of reusable multi-mode components. Compared with other existing component models, the mode-aware component model bears three distinctive features:

- A multi-mode component has unique configurations in different modes. A configuration depends on factors such as the functional behavior of a component, the values of certain mode-dependent properties, the activated subcomponents and their connections.

- A multi-mode component has dedicated mode-switch ports to exchange mode information with its parent and subcomponents during a mode switch.

- A multi-mode component runs a built-in MSRM which governs its mode-switch behavior at runtime.

The mode mapping mechanism, which resides in each composite component, maps the modes of the composite component to the modes of its subcomponents. The mode mapping of each composite component is statically specified at design time by a set of internally synchronized MMAs. The mode mapping of a component provides mode mapping results to its MSRM during a mode switch, telling which components among the composite component itself and its subcomponents should switch mode and what new modes these components should switch to.

The mode transformation technique works when all the software components are deployed on the same hardware platform and the mode information of all components is globally accessible. Mode transformation converts the distributed mode-switch handling of MSL into a centralized mode management by replacing the MSRMs of all components with a single global mode-switch manager. The transformation is conducted in two sequential steps. First, a Mode Combination Tree is generated based on the mode mappings of all composite components. Each path of the tree corresponds to a system mode. With the identified system modes and the specification of all (mode-switch) scenarios as the input, the second step derives a mode transition graph, adding all the possible transitions between different system modes. After mode transformation, a mode switch is a direct transition between system modes in the mode
transit graph. Mode-switch efficiency is improved from two perspectives: (1) the computation overhead of the global mode-switch manager is much lower than the computation overhead of all MSRMIs; (2) a mode-switch can be completed more swiftly without inter-component communication incurred by the MSRM.

**Research Question (RQ) 2:** How do we handle mode switch at runtime at both component and system levels?

MSL handles mode switch using the MSRM presented in Chapter 4. Guided by the MSRM, each component communicates with its parent and subcomponents by exchanging primitives. A mode switch is initiated by the triggering of a (mode-switch) scenario from a component that detects a mode-switch event. The MSRM distinguishes two types of scenarios: emergency scenario and non-emergency scenario. A non-emergency scenario is propagated by an MSP protocol to all the affected components. A mode switch is executed only when all the affected components are ready to switch mode. By contrast, an emergency scenario must be executed as soon as possible, even if some components are not ready to switch mode. The MSRM contains an EMSP protocol for the propagation of an emergency scenario. The execution of a mode switch (either emergency or non-emergency) follows a mode-switch dependency rule which forces a mode switch to be completed bottom-up. If the propagation of a scenario encounters an ongoing atomic execution of a component, the scenario will not be propagated to this component until the completion of its ongoing atomic execution. The MSRM also supports the concurrent triggering of emergency and non-emergency scenarios. Each component is equipped with three queues: MSR queue, MSQ queue, and EMS queue, in ascending priority order. The MSR queue and MSQ queue store an incoming MSR or MSQ primitive which propagates a non-emergency scenario. All pending non-emergency scenarios are handled by each component in a sequential order. The EMS queue stores an incoming EMS which propagates an emergency scenario with a higher priority than all non-emergency scenarios. The mode switch of a component may invalidate certain elements in its MSR queue and MSQ queue. As part of the MSRM, an MSR/MSQ queue updating rule is introduced to remove the invalid MSR and MSQ primitives in the corresponding queues.

**Research Question (RQ) 3:** How can we ensure that the mode-switch handling mechanisms in RQ 2 do not violate the system timing requirements?

Almost all multi-mode systems have timing constraints on the execution
of a mode switch. The mode-switch time must always be bounded and predictable. Chapter 6 provides the mode-switch timing analysis for the MSRM with the assumption of the triggering of a single non-emergency scenario. The timing analysis calculates the worst-case mode-switch time, dividing a complete mode-switch phase into three non-overlapping and continuous phases. Phase 1 starts from the triggering of a scenario and is terminated by the identification of the MSDM, i.e., a component which is authorized to approve or reject the scenario. In Phase 2, the MSDM queries whether all the components affected by a scenario are ready to switch mode. If yes, a mode switch will be triggered by the MSDM in Phase 3 which ends upon mode-switch completion. The duration of Phase 2 can be prolonged by atomic execution. Hence a model-checking approach is used to derive the worst-case atomic execution time of a group of components.

In addition, the mode transformation technique presented in Chapter 8 is able to expedite the handling of a mode switch, thereby increasing the likelihood of fulfilling the system timing requirements.

Research Question (RQ) 4: How do we evaluate the applicability of this framework?

One of our crucial objectives is to verify the correctness of MSL and evaluate its applicability. Chapter 5 presents the formal verification of the MSRM, a pivotal element of MSL. The verification is carried out in two steps: model checking and manual theorem proving. First, the MSRM of a component and its interaction with the parent and subcomponents are modeled and verified using the model checker UPPAAL. The verification results in the first step is subsequently generalized by manual theorem proving.

