Mälardalen University Licentiate Thesis
No.161

Mode switch for component-based multi-mode systems

Hang Yin

December 2012

School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden

ISBN 978-91-7485-088-8
ISSN 1651-9256
Printed by Mälardalen University, Västerås, Sweden

To Hongwan and my family

Abstract

Component-based software engineering is becoming a prominent solution to the development of complex embedded systems. Since it allows a system to be built by reusable and independently developed components, component-based development substantially facilitates the development of a complex embedded system and allows its complexity to be better managed. Meanwhile, partitioning system behavior into multiple operational modes is also an effective approach to reducing system complexity. Combining the component-based approach with the multi-mode approach, we get a component-based multi-mode system, for which a key issue is its mode switch handling. The mode switch of such a system corresponds to the joint mode switches of many hierarchically organized components. Such a mode switch is not trivial as it amounts to the mode switch coordination of different components that are independently developed.

Since most existing approaches to mode switch handling assume that mode switch is a global event of the entire system, they cannot be easily applied to component-based multi-mode systems where both the mode switch of the system and each individual component must be considered, and where components cannot be assumed to have global knowledge of the system. In this thesis, we present a mechanism—the Mode Switch Logic (MSL)—which provides an effective solution to mode switch in component-based multi-mode systems. MSL enables a multi-mode system to be developed in a component-based manner, including (1) a mode-aware component model proposed to suit the multi-mode context; (2) a mode mapping mechanism for the seamless composition of multi-mode components and their mode switch guidance; (3) a mode switch runtime mechanism which coordinates the mode switches of all related components so that the mode switch can be correctly and efficiently performed at the system level; and (4) a timing analysis for mode switches realized by MSL. All the essential elements of MSL are additionally demonstrated by a case study.

Populärvetenskaplig sammanfattning

Vi omges dagligen av inbyggda system; datorsystem som är inbyggda i andra produkter. De finns i allt från smarta telefoner, skrivare, medicinska apparater och industrirobotar till bilar och flygplan. På grund av den ökande efterfrågan på funktionalitet, lägre kostnader och kortare tid till marknaden, utgör programvaran en allt viktigare del av dessa inbyggda system. Den ökande komplexiteten i programvaran ställer i sin tur krav på nya, mer effektiva och skalbara metoder för programvaruutveckling.

En metod som introducerats för att hantera den ökande komplexiteten i programvaran är att bygga programvarusystemen med hjälp av förutvecklade programvarukomponenter; så kallad komponentbaserad programvaru-utveckling som kan liknas vid att bygga med lego. Ett annat sätt att reducera komplexiteten är att dela upp systemet i olika driftlägen och hantera dessa oberoende av varandra. Till exempel kan programvaran som styr ett flygplan vara uppdelad i driftlägen så som taxi, start, flyg och landning. Om man kombinerar komponentbaserad utveckling med uppdelning av systemet i olika driftlägen, så får man ett komponentbaserat system med multipla driftlägen. En central uppgift för ett sådant system är att på ett effektivt och förutsägbart sätt växla mellan systemets driftlägen utifrån växling av de ingående komponenternas driftlägen. Befintliga metoder för växling av driftläge utgår från att växlingen är en global händelse, vilket inte är förenligt med den grundläggande principen om oberoende utvecklade komponenter som gäller för komponentbaserad utveckling.

I denna avhandling presenterar vi en ny metod för växling av driftläge, kallad Mode Switch Logic (MSL). MSL möjliggör driftlägesväxling på ett sätt som är förenligt med oberoende utveckling av de komponenter som ingår i

systemet . Vi har dessutom utvecklat en metod för att analysera hur lång tid det tar att genomföra driftväxling. I en fallstudie illustrerar vi tidsanalysen och de mekanismer vi utvecklat.

摘要

基于组件的软件工程逐渐在复杂嵌入式系统的开发中热门起来。由于它可以用可重复利用的和独立开发的组件来构建系统，组件式开发可以很大程度上提高复杂嵌入式系统的开发进程，并且系统的复杂性也更容易管理。与此同时，将系统不同的功能划分为不同的操作模式也是一个降低系统复杂度的有效办法。将基于组件的方法和多操作模式的方法结合起来，我们就可以得到一种基于组件的多模式系统。这种系统的一个关键问题就是对于它的不同模式转换的处理。这种系统的模式转换等同于许多系统内部各个阶层的组件的模式转换。这种模式转换不容忽视，因为不同组件的模式转换需要很好的相互协调。

目前大多数处理模式转换的方法都假设模式转换是整个系统的一种全局事件，而对于基于组件的多模式系统来说，无论是系统的模式转换还是每一个内部组件的模式转换都需要考虑，而且每个组件不允许了解系统的全局信息，所以这些方法并不能简单的应用到基于组件的多模式系统。本篇论文针对此类问题介绍了一种我们开发的模式转换逻辑(MSL: Mode Switch Logic)。作为一种有效的手段来处理基于组件的多模式系统的模式转换，MSL允许用基于组件的开发方式来开发多模式系统，它包括 (1) 一种适合多模式环境的有模式意识的组件模型; (2) 一种模式映射机制，这种机制使得多模式的组件可以无缝叠加，并且可以引导多模式组件的模式转换; (3) 一种实时模式转换机制，它可以协调所有相关组件的模式转换从而保证整个系统能够正确而且高效的完成模式转换; (4) 对以MSL实现的模式转换进行时间分析。另外，MSL的所有关键元素都在本论文中通过一个案例被演示出来。

Acknowledgements

A couple of months ago, I decided to write a monograph as my Licentiate thesis, without knowing how challenging such an ambitious goal was. Fortunately, I managed to keep this promise after more than two months' torment from the thesis writing. I have been through a really tough period to fulfill this task, however, I still believe that I have made the right decision. The accomplishment of the thesis is actually beyond my own competence and effort. Herein, I shall express my gratitude to those who I ever got help from[1].

First, I would like to thank my supervisor Hans Hansson for assigning such an interesting research topic for me as well as his invaluable guidance even since I was a master student at MDH. Thanks to his enormous help and tutelage, I became more and more capable of doing research, in terms of investigating existing related works, solving hard research problems, writing good academic papers, and many other aspects. I am so impressed that he could know so many technical details of my work and even sometimes point out the hidden problems of my ideas from our discussions. It is definitely my supreme honor to be one of his PhD students! I also would like to thank my assistant supervisors Paul Pettersson and Thomas Nolte. As a UPPAAL expert, Paul has offered many helpful suggestions for solving my UPPAAL problems (hopefully you are not bothered too much by my occasional bizarre questions on UPPAAL). Also, thank Mikael Sjödin for being the examiner of my Licentiate proposal and providing so valuable feedbacks.

Many thanks go to Damir Isovic, whose visit to my home university ECUST in 2007 became the initial and key motivation for me, who had no ambition to go abroad before, to come to MDH. Now I am so pleased to witness the intimate cooperation between MDH and ECUST. Also thank him for giving me the opportunity to be exchanged to TU/e in Eindhoven in 2009. I re-

[1] Occasionally I may thank someone multiple times for different reasons.

ally got a hard time there (a good evidence is that I lost 5kg within five months), yet it became one of my most memorable experiences. I need to thank Damir again for recommending me to Hans for his PhD position when I was still in Eindhoven. In addition, I am grateful to my ex-supervisor in ECUST, Huifeng Wang, for her encouragement of my MDH plan. Moreover, thank Jiale Zhou and his family for their financial assistance before I came to Sweden. Jiale and I flew to Sweden together, exchanged to TU/e together, taking almost the same courses, and now we are both PhD students at MRTC. It is indeed hard to get rid of you:)

I want to express my special thanks to Jan Carlson who has aided me a lot in my research. Even though we are in different research groups, many novel ideas have popped up from our discussions and they are extremely beneficial for me. I sincerely appreciate Etienne Borde, who yet is not at MRTC any more, for his active help at the beginning of my PhD. Thank Björn Lisper for enlightening us about a model-checking approach which afterwards contributed to our ECRTS paper. Additional thanks go to Barbara Gallina and Dag Nyström, who ever inspired me to borrow some ideas from the classic two-phase commit protocol used in distributed databases for my own research. One more thing, thank Dag for his "illegal" buns in the ES meetings:) Besides, I would love to thank Cristina Seceleanu who has been so enthusiastic to help me with theorem proving (sorry that I did not do a good job and have to leave it as my future work for the time being).

Since I started my master study at MDH, I have taken numerous courses that established good foundations for my research. Coincidently, most of the lecturers and lab assistants of these courses now turn out to be my colleagues. Thank Damir Isovic and Moris Behnam for the course Real-Time Systems 1. Thank Gordana Dodig-Crnkovic and Jan Gustafsson for teaching me the research methodology. Thank Thomas Nolte and many others for the course Real-Time Systems 2. Thank Mikael Ekström for the course Sensor Techniques. Thank Lars Asplund, Fredrik Ekstrand, Jörgen Lidholm and the artist Mikael Genbergs for the robot project "Roony". We could not let Roony carry a Swedish house to the moon as required, but it was absolutely a lot of fun to be engaged in this project. Thank Paul Pettersson, Cristina Seceleanu, Hans Hansson and Leo Hatvani for the advanced software verification and validation course. Thank Thomas Nolte and Emma Nehrenheim for the PhD introduction course, which is undoubtedly my favorite PhD course that not only covers such a variety of hot and interesting topics related to research, but also has amused us in the coffee and lunch sessions:) Thank my supervisor Hans Hansson again for the research planning course. Thank Phil McMim from University of Sheffield

and Paul Pettersson for the search-based testing course. Thank Ivica Crnković, Séverine Sentilles and Aneta Vulgarakis for the advanced CBSE course which is rather conducive to my research. Thank Iain Bate from University of York, Sasikumar Punnekkat and Hans Hansson for the course Introduction to Safety-Critical Systems. Furthermore, my heartfelt thanks go to Reinder J. Bril, Uğur Keskin, and other members in the System Architecture and Networking (SAN) group in TU/e for their great help during my stay in Eindhoven.

I have been to six countries in recent two years for international conferences, the ARTIST summer school and the last PROGRESS annual trip. I was never alone for any trip, as I always got accompanied by a few MRTC fellows who made my each single journey hilarious. First I want to convey my gratitude to Ivica Crnković, Magnus Larsson and Jan Carlson who took care of me throughout CompArch 2012 this June. I also would like to thank Guillermo Rodriguez-Navas for being the local guide in Sóller for us in 2010 and now I am delighted to have him here at MRTC. Thank Thomas Nolte again for accompanying me in all the real-time conferences and for being a hot topic to start a conversation with other researchers (most people mentioned your name after identifying my affiliation). In addition, many thanks go to my other conference/summer school fellows: Andreas Gustavsson, Farhang Nemati, Josip Maras, Luka Lednicki, Mikael Åsberg, Mohammad Ashjaei, Moris Behnam, Rafia Inam, Saad Mubeen, Sara Afshar and Yue Lu.

I would love to acknowledge all the administrative staff at IDT who have made my daily affairs a lot easier: in particular, Carola Ryttersson, Susanne Fronnå, Gunnar Widforss, Malin Rosqvist, and Åsa Lundkvist.

I admit that sometimes I concentrate too much on working at the sacrifice of missing countless coffee breaks. However, I never forget the happy moments with so many of you from PhD/Licentiate parties, ES meetings/challenges and other various exciting activities. Apart from those who I have thanked above, I give my sincere thanks to Abhilash Thekkilakattil, Adnan Causevic, Aida Causevic, Andreas Ermedahl, Andreas Johnsen, Daniel Sundmark, Eduard Paul Enoiu, Federico Ciccozzi, Hüseyin Aysan, Irfan Šljivo, Jagadish Suryadevara, Jukka Mäki-Turja, Juraj Feljan, Kan Yu, Kivanc Doganay, Kristina Lundqvist, Mats Björkman, Meng Liu, Nima Moghaddami Khalilzad, Ning Xiong, Radu Dobrin, Raluca Marinescu (as your roommate, I apologize for being a bit taciturn these months due to my thesis writing. In case you have any complaint, please blame the thesis, not me:)), Sigrid Eldh, Stefan Bygde, Svetlana Girs and many others around me. Additional thanks are reserved for some of my former colleagues who are not working at MRTC any more, including Andreas Hjertström, my ex-roommate Stefan Björnander and Johan Kraft.

Working for the ARROWS project, I have been funded by Vetenskapsrådet. I am deeply thankful for its financial support.

I give my wholehearted thanks to my parents for their immense parental love and care. Although I have been away from home for more than eleven years (seven years in Shanghai and more than four years in Sweden) and family reunion has gradually become an annual event for us, they are always encouraging me, respecting my own decisions, and doing their best to support my life when I was still not economically independent. Also, I am so grateful to all my relatives in China. In particular, I cherish the memory of my dear grandmother who unfortunately left us just one week before I came back to China this August, desperate for seeing me in her last minute. She must be proud of me in the heaven right now.

Last but foremost, I would deliver my deepest thanks and love to my girlfriend Hongwan Qin who has brought the true jubilance in my life. 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! We have shared pains and joys while I am supervising your master thesis. I know that sometimes I am not lenient enough. Believe me, there is always a tradeoff between a good boyfriend and a good supervisor:)

Finally, thank everyone who is patient to go through my tedious acknowledgements. This is the only one non-technical part of the thesis and you may realize that it is still the most interesting part after comparing it with the main body of the thesis, which could be even more boring. Due to my limitation to provide an exhaustive list of people's names that deserve my appreciation, it is hard to ensure that I fail to mention any important people here. If this really happens, please do let me know and I will thank you twice next time in my PhD thesis:)

Thank you all!

Hang Yin
Västerås, October 29, 2012

P.S. My name is 尹航 which is literally translated into Yin Hang. According to Chinese tradition, the family name Yin is placed in front. However, my name is frequently written as Hang Yin in accordance with Western tradition.

List of Publications

Papers related to the thesis:

Conference and workshop papers

- *Timing analysis for mode switch in component-based multi-mode systems*. Yin Hang, Hans Hansson. In proceedings of 24th Euromicro Conference on Real-Time Systems (ECRTS'12), pages 255-264, Pisa, Italy, July, 2012.

- *Towards mode switch handling in component-based multi-mode systems*. Yin Hang, Jan Carlson, Hans Hansson. In proceedings of 15th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE'12), pages 183-188, Bertinoro, Italy, June, 2012.

- *A mode mapping mechanism for component-based multi-mode systems*. Yin Hang, Hans Hansson. In proceedings of 4th Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'11), pages 38-45, Vienna, Austria, November, 2011.

- *Timing analysis for a composable mode switch*. Yin Hang, Hans Hansson. In proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS'11), WIP, pages 15-18, Porto, Portugal, July, 2011.

- *Composable mode switch for component-based systems*. Yin Hang, Etienne Borde, Hans Hansson. In proceedings of 3rd Workshop on Adaptive and Reconfigurable Embedded Systems (APRES'11), pages 19-22, Chicago, April, 2011.

Technical reports

- *A UPPAAL model for timing analysis of atomic execution in component-based multi-mode systems*. Yin Hang, Hans Hansson, Technical Report, Mälardalen Real-Time Research Center (MRTC), Mälardalen University, Sweden, February, 2012.
- *A Mode Switch Logic for component-based multi-mode systems*. Yin Hang, Hans Hansson, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-261/2012-1-SE, Mälardalen Real-Time Research Centre (MRTC), Mälardalen University, Sweden, January, 2012.

Papers not related to the thesis:

Conference and workshop papers

- *Accelerating exact schedulability analysis for fixed-priority scheduling*. Yin Hang, Zhou Jiale, Uğur Keskin, Reinder J. Bril. In proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS'10), WIP, pages 5-8, Brussels, Belgium, July, 2010.

List of Acronyms

ACC	Adaptive Cruise Control, page 99
AEG	Atomic Execution Group, page 61
CBD	Component-Based Development, page 4
CBMMS	Component-Based Multi-Mode System, page 5
CBSE	Component-Based Software Engineering, page 1
DDM	Dominant Default Mode, page 49
DPS	Data Processing Status, page 64
EPF	Extra-Functional Property, page 18
MMA	Mode Mapping Automata, page 49
MSC	Mode Switch Completion, page 30
MSD	Mode Switch Denial, page 29
MSDM	Mode Switch Decision Maker, page 28
MSDT	Mode Switch Detecting Time, page 78
MSI	Mode Switch Instruction, page 29
MSL	Mode Switch Logic, page 8
MSP protocol	Mode Switch Propagation protocol, page 30
MSQ	Mode Switch Query, page 29

MSR	Mode Switch Request, page 29
MSS	Mode Switch Source, page 27
MST	Mode Switch Time, page 79
PHT	Primitive Handling Time, page 78
QoS	Quality of Service, page 3
QRT	Query Response Time, page 78
SCT	State Checking Time, page 78
WCET	Worst-Case Execution Time, page 3

List of Notations

$c_i \in PC$	c_i is a primitive component
$c_i \in CC$	c_i is a composite component
Top	the component at the top level of the component hierarchy
$c_i.p^{MSX}$	the dedicated mode switch port of c_i for exchanging mode switch information with the parent
$c_i.p_{in}^{MSX}$	the dedicated mode switch port of c_i for exchanging mode switch information with the subcomponents
$c_i.M$	the set of supported modes of c_i
$c_i.m \in M$	the current mode of c_i
$c_i.m^0 \in M$	the initial mode of c_i
P_{c_i}	the parent of c_i
SC_{c_i}	the set of subcomponents of c_i
$ASC_{c_i}^m$	the set of activated subcomponents of c_i when c_i is in m
$DSC_{c_i}^m$	the set of deactivated subcomponents of c_i when c_i is in m
l_{c_i}	the depth level of c_i
A_{c_i}	the set of ancestors of c_i
D_{c_i}	the set of descendants of c_i
$AD_{c_i}^m$	the set of activated descendants of c_i when c_i is in m
$DD_{c_i}^m$	the set of deactivated descendants of c_i when c_i is in m
$c_k : m_{c_k}^j \rightarrow m_{c_k}^j$	a mode switch scenario triggered by c_k from $m_{c_k}^i$ to $m_{c_k}^j$
$T_{c_i} = A \wedge B$	c_i is a Type A or Type B component in the current mode for a specific mode switch scenario
$c_i.t_\lambda$	the transmission time of the primitive λ within c_i
$c_i.msdt$	the MSDT of c_i as an MSS
$c_i.pht$	the PHT of c_i
$c_i.sct$	the SCT of c_i
$c_i.qrt$	the QRT of c_i

$c_i.rct$	the reconfiguration time of c_i
$c_i.mst$	the MST of c_i
AE_{G_i}	the atomic execution time of an AEG G_i

Contents

List of Figures

List of Tables

Chapter 1

Introduction

New techniques are demanded to manage the growing software complexity of embedded systems. A common and effective approach to reducing such complexity is to partition the system behavior into different operational modes. Each mode implies a distinctive system behavior and the system can switch between different modes at runtime as required. This multi-mode approach has already been successfully applied to real-world systems such as avionic and multimedia systems. Meanwhile, Component-Based Software Engineering (CBSE) is getting more and more attention by virtue of its approach to building a system by reusable software components. Since multi-mode and CBSE are complementary in the way they handle software complexity, the combination of both strategies, i.e. to build a multi-mode system in a component-based manner, is expected to bring added value for the development and execution of embedded systems. A key challenge in such a combination is to guarantee a correct and bounded mode switch. The prime aim of this thesis is to address this challenge.

In this first chapter, the background and motivation of this thesis will be introduced. We also identify our research challenges and sub-goals and summarize our contributions.

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 air-

crafts. Moreover, it has been witnessed in recent years that the never stopping evolution of embedded systems gives rise to more advanced and innovative functionalities. Unfortunately, an inevitable negative side effect is the rising software complexity, which is potentially a huge impediment to system design, as well as to system verification and validation. Obviously, traditional and classical software development methods become less and less appropriate for the more complex software. New techniques are required in order to manage such complexity.

A common approach in embedded system design is to partition the system behavior into different operational modes. Each mode corresponds to a specific system behavior. Some subsystems may run in all modes while there are also mode-specific subsystems, which run only in selected modes. A multi-mode system starts by running in a default mode and switches to another appropriate mode when some condition changes. A representative multi-mode system is the control software of an airplane, which could run in the modes *taxi* (the initial mode), *taking off*, *flight* and *landing*. Different subsystems are running 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.

There are various reasons motivating the use of multi-mode systems, including:

1. Reducing software complexity, as is mentioned above. Once a system behavior is partitioned into a number of modes, the associated software in different modes will be loosely coupled, meaning that they essentially can be independently developed, tested, analyzed and maintained. Hence, for each mode the software will be less complex compared to a monolithic system. Only the intermediate stage during a mode switch deserves extra care.

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. Multiple operational phases. The execution of many types of systems follows a series of sequential operational phases. Apart from the airplane example introduced above, most autonomous robotic systems run through different phases. For instance, an object-searching robot can

start by running in the initialization phase. After that it enters the object-searching phase, in which it moves and tries to detect any object. When an object is detected, the robot enters the next phase that can be pre-defined according to the system requirements, e.g. taking a photo of the object and then transmitting the photo to the monitoring centre. For such a multi-phase system, each phase is equivalent to a mode and the phase transition could be regarded as mode switch.

4. Adaptivity. A multi-mode system can be considered as a special type of adaptive system that actively adjusts its behavior and performance to accommodate the new condition. This is rather common for multimedia systems. For instance, an adaptive media player in an embedded device with constrained resource may run in the degraded Quality of Service (QoS) mode when the network bandwidth is becoming low and switch to the standard playing mode when the network condition returns to normal.

5. Saving resource. Since some tasks of a system are only running in certain conditions, a lot of resources would be wasted if all tasks are running all the time. A more efficient design is to deactivate the tasks that are not currently used in the system. The task running status can be taken as a criterium to specify modes. A drastic change of the set of running tasks is most likely to be a suitable instant for mode switch.

6. Fault-tolerance. Most safety-critical systems are guaranteed to be fault-tolerant so as to prevent catastrophic consequences. A practical fault-tolerant strategy is to define a *Safe* mode in case a fault occurs. This *Safe* mode can minimize the impact from a fault, or even eliminate the fault and guide the system to the normal running state.

7. More precise analysis of system properties. Many system properties, especially non-functional properties, are associated with a wide range of values, however, the maximum and the minimum 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. In contrast, different modes can be defined based on the execution time distribution or variation of some important task. Then the WCET of this task can be calculated or measured for each mode. For

instance, for a multi-mode system running in mode m_1, m_2 and m_3 and the WCET of one of its running tasks τ can be 5 in mode m_1, 10 in mode m_2 and 15 in mode m_3. Obviously, the use of modes gives more precise WCET analysis and thus improves schedulability. The same benefit applies to other system properties as well.

8. 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, a new mode can be introduced for the new function. In this way, 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.

Of course, multiple modes and mode switch is not the sole technique for managing software complexity. Another option is Component-Based Software Engineering (CBSE) (also known as Component-Based Development (CBD)) [12]. CBSE provides a promising design paradigm for the development of complex embedded systems. The key idea of CBSE is to build a system by reusable software components which can be independently developed and reused in different applications. In other words, a system does not have to be developed from scratch. Instead, some of its components 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 quite a number of appealing features, such as

- Reduced complexity. 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 the developer can focus more on how to compose components into a bigger system. Even though component reuse may increase the size of software, software complexity is reduced a lot thanks to the hierarchical structure and the abstraction at each level.

- 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, and thus the shorter time to market.

- Improved software quality. Reusing well tested and qualified components increases the chance of building a more reliable system.

The success of CBSE is evident in several well-known component models, such as Microsoft Component Object Model (COM) [9], Enterprise JavaBeans (EJB) [43] and Open Services Gateway Initiative (OSGi) [3]. Compared with desktop systems, component reuse in embedded systems is relatively more challenging, but is getting more and more attention in the industry of embedded systems, including component models such as Koala [47], Rubus [25], and AUTOSAR [1].

With the ambition of getting the benefits from multi-mode systems and CBSE, the target of this thesis is the combination of both approaches, which is Component-Based Multi-Mode Systems (CBMMSs):

Definition 1. ***[Component-Based Multi-Mode System (CBMMS)]:*** *A Component-Based Multi-Mode System (CBMMS) is a system which supports multiple operational modes and is built by reusable components.*

One may wonder what such a system looks like. A vivid and impressive representation is the *Transformers* toys, which are artificial but intuitive. The leading character of Transformers is *Optimus Prime* who can switch between the human mode and car mode. Optimus Prime is a CBMMS for two reasons: (1) he supports two operational modes and the mode switch between them; (2) he is composed by a vast number of (mostly hardware) components. Some components can also be further decomposed into smaller components. For instance, a tyre is the composition of the rubber part and the metal part. When Optimus Prime switches mode, some of his components will be reconfigured, e.g. by changing their physical layouts or shapes. Some components may also alter their roles after a mode switch. For example, his four tires are not used in the human mode, whereas in the car mode, the tires must be activated. Component composition can be further demonstrated by another character of the Transformers, *Devastator*, composed by six smaller Transformers. Each of these six Transformers is a multi-mode component (supporting two modes, say the human mode and the composition mode). All these six Transformers must switch to their composition mode to construct Devastator.

The Transformer example provides a first impression of a CBMMS built by hardware components. In reality, the behavior of these hardware components must be realized by software components. Our focus is on software components rather than hardware components. As a conceptual example, the left part of Figure 1.1 illustrates the hierarchical component structure of a typical CBMMS. The system consists of three components: a, b and c. Component b is

composed of two other components: *d* and *e*. With respect to the terminology of CBSE, we distinguish two basic types of components: (1) a *primitive component*, whose behavior is given directly by associated software, thus cannot be further decomposed into other components; (2) a *composite component* which is a composition of other components. In Figure 1.1, *a*, *c*, *d*, and *e* are primitive components whereas *Top* and *b* are composite components. Since the component hierarchy has a tree structure, a composite component and its subcomponents have a parent-and-children relationship. For instance, *b* is the parent of *d* and *e*, which in turn are the children of *b*. Moreover, the system, i.e. Component *Top*, supports two modes: m^1_{Top} and m^2_{Top}. When the system is in mode m^1_{Top}, Component *c* is deactivated (i.e. not running), as the component hierarchy in Figure 1.1 shows by not displaying *c* in mode m^1_{Top}. In contrast, when the system is in mode m^2_{Top}, *c* is activated whilst *e* is deactivated. Besides, Component *a* has different mode-specific behaviors represented by black and grey colors in Figure 1.1. The right part of Figure 1.1, consistent with the left part, presents the connections between these components. A component is connected to another component via its input or output ports (squares in Figure 1.1).

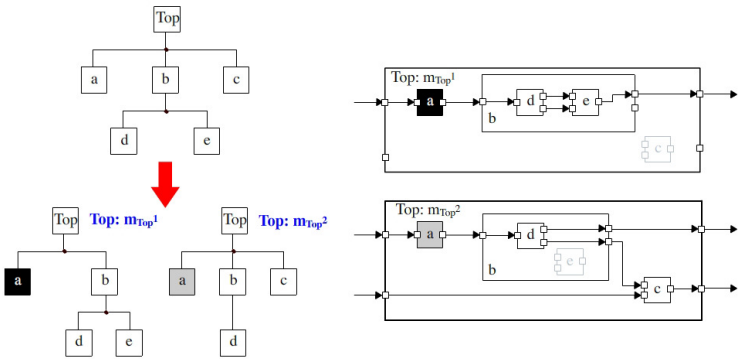

Figure 1.1: A component-based multi-mode system

The system in Figure 1.1 is able to switch between its supported modes m^1_{Top} and m^2_{Top}. Figure 1.2 depicts its mode switch from m^1_{Top} to m^2_{Top}, starting at time t_1 and ending at time t_2. We call m^1_{Top} and m^2_{Top} *stable modes*. During the mode switch within the interval [t_1,t_2], the system is in an intermediate transition state, thus not running in any stable mode. From

Figure 1.1, it can be observed that a CBMMS exhibits unique system behavior in each mode reflected by particular configurations of its components at various levels. The system mode switch is achieved by the joint mode switches of components composed together to form the complete system. For instance, the mode switch of the system from m^1_{Top} to m^2_{Top} in Figure 1.1 corresponds to the following component mode switches:

- Component *a* changes its mode-specific behavior (indicated by the color change from black to grey).
- Component *e* is informed to change its running status from "activated" to "deactivated".
- Component *b* changes the running statuses and connections of its subcomponents (*d* and *e*) as well as the configuration of its ports (its second output port becomes activated in the new mode).
- Component *c* is informed to change its running status from "deactivated" to "activated".
- Component *Top* changes the running statuses and connections of its subcomponents as well as the configuration of its ports.

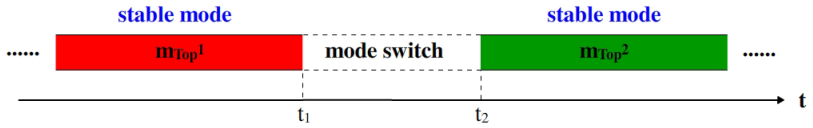

Figure 1.2: Mode switch illustration

We call this *composable mode switch*:

Definition 2. ***[Composable mode switch]:*** *A composable mode switch is the mode switch of an entire CBMMS or its subsystem represented by the joint mode switches of its components at various hierarchical levels.*

A successful composable mode switch not only relies on the successful mode switch of each single component, but also relies on the correct synchronization and coordination of the mode switches of different components. Unlike the traditional mode switch problem, composable mode switch is novel.

Traditional mode switch handling usually does not assume systems built by reusable components, thus the mode switch is not composable and traditional approaches cannot be directly applied to composable mode switch. Furthermore, traditional component-based systems usually do not consider multiple operational modes and mode switch. In this thesis, we have presented a mode switch mechanism, which we call the Mode Switch Logic (MSL). Parts and preliminary versions of our MSL have been presented in [18] [19]. MSL does not only allow the seamless composition of single-mode or multi-mode components but is also capable of handling the composable mode switch of CBMMSs.

1.2 Problem description

1.2.1 Challenges of composable mode switch

The handling of composable mode switch is not trivial, as it poses multiple potential problems and challenges. Many contributing factors need to be considered, such as component model, component hierarchy, component connection, system architecture and component execution pattern. The following are the major challenges that have to be addressed:

The component model

CBSE specifies that a software component must conform to a component model. A large number of component models have already been proposed targeting various application domains [13] [32]. Among these component models, only a few have multi-mode support, e.g. COMDES-II [35], MyCCM-HI [8], Rubus [25] and SaveCCM [26]. To compose multi-mode components in a CBMMS, traditional component models must be extended, since in a CBMMS a component must be aware of its own modes which could be switched at runtime. In our MSL, we propose a mode-aware component model taking into account both the mode switch of a single component and the mode switch synchronization between different components.

Multi-mode component composition

A CBMMS or multi-mode component can be composed of both single-mode and multi-mode components. For a multi-mode component, its supported modes should also be composable. Since reusable components typically are

developed independently, it is very likely that the supported modes and the number of supported modes differ between components. Hence, when a composite component is built by reusable components, the supported modes of each component are likely to be inconsistent with the desired modes of the composed component. Therefore, the mapping between the modes of different components must be defined during composition. In our MSL, this problem is settled by our mode mapping mechanism [20].

The mode switch runtime mechanism

A composable mode switch must be carried out under the guidance of some runtime mechanism that ensures correctness and efficiency. This is to a large extent neglected by existing mode switch handling approaches. Such a runtime mechanism includes many aspects, such as

- **Mode switch propagation:** A mode switch can be triggered by any component and should be propagated to other components that also need to change their modes as a consequence. Since a component has no global knowledge of the system component hierarchy, a mode switch event cannot simply be broadcast from one component to all the other components. Instead, a stepwise propagation method is required. By taking advantage of the hierarchical component structure, a mode switch event can be directly or indirectly propagated to any related component. Later we shall show how this can be achieved by our Mode Switch Propagation (MSP) protocol[1].
- **The guarantee of consistent mode switch:** The mode switch of a system may correspond to the mode switches of many components. When the system completes a mode switch, all its components must be in a consistent state. For instance, components supposed to run in the new mode must not run in the old mode after the system has completed a mode switch. The MSP protocol plays a significant role in notifying different components of the mode switch event, however, additional rules should be applied to guarantee the overall mode switch consistency. In our MSL, we have defined a mode switch dependency rule that guarantees a consistent mode switch.
- **Mode switch and atomic execution:** When a mode switch is triggered, the CBMMS is supposed to stop running in the old mode and start its

[1] Our MSP protocol was previously [19] called MS propagation mechanism.

mode switch as soon as possible. However, there could be ongoing executions in one or a set of components that cannot be aborted by a mode switch triggering. Our MSL handles such atomic component execution by an extended MSP protocol [23].

- **Conflict handling for multiple mode switch triggering:** Normally, a CBMMS performs a mode switch rather swiftly, yet not instantaneously. Multiple mode switch triggering could happen either simultaneously or within a short interval. Without proper treatment, an ongoing mode switch could be compromised by the triggering of a new mode switch. In our MSL, the initial solution is to use an arbitration mechanism managed by a component with high authority to resolve the conflict [23]. Further details and a more general in-depth solution are included in our future work.

The timing analysis of a composable mode switch

Since most CBMMSs are also real-time systems, not only must the system functionality be correct, but also the timing constraints must be satisfied. In particular, the mode switch time must be bounded and analyzable. The mode switch time highly depends upon the mechanism implementing the composable mode switch.

1.2.2 Research goal

Since the composable mode switch of CBMMSs has been rarely exploited to date, our research goal boils down to:

Development of a systematic and practical approach which not only effectively and efficiently copes with the composable mode switch of CBMMSs, but also is amenable to analysis and verification of timing and functional correctness.

1.2.3 Research sub-goals

Our research goal is ambitious in the sense that it is the first attempt to realize the seamless composition of multi-mode components into CBMMSs as well as its mode switch handling. We decompose our research goal into a set of major research sub-goals (SGs):

SG1: *To build a multi-mode system by single-mode or multi-mode components in a composable manner.*

SG2: *To efficiently achieve a composable mode switch of a CBMMS.*

SG3: *To analyze the mode switch time of a CBMMS.*

1.3 Thesis contributions

The contribution of this thesis realizes the research sub-goals formulated above, including:

- A generic mode-aware component model which enables a multi-mode component to support composable mode switch (for **SG1**).
- A mode mapping mechanism which can be easily implemented in each composite component and overcome the mode incompatibility problem during composition (for **SG1**).
- A mode switch runtime mechanism capable of efficiently handling a composable mode switch (for **SG2**).
- The timing analysis of composable mode switch guided by our MSL (for **SG3**).

1.4 Publications

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

1. **Composable mode switch for component-based systems,** Yin Hang, Etienne Borde, Hans Hansson, 3rd Workshop on Adaptive and Reconfigurable Embedded Systems (APRES 2011), p 19-22, Editor(s): Sebastian Fischmeister, Linh TX Phan, April, 2011

Abstract: Component-Based Development (CBD) reduces development time

and effort by allowing systems to be built from pre-developed reusable components. A classical approach to reduce embedded systems design and runtime complexity is to partition the behavior into a set of major system modes. In supporting system modes in CBD, a key issue is seamless composition of multi-mode components into systems.

In addressing this issue, we present a mode switch logic and algorithm for component-based multi-mode systems. The algorithm implements seamless coordination and synchronization of mode switch in systems composed of independently developed components. The paper provides formally defined semantics covering aspects relevant for mode switch, together with algorithms implementing mode switch rules for different types of components. The approach is illustrated by a simple example.

Thesis contribution: This paper proposes the initial version of our MSL, including the mode-aware component model, mode switch propagation and the mode switch dependency rule. It contributes to chapters 2 and 3.

My contribution: Yin Hang is the main contributor and main author of this paper, inspired by the useful comments and suggestions from Etienne Borde and Hans Hansson.

2. **A mode mapping mechanism for component-based multi-mode systems,** Yin Hang, Hans Hansson, 4th Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS 2011), p 38-45, Editor(s): Robert I. Davis and Linh T.X. Phan, November, 2011

Abstract: Component-Based Development (CBD) reduces development time and effort by allowing systems to be built from pre-developed reusable components. A classical approach to reduce embedded systems design and runtime complexity is to partition the behavior into a set of major system modes. In supporting system modes in CBD, a key issue is seamless composition of multi-mode components into systems. In addressing this issue, we previously developed a Mode Switch Logic (MSL) for component-based multi-mode systems. Our MSL implements seamless coordination and synchronization of mode switch in systems composed of independently developed components. However, our original MSL is based on the, in a setting of reusable components, unrealistic assumption, that all the components of a system support the same modes. This considerably limits the feasibility of our MSL. In this paper we lift this assumption and propose a mode mapping mechanism that enables assembly of components supporting different sets of modes. We demonstrate our mode mapping mechanism by a simple example application.

Thesis contribution: This paper identifies the mode incompatibility problem of composing multi-mode components and proposes a mode mapping mechanism as the corresponding solution. It contributes to Chapter 4.

My contribution: Yin Hang is the main contributor and main author of this paper. This work is carried out in close cooperation with Hans Hansson.

3. **Towards mode switch handling in component-based multi-mode systems,** Yin Hang, Jan Carlson, Hans Hansson, 15th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE 2012), p 183-188, Editor(s): Nenad Medvidovic and Magnus Larsson, June, 2012

Abstract: Component-Based Software Engineering (CBSE) is becoming a prominent solution to the development of complex embedded systems. Meanwhile, partitioning system behavior into different modes is an effective approach to reduce system complexity. Combining the two, we get a component-based multi-mode system, for which a key issue is its mode switch handling. The mode switch of such a system corresponds to the joint mode switches of many hierarchically organized components. Such a composable mode switch is not trivial as it amounts to the mode switch coordination of different components. In this paper, we identify the major challenges of the composable mode switch handling and classify existing approaches with respect to how they handle these challenges. We also provide a more detailed presentation of the corresponding solutions included in our approach – the Mode Switch Logic (MSL).

Thesis contribution: This paper identifies the major challenges of handling the mode switch of a CBMMS and classifies a number of existing component models in terms of their mode switch handling. Moreover, this paper also revises the mode-aware component model, the MSP protocol and the mode switch dependency rule of MSL. It contributes to chapters 1, 2 and 3.

My contribution: Yin Hang is the main contributor and main author of this paper. This work is inspired by the intensive discussion with Jan Carlson and Hans Hansson.

4. **Timing analysis for mode switch in component-based multi-mode systems,** Yin Hang, Hans Hansson, 24th Euromicro Conference on Real-Time Systems (ECRTS 2012), p 255-264, Editor(s): Robert I. Davis, July, 2012.

Abstract: Component-Based Development (CBD) reduces development time and effort by allowing systems to be built from pre-developed reusable compo-

nents. Partitioning the behavior into a set of major *operational modes* is a classical approach to reduce complexity of embedded systems design and execution. In supporting system modes in CBD, a key issue is seamless composition of pre-developed multi-mode components into systems. We have previously developed a Mode Switch Logic (MSL) for component-based multi-mode systems implementing such seamless composition.

In this paper we extend our MSL to cope with atomic transactions, i.e., to handle sets of components that must not be aborted in the middle of the processing of data. This is in contrast with our original MSL, in which components are immediately aborted to perform a mode switch. Based on our extended MSL, we provide analysis of the mode switch timing.

Thesis contribution: This paper extends MSL by taking atomic component execution into account and performs the mode switch timing analysis for MSL. A preliminary approach is also proposed to resolve the conflict due to multiple mode switch triggering. It mainly contributes to chapters 5 and 7.

My contribution: Yin Hang is the main contributor and main author of this paper. This work is carried out in close cooperation with Hans Hansson.

1.5 Research methodology

The methodology adopted in our research can be summarized as follows:

- Literature review is always a key preliminary step before a research goal or sub-goal is established. Investigated works are mostly related to mode switch problems, CBSE, and adaptive embedded systems.
- Once the research goal is identified, a couple of sub-goals are proposed. A research sub-goal is usually further divided into a couple of sub-problems which can be handled separately. We can simplify a problem by initially making simplifying assumptions. Once we come up with a solution, some assumptions will be lifted to yield a more realistic solution. This process may iterate several times as the solution is incrementally developed.
- Each proposed solution is initially evaluated and verified via the modeling of some representative conceptual examples which are helpful for spotting potential design errors and the refinement and revision of the

corresponding solution. Currently the evaluation is mostly based on the extensive use of UPPAAL [38] modeling and verification. However, we also intend to perform extensive evaluation in future.

- We publish our research results yielded at different stages. This dissemination provides feedback and discussion that helps refine and shape our research. For instance, reviewers' feedback and discussions with fellow researchers at conferences is instrumental in identifying underlying problems that we have overlooked, and could also suggest improvements and relevant literature in neighboring areas.

The research methodology is further illustrated in Figure 1.3.

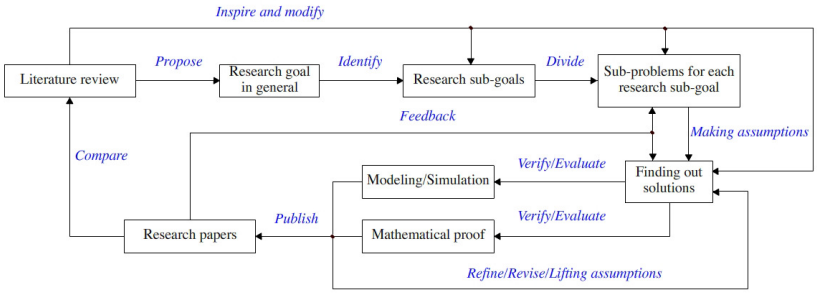

Figure 1.3: Research methodology

1.6 Thesis outline

This thesis is a monograph consisting of 10 chapters and the outline of the rest of the thesis is organized as follows:

- **Chapter 2 — The mode-aware component model:** introduces the mode-aware component model for both primitive and composite components. The mode-aware component model is generic and can be applied as a mode switch extension to many existing component models.

- **Chapter 3 — The mode switch runtime mechanism:** elaborates how a composable mode switch is handled at runtime. The mode switch runtime mechanism in this chapter includes the Mode Switch Propagation

(MSP) protocol and the mode switch dependency rule. The MSP protocol propagates the mode switch request from a component to all other related components. The mode switch dependency rule guarantees the mode consistency between different components after each mode switch.

- **Chapter 4 — Mode mapping:** describes the mode mapping mechanism of MSL for overcoming the mode incompatibility problem for composing multi-mode components. The purpose of mode mapping is twofold. First, it maps the modes of different components during the composition. Second, it derives the new mode of each component at runtime in cooperation with the MSP protocol.

- **Chapter 5 — The handling of atomic component execution:** extends the MSP protocol by introducing the support for atomic component execution.

- **Chapter 6 — Algorithms for the mode switch runtime mechanism:** implements the mode switch runtime mechanism as algorithms for primitive components, non-composite components and the top component, respectively.

- **Chapter 7 — The composable mode switch timing analysis:** performs the mode switch timing analysis based on our MSL. Furthermore, a model-checking approach is devised to obtain the worst-case atomic component execution time assuming the pipe-and-filter architectural style.

- **Chapter 8 — Case study–An Adaptive Cruise Control system:** demonstrates the usage of MSL in a case study, an Adaptive Cruise Control (ACC) system which is theoretically designed as a CBMMS in the thesis.

- **Chapter 9 — Related work:** presents existing work related to our MSL in various research domains such as component models, real-time systems and specification languages.

- **Chapter 10 — Conclusions and future work:** summarizes the thesis and discusses the future work. Additionally, the evolution trace of MSL is also provided.

Chapter 2

The mode-aware component model

Traditionally, CBSE assumes the composition of single-mode components. In addition to this, a CBMMS or a multi-mode component should allow the composition of both single-mode and multi-mode components. This imposes extra requirement on the component model and the composition process. A mode-aware (or multi-mode) component is different from traditional components in terms of the interface, internal properties, and many other aspects. In this chapter, we propose a mode-aware component model for both primitive and composite components. This component model is rather generic, with emphasis on modes and mode switch. It is not intended as a complete component model, rather as a basis for mode-aware extensions of existing component models that cover a more complete set of features.

2.1 The mode-aware component model and its definition

Just like traditional port-based component models, both primitive and composite components in our component model have a set of ports to communicate with neighboring components. Furthermore,

- Both primitive and composite components support one or more modes. Each mode of a multi-mode component is associated with a unique mode

identifier (ID).

- A primitive component has a dedicated mode switch port for the communication with its parent during a mode switch. A composite component has an additional dedicated port for the communication with its subcomponents/children.
- Both primitive and composite components have a separate configuration in each mode. A component configuration is a collection of mode-specific information such as the mode-specific functional behavior and mode-related properties of a component, as well as mode-independent information.
- A component starts to reconfigure itself by changing its configuration to the configuration in the new mode during a mode switch. The mode switch of a component is controlled by the runtime mechanism of MSL integrated in the component.

Let PC denote the set of primitive components and CC denote the set of composite components of a CBMMS. In the following, the formal definition of the mode-aware component model will be given for primitive components and composite components, respectively.