The mode-aware component model, the mode mapping mechanism, and the mode transformation technique are implemented in a tool MCORE (the Multi-mode COmponent Reuse Environment) presented in Chapter 9. MCORE provides a graphical user interface for the development, composition, and reuse of multi-mode components. Furthermore, MCORE can generate the system mode transition graph and all global configurations by applying mode transformation. The ambition of MCORE is to export the system model of a multi-mode system to Rubus ICE [114], an industrial tool for software development of vehicular systems.

The integration of MSL in the ProCom component model, explained in Chapter 7, is another evidence of the practical value of MSL. Although the original ProCom component model is not intended for development of multi-
mode systems, we are able to support reuse of multi-mode components and distributed mode-switch handling in ProCom with slight extension.

Moreover, Chapter 10 demonstrates the practical usage of MSL based on two proof-of-concept case studies, an Adaptive Cruise Control (ACC) system and a healthcare monitoring system. They are traditionally developed as monolithic systems that can run in multiple modes. Using MSL, we redesign both systems with a more scalable and flexible scheme that includes reuse of multi-mode components.

12.2 Future research directions

Currently, there are still many remaining issues worth a forthcoming exploration. We envision a number of interesting future research directions stemming from our current contributions, as explained below.

Reinforcement in the evaluation

First, performance analysis is still missing in the formal verification of the MSRM. A valuable future work is to develop a simulator for the extensive simulation and evaluation of the MSRM. The simulator is expected to support the automatic generation of a hierarchy of multi-mode components based on the specification of various parameters such as the maximum depth level of each primitive component, and the maximum number of subcomponents of each composite component. Different scenarios are randomly triggered from an arbitrary component. The coordination between the mode switches of different components can be clearly observed by visualizing the inter-component communication with transmission of different primitives in the simulator. Extensive simulation also allows us to analyze performance. For instance, it is possible to study how the component hierarchy impacts the number of transmitted primitives and mode-switch time.

Our tool MCORE is at a work-in-progress stage, lacking several key functionalities such as the MSRM implementation and exporting a system model as xml files to Rubus ICE. Additional efforts must be devoted to the completion of a full-fledged version. Another interesting work is the continuation of the ProCom component model extended by MSL to support modes. It would be worthwhile to implement our theoretical foundation in PRIDE [14], an IDE for ProCom, thereby obtaining a tool environment for performing more extensive evaluations and case studies.
As far as we know, no existing multi-mode systems are yet built by multi-mode components. The ACC system and healthcare monitoring system presented in the thesis are only proof-of-concept implementations. Our ultimate goal is to explore the applicability of MSL in real-world systems.

**Extension of the MSRM**

The mode-switch dependency rule of our MSRM assumes that the mode switches of sibling components, i.e., components with the same parent, are independent of each other. This implies that a composite component does not care about the order of mode-switch completion among its subcomponents. However, under certain conditions, the mode switches of sibling components are expected to follow a specific order. That is to say, the mode switch of a component may not be allowed until the mode-switch completion of another sibling component. Our MSRM needs to be extended to respect such constraints.

Another assumption of our MSRM is that a system contains at most one emergency scenario which can be recurrently triggered. We intend to extend the MSRM by supporting the triggering of multiple emergency scenarios with different criticality levels. Besides, the MSRM does not allow an emergency scenario to abort an ongoing component reconfiguration, thus incurring an unacceptable delay to the handling of an emergency scenario if some component has extremely long reconfiguration time. Therefore, we need to investigate how an emergency scenario can be immediately handled without delay, even at the sacrifice of aborting an ongoing reconfiguration.

**Allocation of multi-mode components to tasks**

An essential step from component-based design to implementation is to allocate software components to runtime tasks. Techniques for the allocation of single-mode components to tasks have been reported in a few publications [42, 41, 108]. There are no standard guidelines for the allocation, since different component models call for different allocation techniques. In general, the hierarchical structure of components must be flattened [18, 82]. All composite components are removed in the flattening process, while new elements are generated to preserve the original operational semantics of the system. After flattening, the component-to-task allocation is performed based on the transaction flows identified in the flattened system.

The allocation of multi-mode components to tasks is still an open research
problem. Components deactivated in certain modes and the mode-dependent component connections may substantially complicate the allocation. We are looking forward to new allocation techniques allowing for modes.