Definition 3. ***[Mode-aware primitive component]:*** *A mode-aware primitive component $c_i \in PC$ is a tuple:*

$$< \text{IP}, \text{OP}, p^{\text{MSX}}, \text{B}, \text{MIP}, \text{MDEFP}, \text{m}, \text{MB}, \text{AIP}, \text{AOP}, \text{MS}, \text{MP}, \text{MSRM} >$$

where IP *is the set of input ports of c_i;* OP *is the set of output ports of c_i; $p^{\text{MSX}} \notin (\text{IP} \cup \text{OP})$ is a dedicated bidirectional mode switch port for exchanging mode-related information with the parent;* B *is the set of mode-specific behaviors of c_i;* MIP *is the set of mode-independent properties of c_i (including* M*, the set of supported modes of c_i.);* MDEFP *is the set of mode-dependent Extra-Functional Properties (EFPs) of c_i;* m *is the current mode of c_i; the function* $\text{MB} : \text{M} \rightarrow \text{B}$ *defines the functional behavior of c_i in each mode; the function* $\text{AIP} : \text{M} \rightarrow 2^{\text{IP}}$ *defines the set of activated input ports of c_i in each mode; the function* $\text{AOP} : \text{M} \rightarrow 2^{\text{OP}}$ *defines the set of activated output ports of c_i in each mode; the function* $\text{MS} : \text{M} \rightarrow \{\textit{Activated}, \textit{Deactivated}\}$ *indicates the running status*[1] *(either activated or deactivated) of c_i in each mode; the*

[1] The running status of a component c_i can be considered as a property in a specific mode, as is defined here. Alternatively, the deactivated running status of c_i may also be considered as a dedicated mode.

function MP : M $\times$ MDEFP $\rightarrow$ Q *assigns values to each mode-dependent EFP of* c_i *in each mode, where the set* Q *is such that for a mode* m *and a mode-dependent EFP* e, MP$(m, e) \in dom(e)$*—the range of values of* e*;* MSRM *is the mode switch runtime mechanism of* c_i.

Since p^{MSX} is dedicated to mode switch, we separate it from *IP* and *OP*. *MSRM* is used to control the mode switch of c_i and will be defined in Chapter 3. It should be noted that *MSRM* in the tuple above can be considered as an internal controller of c_i. When c_i is used in a specific context, the *MSRM* of c_i should not be exposed like its mode information and EFPs. Besides,

Definition 4. *The* MIP *of a mode-aware primitive component* c_i *is another tuple:*

$$< \text{M}, m^0, \text{MIEFP} >$$

where M *as explained above is the set of supported modes of* c_i*;* m^0 *is the initial mode of* c_i*;* MIEFP *is the set of mode-independent EFPs of* c_i.

Actually, *MIEFP* and *MDEFP* of c_i are two disjoint sets and their union is *EFP*, i.e. the set of all EFPs of c_i. EFP, sometimes also called non-functional property, is a key concept in CBSE. An EFP is not related to the functional aspect of a system. Instead, it covers aspects such as performance and dependability. An EFP can be associated with one or more parameters. When the parameter value(s) of an EFP is(are) mode-dependent, this EFP belongs to *MDEFP*. Otherwise, it belongs to *MIEFP*. The following lists a couple of typical EFPs:

- Resource consumption. This EFP defines the resource requirement of a component. The resource here can be power, CPU cycles, memory, bandwidth or others. If a multi-mode component consumes different amount of resources in different modes, then resource consumption belongs to *MDEFP*.
- Atomicity. If the execution of a component is atomic, it cannot be interrupted by any other event before completion. If the execution of a component is non-atomic, then it can be interrupted by another event at any time. In Chapter 5, atomic component execution will be revisited and we shall present how it is handled in our MSL.
- Real-time properties, such as Worst-Case Execution Time (WCET), triggering period, maximum and minimum time for the component to stay in one mode, and even scheduling policy.

Please note that it is possible to define the scheduling policy as an EFP for a component which then can have its own scheduler. This is to some extent congruous with hierarchical scheduling [14]. Compared with a primitive component, the mode-aware component model of a composite component is more complex, since a composite component must know the essential information of itself, its subcomponents and how they are composed.

Definition 5. ***[Mode-aware composite component]:*** *A mode-aware composite component $c_i \in CC$ is a tuple:*

$$<\text{IP}, \text{OP}, p^{\text{MSX}}, p_{\text{in}}^{\text{MSX}}, \text{SC}, \text{Con}, \text{MIP}, \text{MDEFP}, \text{m}, \\ \text{AIP}, \text{AOP}, \text{MS}, \text{ASC}, \text{DSC}, \text{ACon}, \text{MP}, \text{MSRM} >$$

where IP*,* OP*,* p^{MSX}*,* MDEFP*,* m*,* AIP*,* AOP*,* MS*,* MP *and* MSRM *are defined as in Definition 3;* $p_{\text{in}}^{\text{MSX}}$ *is a second dedicated mode switch port for exchanging mode-related information with the children of* c_i*;* SC *is the set the subcomponents of* c_i*;* $\text{Con} \subseteq (P_{SC} \cup \text{IP} \cup \text{OP}) \times (P_{SC} \cup \text{IP} \cup \text{OP})$ *is the set of all inner component connections of* c_i *in all modes, where* P_{SC} *is the set of ports of* SC*:*

$$P_{SC} = \bigcup_{\forall c_j \in SC_{c_i}} \text{IP}_{c_j} \cup \text{OP}_{c_j};$$

MIP *is the set of mode-independent EFPs of* c_i *as defined in Definition 6; the function* $\text{ASC} : \text{M} \rightarrow 2^{\text{SC}}$ *indicates the activated subcomponents of* c_i *in each mode; the function* $\text{DSC} : \text{M} \rightarrow 2^{\text{SC}}$ *indicates the deactivated subcomponents of* c_i *in each mode; the function* $\text{ACon} : \text{M} \rightarrow 2^{\text{Con}}$ *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. It should be noted that some of the elements in the tuple for $c_i \in CC$ are essential for the mode switch handling of c_i but should not be exposed for component reuse, including *SC*, *Con*, *ASC*, *DSC*, *ACon*, and *MSRM*. Besides,

Definition 6. *The* MIP *of a mode-aware composite component* c_i *is another tuple:*

$$< \text{M}, m^0, \text{M}_{SC}, m_{SC}^0, m_{SC}, \text{MM}, \text{MIEFP} >$$

where M*,* m_0 *and* MIEFP *are defined as in Definition 4; the function* $\text{M}_{SC} : SC \rightarrow 2^{\mathcal{M}}$ *maps each subcomponent of* c_i *to the corresponding set of supported*

modes ($\mathcal{M}$ is the set of all modes supported by c_i and SC_{c_i}); the function m^0_{SC} : SC $\rightarrow 2^{\mathcal{M}}$ maps each subcomponent of c_i to the corresponding initial mode; the function m_{SC} : SC $\rightarrow 2^{\mathcal{M}}$ maps each subcomponent of c_i to the corresponding current mode; MM *is the mode mapping between c_i and SC_{c_i} and will be further explained later in Chapter 4.*

In the tuple for the *MIP* of $c_i \in CC$, elements M_{SC}, m^0_{SC}, m_{SC}, and *MM* should not be exposed for component reuse. The *MIEFP* and *MDEFP* of $c_i \in CC$ can be exposed for component reuse because the EFPs of c_i has been derived based on the EFPs of its subcomponents.

Our mode-aware component model for primitive and composite components are illustrated in figures 2.1 and 2.2, where the similarity and discrepancy of primitive and composite components can be easily perceived. Generally speaking, this component model assumes the pipe-and-filter type of communication and it is data-driven rather than control-driven [39]. Yet the model itself should be able to support both data-driven and control-driven computation patterns. In both figures, $p^0_{in} \cdots p^m_{in}$ denote the input ports and $p^0_{out} \cdots p^n_{out}$ denote the output ports. p^{MSX} and p^{MSX}_{in} are the dedicated mode switch ports. A unique configuration is defined for each mode k and the included elements for each configuration are rather consistent with the aforementioned definitions.

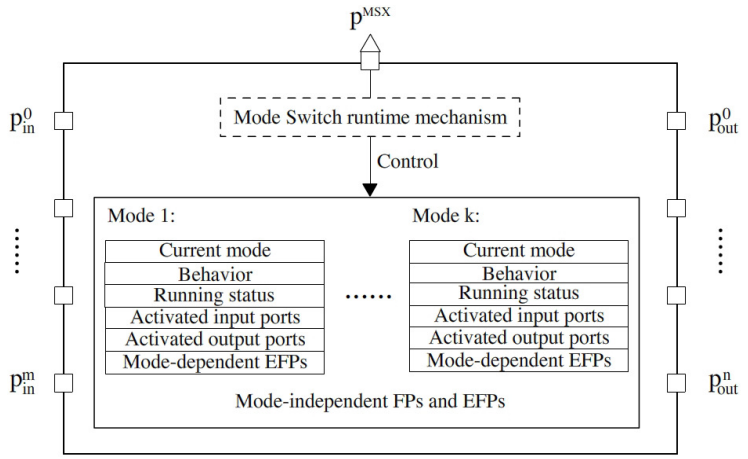

Figure 2.1: The mode-aware component model for primitive components

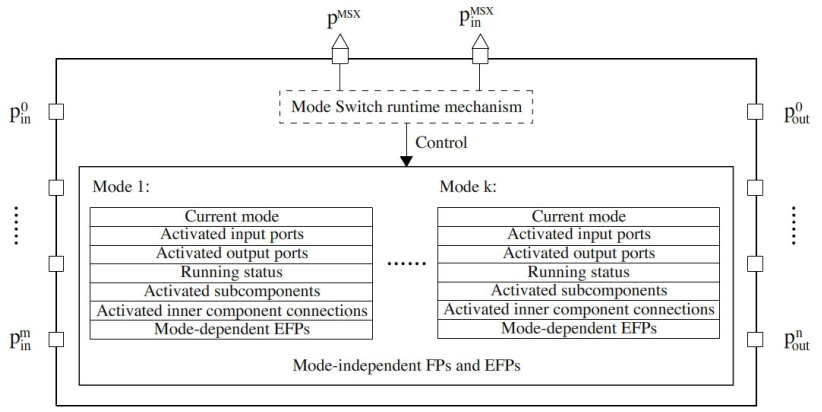

Figure 2.2: The mode-aware component model for composite components

2.2 An example

Our mode-aware component model can be used for the formal definition of a multi-mode component. For instance, let's formally define the primitive component *a* and the composite component *b* in the example introduced in Figure 1.1. Figure 2.3 labels each component with its port names in red. Suppose the set of supported modes of *a* is denoted by $a.M = \{m_a^1, m_a^2\}$ and the mode-specific behavior of *a* is α in m_a^1 and β in m_a^2. The behaviors α and β can, for instance, be two different video decoding schemes for a multimedia application. Component *a* has a mode-independent EFP, CPU consumption (denoted by cpu) that is the same in both modes, e.g. $cpu = 8$, and *a* also has a mode-dependent EFP, memory consumption (denoted by mem) that is different in different modes, e.g. $mem = 5$ in m_a^1 and $mem = 10$ in m_a^2. Then Component *a* can be formally defined by the tuple,

$$<a.IP, a.OP, a.p^{MSX}, a.B, a.MIP, a.MDEFP, a.m,$$
$$a.MB, a.AIP, a.AOP, a.MS, a.MP, a.MSRM >$$

where

$$
\begin{array}{lcl}
a.IP & = & \{a.p_{in}^0\} \\
a.OP & = & \{a.p_{out}^0\} \\
a.B & = & \{\alpha, \beta\} \\
a.MDEFP & = & \{mem\} \\
a.m & = & m_a^1 \\
a.MB & = & \{m_a^1 \rightarrow \alpha, m_a^2 \rightarrow \beta\} \\
a.AIP & = & \{m_a^1 \rightarrow \{a.p_{in}^0\}, m_a^2 \rightarrow \{a.p_{in}^0\}\} \\
a.AOP & = & \{m_a^1 \rightarrow \{a.p_{out}^0\}, m_a^2 \rightarrow \{a.p_{out}^0\}\} \\
a.MS & = & \{m_a^1 \rightarrow Activated, m_a^2 \rightarrow Activated\} \\
a.MP & = & \{(m_a^1, mem) \rightarrow 5, (m_a^2, mem) \rightarrow 10\}
\end{array}
$$

In addition, $a.MIP$ is another tuple,

$$< a.M, a.m^0, a.MIEFP >$$

where

$$
\begin{array}{lcl}
a.M & = & \{m_a^1, m_a^2\} \\
a.m^0 & = & m_a^1 \\
a.MIEFP & = & \{cpu = 8\}
\end{array}
$$

Here $a.MSRM$ cannot be simply presented in the tuple and it will be further discussed in the next chapter. The dedicated mode switch port $a.p^{MSX}$ is the same for all primitive components, thus there is no need to describe it further.

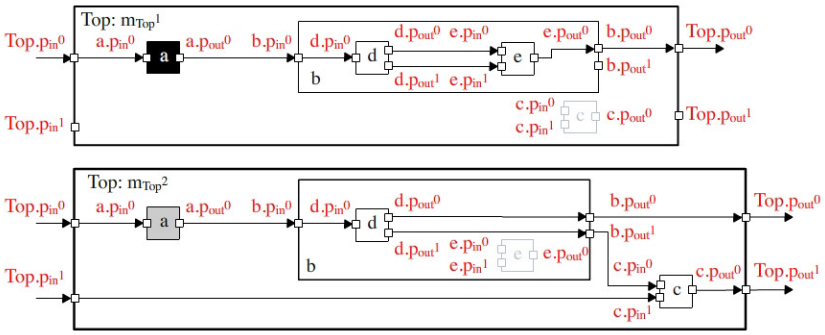

Figure 2.3: A CBMMS marked with port names

Suppose the set of supported modes of b is $b.M$=$\{m_b^1, m_b^2\}$. Component b has WCET (denoted as $wcet$) as its *MDEFP* such that $wcet = 75$ in m_b^1 and

$wcet = 50$ in m_b^2. Also, *b* has a mode-independent EFP, i.e. its activation period $T = 100$ in all modes. As a composite component, *b* can be formally defined by the tuple,

$$<b.IP, b.OP, b.p^{MSX}, b.p_{in}^{MSX}, b.SC, b.Con, b.MIP, b.MDEFP, b.m,$$
$$b.AIP, b.AOP, b.MS, b.ASC, b.DSC, b.ACon, b.MP, b.MSRM >$$

where there is no need to describe b.p^{MSX} and b.p_{in}^{MSX}, and $b.MSRM$ will be further discussed in the next chapter, and

$$\begin{array}{lcl}
b.IP & = & \{b.p_{in}^0\} \\
b.OP & = & \{b.p_{out}^0, b.p_{out}^1\} \\
b.SC & = & \{d, e\} \\
b.Con & = & \{(b.p_{in}^0, d.p_{in}^0), (d.p_{out}^0, e.p_{in}^0), (d.p_{out}^1, e.p_{in}1), \\
 & & (e.p_{out}^0, b.p_{out}^0), (d.p_{out}^0, b.p_{out^0}), (d.p_{out}^1, b.p_{out}^1)\} \\
b.MDEFP & = & \{wcet\} \\
b.m & = & m_b^2 \\
b.AIP & = & \{m_b^1 \rightarrow \{b.p_{in}^0\}, m_b^2 \rightarrow \{b.p_{in}^0\}\} \\
b.AOP & = & \{m_b^1 \rightarrow \{b.p_{out}^0\}, m_b^2 \rightarrow \{b.p_{out}^0, b.p_{out}^1\}\} \\
b.MS & = & \{m_b^1 \rightarrow Activated, m_b^2 \rightarrow Activated\} \\
b.ASC & = & \{m_b^1 \rightarrow \{d, e\}, m_b^2 \rightarrow \{d\}\} \\
b.DSC & = & \{m_b^1 \rightarrow \varnothing, m_b^2 \rightarrow \{e\}\} \\
b.ACon & = & \{m_b^1 \rightarrow \{(b.p_{in}^0, d.p_{in}^0), (d.p_{out}^0, e.p_{in}^0), (d.p_{out}^1, e.p_{in}1), \\
 & & (e.p_{out}^0, b.p_{out}^0)\}, m_b^2 \rightarrow \{(b.p_{in}^0, d.p_{in}^0), (d.p_{out}^0, b.p_{out^0}), \\
 & & (d.p_{out}^1, b.p_{out}^1)\}\} \\
b.MP & = & \{(m_b^1, wcet) \rightarrow 75, (m_b^2, wcet) \rightarrow 50\}
\end{array}$$

In addition, $b.MIP$ is another tuple,

$$< b.M, b.m^0, b.M_{\text{SC}}, b.m_{\text{SC}}^0, b.m_{\text{SC}}, b.MM, b.MIEFP >$$

where $b.MM$ presents the mode mapping of *b* and will be explained in Chapter 4, and

$$\begin{array}{lcl}
b.M & = & \{m_b^1, m_b^2\} \\
b.m^0 & = & m_b^1 \\
b.M_{\text{SC}} & = & \{d \rightarrow \{m_d^1\}, e \rightarrow \{m_e^1, m_e^2\}\} \\
b.m_{\text{SC}}^0 & = & \{d \rightarrow m_d^1, e \rightarrow m_e^1\} \\
b.m_{\text{SC}} & = & \{d \rightarrow m_d^1, e \rightarrow m_e^2\} \\
b.MIEFP & = & \{T = 100\}
\end{array}$$

The formal definition of *b* can be further explained as follows:

- Each pair "(x, y)" in $b.Con$ defines a connection with direction from port x to port y. For instance, $(b.p^0_{in}, d.p^0_{in})$ defines the connection from port p^0_{in} of *b* to port p^0_{in} of *d*.
- The initial mode of *b* is m^1_b and the current mode of *b* is m^2_b.
- As the subcomponents of *b*, *d* supports one mode m^1_d and *e* supports two modes m^1_e and m^2_e.
- The initial mode of *d* is m^1_d which is also its current mode.
- The initial mode of *e* is m^1_e and the current mode of *e* is m^2_e.

All the elements included in a tuple together with their values in each mode define a specific configuration of a component. The mode switch of a component is equivalent to the change of its configuration, viz. reconfiguration. The mode-aware component model in our MSL is not based on any existing component model. Instead, it is rather generic and most existing component models can be extended to be mode-aware guided by the principles of MSL.

2.3 Summary

In this chapter, we have introduced a mode-aware component model for both primitive and composite components which can be formally defined as tuples. Each multi-mode component has a unique configuration for each of its supported modes and can reconfigure itself when a mode switch occurs. Besides, dedicated mode switch ports have been defined for the cross-level communication during a mode switch which is totally independent of system functionality. The mode-aware component model itself does not serve as a complete component model. Instead, it emphasizes the key features for a component to support mode switch in general. In principle, many existing component models can be extended to support mode switch by referring to the mode-aware component model.

Chapter 3

The mode switch runtime mechanism

As the principal ingredient of our MSL, the mode switch runtime mechanism synchronizes and coordinates the mode switches of different components to achieve a correct mode switch at the system level. A mode switch starts when a mode switch decision is made and is terminated when all involved components have completed their mode switches. In this chapter, we first begin with a description of the notations intensively used in the rest of this thesis and then look into this runtime mechanism with regard to mode switch propagation and the guarantee of mode consistency.

3.1 System model and notations

A CBMMS consists of a set of hierarchically organized components. Let PC denote the set of primitive components and CC denote the set of composite components of a CBMMS: $PC \cap CC = \varnothing$. The top component is denoted by Top. Let $\widetilde{CC}$ denote $CC \backslash \{Top\}$. Consider a composite component c_i composed by a set of components $Com = \{c_j^1, c_j^2, \cdots, c_j^n\}$ ($n \in \mathbb{N}$), $\forall c_j^k \in Com (k = [1, n])$, c_i is the parent of c_j^k, denoted by $P_{c_j^k}$. Reversely, Com is the set of subcomponents of c_i, denoted by SC_{c_i}. For each mode $m \in M_{c_i}$, the set SC_{c_i} can be divided into two disjoint parts: the set of subcomponents $ASC_{c_i}^m$ activated in m and the set of subcomponents $DSC_{c_i}^m$ deactivated in m. $SC_{c_i} = ASC_{c_i}^m \cup DSC_{c_i}^m$. Each component c_i is associated with a depth level

l_{c_i} in the component hierarchy, with $l_{Top} = 0$ and $l_{c_j} = l_{c_i} + 1$ if $c_j \in SC_{c_i}$. For two components c_i and c_j, $l_{c_j} > l_{c_i}$, if c_j is inside c_i (i.e. c_i is directly or indirectly composed by c_j), then c_i is an ancestor of c_j. If A_{c_j} denotes the set of ancestors of c_j, then $c_i \in A_{c_j}$. Reversely, c_j is a descendant of c_i. If D_{c_i} denotes the set of descendants of c_i, then $c_j \in D_{c_i}$. As a special case, $c_i = P_{c_j}$ when $l_{c_j} = l_{c_i} + 1$. The sets of activated and deactivated descendants of c_i in $m \in M_{c_i}$ are denoted by $AD^m_{c_i}$ and $DD^m_{c_i}$ respectively.

3.2 Mode switch propagation

Generally speaking, a mode switch can be either time-triggered or event-triggered. In this paper we mainly consider event-triggered mode switch, where a mode switch is triggered by a mode switch event (e.g. when the value of a sensor reaches a threshold). Time-triggered mode switch can be easily considered as a special case of event-triggered mode switch (by considering the advancement of the time at a certain point to be an event). In practice, different mode switch events could be simultaneously detected, incurring a mode switch conflict which must be resolved appropriately. In this thesis we assume that such conflict never occurs.

A mode switch event is originally detected by a Mode Switch Source (MSS):

Definition 7. ***[Mode Switch Source (MSS)]:*** *For a component $c_i \in PC \cup CC$ with the set of supported modes M_{c_i}. If $\exists m_{c_i} \in M_{c_i}$ such that c_i can actively detect a mode switch event in m_{c_i} and intend to switch to a new mode $m^{new}_{c_i} \neq m_{c_i}$, c_i is called a Mode Switch Source (MSS) in m_{c_i}.*

An MSS c_i is defined together with its mode(s). The same component can be an MSS in different modes and a system can have multiple MSSs that are predefined at design time. An MSS can trigger one or more *mode switch scenarios*:

Definition 8. ***[Mode switch scenario]:*** *A mode switch scenario is an event that an MSS c_k detects a mode switch event in $m^i_{c_k}$ and requests to switch from $m^i_{c_k}$ to $m^j_{c_k}$ ($m^i_{c_k}, m^j_{c_k} \in M_{c_k}$). We will use "$c_k : m^i_{c_k} \rightarrow m^j_{c_k}$" to denote this mode switch scenario.*

Although an MSS is the origin of a mode switch scenario, it may not have the authority to trigger a mode switch. This is the duty of another special role:

Definition 9. ***[Mode Switch Decision Maker (MSDM)]:*** *For a mode switch scenario $c_k : m^i_{c_k} \rightarrow m^j_{c_k}$, if $\exists c_p$ in $m^q_{c_p} \in M_{c_p}$ such that c_p either approves the mode switch request from c_k and triggers a mode switch or rejects the mode switch request from c_k, c_p is called the Mode Switch Decision Maker (MSDM) in $m^q_{c_p}$ for this specific mode switch scenario.*

Comparing the two special roles, i.e. MSS and MSDM, one can conclude that an MSDM must have a higher authority than the corresponding MSS. In a component-based system, a natural assertion is: the higher level (viz. lower depth level) in the component hierarchy, the higher authority. Therefore, for an MSS c_i triggering the mode switch scenario $c_i : m^p_{c_i} \rightarrow m^q_{c_i}$, if c_j is the MSDM, then $l_{c_i} > l_{c_j}$. If the top component happens to be the MSS of a mode switch scenario, then it is also the corresponding MSDM.

In Chapter 1, the example in Figure 1.1 has reflected that the mode switches of different components are not independent but correlated. This means that the mode switch of a component may be accompanied by the mode switches of some other components so as to achieve a correct mode switch at the system level. For instance, if Component *a* is defined as an MSS in Figure 1.1, the mode switch scenario triggered by *a* will force Component *b* to switch mode. Yet, Component *d* stays unaffected. Therefore, we distinguish two types of components for each mode switch scenario:

Definition 10. ***[Type A and Type B components]:*** *For a mode switch scenario $c_k : m^i_{c_k} \rightarrow m^j_{c_k}$ and any other component c_p running in $m^q_{c_p}$, if c_p must switch to a new mode $c^{new}_{c_p} \neq c^q_{c_p}$ as a consequence, c_p is a Type A component in $m^q_{c_p}$ for this scenario, denoted as $T_{c_p} = A$. Otherwise, if c_p keeps running in $m^q_{c_p}$ without being affected, c_p is a Type B component in $m^q_{c_p}$ for this scenario, denoted as $T_{c_p} = B$.*

The identification of Type A and Type B components for each mode switch scenario is realized by mode mapping which will be elaborated on in Chapter 4. In this chapter, we assume Type A and Type B components are already identified for each mode switch scenario. Then a key issue is how to propagate a mode switch event to all the Type A components without affecting Type B components. Mode switch propagation can be carried out in different ways. Nevertheless, to achieve a satisfying propagation outcome, several criteria should be met:

- Stepwise propagation. Since no component has the global knowledge of the complete component hierarchy of a CBMMS, mode switch propaga-

tion must be stepwise. In other words, a component is only allowed to propagate a mode switch event to its parent or children.

- Precise coverage. A mode switch event is propagated to all Type A components while no Type B components are disturbed.

- No redundant propagation. It suffices for the same Type A component to be notified by a mode switch event just once. Redundant propagation should be avoided.

- Bounded propagation time. Mode switch propagation time should be bounded and as short as possible.

Taking all these criteria into account, we have developed a Mode Switch Propagation (MSP) protocol, which can be considered as a distributed algorithm. Thanks to the dedicated mode switch ports p^{MSX} and p_{in}^{MSX} introduced in the mode-aware component model, there is clear separation between mode switch propagation and a system's functional behavior. In essence, mode switch propagation is performed by sending and receiving primitives between components at adjacent levels via their dedicated mode switch ports. We have defined the following primitives that will be used by our mode switch runtime mechanism:

- **Mode Switch Request** (**MSR**): An **`MSR`** is originally issued from an MSS and it has to be transmitted from a component to its parent.

- **Mode Switch Query** (**MSQ**): An **`MSQ`** is originally issued from an MSDM as an approval of the **`MSR`** from one subcomponent, and it has to be transmitted from a composite component to its subcomponents.

- **MSOK**: A positive feedback from a component to its parent with response to the **`MSQ`**, indicating the readiness for mode switch.

- **MSNOK**: A negative feedback from a component to its parent with response to the **`MSQ`**, indicating the unreadiness for mode switch.

- **Mode Switch Instruction** (**MSI**): An **`MSI`** is originally issued from an MSDM after it has received all expected primitives **`MSOK`** from its subcomponents. An **`MSI`** triggers a mode switch and has to be transmitted from a composite component to its subcomponents.

- **Mode Switch Denial** (**MSD**): An **MSD** is originally issued from an MSDM who receives at least one **MSNOK** from its subcomponents. An **MSD** aborts the mode switch to be triggered and has to be transmitted from a composite component to its subcomponents.

- **Mode Switch Completion** (**MSC**): An **MSC** is transmitted from a component to its parent after its mode switch completion. **MSC** is the only primitive not used for mode switch propagation. Instead, it is used after a mode switch is triggered as will be discussed later in this chapter.

Since the names of these primitives all begin with "MS", we shall use "MSX" to represent a generic primitive and this is where the "MSX" of the two dedicated mode switch ports p^{MSX} and p^{MSX}_{in} comes from. Next we shall describe our MSP protocol in terms of three facets: the global interpretation, primitive components and composite components.

Definition 11. ***[MSP protocol—the global interpretation]:*** *Let c_i be an MSS and $c_j \in A_{c_i}$ be the corresponding MSDM, with C_M as the set of vertically intermediate components between c_i and c_j, while $\forall c_k \in C_M$, $c_k \in A_{c_i} \cap D_{c_j}$. Mode switch propagation is taken in two phases. The first phase determines if a detected mode switch event can trigger a mode switch, while the second phase either performs the mode switch or aborts the mode switch. In the first phase, a mode switch event is initially detected by c_i which will trigger a mode switch scenario by issuing an **MSR** to P_{c_i}. Then $\forall c_k \in C_M$, the **MSR** is forwarded from c_k to P_{c_k}. When c_j receives the **MSR**, it makes a decision based on its mode mapping and current state:*

- *c_j approves the **MSR**. In order to ensure all related components are ready for mode switch, c_j will issue an **MSQ** to all $c_k \in SC_{c_j}, T_{c_k} = A$. In the same way, $\forall c_p \in D_{c_j}$ ($c_p \in CC$ and $T_{c_p} = A$), the **MSQ** is propagated from c_p to all $c_q \in SC_{c_p}, T_{c_q} = A$. Components receiving the **MSQ** are required to send either **MSOK** or **MSNOK** back. Mode switch propagation enters the second phase when c_j has received all feedbacks. If c_j receives at least one **MSNOK**, it has to abort the mode switch to be triggered and sends an **MSD** that is transmitted to all components having received the **MSQ**, following the propagation trace of **MSQ**. Then c_j resumes execution in the current mode. If c_j receives all expected **MSOK**, it will trigger a mode switch by sending an **MSI** which also follows the propagation trace of **MSQ**. Mode switch propagation is over when all Type A components receive the **MSD** or **MSI**.*

- c_j *rejects the* **MSR** *by doing nothing.*

Generally speaking, for an MSS c_i triggering the mode switch scenario $c_i : m^p_{c_i} \rightarrow m^q_{c_i}$, the corresponding MSDM c_j, as well as the set C_M of vertically intermediate components between c_i and c_j, the MSP protocol can result in one of the following scenarios:

1. c_j directly rejects the **MSR** originating from c_i by doing nothing and mode switch propagation is terminated without entering the second phase.

2. c_j approves the **MSR** originating from c_i by issuing an **MSQ** and later on receives all expected **MSOK**. In the second phase, c_j issues an **MSI** to trigger the mode switch. The propagation trace of the **MSI** follows the propagation trace of the **MSQ**.

3. c_j approves the **MSR** originating from c_i by issuing an **MSQ** and later on receives at least one **MSNOK**. Consequently, in the second phase, c_j issues an **MSD** whose propagation trace follows the propagation trace of the **MSQ**.

Figures 3.1-3.3 illustrate these three different scenarios respectively. The red node is an MSS and the blue node is the MSDM, while all the other nodes are Type A components. Please note that in Figure 3.3 the **MSQ** propagation is terminated by the composite component first replying with **MSNOK** because further **MSQ** propagation contributes nothing.

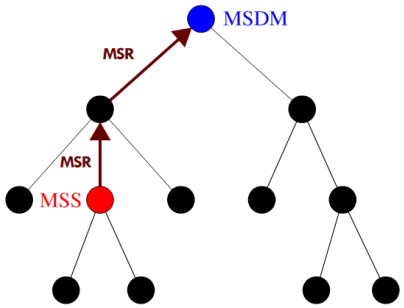

Figure 3.1: The global interpretation of the MSP protocol—Scenario 1

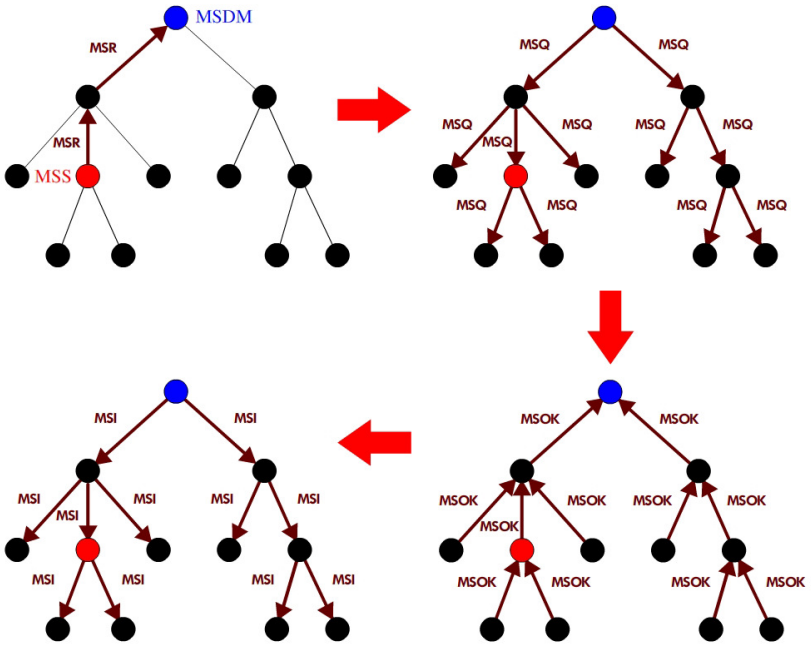

Figure 3.2: The global interpretation of the MSP protocol—Scenario 2

The global interpretation of the MSP protocol specifies how different primitives are exchanged between components during the mode switch propagation. However, it is insufficient to address the behavior of each single component. The other two facets of the MSP protocol provide the missing information:

Definition 12. ***[MSP protocol—primitive component]:*** *If a component $c_i \in PC$ is not an MSS, the first primitive it can receive is an* ***MSQ*** *asking its readiness for mode switch. First c_i will stop running and check its current state. If the current state of c_i does not allow a mode switch, it will reply with* ***MSNOK*** *and expects an* ***MSD****. Conversely, if the current state of c_i allows a mode switch, it will reply with* ***MSOK*** *and then expects either an* ***MSI*** *or* ***MSD****. Component c_i starts its mode switch upon receiving an* ***MSI****, and resumes execution in the current mode upon receiving an* ***MSD****.*

If c_i is an MSS, it can actively send an ***MSR*** *to P_{c_i} without stopping its current execution.*

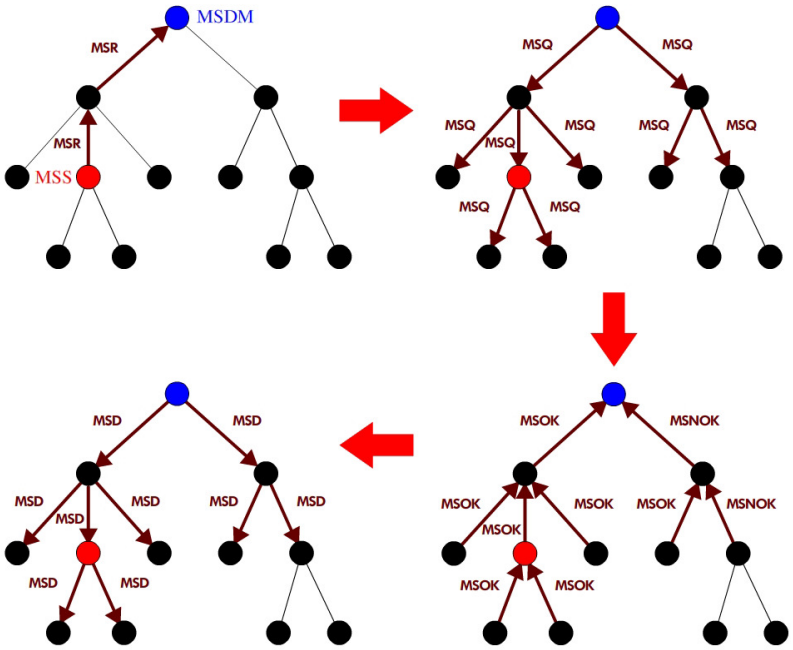

Figure 3.3: The global interpretation of the MSP protocol—Scenario 3

Definition 13. ***[MSP protocol—composite component]:*** *If a component $c_i \in CC$ is not an MSS, the first primitive it can receive is either an* ***MSQ*** *from the parent or an* ***MSR*** *from one subcomponent:*

- *If the first primitive c_i receives is* ***MSQ*** *($c_i \neq Top$), it stops running in the current mode and checks if its current state allows a mode switch. If yes, then it propagates the* ***MSQ*** *to all $c_j \in SC_{c_i}, T_{c_j} = A$ and waits for an* ***MSOK*** *or* ***MSNOK*** *from each one of them. Component c_i sends an* ***MSOK*** *back to P_{c_i} only when all c_j have replied with the primitive* ***MSOK****. Conversely, if the current state of c_i does not allow a mode switch, it will directly send an* ***MSNOK*** *back to P_{c_i} without propagating the* ***MSQ*** *further. After sending an* ***MSOK*** *to P_{c_i}, c_i expects either an* ***MSI*** *or* ***MSD*** *from P_{c_i}. In contrast, after sending an* ***MSNOK*** *to P_{c_i}, c_i only expects an* ***MSD****. Upon receiving an* ***MSI****, c_i will propagate the* ***MSI*** *to the same* ***MSQ*** *recipients and start its mode switch. Upon receiving an* ***MSD****, if c_i has*

*replied with an **MSOK**, it will propagate the **MSD** to its **MSQ** recipients and resume execution in the current mode. Otherwise, if c_i has replied with an **MSNOK**, it will immediately resume execution in the current mode.*

- *If the first primitive c_i receives is **MSR**, c_i refers to its mode mapping, checks its current state, and makes one of the following decisions:*
  - *If $T_{c_i} = B$, c_i becomes the MSDM, stops its current execution, approves the **MSR**, and follows the global interpretation as an MSDM.*
  - *If $T_{c_i} = A$, but the current state of c_i does not allow a mode switch, c_i becomes the MSDM and rejects the **MSR** without stopping its current execution.*
  - *If $T_{c_i} = A, c_i \neq Top$ and the current state of c_i allows a mode switch, c_i forwards the **MSR** to P_{c_i} without stopping its execution and let P_{c_i} make further decisions. If $c_i = Top$, c_i becomes the MSDM, stops its current execution, and approves the **MSR**, following the global interpretation as an MSDM.*

*If c_i is an MSS and $c_i \neq Top$, it can actively send an **MSR** to P_{c_i} and then behaves as a normal composite component. If $c_i = Top$, it is also the MSDM and can actively stop running and send an **MSQ**, following the global interpretation.*

What deserves extra explanation is that the propagation decision of a composite component c_i after receiving an **MSR** or **MSQ** is based on the mode mapping of c_i which identifies which components among c_i and SC_{c_i} are of Type A and which are of Type B. The thorough explanation of mode mapping can be found in Chapter 4. The propagation of an **MSI** or **MSD** does not require mode mapping because their propagation traces are exactly the same as the propagation trace of the preceding **MSQ**, which can be stored and retrieved.

All the three facets above are merged into the complete version of the MSP protocol. For the sake of better illustration, an example is used to demonstrate the MSP protocol. Figure 3.4 depicts a CBMMS with Component *h* as an MSS and Component *a* as the MSDM for the mode switch scenario triggered by *h*. Mode switch is a local event within *a*. Components filled with grey color are Type A components while others are Type B components. Figures 3.5-3.7 demonstrate scenarios 1, 2 and 3 respectively. In figures 3.6 and 3.7, it can be observed that the propagation traces of the **MSQ**, **MSI** and **MSD** are the same. The only difference between Scenario 2 and Scenario 3 is that Component *g* replies with an **MSNOK** in response to the **MSQ** in Scenario 3, leading to the **MSD**

propagation instead of **MSI**. Actually, Scenario 3 in Figure 3.7 can be further optimized by shortening the propagation time in the first phase. For instance, when *c* receives the **MSNOK** from *g*, *c* can immediately send an **MSNOK** to *a* without waiting for the response from *h*.

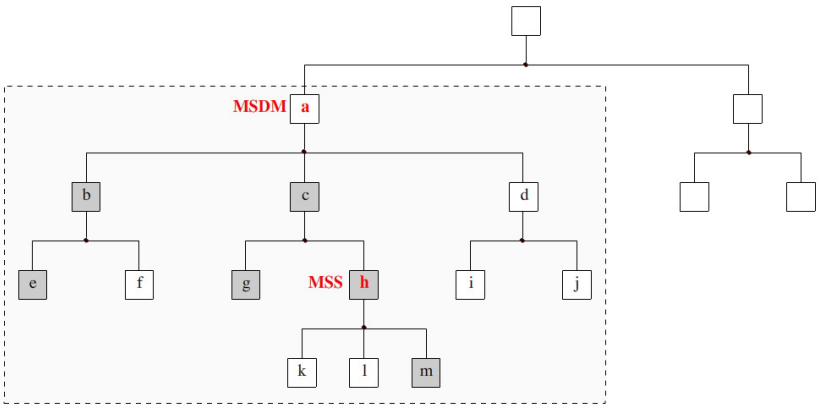

Figure 3.4: The MSS, MSDM and Type A components in a system

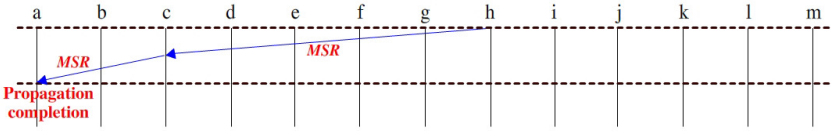

Figure 3.5: Mode switch propagation-Scenario 1

Next we shall prove the correctness and efficiency of the MSP protocol by showing that it meets all the anticipated criteria, including (1) stepwise propagation; (2) precise coverage; (3) no redundant propagation; and (4) bounded propagation time. Here we assume that there is no conflict due to multiple mode switch triggering and that the mode mapping of each composite component is correct.

Theorem 1. *The MSP protocol is distributed and stepwise, without violating the key principle of CBSE that a component only has the knowledge of itself and its immediate subcomponents.*

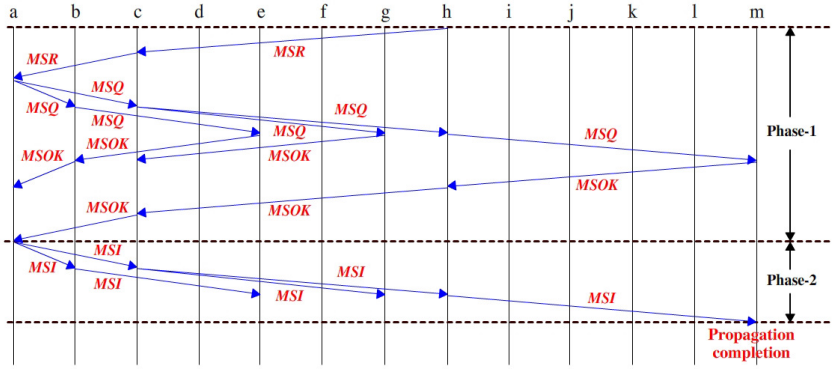

Figure 3.6: Mode switch propagation-Scenario 2

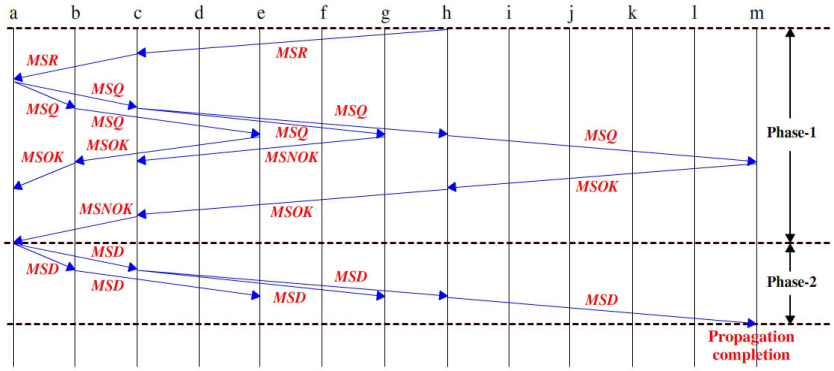

Figure 3.7: Mode switch propagation-Scenario 3

Proof. The MSP protocol is distributed among components. Its facet for primitive components (Definition 12) indicates that all primitive components follow the same propagation rules. Likewise, its facet for composite components (Definition 13) indicates that all composite components follow the same propagation rules. An MSS, MSDM or the top component have extra concerns, yet they are already included in Definition 12 and Definition 13. Since a component can only send a primitive to its parent or subcomponents, the MSP protocol is stepwise. A component sends a primitive to its parent via its p^{MSX} port, without needing to know the identity of its parent. When a composite com-

ponent sends a primitive to its subcomponents, it can be guided by its mode mapping to select the right receivers among its subcomponents. Therefore, the MSP protocol conforms to the key principle of CBSE that a component only has the knowledge of itself and its immediate subcomponents. □

Theorem 2. *The MSP protocol guarantees that the* ***MSR/MSQ*** *issued by an MSS is propagated to all Type A components, without disturbing Type B components. If no* ***MSI*** *is issued, no component will switch mode.*

Proof. Considering a CBMMS with an MSS c_i issuing an **MSR**, the corresponding MSDM c_j, as well as the set C_M of vertically intermediate components between c_i and c_j. Definition 11 actually indicates that all Type A components belong to D_{c_j} and a Type B component either belongs to D_{c_j} or is outside c_j. When c_j decides to trigger a mode switch by issuing an **MSI**, c_j must have approved the **MSR** from c_i, and c_j has received all expected **MSOK** from SC_{c_j}. Since the **MSI** exactly follows the propagation trace of the **MSQ**, the "precise coverage" property can be proved by showing that the **MSQ** from c_j must be propagated to all Type A components while no Type B components will receive the **MSQ**. Again, Definition 11 states that the **MSQ** from c_j is always propagated to lower level components. Without considering mode mapping, the **MSQ** is able to reach all components belonging to D_{c_j}. Mode mapping enables selective propagation of **MSQ** so that the **MSQ** is only propagated to Type A components and no Type A components will miss the **MSQ**. Hence Type B components belonging to D_{c_j} will not receive the **MSQ**. A Type B component can also be outside c_j. In this case, it will not receive any **MSQ** neither because c_j does not propagate the **MSQ** upwards. Therefore, this protocol guarantees that an **MSQ** or **MSI** is propagated to all Type A components without disturbing Type B components. Besides, a special case is that c_j is also the top component. Consequently, MSR transmission is skipped, but this does not affect the precise coverage of **MSQ** or **MSI**. This proves the first half of Theorem 2.