**A refined mode-switch timing analysis**

The mode-switch timing analysis presented in Chapter 6 of this thesis assumes the triggering of a single non-emergency scenario. A natural next step would be to extend the timing analysis to cover the concurrent triggering of both emergency and non-emergency scenarios. Furthermore, the mode-switch timing analysis at component level only gives a coarse-grained value of the worst-case mode-switch time due to the abstraction of low-level information such as scheduling and hardware platform. The allocation of multi-mode components to tasks plays an important role in achieving a more precise mode-switch timing analysis, since the timing analysis at task level provides much more fine-grained results.

**The support of runtime software evolution**

In MSL, the modes of each component and the corresponding configurations are all statically specified at design time. Hence it is impossible to change the component structure or mode specification at runtime. Nonetheless, dynamic software evolution [76] at runtime is a charming feature raising a growing attention in software engineering. It would be noteworthy to embrace runtime software evolution in MSL so that one can dynamically add, remove, replace components, or even change the mode mapping and MSRM of a component on the fly. A even more evolvable system should also be able to automatically introduce new modes and configurations or remove existing modes whenever appropriate. However, there is always a tradeoff between flexibility and predictability. The more functionalities that we transfer from design time to runtime, the less predictable and analyzable the system may become. It would be a challenging task to achieve predictable runtime software evolution in MSL.

**The adaptation of MSL to more architectural styles**

MSL is largely based on the pipes and filters architectural style. We aspire to future adaptation of MSL to other architectural styles such as client-server [32].


Bibliography


[18] M. Bozga, M. Jaber, and J. Sifakis. Source-to-source architecture transformation for performance optimization in BIP. In Proceedings of


Index

2-phase commit protocol, 56
AADL, 299
Ada, 298
adaptive cruise control, 261
atomic execution, 68
atomic execution group, 70
autonomic manager, 293
AUTOSAR, 297
BlueArX, 294
COMDES-II, 295
component, 1
component model, 4
component reuse, 4
component target mode table, 239
component-based development, 4
component-based software engineering, 4
composite component, 5
composition level, 143
configuration, 19
control flow, 200
Darwin, 300
data flow, 200
dedicated mode-switch port, 20
depth level, 143
downstream primitive, 51
dynamic software product line, 301
electronic control unit, 250
embedded system, 1
emergency mode switch, 99
emergency scenario triggering rule, 110
emergency transition state, 104
EMS queue, 100
EMS receiving rule, 116
EMSP protocol, 101
Event-B, 294
event-triggered, 48
external mode-switch behavior, 142
fault-tolerant, 3
Fractal, 297
Gaspard2, 292
Giotto, 299
internal mode-switch behavior, 142
JointJS, 255
Koala, 295
Lustre, 300
MARTE, 292
mathematical induction, 145
MCORE, 255
meta-model, 20
mixed-criticality system, 304
mode, 2
mode combination tree, 232
mode consistency, 66
mode mapping, 29
mode mapping automata
  child MMA, 32
  MMA composition, 39
  MMA synchronization, 34
  parent MMA, 32
  signal, 33
  state, 33
  transition, 33
mode mapping automaton, 33
mode switch, 2
mode transformation, 230
mode transition graph, 238
mode-automata, 300
mode-aware component model, 19
mode-dependent property, 20
mode-independent property, 20
mode-switch decision maker, 50
mode-switch dependency rule, 66
mode-switch logic, 8
mode-switch manager, 230
mode-switch port, 22
mode-switch protocol, 302
mode-switch runtime mechanism, 47
mode-switch scenario, 49
mode-switch source, 48
Modechart, 300
model checking, 132
model-driven engineering, 200
MSA handling rule, 112
MSP protocol, 52
MSQ delaying rule, 72
MSQ enqueuing rule, 97
MSR and MSQ queues, 75
MSR enqueuing rule, 96
MSR/MSQ queue checking rule, 76
MSR/MSQ queue updating rule, 81
multi-mode component, 5
multi-mode system, 2
MyCCM-HI, 296
MechatronicUML, 289
normal transition state, 104
parent and child stubs, 132
pending scenario checking rule, 104
pipes and filters, 22, 70, 200
port, 20
preliminary EMS handling rule, 120
PRIDE, 309
primitive component, 5
ProCom, 200
  connector, 201
  data port, 200
  message channel, 203
  message passing, 203
  message port, 203
  port group, 200
  ProSave layer, 200
  ProSys layer, 203
  service, 200
  trigger port, 200
property network, 291
real time logic, 300
real-time system, 175
reconfiguration, 19
Rubus, 250, 296
Rubus ICE, 257
runtime software evolution, 311
safety-critical, 3
SaveCCM, 275, 296
schedulability analysis, 302
software architecture, 300
subcomponent, 5

task, 3, 302
TDL, 299
time-triggered, 48
timed automata, 135
timed computation tree logic, 135
timing analysis, 175
transition state, 76
Type A/B component, 49

UPPAAL, 134
upstream primitive, 50

valid local mode combination, 232
verification, 131

worst-case execution time, 3