If c_j does not issue any **MSI**, either Scenario 1 or 3 will take place. Since a component will not switch mode until it issues/receives an **MSI**, no component will switch mode. This proves the latter half of Theorem 2. □

Theorem 3. *For each mode switch scenario, the MSP protocol guarantees that the same primitive from the same sender is never transmitted to the same component more than one time.*

Proof. Again, let's consider the CBMMS with an MSS c_i issuing an **MSR**, the corresponding MSDM c_j, as well as the set C_M of vertically intermediate

components between c_i and c_j. we can enumerate all the primitive types used in the MSP protocol and prove the correctness of Theorem 3 for each primitive type, including **MSR**, **MSQ**, **MSI**, **MSD**, **MSOK** and **MSNOK**:

- An **MSR** is issued from c_i, forwarded by C_M, and finally reaches c_j. According to Definition 11, an **MSR** is always transmitted upstream, with a single directional and non-cyclic propagation trace. In addition, according to Definition 11, c_i only issues an **MSR** once after detecting a mode switch event, and $\forall c_k \in C_M$, c_k forwards the **MSR** to P_{c_k} only once. Therefore, the same **MSR** is transmitted to a component at most once from the same sender.

- An **MSQ** is issued from c_j and propagated downstream to all Type A components. According to Definition 11 and the component hierarchical structure, the propagation trace of an **MSQ** is a tree, with c_j as its root and spreading but non-interweaving branches. In addition, c_j issues an **MSQ** only once before it receives all the **MSOK** or **MSNOK**, and all Type A components transmit the **MSQ** to its Type A subcomponents only once. Due to the acyclic structure of the propagation trace of **MSQ**, the same **MSQ** is transmitted from the same sender to a component at most once. Furthermore, since an **MSI** or **MSD** has the same propagation trace as the **MSQ**, this property holds also for an **MSI** or **MSD**.

- Referring to all facets of the MSP protocol (i.e. Definitions 11-13), we are sure that each component will reply with an **MSOK** or **MSNOK** only once for each **MSQ**. Since the same **MSQ** is propagated to the same component only once, there is no redundant transmission of either **MSOK** or **MSNOK**.

Since Theorem 3 holds for all types of primitives used in the MSP protocol, Theorem 3 is proved. □

Actually, Theorem 3 is related to efficiency rather than correctness. Assuming no transmission error, avoiding redundant transmission plays a significant role in lowering the communication overhead between different components during the mode switch propagation.

Theorem 4. *Assuming bounded primitive handling time, bounded primitive transmission time and no primitive transmission error, the MSP protocol guarantees bounded propagation time.*

Proof. We still consider the CBMMS with an MSS c_i issuing an **MSR**, the corresponding MSDM c_j, as well as the set C_M of vertically intermediate components between c_i and c_j. Since both the primitive handling time and the primitive transmission time are assumed to be bounded, we only need to prove that the number of propagation steps is bounded. Let N_{MSR}, N_{MSQ}, N_{MSI}, and N_{MSD} denote the number of steps of the propagation of the corresponding primitives. Also, let $N_{(N)OK}$ denote the number of steps of **MSOK/MSNOK** collection. The proof can be split based on the three mode switch propagation scenarios identified previously:

- Scenario 1: **MSR** is the only primitive being propagated. Since the propagation trace of an **MSR** is a straight single directional path from c_i to c_j with bounded length, N_{MSR}, must be bounded.

- Scenario 2: The first phase consists of three non-overlapping sub-phases: the upstream **MSR** propagation, the downstream **MSQ** propagation and the upstream **MSOK** collection. The propagation time in the first phase is the sum of all three sub-phases. It has been proved that N_{MSR} is bounded in the first sub-phase. In the second sub-phase, since the propagation trace of an **MSQ** is a tree rooted in c_j, N_{MSQ} is in proportion to the length of the longest branch, which is the maximum depth level difference between c_j and a Type A component. Since the propagation tree of an **MSQ** has bounded branch lengths, N_{MSQ} is bounded. In the third sub-phase, the **MSOK** collection can be considered as the reverse process of the **MSQ** propagation, thus $N_{(N)OK}$ is also bounded. Therefore, the number of propagation steps in the first phase is bounded. In the second phase, due to the same propagation trace of an **MSQ** and the corresponding **MSI**, N_{MSI} is also bounded. The total number of propagation steps in Scenario 2 is thus bounded.

- Scenario 3: According to Definition 11, the total number of propagation steps in Scenario 3 cannot be bigger than Scenario 2. Suppose there exists a Type A composite component c_k with at least one Type A subcomponent and the current state of c_k does not allow a mode switch when c_k receives an **MSQ**. Then the **MSQ** will not be further propagated from c_k to its Type A subcomponents. This actually reduces the number of propagation steps compared with Scenario 2. In the same manner, the number of steps of the **MSD** propagation will thus be smaller than the **MSI** propagation in Scenario 2. Since the total number of propagation

steps in Scenario 2 is bounded, the total number of propagation steps in Scenario 3 is also bounded.

Since all the three scenarios above imply bounded number of propagation steps and since each step is assumed to be bounded, the propagation time must always be bounded. □

3.3 Guaranteeing mode consistency

As is stated before, a mode switch scenario is triggered by an MSS c_i, which issues an **MSR** (if $c_i \in \widetilde{CC}$) or an **MSQ** (if $c_i = Top$). A mode switch will be triggered when the MSDM decides to issue an **MSI**. This necessitates the satisfaction of two conditions: (1) the MSDM approves the **MSR**; (2) the MSDM receives all expected **MSOK** after issuing the **MSQ**. If $c_i = Top$, only the second condition must be met. When the MSDM triggers a mode switch by issuing an **MSI**, the MSP protocol will propagate this **MSI** to all Type A components. Each Type A component propagates an incoming **MSI** to its Type A subcomponents and then start its mode switch. From the perspective of a single Type A component c_j, its mode switch is essentially equivalent to its reconfiguration, i.e. changing its current configuration to the configuration in the new mode (see Chapter 2). And the mode switch of c_j should be completed when c_j completes its reconfiguration. Nevertheless, from the perspective of the entire system, its mode switch is completed when all the Type A components have completed their mode switches. A mode inconsistency anomaly will occur if one or more Type A components are still running in the old modes or are in the process of switching mode even after the mode switch completion of the system. To guarantee mode consistency, the reconfigurations of different Type A components must be properly synchronized. To this end, we define the following mode switch dependency rule:

Definition 14. ***[Mode switch dependency rule]:*** *Any component receiving an* ***MSI*** *from the parent must send a primitive Mode Switch Completion (****MSC****) back to its parent as a feedback upon its mode switch completion. For a component c_i which starts its mode switch, the following applies*

- *If $c_i \in PC$, its mode switch starts after receiving an* ***MSI*** *from P_{c_i}. Component c_i will send an* ***MSC*** *to P_{c_i} after its reconfiguration to indicate mode switch completion.*

- *If $c_i \in \widetilde{CC}$ and c_i is not the MSDM, its mode switch starts after receiving an* **MSI** *from P_{c_i} and propagating the* **MSI** *to its Type A subcomponents. The mode switch of c_i is completed after the reconfiguration of c_i and its* **MSC** *collection from its Type A subcomponents. Then c_i sends an* **MSC** *to P_{c_i} after its mode switch completion. If c_i is the MSDM, it does not switch mode but expects an* **MSC** *from all its Type A subcomponents.*

- *If $c_i = Top$ and c_i is the MSDM, then $T_{c_i} = A$ or $T_{c_i} = B$. If $T_{c_i} = A$, its mode switch starts after issuing an* **MSI** *to its Type A subcomponents. The mode switch of c_i is completed after the reconfiguration of c_i and its* **MSC** *collection from its Type A subcomponents. If $T_{c_i} = B$, it does not switch mode but expects an* **MSC** *from its Type A subcomponents.*

The system mode switch is completed when the MSDM c_k completes its mode switch ($T_{c_k} = A$) or when c_k completes its **MSC** *collection ($T_{c_k} = B$).*

The mode switch dependency rule enforces a dependency of the mode switch of a composite component on the mode switches of its Type A subcomponents. Therefore, a system mode switch is taken in a bottom-up manner, ensuring the well synchronization between the mode switches of different components and mode consistency. The mode switch dependency rule exhibits the following property:

Lemma 1. *Assuming bounded reconfiguration time of each component, the mode switch dependency rule guarantees that any component will complete its mode switch in bounded time.*

Proof. A non-top component starts its mode switch by receiving an **MSI** and the top component starts its mode switch by issuing an **MSI**. In Theorem 2, it has been proved that an **MSI** is propagated to all Type A components without disturbing Type B components. When a component c_i starts its mode switch, it has at most two things to do before its mode switch completion: reconfiguration and **MSC** collection.

If $c_i \in PC$, its mode switch equals its reconfiguration. Assuming the bounded reconfiguration time of c_i, c_i will complete its mode switch in bounded time.

If $c_i \in CC$, its mode switch completion relies on the satisfaction of two conditions: (1) c_i completes its reconfiguration; (2) c_i has received all expected primitives **MSC** from $c_j \in SC_{c_i}, T_{c_j} = A$. Since the reconfiguration time of c_i is always bounded, Condition (1) is satisfied. If Condition (2) is proved, c_i will be guaranteed to complete its mode switch as a composite component.

Condition (2) can be proved by contradiction. Suppose c_i expects an **MSC** from $c_p \in SC_{c_i}, T_{c_p} = A$ which never sends the **MSC**, thus c_i can never complete its mode switch. The MSP protocol guarantees that c_p must have received an **MSI** from c_i, thus the only reason why c_p never sends the **MSC** to c_i is that c_p expects at least one **MSC** from $c_q \in SC_{c_p}$. As the depth level grows ($l_{c_p} = l_{c_i} + 1$ and $l_{c_q} = l_{c_p} + 1$), following the propagation trace of the **MSI**, we will infer that $\exists c_k \in D_{c_i}$, $c_k \in PC$, which never sends an **MSC** to P_{c_k} upon mode switch completion. This is in contradiction with the mode switch dependency rule, according to which c_k must send an **MSC** back to P_{c_k} after its reconfiguration in response to the **MSI**. Therefore, Condition (2) is also satisfied and $c_i \in CC$ will complete its mode switch in bounded time.

Since both primitive and composite components will complete the mode switch in bounded time, Lemma 1 is proved. □

Lemma 1 can further imply:

Lemma 2. *Assuming bounded reconfiguration time of each component, the mode switch time of a CBMMS is always bounded.*

Proof. According to Definition 14, the mode switch of a CBMMS is completed either when the MSDM c_i completes its mode switch if $T_{c_i} = A$, or when the MSDM c_i has received an **MSC** from all $c_j \in SC_{c_i}, T_{c_j} = A$ if $T_{c_i} = B$. The MSP protocol (Definition 11) indicates that the MSDM c_i must exist for any mode switch scenario $c_k : m^p_{c_k} \rightarrow m^q_{c_k}$.

If $T_{c_i} = A$, then $c_i = Top$. Otherwise if $c_i \neq Top$, c_i will forward the **MSR** issued by c_k to P_{c_i} and c_i will not be the MSDM. It has been proven in Lemma 1 that any component can complete its mode switch in bounded time, including c_i. Therefore, the system mode switch is also bounded.

If $T_{c_i} = B$, then c_i may or may not be the top component. In either case, c_i does not switch mode and only expects to receive an **MSC** from all $c_j \in SC_{c_i}, T_{c_j} = A$. Then we only need to prove the **MSC** collection time of c_i is bounded. From Theorem 2, it is assured that all c_j have received an **MSI** from c_i. Also, Lemma 1 assures that the mode switch time of all c_j is bounded. Then c_j must send an **MSC** to c_i in bounded time according to the mode switch dependency rule. Hence the **MSC** collection time of c_i must be bounded and the system mode switch is also bounded.

The analysis of the two cases above suffices to prove Lemma 2. □

Apart from bounded mode switch time, the mode switch dependency rule satisfies the most important property—mode consistency:

Theorem 5. *The mode switch dependency rule guarantees that all Type A components are running in their new modes after the mode switch completion of the system.*

Proof. Let c_i be the MSDM for a specific mode switch scenario. Theorem 5 can be easily proved by contradiction. First, according to Lemma 2, there must exist a time t_1 when a system completes its mode switch. Suppose there exists a Type A component c_j not running in its new mode even after the mode switch of the system at time t_1. It is certain that $c_j \neq c_i$. Otherwise, if $c_j = c_i$, as the MSDM and a Type A component, c_i must be the top component and must have already completed its mode switch according to Lemma 1. Also, $c_j \in D_{c_i}$. Otherwise, if $c_j \notin D_{c_i}$, c_j cannot be a Type A component.

It can be implied that c_j is either still running in its old mode or is in the process of switching to its new mode at t_1. If c_j is still running in its old mode at t_1, the only possible reason is that c_j has not received an **`MSI`**. According to Theorem 2, this is impossible because an MSI must be propagated to all Type A components.

If c_j is still switching mode at t_1, then according to Lemma 1, there must exist a time $t_2 > t_1$ when c_j completes its mode switch. And then according to Definition 14, c_j must send an MSC to P_{c_j} at t_2. Since $c_j \in D_{c_i}$, it can be implied that c_i has not received an **`MSC`** from $c_k \in SC_{c_i}$ at t_2, $c_k \in A_{c_j}$. Therefore, it is concluded that the MSDM c_i has not collected all the expected primitives **`MSC`** at t_2. That is to say, the mode switch of the system is not completed at t_2. Since $t_2 > t_1$, the system mode switch cannot be completed at t_1. This is against the assumption that the mode switch of the system is completed at t_1. Hence, the mode switch dependency rule guarantees that there is no such Type A component which is not running in its new mode after the mode switch completion of the system and Theorem 5 is proved. □

Figure 3.8 demonstrates the mode switch dependency rule based on the example in Figure 3.4 and its mode switch propagation—Scenario 2 in Figure 3.6. For the sake of avoiding redundant illustration, the first phase of the mode switch propagation in Figure 3.6 is omitted in Figure 3.8. Each component starts its reconfiguration after receiving an **`MSI`** and propagating the **`MSI`** to its Type A subcomponents. Component reconfiguration is represented by black bars. The length of each black bar specifies the reconfiguration time of a component. Each primitive component (e.g. *e*, *g* and *m*) will send an **`MSC`** to the parent after reconfiguration. Each composite component (e.g. *b*, *c* and *h*) will not send an **`MSC`** to the parent until it completes its reconfiguration and has received an **`MSC`** from all its Type A subcomponents. White bars imply

that a composite component completes its reconfiguration earlier than its **MSC** collection such that its mode switch is temporarily blocked by the **MSC** from its subcomponents. When the MSDM *a* receives the **MSC** from both *b* and *c*, the system mode switch is completed.

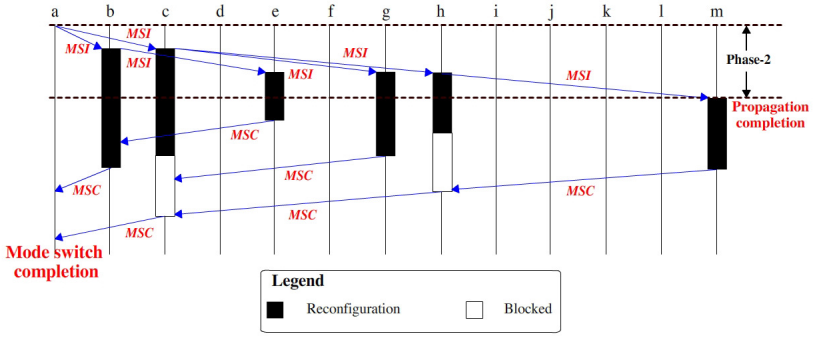

Figure 3.8: Demonstration of the mode switch dependency rule

3.4 Summary

In this chapter, a mode switch runtime mechanism is presented to handle the composable mode switch of a CBMMS. The mode switch runtime mechanism includes the MSP protocol and the mode switch dependency rule. The MSP protocol is able to propagate the mode switch request initiated from an MSS to all the components which must switch mode as a consequence. The mode switch dependency rule is applied after a mode switch is triggered, guaranteeing the mode consistency between different components after each mode switch. The correctness of the MSP protocol and the mode switch dependency rule has been formulated into a number of properties which have all been proven to be satisfied.

Chapter 4

Mode mapping

In Chapter 3, we have identified Type A and Type B components for each mode switch scenario. Type A components must switch mode as a consequence while Type B components are not affected. Then another problem emerges: How do we know which components are of Type A and which are of Type B? In addition, even if Type A and Type B components are already identified, for each Type A component, what is the new mode that it should switch to? In this chapter, we propose a mode mapping mechanism that handles these problems. Mode mapping plays a vital role in the composition of multi-mode components and mode switch propagation at runtime. The mechanism presented here was originally introduced in [20]; here we present an improved version of that mechanism.

4.1 Mode mapping for component composition

Since a multi-mode component is assumed to be independently developed without knowing the context in which it will be used/reused, the composition of multi-mode components may be subject to incompatibility between the modes of composed components, as the modes of these components are not well-matched. This mode incompatibility problem is illustrated by Table 4.1, listing the components with the same hierarchy as in Figure 1.1 but with different component connections and different supported modes. Figures 4.1 and 4.2 illustrate how these components are connected in each mode after composition. It should be noted that Table 4.1 only lists the supported modes of each

component for this particular composition. When a multi-mode component is reused, either all or a subset of its supported modes can be selected for the particular context. For instance, Component *c* in Table 4.1 may originally support several modes; still only m_c^1 is used for composing *b* in this context.

Component	Supported modes
Top	m_{Top}^1, m_{Top}^2
a	m_a^1, m_a^2
b	m_b^1, m_b^2, m_b^3
c	m_c^1
d	m_d^1, m_d^2, m_d^3, m_d^4
e	m_e^1

Table 4.1: The mode incompatibility problem

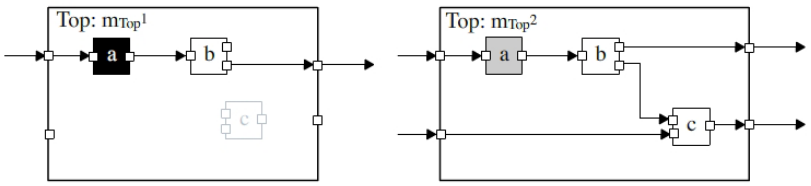

Figure 4.1: The inner component connection of *Top*

In order to tackle the mode incompatibility problem, the supported modes of a composite component and its subcomponents must be properly mapped. We have proposed a mode mapping mechanism for this purpose, based on the following:

- A primitive component knows its supported modes, its initial mode and its current mode, but knows nothing about the mode information of other components in the system.

- A composite component knows the mode information (supported modes, initial mode and current mode) of itself and its immediate subcomponents, but knows nothing about the mode information of other components in the system.

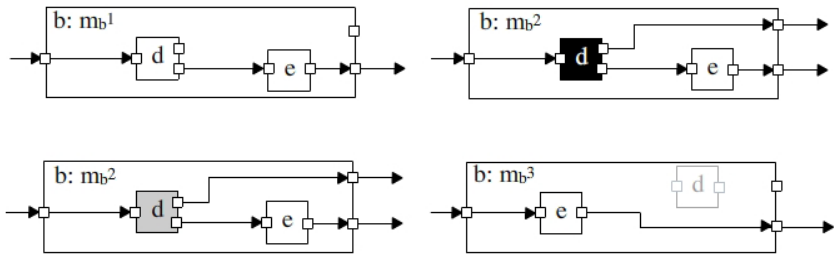

Figure 4.2: The inner component connection of *b*

Mode mapping is always locally managed by a composite component according to a set of local mode mapping rules. These mode mapping rules not only consider the mode mapping in stable modes, but also the mode mapping for each mode switch scenario at runtime. There is no need for a primitive component to consider mode mapping in that it has no subcomponents. The mode mapping between a composite component and its subcomponents in stable modes can be presented by a *mode mapping table*. Tables 4.2 and 4.3 present the mode mapping tables of *Top* and *b*, respectively, with regard to figures 4.1 and 4.2. Modes in the same column are mapped to each other. For example, when *Top* is in m^1_{Top}, among its subcomponents, *a* is in m^1_a, *b* can be in either m^1_b or m^3_b, and *c* is deactivated, i.e. not running in any mode.

Component	Supported modes		
Top	m^1_{Top}		m^2_{Top}
a	m^1_a		m^2_a
b	m^1_b	m^3_b	m^2_b
c	Deactivated		m^1_c

Table 4.2: The mode mapping table of *Top*

The mode mapping table is straightforward and intuitive, although lacking expressiveness. Mode mapping is not only essential for the composition of multi-mode components, but also crucial for distinguishing Type A and Type B components for each mode switch scenario and deriving the new modes of

Component	Supported modes			
b	m_b^1	m_b^2		m_b^3
d	m_d^1	m_d^2	m_d^3	Deactivated
e	m_e^1			

Table 4.3: The mode mapping table of b

Type A components. In the latter case, mode mapping tables will be insufficient to express the required set of mode mapping rules since it is hard to include different mode switch scenarios in a mode mapping table. In the next section, we shall propose a new and more expressive presentation of mode mapping.

4.2 Mode mapping at runtime

We have pointed out that mode mapping is not only crucial for the composition of multi-mode components, but also essential for mode switch at runtime. Section 3.2 has implied the intimate correlation between mode switch propagation and mode mapping. Figure 4.3 further shows the relationship between the MSP protocol and the mode mapping of a composite component. The MSP protocol together with mode mapping can be regarded as a black box interacting with the dedicated mode switch ports p^{MSX} and p_{in}^{MSX}. The mode mapping acts as a function which can be called by the MSP protocol to return the right mode mapping results. The type of primitive, the propagation direction and the current state of the composite component determine whether the mode mapping should be called or not.

The mode mapping of each composite component plays a predominant role in determining the propagation trace of primitives. In Chapter 3, it has been mentioned that only an **MSR** or **MSQ** requires mode mapping, as the propagation trace of an **MSI** or **MSD** exactly follows the propagation trace of the **MSQ**. Each **MSR** contains the identity of the sender, the current mode of the sender and its desired new mode. According to the MSP protocol, the response of an **MSR** receiver is based on its mode mapping and its current state. Different from **MSR**, an **MSQ** contains the identity of the receiver and the derived new mode for the receiver based on the mode mapping result of the **MSQ** sender. If a receiver is supposed to be deactivated, this derived new mode will be replaced by a deactivation mode or state. The mode mapping of a component is represented by a set of mode mapping rules which can be divided into two parts: static

mode mapping rules and dynamic mode mapping rules. Static mode mapping rules define the mode mapping in stable modes and can essentially be presented by mode mapping tables introduced in Section 4.1. Dynamic mode mapping rules are complementary to static mode mapping rules and define the mode mapping at runtime.

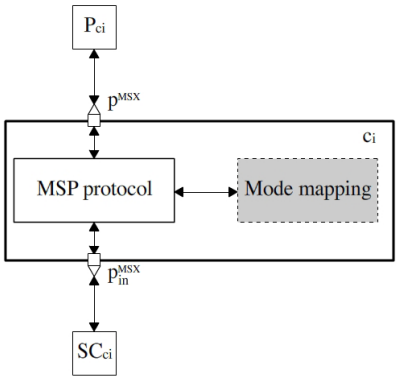

Figure 4.3: Mode switch propagation and mode mapping

The key purpose of dynamic mode mapping rules is to identify Type A components for each mode switch scenario and derive the new mode of each Type A component. This is beyond the description of the mode mapping table. For instance, considering Table 4.3, when Component *b* is asked to switch from m_b^1 to m_b^2 as it receives an **`MSI`** from *Top*, *b* is supposed to propagate this **`MSI`** to its Type A subcomponents. According to the mode mapping table of *b*, *d* is a Type A component and *e* is a Type B component. Moreover, *d* can switch from m_d^1 to either m_d^2 or m_d^3 to satisfy the static mode mapping between *b* and *d*. The problem is that the new mode of *d* is not uniquely defined by the table as there are two candidates. One effective solution is to introduce the Dominant Default Modes (DDMs):

Definition 15. ***[Dominant Default Mode (DDM)]:*** *Let c_i and c_j be two components so that either $c_i = P_{c_j}, c_j = P_{c_i}$ or $P_{c_i} = P_{c_j}$ (i.e. c_i and c_j are siblings). If the mode switch of c_i from $m_{c_i}^p$ to $m_{c_i}^q$ ($m_{c_i}^p, m_{c_i}^q \in M_{c_i}$) implies one and only one new mode $m_{c_j}^{new}$ of c_j ($m_{c_j}^{new} \in M_{c_j}$), then $m_{c_j}^{new}$ is called the Dominant Default Mode (DDM) of c_j for this mode switch scenario.*

It is obvious that the DDM of a component is context-dependent. Different mode switch scenarios may result in different DDMs for a component. The

static mode mapping rules of a component may define a couple of DDM candidates for each component and it is the duty of the dynamic mode mapping rules to specify all the DDMs of a composite component itself and its subcomponents. Regarding the example in Table 4.3, either m_d^2 or m_d^3 can be defined as a DDM of *d* when *b* switches from m_b^1 to m_b^2 for the sake of mode switch predictability.

We have proposed the use of Mode Mapping Automata (MMAs)[1] to express both static and dynamic mode mapping rules [20]. The mode mapping of a component $c_i \in CC$ can be formally presented by a set of MMAs, which consist of one Mode Mapping Automaton (MMA) of c_i as the parent (denoted as $\text{MMA}_{c_i}^p$) and one MMA of each subcomponent of c_i (denoted as $\text{MMA}_{c_j}^c$, $c_j \in SC_{c_i}$). Here we call $\text{MMA}_{c_i}^p$ a parent MMA and $\text{MMA}_{c_j}^c$ a child MMA. Each MMA 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 MSP protocol. A parent or child MMA can be formally defined as follows:

Definition 16. ***[Mode Mapping Automaton (MMA)]:*** *An MMA is defined as a tuple:*

$$< \text{S}, s^0, \text{SI}, expr(\text{BV}), \text{T} >$$

where S *is a set of states;* $s^0 \in$S *is the initial state;* SI=I$\cup$E (I$\cap$E$= \varnothing$) *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 boolean variables and expr(*BV*) is a set of boolean expressions over* BV*;* $\text{T} = \text{S} \times \text{SI} \times expr(\text{BV}) \times 2^{\text{SI}} \times \text{S}$ *is a set of transitions of the MMA, where expr(*BV*) must evaluate to true to enable a transition.*

Based on Definition 16, we shall use $s \xrightarrow{si \;\&\&\; cond/O} s'$ to denote a transition $(s, si, cond, O, s') \in T$. Please note that for a child MMA, *BV*$= \varnothing$. Moreover, although the output of a transition is a set O of signals for both parent and child MMAs, O always has a single element for a child MMA.

For better apprehension of the parent and child MMAs and their formal semantics, we can use MMAs to describe the mode mapping of Component *b* based on Table 4.3. Figure 4.4 illustrates the relation between the MSP protocol and the mode mapping of Component *b*. Compared with Figure 4.3, Figure 4.4 additionally reveals the MMA structure of the mode mapping of

[1] In this thesis, a major revision has been taken for the MMAs, compared with the initial version introduced in [20].

b. There is one parent MMA (MMA_b^p) and two child MMAs (MMA_d^c and MMA_e^c). These MMAs are hierarchically organized in the same way as the corresponding components. The only difference is that all these MMAs reside in the parent Component b. Both MMA_d^c and MMA_e^c are internally synchronized with MMA_b^p. Moreover, all three MMAs can directly communicate with the MSP protocol.

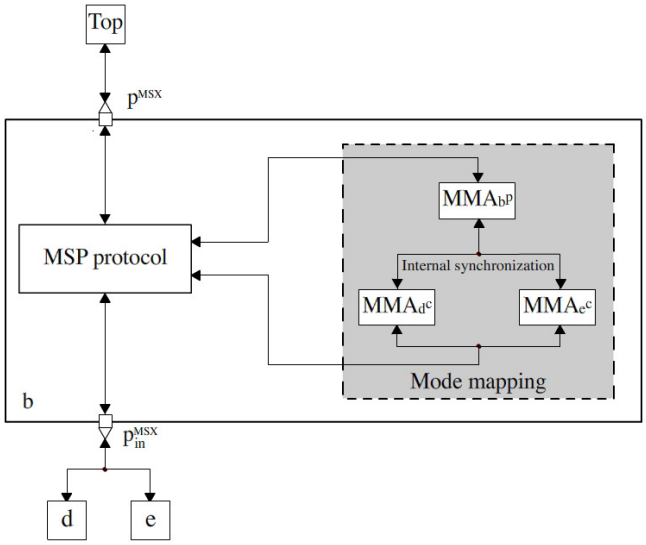

Figure 4.4: Mode switch propagation and the mode mapping of Component b

Figures 4.5 and 4.6 provide the graphic presentations for MMA_b^p and MMA_d^c respectively. In general, the set of states of MMA_{c_i}, corresponding to the supported modes of c_i, can be graphically presented by locations with circles, the location with double circles being the initial state which corresponds to the initial mode of c_i. Each transition $s_1 \rightarrow s_2$ is presented by an arrow starting from state s_1 and ending in state s_2. A transition can be associated with input and output signals which can be either external or internal. These signals are denoted by the following notations:

- $x.I(y)$, an internal signal emitted by a parent MMA to the recipient MMA_x^c, which is asked to change location to y.
- $x.I(y \rightarrow z)$, an internal signal emitted by MMA_x^c which actively

changes location from y to z.

- $x.E(y)$, an external signal asking MMA$_x$, which is either a parent MMA or a child MMA, to change location to y.

In addition, for a parent MMA, if multiple transitions share the same output, starting and ending locations, they can be combined into one transition where different input signals are connected by the notation "||". For instance, MMA$_b^p$ in Figure 4.5 has two transitions $m_b^1 \xrightarrow{d.I(m_d^1 \to m_d^2)/\{b.E(m_b^2)\}} m_b^2$ and $m_b^1 \xrightarrow{d.I(m_d^1 \to m_d^3)/\{b.E(m_b^2)\}} m_b^2$, which are combined into one transition $m_b^1 \xrightarrow{d.I(m_d^1 \to m_d^2) \;||\; d.I(m_d^1 \to m_d^3)/\{b.E(m_b^2)\}} m_b^2$. Actually the output of each transition in MMA$_b^p$ defines the DDMs of b and SC_b (i.e. d and e) for a specific mode switch scenario. For instance, when b receives an **MSQ** requesting to switch from m_b^1 to m_b^2, an external signal $b.E(m_b^2)$ will be injected into MMA$_b^p$ as an input. Then the DDM of d, which can be either m_d^2 or m_d^3, is defined based on the boolean expressions, $cond\ 1$ and $cond\ 2$ marked in red in Figure 4.5. According to the transition $m_b^1 \xrightarrow{b.E(m_b^2)\ \&\&\ cond\ 1/\{d.I(m_d^2)\}} m_b^2$, the DDM of d is defined as m_d^2 when $cond\ 1$ evaluates to true. Subsequently, the internal signal $d.I(m_d^2)$ will be synchronized with the transition $m_d^1 \xrightarrow{d.I(m_d^2)/\{d.E(m_d^2)\}} m_d^2$ of MMA$_d^c$ depicted in Figure 4.6. Similarly, the DDM of d is defined as m_d^3 when $cond\ 2$ evaluates to true. The boolean expressions $cond\ 1$ and $cond\ 2$ are abstracted from the system's functional code unrelated to mode switch. Serving as an interface linking mode mapping and the functional behavior of a system, a boolean expression can be as simple as a single boolean variable or a much more complex expression.

Please note that if a transition of a parent MMA produces no output, it is denoted as $\varnothing$. Besides, Figure 4.6 implies that the deactivated running status of a component can be treated as a separate location D in the corresponding child MMA. Here we skip the illustration of MMA$_e^c$ which is simply a single location m_e^1 without any transition.

Similarly, figures 4.7-4.10 display MMA$_{Top}^p$, MMA$_a^c$, MMA$_b^c$ and MMA$_c^c$ which jointly present the mode mapping of *Top* in Table 4.2. MMA$_{Top}^p$ is the parent MMA while the others are the child MMAs. What deserves particular attention is the discrepancy between MMA$_b^p$ in Figure 4.5 and MMA$_b^c$ in Figure 4.9, as the former is a parent MMA included in the mode mapping of *b* whereas the latter is a child MMA included in the mode mapping of *Top*.

The MMAs of *Top* and *b* take all possible mode switch scenarios into ac-

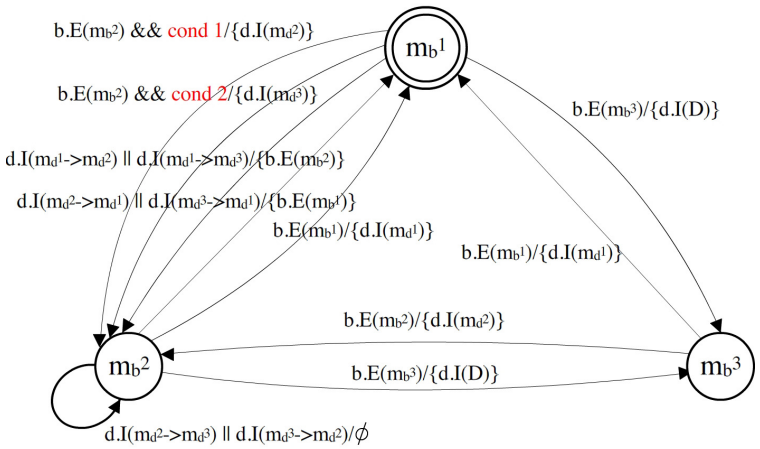

Figure 4.5: The parent Mode Mapping Automaton of *b*

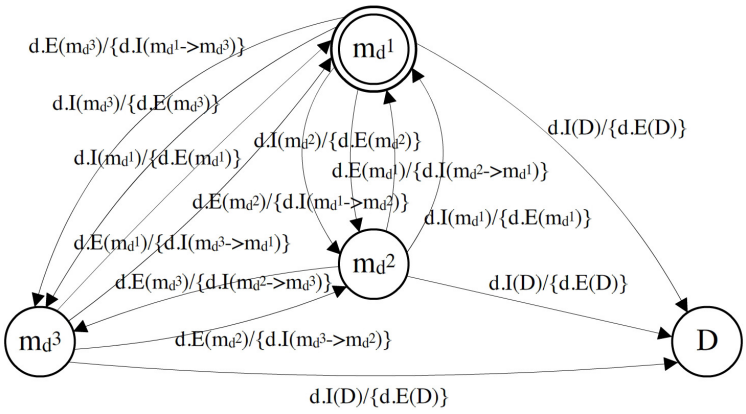

Figure 4.6: The child Mode Mapping Automaton of *d*

count for the sake of better component reuse. In a specific system, only certain components are defined as the MSSs and there are much fewer mode switch scenarios, thus a parent MMA usually contains much fewer mode mapping rules in a particular system.

It should be noted that it is not the obligation of a set of MMAs to distin-

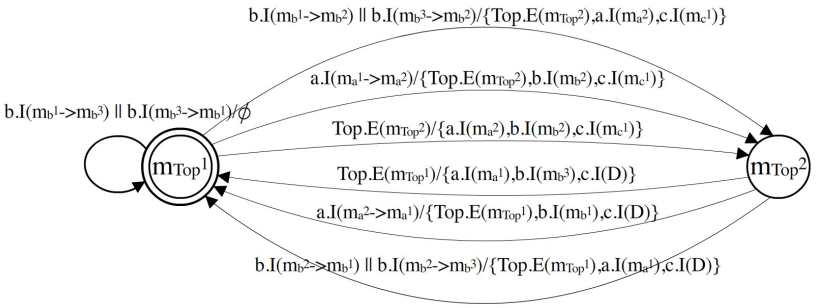

Figure 4.7: The parent Mode Mapping Automaton of *Top*

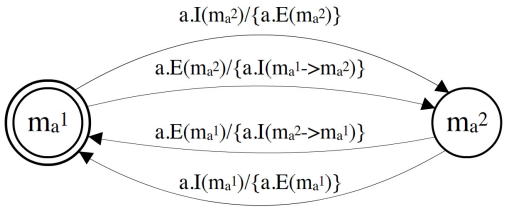

Figure 4.8: The child Mode Mapping Automaton of *a*

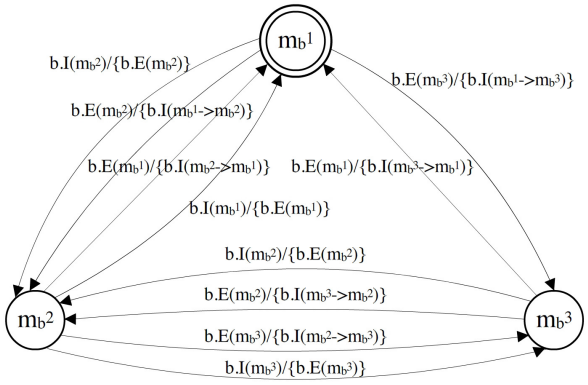

Figure 4.9: The child Mode Mapping Automaton of *b*

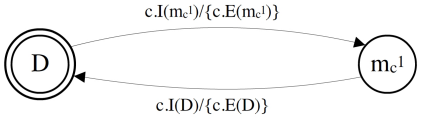

Figure 4.10: The child Mode Mapping Automaton of c

guish different types of primitives. An MMA only needs to know if an input signal is internal or external. An internal signal comes from another MMA and an external signal comes from the MSP protocol. When a composite component c_i receives an **MSR** from a subcomponent c_j, the MSP protocol will translate the **MSR** into an external signal which is then sent to $\text{MMA}^{c}_{c_j}$. The mode mapping result from the MMAs only tells which components are Type A or Type B among c_i or $SC_{c_i} \setminus \{c_j\}$ as well as the new modes of these identified Type A components for this mode switch scenario. If $T_{c_i} = B$, then according to the MSP protocol, c_i is the MSDM and will approve the **MSR**. Otherwise, c_i will either reject the **MSR** or forward the **MSR** to P_{c_i} (if $c_i \neq \mathit{Top}$), depending on its current state. This decision is taken by the MSP protocol rather than the set of MMAs of c_i which never considers whether an **MSR** is rejected or not.

Moreover, the state transition of an MMA is not necessarily an evidence of mode switch. For instance, when an **MSR** is rejected, state transitions must have been completed in the set of MMAs of c_i, yet no component will switch mode. This inconsistency can be easily tackled by updating the MMAs after c_i makes a propagation decision. When c_i makes a decision after having received an **MSR** or **MSQ**, the MMAs of c_i must be reset to the previous configuration. This requires the support of at least a one-step rollback.

Another important issue is that c_i must be able to tell if an incoming **MSQ** is associated with an **MSR** that c_i has just forwarded to P_{c_i}. If yes, then the MSP protocol should not translate this **MSQ** into an external signal and send it to the MMAs. Instead, it can simply perform a rollback of the MMAs and refer to the mode mapping results based on the incoming **MSR** whose approval by the MSDM results in this **MSQ**. This is because the **MSR** is considered as an external signal injected into a child MMA whilst the **MSQ** is considered as an external signal injected into the parent MMA. Consequently, the **MSR** and the **MSQ** issued due to the approval of this **MSR** can trigger two different mode switch scenarios, thus potentially producing different mode mapping results. In this case, the mode mapping results in response to the **MSR** should be the correct one and can be retrieved. Otherwise, if c_i receives an **MSQ** without re-

ceiving the corresponding **MSR** first, it will directly produce the mode mapping result in response to the **MSQ**.

The interaction between the MSP protocol and the MMAs can be further explicated by following a specific mode switch scenario. Suppose the current mode of *Top* is m^2_{Top}. Then given tables 4.2 and 4.3, the current modes of *a*, *b*, *c*, *d* and *e* are m^2_a, m^2_b, m^1_c, m^2_d or m^3_d, and m^1_e, respectively. The mode switch propagation can be described as follows:

1. Component a is assumed to be an MSS initiating an **MSR** from m^2_a to m^1_a. As a consequence, MMA^c_a, as part of the mode mapping of *Top*, will receive an external signal $a.E(m^1_a)$. This triggers the transition $m^2_a \xrightarrow{a.E(m^1_a)/\{a.I(m^2_a \to m^1_a)\}} m^1_a$ of MMA^c_a.

2. To be synchronized with the transition of MMA^c_a, MMA^p_{Top} will undergo the transition $m^2_{Top} \xrightarrow{a.I(m^2_a \to m^1_a)/\{Top.E(m^1_{Top}), b.I(m^1_b), c.I(D)\}} m^1_{Top}$. This means that *Top*, *b* and *c* are all Type A components for this mode switch scenario, and m^1_{Top} and m^1_b are the DDMs of *Top* and *b*, with *c* being deactivated.

3. Among the output set of the transition of MMA^p_{Top}, the internal signals $b.I(m^1_b)$ and $c.I(D)$ are synchronized with the transition $m^2_b \xrightarrow{b.I(m^1_b)/\{b.E(m^1_b)\}} m^1_b$ of MMA^c_b and the transition $m^1_c \xrightarrow{c.I(D)/\{c.E(D)\}} D$ of MMA^c_c. All the external output signals, including $Top.E(m^1_{Top})$, $b.E(m^1_b)$ and $c.E(D)$ will be send to the MSP protocol.

4. The external signal $Top.E(m^1_{Top})$ implies that *Top* is a Type A component and it has to check its current state as the MSDM. Suppose *Top* approves the **MSR** by issuing an **MSQ** based on the mode mapping result. First all the MMAs of *Top* should be reset and save the new configuration as the history configuration. Then *Top* decides to issue an **MSQ**. Since the **MSQ** is associated with the **MSR**, *Top* will update the MMAs by retrieving the history configuration without referring to the mode mapping.

5. The **MSQ** is propagated to a, b and c. According to the mode mapping result of the **MSR**, *Top* will switch to m^1_{Top}, with *b* switching to m^1_b and *c* becoming deactivated. Additionally, *Top* should also send the **MSQ** to a which is the **MSR** sender, telling a to switch to m^1_a, though this decision is not explicitly stated in MMA^p_{Top}.

6. Let's assume that both a and c reply with an **MSOK**. When b receives the **MSQ**, it realizes that it has not forwarded any **MSR** associated with this **MSQ**, thus it will refer to the mode mapping by sending an external signal $b.E(m_b^1)$ to MMA_b^p (see Figure 4.5), triggering the transition $m_b^2 \xrightarrow{b.E(m_b^1)/\{d.I(m_d^1)\}} m_b^1$.

7. Suppose the current mode of d is m_d^2. The internal signal $d.I(m_d^1)$ produced by the transition of MMA_b^p leads to the transition $m_d^2 \xrightarrow{d.I(m_d^1)/\{d.E(m_d^1)\}} m_d^1$, indicating that m_d^1 is the DDM of *d*.

8. Component *b* propagates an **MSQ** to d without affecting *e* and resets its MMAs.

9. Assuming *d* replies with an **MSOK**, *b* will also send an **MSOK** back to *Top*. Since *a*, *b* and *c* all reply with an **MSOK**, *Top* will trigger a mode switch by issuing an **MSI** which follows the propagation trace of the **MSQ**. The MMAs of *Top* and *b* which have been reset will be updated to the new configuration again. Mode switch propagation is completed when the **MSI** arrives at all Type A components, i.e. a, b, c and d (the MSS a and the MSDM *Top* are also Type A components for this mode switch scenario).

4.3 MMA composition

In this section, we present 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 17. ***[MMA composition]:*** *For a set $\mathcal{A} = \{\mathcal{A}_0, \mathcal{A}_1, \cdots, \mathcal{A}_n\} (n \in \mathbb{N})$ of MMAs, where $\mathcal{A}_0 =< \text{S}_0, s_0^0, \text{SI}_0, expr(\text{BV}_0), \text{T}_0 >$ corresponds to a parent MMA and $\forall k = [1, n]$, $\mathcal{A}_k =< \text{S}_k, s_k^0, \text{SI}_k, expr(\text{BV}_k), \text{T}_k >$ correspond to the child MMAs synchronized with $\mathcal{A}_0$, the MMA composition of $\mathcal{A}$ is an MMA defined by the tuple*

$$< \text{S}, s^0, \text{SI}, expr(\text{BV}), \text{T} >$$

where

$$\begin{array}{lcl} \mathrm{S} & \subseteq & \mathrm{S}_0 \times \mathrm{S}_1 \times \cdots \times \mathrm{S}_n \\ s^0 & = & (s_0^0, s_1^0, \cdots, s_n^0) \\ \mathrm{SI} & \subseteq & \bigcup_{i=[0,n]} \mathrm{E}_i \\ expr(\mathrm{BV}) & = & expr(\mathrm{BV}_0) \end{array}$$

In defining T*, two cases are considered based on the origin of an external signal injected in $\mathcal{A}$:*

(1) An external signal from above. For an external signal $es_0 \in \mathrm{E}_0$, if

$$\exists s = (s_0, s_1, \cdots, s_n) \in S \ \wedge \ s_0 \xrightarrow{es_0 \ \&\& \ cond/O_0} s_0' \in T_0$$

where $cond \in expr(\mathrm{BV}_0)$ evaluates to true in state s_0, then

$$s \xrightarrow{es_0 \ \&\& \ cond/O} s' \in T \ \wedge \ s' = (s_0', s_1', \cdots, s_n') \in S$$

where O and $s_k'(k = [1, n])$ are defined by the following

$$O = \biguplus_{k=[1,n]} O_k$$

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 $O_k = O_k' \ \wedge \ s_k' = s_k''$;*
- *Else $O_k = \varnothing \ \wedge \ s_k' = s_k$.*

(2) An external signal from below. For $es_i \in \mathrm{E}_i(i = [1, n])$, if

$$\exists s = (s_0, s_1, \cdots, s_n) \in S \wedge s_i \xrightarrow{es_i/O_i} s_i' \in T_i \wedge s_0 \xrightarrow{O_i \ \&\& \ cond/O_0} s_0' \in T_0$$

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

$$s \xrightarrow{es_i \ \&\& \ cond/O} s' \in T \ \wedge \ s' = (s_0', s_1', \cdots, s_n') \in S$$

where O and $s_k'(k = [1, n], k \neq i)$ are defined by the following

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

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* $O_k = O'_k \ \wedge \ s'_k = s''_k$;
- *Else* $O_k = \varnothing \ \wedge \ s'_k = s_k$.

Definition 17 covers in total six scenarios:

1. An external signal comes from above and $O_0 = \varnothing$. This corresponds to the case when the mode switch of a composite component implies no mode switch among its subcomponents.
2. An external signal comes from above and $O_0 \neq \varnothing$. This corresponds to the case when the mode switch of a composite component implies the mode switch of at least one of its subcomponents.
3. An external signal comes from below and $O_0 = \varnothing$. This corresponds to the case when the mode switch of a component implies no mode switch among its parent and its siblings (two components are siblings if they share the same parent).
4. An external signal comes from below and O_0 only contains an external signal es_0. This corresponds to the case when the mode switch of a component only implies the mode switch of its parent but not its siblings.
5. An external signal comes from below and O_0 only contains a set of internal signals $is_k(k = [1, n])$. This corresponds to the case 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 comes from below and O_0 contains both es_0 and $is_k(k = [1, n])$. This corresponds to the case when the mode switch of a component not only implies the mode switch of its parent, but also implies the mode switch of at least one of its siblings.

Next let's demonstrate each scenario by a simple example, where a composite component *a* has two subcomponents *b* and *c*. Scenario 1 can be demonstrated in Figure 4.11, which includes the mode mapping table of *a*, the MMA of each component and the MMA after composition. Suppose an external signal $a.E(m_a^2)$ arrives at MMA_a (the parent MMA), triggering the transition $m_a^1 \xrightarrow{a.E(m_a^2)/\varnothing} m_a^2$. Since this mode switch scenario does not imply the mode switch of *b* or *c*, no state transition will occur for MMA_b or MMA_c. Let $\mathcal{A}$ be the MMA after composition, with s_1 as the state before the transition of MMA_a

and s_2 as the state after the transition of MMA$_a$. Then according to Definition 17, $\mathcal{A}$ will undergo the transition $s_1 \xrightarrow{a.E(m_a^2)/\varnothing} s_2$ where $s_1 = (m_a^1, m_b^1, m_c^1)$ and $s_2 = (m_a^2, m_b^1, m_c^1)$.

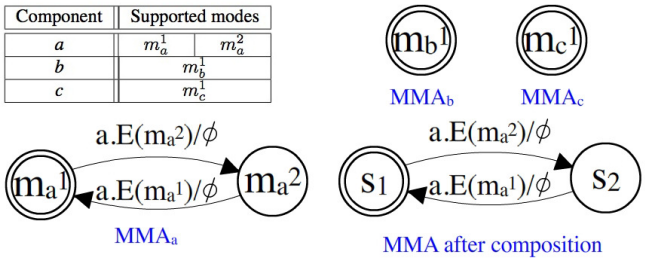

Figure 4.11: MMA composition—Scenario 1

Figure 4.12 demonstrates both Scenario 2 and Scenario 6, based on a mode mapping table different from that in Figure 4.11. Scenario 2 occurs as an external signal $a.E(m_a^2)$ arrives at MMA$_a$, triggering the transition $m_a^1 \xrightarrow{a.E(m_a^2)/\{b.I(m_b^2),c.I(m_c^2)\}} m_a^2$. This leads to the transition $m_b^1 \xrightarrow{b.I(m_b^2)/\{b.E(m_b^2)\}} m_b^2$ of MMA$_b$ and the transition $m_c^1 \xrightarrow{c.I(m_c^2)/\{c.E(m_c^2)\}} m_c^2$ of MMA$_c$. Then $\mathcal{A}$ will undergo the transition $s_1 \xrightarrow{a.E(m_a^2)/\{b.E(m_b^2),c.E(m_c^2)\}} s_2$ where $s_1 = (m_a^1, m_b^1, m_c^1)$ and $s_2 = (m_a^2, m_b^2, m_c^2)$.

Scenario 6 occurs as an external signal $b.E(m_b^1)$ arrives at MMA$_b$, triggering the transition $m_b^2 \xrightarrow{b.E(m_b^1)/\{b.I(m_b^2 \to m_b^1)\})} m_b^1$. This leads to the transition $m_a^2 \xrightarrow{b.I(m_b^2 \to m_b^1)\ \&\&\ cond/\{a.E(m_a^1),c.I(m_c^1)\}} m_a^1$ of MMA$_a$, which further leads to the transition $m_c^2 \xrightarrow{c.I(m_c^1)/\{c.E(m_c^1)\}} m_c^1$ of MMA$_c$. Then $\mathcal{A}$ will undergo the transition $s_2 \xrightarrow{b.E(m_b^1)\ \&\&\ cond/\{a.E(m_a^1),c.E(m_c^1)\}} s_1$ where $s_2 = (m_a^2, m_b^2, m_c^2)$ and $s_1 = (m_a^1, m_b^1, m_c^1)$.

Figure 4.13 demonstrates Scenario 3. Suppose an external signal $b.E(m_b^2)$ arrives at MMA$_b$, triggering the transition $m_b^1 \xrightarrow{b.E(m_b^2)/\{b.I(m_b^1 \to m_b^2)\}} m_b^2$. This leads to the transition $m_a^1 \xrightarrow{b.I(m_b^1 \to m_b^2)/\varnothing} m_a^1$ of MMA$_a$ without affecting MMA$_c$. Then $\mathcal{A}$ will undergo the transition $s_1 \xrightarrow{b.E(m_b^2)/\varnothing} s_2$ where $s_1 =$

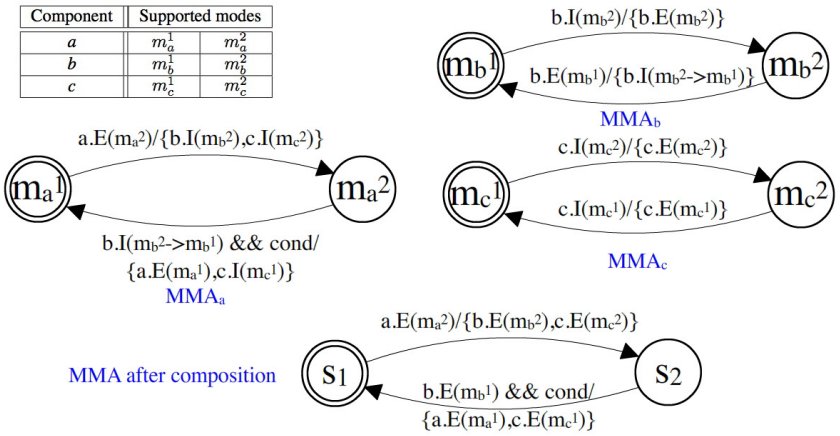

Figure 4.12: MMA composition—Scenarios 2 and 6

(m_a^1, m_b^1, m_c^1) and $s_2 = (m_a^1, m_b^2, m_c^1)$.

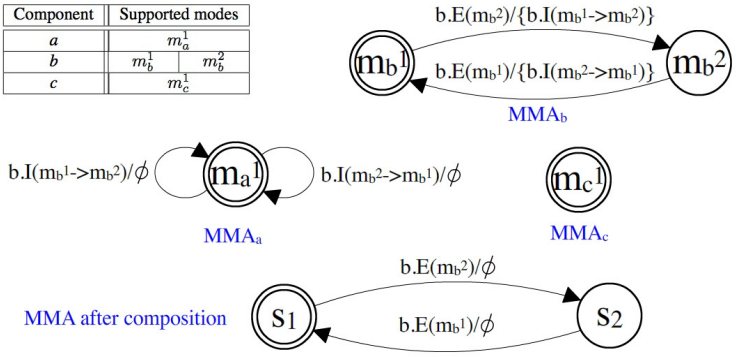

Figure 4.13: MMA composition—Scenario 3

Figure 4.14 demonstrates Scenario 4. Suppose an external signal $c.E(m_c^2)$ arrives at MMA$_c$, triggering the transition $m_c^1 \xrightarrow{c.E(m_c^2)/\{c.I(m_c^1 \to m_c^2)\}} m_c^2$. This leads to the transition $m_a^1 \xrightarrow{c.I(m_c^1 \to m_c^2)/\{a.E(m_a^2)\}} m_a^2$ of MMA$_a$ without affecting MMA$_b$. Then $\mathcal{A}$ will undergo the transition $s_1 \xrightarrow{c.E(m_c^2)/\{a.E(m_a^2)\}} s_2$

where $s_1 = (m_a^1, m_b^1, m_c^1)$ and $s_2 = (m_a^2, m_b^1, m_c^2)$.

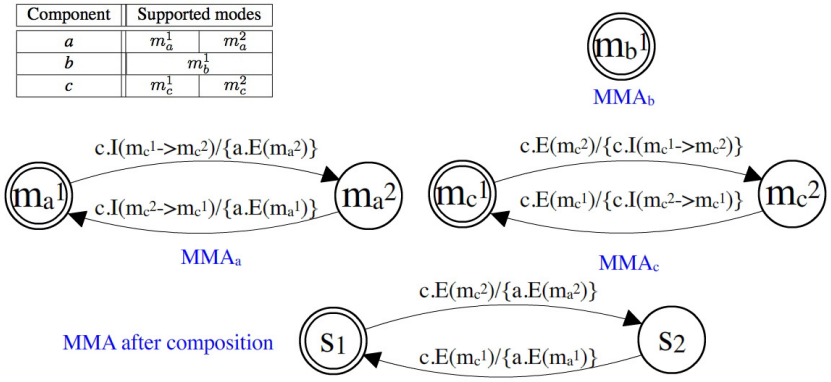

Figure 4.14: MMA composition—Scenario 4

Finally, Figure 4.15 demonstrates Scenario 5. Suppose an external signal $c.E(m_c^2)$ arrives at MMA$_c$, triggering the transition $m_c^1 \xrightarrow{c.E(m_c^2)/\{c.I(m_c^1 \to m_c^2)\}} m_c^2$. This leads to the transition $m_a^1 \xrightarrow{c.I(m_c^1 \to m_c^2)\ \&\&\ cond\ 1/\{b.I(m_b^2)\}} m_a^1$ of MMA$_a$, which further leads to the transition $m_b^1 \xrightarrow{b.I(m_b^2)/\{b.E(m_b^2)\}} m_b^2$. Then $\mathcal{A}$ will undergo the transition $s_1 \xrightarrow{c.E(m_c^2)\ \&\&\ cond\ 1/\{b.E(m_b^2)\}} s_2$ where $s_1 = (m_a^1, m_b^1, m_c^1)$ and $s_2 = (m_a^1, m_b^2, m_c^2)$.

4.4 Summary

This chapter reveals the mode incompatibility problem for composing multi-mode components and the demand for mode mapping for a CBMMS. A mode mapping mechanism is proposed as the corresponding solution. The mode mapping should always be locally handled by each composite component which contains a set of mode mapping rules which are either static or dynamic. Static mode mapping rules can be intuitively represented by mode mapping tables for component composition. However, dynamic mode mapping rules, referred to the MSP protocol at runtime, are beyond the description of the mode mapping table. Therefore, we suggest the usage of a more powerful

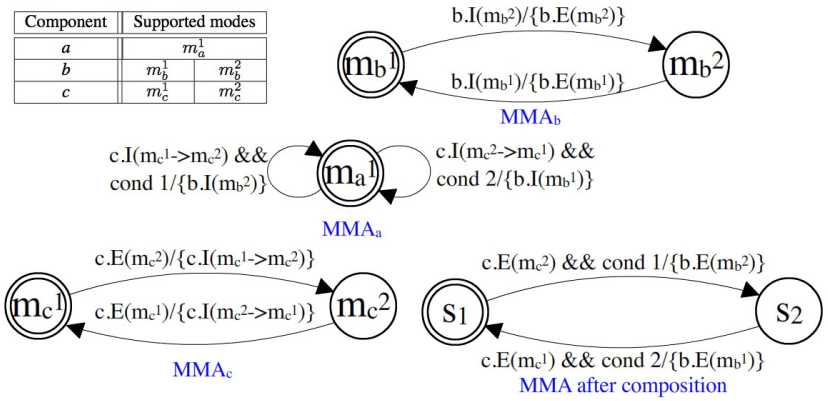

Figure 4.15: MMA composition—Scenario 5

presentation, Mode Mapping Automata (MMA), for expressing both static and dynamic mode mapping rules. Two types of MMAs can be distinguished, i.e. the parent MMA and the child MMA. The formal semantics of both types have been provided. The relation between the MSP protocol and a set of MMAs is explained. Furthermore, we formally define MMA composition which is demonstrated by a simple example based on six scenarios.

Chapter 5

The handling of atomic component execution

Until now we have assumed that the current execution of a component can be immediately aborted to perform a mode switch. However, on many occasions, a component can have executions which cannot be aborted before its completion. Such atomic execution is quite common in real-world applications. For instance, each transaction of the online transfer system of a bank is typically atomic. The transaction should be either successfully completed or aborted in such a way that the system remains in the same state as before the transaction. If the transaction is aborted midway, 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. For a CBMMS with atomic component execution, the mode switch handling becomes more challenging as the atomic execution of a component should not be interrupted by any mode switch event. In this chapter, we extend our MSP protocol to cope with atomic component execution, assuming a pipe-and-filter execution pattern for the CBMMS.

5.1 The pipe-and-filter CBMMS model with atomic component execution

Our mode-aware component model introduced in Chapter 2 implicitly assumes the pipe-and-filter execution pattern. A component has a number of input and output ports. All components follow the same execution pattern of a pipe-and-filter system: wait for the input data, process the data, and produce the output data. The example introduced in Figure 1.1 at the very beginning of this thesis is a typical pipe-and-filter CBMMS. Our mode-aware component model and the mode switch runtime mechanism also work for other software architectures. The reason why we particularly focus on the pipe-and-filter architecture is that a pipe-and-filter system is very suitable to demonstrate the problem of atomic execution. Likewise, atomic component execution can be easily demonstrated in a pipe-and-filter component-based system.

Figure 5.1 (A) presents a pipe-and-filter component-based system. The input data is processed by different components of the system, starting from *a* and going out through the output of *g* and *i*. There are diverging and converging branches that enable more complex data flow structure. In practice, even feedback loops can be used for more advanced data flow control. A real-world example of such a pipe-and-filter component-based system is a fuel gauge on a car's dashboard [39]. Actually, the pipe-and-filter interaction paradigm is fairly common in multimedia systems and telecommunications systems. Concurrent data processing is a typical advantage of such type of systems in that different data can be processed simultaneously by different components.

Atomic component execution in a pipe-and-filter component-based system can be specified by *Atomic Execution Groups (AEGs)*:

Definition 18. ***[Atomic Execution Group (AEG)]:*** *An Atomic Execution Group (AEG) is a single component or a group of horizontally contiguous components with atomic execution. For an AEG G, a component $c_k \in G$, and an input port p_i of G, there must exist a sequence of connections ρ through which any input data from p_i can reach c_k without flowing out of G.*

Figure 5.1 (B) specifies two AEGs for the system in Figure 5.1 (A). Component *e* is defined as an AEG alone, i.e. AEG 1. Components *h* and *i* are together defined as the other AEG, i.e. AEG 2. This example suggests that it is possible to define multiple AEGs within a composite component. Yet multiple AEGs should not overlap. Otherwise, two overlapping AEGs become contiguous and should be considered as a single bigger AEG. It is also allowed to have an AEG included by a bigger AEG, which then absorbs the smaller one.

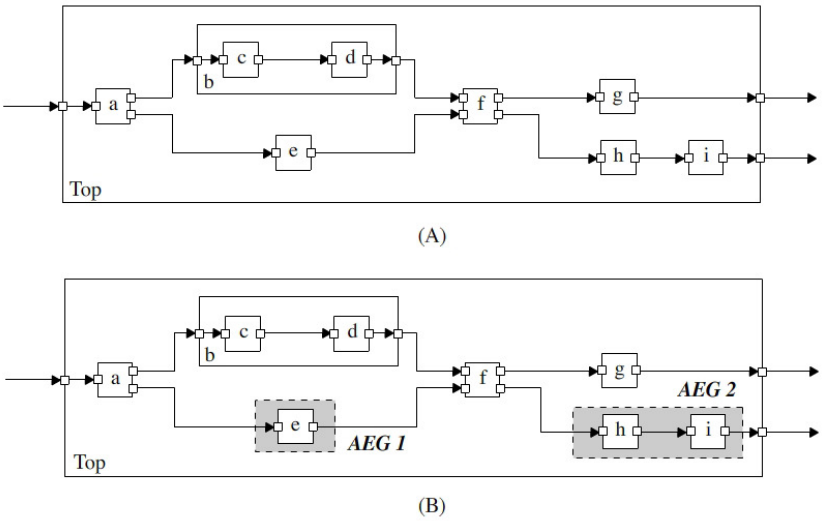

Figure 5.1: Atomic component execution in a pipe-and-filter component-based system

AEGs in a pipe-and-filter CBMMS can be defined in the same fashion. The extra concern is that an AEG can be mode-dependent. Figure 5.2 extends the example in Figure 5.1 (B) into a pipe-and-filter CBMMS. The system (Component *Top*) supports two modes: m^1_{Top} and m^2_{Top}. In m^1_{Top}, the system configuration is the same as the system in Figure 5.1 (B). In m^2_{Top}, the internal behavior of Component *a* is changed and Component *c* becomes deactivated. Moreover, AEG 1 and 2 are not valid any more in m^2_{Top}. Instead, a new AEG (AEG 3) is defined, including *a*, *b*, *e* and *f*. This reveals the fact that an AEG can consist of components with mixed-granularity, since the subcomponent of *b*, i.e. *d* also belongs to AEG 3, yet at a lower level of the component hierarchy. Different from *d*, *c* is deactivated when *Top* is in m^2_{Top}, and thus does not belong to any AEG. Generally speaking, if $c_i \in CC$ is an AEG in mode m, then $\forall c_k \in AD^m_{c_i}$ (i.e. all the activated descendants of c_i) must be in the same AEG as well.

In our mode-aware component model, AEGs can be defined as mode-dependent or mode-independent EFPs, e.g. atomicity which could be either atomic or non-atomic for a given mode. A primitive multi-mode component

can expose its atomicity property in each mode when it is reused. A composite multi-mode component not only exposes its own atomicity property in each mode, but also has the knowledge of the AEG specification among its subcomponents. The AEG specification is not constrained by the atomicity property, as even non-atomic components can be included in an AEG and become atomic in the context where it is reused. The following defines the atomicity property and AEGs of *Top* in the system in Figure 5.2 based on the mode-aware component model:

$$\begin{aligned} Top.Atomicity &= \{m^1_{Top} \rightarrow \text{Non-atomic}, m^2_{Top} \rightarrow \text{Non-atomic}\} \\ Top.AEG &= \{m^1_{Top} \rightarrow \{\{e\},\{h,i\}\}, m^2_{Top} \rightarrow \{\{a,b,e,f\}\}\} \end{aligned}$$

According to the definition above, *Atomicity* is a function mapping each supported mode of *Top* to either atomic or non-atomic; *AEG* is another function mapping each supported mode of *Top* to the specified AEGs among the subcomponents of *Top*. These two functions are preserved in the component configuration of *Top*. This definition also works for all the other components. For a primitive component, the function *AEG* is omitted.

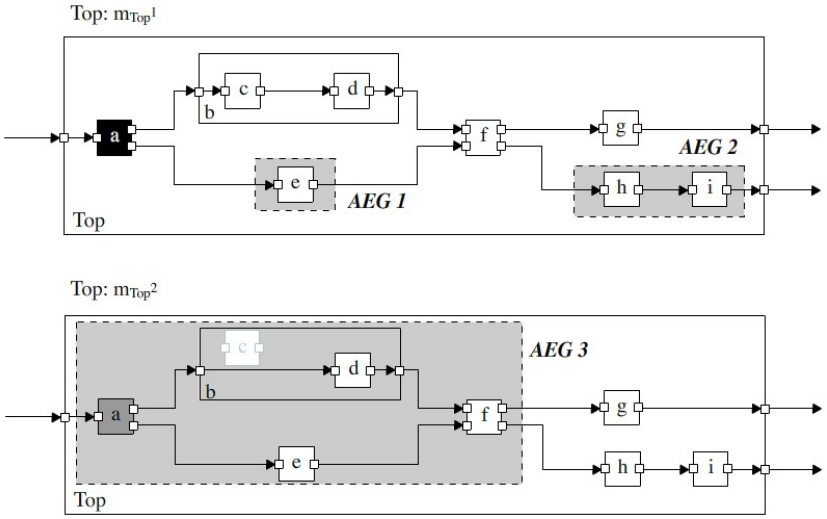

Figure 5.2: Atomic component execution in a pipe-and-filter CBMMS

5.2 The handling of atomic component execution

Atomic component execution inevitably imposes additional challenge on the mode switch handling of a CBMMS in the sense that an ongoing atomic execution cannot be interrupted to perform a mode switch. The MSP protocol requires that a component must stop running in the current mode immediately when it receives an **MSQ**, or when it actively issues an **MSQ** as both the top component and an MSS, whereas this requirement may not be met if an **MSQ** is sent to an AEG. This indicates that the atomic component execution problem should be handled during the mode switch propagation. The mode mapping mechanism and the mode switch dependency rule do not need to be altered. The handling of atomic component execution could be handled by revising the MSP protocol. The question is how the MSP protocol can be extended with minimum modification.

The basic idea of extending the MSP protocol to cater for the handling of atomic component execution is to avoid **MSQ** transmission to the Type A components in an AEG with ongoing atomic execution. If components belonging to an AEG receive an **MSQ** while the AEG has no ongoing atomic execution, these components can still follow the original MSP protocol, since they can abort their execution in the current mode just like other components excluded from any AEG. Evidently, this idea should be applied to a non-atomic composite component with at least one AEG as its subcomponent(s) whilst all the other components can simply disregard the handling of atomic component execution. Since **MSQ** propagation is in the first phase of the mode switch propagation, the focus of the extended MSP protocol should also be on the first phase. In the second phase, all Type A components have stopped running in the current mode (i.e. without ongoing execution) and await either an **MSI** or **MSD**, thus AEGs do not require special treatment.

For a non-atomic composite component c_i containing at least one AEG, in order to send an **MSQ** to its Type A subcomponents in each AEG at the proper time, we assume that c_i is able to distinguish the presence or absence of any ongoing atomic execution of any AEG composed by its subcomponents. Here we introduce the concept *Data Processing Status (DPS)*:

Definition 19. ***[Data Processing Status (DPS)]:*** *The Data Processing Status (DPS) of an AEG in a pipe-and-filter CBMMS reflects if there is any data being processed within the AEG. The existence of data being processed in the AEG indicates the presence of atomic execution and the DPS of the AEG returns "Processing". Otherwise, the DPS of the AEG returns "Not processing".*

Once c_i is able to monitor the DPS of each of its AEG(s), it will avoid **MSQ** transmission to its Type A subcomponents in an AEG whose DPS is "Processing". In other words, c_i can delay its **MSQ** transmission to those components until the DPS of the corresponding AEG becomes "Not processing". Besides, when c_i decides to propagate an **MSQ**" to an AEG, it should freeze the input of the AEG so that no new input data will enter the AEG. Without this action, the input data flow of the AEG may never cease and the DPS of the AEG could potentially never change status to "Not processing". Conversely, c_i must be able to unfreeze (to allow the input data to enter the AEG) the input of an AEG whose input has been frozen after the **MSC** collection of c_i or after c_i receives at least one **MSNOK**.

The extended MSP protocol still retains the essence of the original MSP protocol, ergo we shall only present what has been extended and revised compared with the original MSP protocol:

Definition 20. ***[The supplement to the MSP protocol considering atomic component execution]:*** *Let $c_i \in CC$ be a non-atomic component, containing a number of AEGs denoted as the set $AEG_{c_i} = \{G_1, G_2, \cdots, G_n\}$ ($n \in \mathbb{N}$) and a number of non-atomic subcomponents denoted as the set H_{c_i}, $AEG_{c_i} \cup H_{c_i} = SC_{c_i}$ and $AEG_{c_i} \cap H_{c_i} = \varnothing$. When c_i decides to propagate an **MSQ**, it immediately sends the **MSQ** to all $c_h \in H_{c_i}, T_{c_h} = A$. Meanwhile, $\forall G_k (k = [1, n])$, c_i checks the DPS of G_k and freezes its input. If the DPS of G_k returns "Not processing", c_i will immediately send the **MSQ** to all $c_g \in G_k, T_{c_g} = A$. If the DPS of G_k returns "Processing", c_i will delay its **MSQ** transmission to all c_g till the DPS of G_k becomes "Not Processing". Component c_i unfreezes the input of G_k to accept new input data either after the **MSC** collection of c_i or after c_i receives at least one **MSNOK**.*

The extended MSP protocol does not exclude the case when an AEG is included in a bigger AEG. Since the parent of the outer AEG freezes its input and does not propagate the **MSQ** until all data within it have been processed, the DPS of the enclosed AEG is always "Not processing" when its parent is ready to propagate an **MSQ**.

Comparing the original and the extended MSP protocols, one can safely conclude that the key difference is the **MSQ** propagation delay to the Type A components in an AEG. Therefore, all the correctness and safety properties formulated for the original MSP protocol also hold for the extended version, which only prolongs the mode switch time for a bounded delay.

The extended MSP protocol can be demonstrated by the example in Figure 3.4. The entire mode switch process of the system in Figure 3.4 has been

demonstrated by figures 3.6 and 3.8. We shall introduce an AEG to this system for another round of its mode switch demonstration so as to expose the impact of atomic component execution upon the mode switch process. Figure 5.3 displays the component connections of the system in Figure 3.4 when the MSS *h* issues an **MSR**. Component *b* is defined as an AEG. Component *m* is deactivated, yet it is a Type A component as it will be activated in the new mode. The complete mode switch process of this system is illustrated in Figure 5.4. Represented by the grey bar, the atomic execution of the AEG *b* stems from the data processing time of its subcomponents *e* and *f*. As is indicated in Figure 5.4, *a* has the responsibility of sending an **MSQ** to the AEG *b* at the right instant. When *a* decides to propagate the **MSQ**, it sends the **MSQ** to its non-atomic subcomponent *c* right away. Additionally, *a* freezes the input of *b* and checks its DPS, which returns "Processing". Consequently, *a* reserves the **MSQ** for *b*, keeping track of the DPS of *b*. When the DPS of *b* is changed to "Not processing" (this corresponds to the termination of the grey bar in Figure 5.4), *a* will send the **MSQ** to *b*, which thereafter will be treated as a normal composite component. When *a* receives the **MSC** from *b*, *a* will unfreeze the input of *b*. From the comparison between the entire mode switch process of the system with or without atomic component execution, the impact of atomic component execution can be perceived without too much effort. An AEG potentially prolongs the mode switch propagation time in the first phase, however, the rest of the mode switch process remains intact.

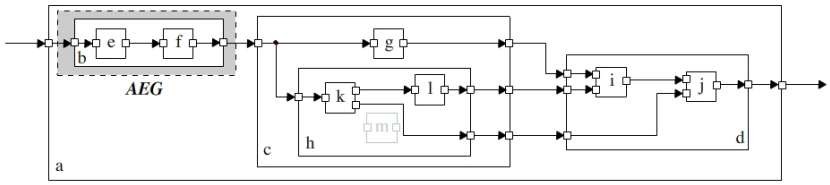

Figure 5.3: The inner component connections of Component *a* in its current mode based on the system in Figure 3.4

It is worthy to mention that we have not addressed how a parent should freeze or unfreeze the input of an AEG composed by its subcomponents, or how the parent keeps track of the DPS of the AEG. Of course, the communication between the parent and the AEG can be established by primitives such as an **MSQ** and an **MSOK**, but too many new primitive types may complicate the understanding of the essence of the MSP protocol. For that reason, it is

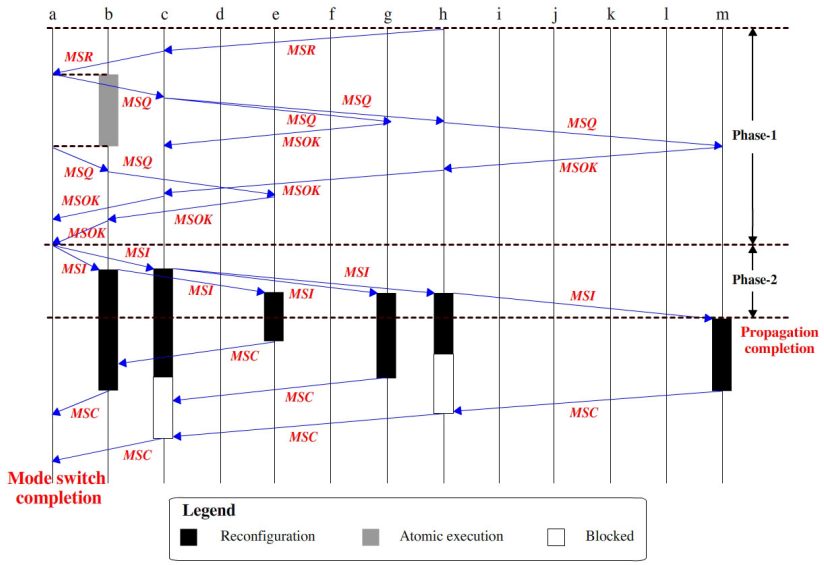

Figure 5.4: The mode switch process with atomic component execution

assumed that freezing or unfreezing the input of an AEG and monitoring the DPS of the AEG can be directly and instantaneously handled by the system. This is rather application-dependent and is out of the scope of this thesis.

5.3 Summary

This chapter is devoted to the handling of atomic component execution during a mode switch. Although such handling belongs to the mode switch runtime mechanism, we use a separate chapter to discuss this specific problem. In MSL, atomic component execution is defined as Atomic Execution Groups (AEGs) which potentially affect the mode switch propagation. The MSP protocol is extended to cater for the presence of AEGs. Compared with the original MSP protocol described in Chapter 3, the extended version can delay an **`MSQ`** propagation to an AEG with ongoing atomic execution. In this chapter, the pipe-and-filter architecture of a CBMMS is assumed in that it presents atomic component execution well. However, our handling of atomic component execution should work for other software architectures as well.

Chapter 6

Algorithms for the mode switch runtime mechanism

The mode switch runtime mechanism of our MSL can be easily implemented for each component of a CBMMS. We distinguish three types of components: (1) primitive component; (2) non-top composite component; (3) the top component[1]. The mode switch runtime mechanism of components of the same type is exactly the same, yet different from the other two types. Additionally, any component can be an MSS and this must be taken into account. In this chapter, the mode switch runtime mechanism of MSL is implemented as algorithms for all these types of components.

6.1 Implementing MSL in a primitive component

Algorithm 1 implements the mode switch runtime mechanism of $c_i \in PC$. As a primitive component, the mode switch behavior of c_i is quite simple. If c_i is not an MSS in is current mode, the only primitive that it can receive is an **`MSQ`**, which makes c_i stop running in the current mode and reply with either an **`MSOK`** or an **`MSNOK`** depending on its current state. If c_i replies with an **`MSOK`**, it will expect either an **`MSI`** or **`MSD`**. Component c_i starts its reconfiguration after receiving an **`MSI`**. After reconfiguration, c_i sends an **`MSC`** back and starts

[1] In principle, the top component can be either primitive or composite. However, if it is primitive, it makes not any sense to call it a component-based system because the system itself is just one single component. Therefore, in this thesis, we assume the top component must be composite.

running in the new mode. If c_i receives an **MSD** after replying with an **MSOK**, it can resume its execution in its current mode. In opposition, if c_i replies with an **MSNOK**, it will only expect an **MSD**. Furthermore, if c_i is an MSS in its current mode, when it detects a mode switch event, it can actively issue an **MSR** to P_{c_i} without stopping its current execution. Since c_i has no subcomponents, there is no need to consider mode mapping or communication with components at lower levels. A few points deserve further explanation in the algorithm:

- p^{MSX} is the dedicated mode switch port of c_i for the communication with P_{c_i}. More details can be found in the description of the mode-aware component model in Chapter 2.

- *Wait*(A, B) and *Signal*(A, B) are used for receiving and sending a primitive. The parameter "A" is the dedicated mode switch port through which a primitive is sent or received while the parameter "B" specifies the primitive type. An **MSQ/MSI/MSD** is sent via p_{in}^{MSX} and received via p^{MSX}; an **MSR/MSOK/MSNOK/MSC** is sent via p^{MSX} and received via p_{in}^{MSX}.

- *MSR*$(c_i, m_{c_i}, m_{c_i}^{new})$ represents an **MSR** associated with the mode switch scenario: $c_i : m_{c_i} \rightarrow m_{c_i}^{new}$. Subsequently, *MSX*$(c_i, m_{c_i}^{new})$ (*MSX* can be *MSQ*, *MSI*, *MSD* or *MSC*) represents the primitive from or to c_i in the same mode switch scenario.

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

- *Stop_running*(c_i, m_{c_i}) means c_i stops running in its current mode m_{c_i}. Similarly, *Resume*(c_i, m_{c_i}) means c_i resumes its execution in its current mode and *Start_running*$(c_i, m_{c_i}^{new})$ means c_i starts to run in its new mode.

- *MS_event_detected* is a boolean variable set to true when c_i detects a mode switch event as an MSS; *MS_ready* is a boolean variable set to true when the current state of c_i allows a mode switch in response to an **MSQ**. The current state checking of c_i is realized by the function *Check_state*(c_i, m_{c_i}).

Algorithm 1 *MS_Handling*$(c_i \in PC)$

```
loop
  if c_i = MSS then
    if MS_event_detected then
      Derive_new_mode;
      Signal(p^MSX, MSR(c_i, m_{c_i}, m_{c_i}^{new}));
    end if
  end if
  Wait(p^MSX, MSQ(c_i, m_{c_i}^{new}));
  Stop_running(c_i, m_{c_i});
  Check_state(c_i, m_{c_i});
  if MS_ready then
    Signal(p^MSX, MSOK(c_i, m_{c_i}^{new}));
    Wait(p^MSX, MSI(c_i, m_{c_i}^{new}) ∨ MSD(c_i, m_{c_i}^{new}));
    if MSI then
      Reconfiguration(c_i, m_{c_i}, m_{c_i}^{new});
      Signal(p^MSX, MSC(c_i, m_{c_i}^{new}));
      Start_running(c_i, m_{c_i}^{new});
    else
      Resume(c_i, m_{c_i});
    end if
  else
    Signal(p^MSX, MSNOK(c_i, m_{c_i}^{new}));
    Wait(p^MSX, MSD(c_i, m_{c_i}^{new}));
    Resume(c_i, m_{c_i});
  end if
end loop
```

6.2 Implementing MSL in a non-top composite component

Algorithm 2 implements the mode switch runtime mechanism of a non-top composite component $c_i \in \widetilde{CC}$. Since c_i interacts with both P_{c_i} and SC_{c_i}, its mode switch handling is much more complex than a primitive component. Algorithm 2 additionally calls Algorithm 3 which will be explained later. Algorithms 2 and 3 rigorously adhere to the MSP protocol, mode mapping, the mode switch dependency rule and the handling of atomic component execution discussed in previous chapters. Further details are explained as follows:

- Apart from p^{MSX}, the port p^{MSX}_{in} is the other dedicated mode switch port of c_i for the communication with SC_{c_i}. More details can be found in the description of the mode-aware component model in Chapter 2.

- *ModeSwitch*$(c_i, MSDM, Top)$ is a function (i.e. Algorithm 3) dealing with the mode switch of c_i since c_i is ready to propagate an **`MSQ`**. *MSDM* is a boolean variable set to true when c_i is the MSDM for a specific mode switch scenario; *Top* is a boolean variable set to true when $c_i = Top$.

- *ModeMapping* is a function which returns the mode mapping result of c_i.

- H_{c_i} denotes the set of SC_{c_i} which do not belong to any AEG, as is introduced in the extended MSP protocol; AEG_{c_i} denotes the set of AEGs defined among SC_{c_i} and G_k denotes one specific AEG in AEG_{c_i}.

- *FreezeInput*(G_k) and *UnfreezeInput*(G_k) freeze and unfreeze the input of AEG G_k. The reason derives from the extended MSP protocol.

- *Check_DPS*(G_k) is used when c_i checks the DPS of its AEG G_k, i.e. DPS_{G_k} whose value is either "Processing" or "Not processing".

- *MSOKNOK_Collection* is used when c_i collects the feedback (**`MSOK`** or **`MSNOK`**) from its Type A subcomponents. The result is expressed by the boolean variable *all_MSOK* which is set to true if all the Type A subcomponents of c_i reply with an **`MSOK`**. In a similar fashion, *MSC_Collection* is a function for collecting the **`MSC`** from SC_{c_i} and *all_MSC* is a boolean variable set to true when the **`MSC`** collection of c_i is done.

Algorithm 2 *MS_Handling*$(c_i \in \widetilde{CC})$

```
loop
  if c_i = MSS then
    if MS_event_detected then
      Derive_new_mode;
      Signal(p^MSX, MSR(c_i, m_{c_i}, m_{c_i}^{new}));
    end if
  end if
  Wait(p^MSX ∨ p_in^MSX, MSR(c_o ∈ SC_{c_i}, m_{c_o}, m_{c_o}^{new}) ∨ MSQ(c_i, m_{c_i}^{new}));
  if MSR then
    ModeMapping;
    if m_{c_i}^{new} = m_{c_i} then
      ModeSwitch(c_i, true, false);
    else
      Check_state(c_i, m_{c_i});
      if MS_Ready then
        Signal(p^MSX, MSR(c_i, m_{c_i}, m_{c_i}^{new}));
      end if
    end if
  else
    Stop_running(c_i, m_{c_i});
    Check_state(c_i, m_{c_i});
    if MS_Ready then
      ModeMapping;
      ModeSwitch(c_i, false, false);
    else
      Signal(p^MSX, MSNOK(c_i, m_{c_i}^{new}));
      Wait(p^MSX, MSD(c_i, m_{c_i}^{new}));
      Resume(c_i, m_{c_i});
    end if
  end if
end loop
```

Algorithm 3 *ModeSwitch*$(c_i \in CC, MSDM, Top)$

Signal$(p_{in}^{MSX}, MSQ(c_p, m_{c_p}^{new})) \rightarrow \forall c_p \in H_{c_i}, T_{c_p} = A$;
$\forall G_k \in AEG_{c_i}$: *FreezeInput*(G_k);
repeat
  $\forall G_k \in AEG_{c_i}$: *Check_DPS*(G_k);
until $DPS_{G_k} = \neg$Processing
Signal$(p_{in}^{MSX}, MSQ(c_k, m_{c_k}^{new})) \rightarrow \forall c_k \in G_k, T_{c_k} = A$;
MSOKNOK_Collection;
if *all_MSOK* **then**
  if *MSDM* || *Top* **then**
    Signal$(p_{in}^{MSX}, MSI(c_j, m_{c_j}^{new})) \rightarrow \forall c_j \in SC_{c_i}, T_{c_j} = A$;
    if $T_{c_i} = A$ **then**
      Reconfiguration$(c_i, m_{c_i}, m_{c_i}^{new})$;
    end if
    repeat
      MSC_Collection;
    until *all_MSC*
    $\forall G_k \in AEG_{c_i}$: *UnfreezeInput*(G_k);
    if $T_{c_i} = A$ **then**
      Start_running$(c_i, m_{c_i}^{new})$;
    else
      Resume(c_i, m_{c_i});
    end if
  else
    Signal$(p^{MSX}, MSOK(c_i, m_{c_i}^{new}))$;
    Wait$(p^{MSX}, MSI(c_i, m_{c_i}^{new}) \vee MSD(c_i, m_{c_i}^{new}))$;
    if *MSI* **then**
      Signal$(p_{in}^{MSX}, MSI(c_j, m_{c_j}^{new})) \rightarrow \forall c_j \in SC_{c_i}, T_{c_j} = A$;
      Reconfiguration$(c_i, m_{c_i}, m_{c_i}^{new})$;
      repeat
        MSC_Collection;
      until *all_MSC*
      Signal$(p^{MSX}, MSC(c_i, m_{c_i}^{new}))$;
      $\forall G_k \in AEG_{c_i}$: *UnfreezeInput*(G_k);
      Start_running$(c_i, m_{c_i}^{new})$;
    else
      Signal$(p_{in}^{MSX}, MSD(c_j, m_{c_j}^{new})) \rightarrow \forall c_j \in SC_{c_i}, T_{c_j} = A$;
      $\forall G_k \in AEG_{c_i}$: *UnfreezeInput*(G_k);
      Resume(c_i, m_{c_i});
    end if
  end if

else
  if *MSDM* || *Top* **then**
    Signal$(p_{in}^{MSX}, MSD(c_j, m_{c_j}^{new})) \rightarrow \forall c_j \in SC_{c_i}, T_{c_j} = A$;
    $\forall G_k \in AEG_{c_i}$: *UnfreezeInput*(G_k);
    Resume(c_i, m_{c_i});
  else
    Signal$(p^{MSX}, MSNOK(c_i, m_{c_i}^{new}))$;
    Wait$(p^{MSX}, MSD(c_i, m_{c_i}^{new}))$;
    Signal$(p_{in}^{MSX}, MSD(c_j, m_{c_j}^{new})) \rightarrow \forall c_j \in SC_{c_i}, T_{c_j} = A$;
    $\forall G_k \in AEG_{c_i}$: *UnfreezeInput*(G_k);
    Resume(c_i, m_{c_i});
  end if
end if

6.3 Implementing MSL in the top component

Algorithm 4 implements the mode switch runtime mechanism of the top component, which can be treated as a special composite component without parent. Compared with a normal composite component, the mode switch handling of the top component is simpler. Since the top component has no parent, it can never receive an **`MSI`**/**`MSQ`**/**`MSD`** nor forward an **`MSR`** upwards. Therefore, Algorithm 4 can be regarded as a pruned-version of Algorithm 2. However, extra attention must be paid when the top component actively issues an **`MSQ`** as an MSS. Another special case is when the top component is an AEG. Usually if a single component is defined as an AEG, its parent has the duty to freeze/unfreeze its input and monitor its DPS. Nevertheless, if the top component itself is an AEG, it must be able to freeze/unfreeze its input and monitor its own DPS. Algorithm 4 also calls Algorithm 3 with the parameters *MSDM* and *Top* always set to true.

6.4 Summary

In this chapter, the mode switch runtime mechanism of MSL is implemented as algorithms (pseudo code) for primitive components, non-top composite components and the top component. This reflects the decentralized feature of our MSL. There is no such component which can coordinate the mode switches of all the other components. These algorithms are utterly congruous with the

Algorithm 4 *MS_Handling*$(c_i = Top)$

```
loop
  if c_i = MSS && MS_event_detected then
    Derive_new_mode;
    ModeMapping;
    if c_i = AEG then
      FreezeInput(c_i);
      repeat
        Check_DPS(c_i);
      until DPS_{c_i} = ¬Processing
    end if
    Stop_running(c_i, m_{c_i});
    ModeSwitch(c_i, true, true);
    UnfreezeInput(c_i);
  else
    Wait(p_in^MSX, MSR(c_j ∈ SC_{c_i}, m_{c_j}, m_{c_j}^new));
    ModeMapping;
    Check_state(c_i, m_{c_i});
    if m_{c_i}^new = m_{c_i} || MS_Ready then
      if m_{c_i}^new ≠ m_{c_i} then
        Stop_running(c_i, m_{c_i});
      end if
      if c_i = AEG then
        FreezeInput(c_i);
        repeat
          Check_DPS(c_i);
        until DPS_{c_i} = ¬Processing
      end if
      ModeSwitch(c_i, true, true);
      UnfreezeInput(c_i);
    end if
  end if
end loop
```

related theories spread over previous chapters, including the MSP protocol, the mode switch dependency rule, the mode mapping mechanism, and the handling of atomic component execution. Since the algorithms are quite close to real codes, it should be rather easy to implement MSL in practice. In fact, these algorithms have been successfully tested in UPPAAL modeling and verification via a couple of small but representative examples. Extensive evaluation of the algorithms is considered to be our future work.

Chapter 7

The composable mode switch timing analysis

In previous chapters, our mode switch runtime mechanism has been thoroughly elaborated for handling the composable mode switch of a CBMMS. Much effort has been put into guaranteeing the correct mode switch behavior of each individual component and the system as a whole, whereas less attention was paid to the timing of a composable mode switch. The timing analysis of a CBMMS is however important since we believe that most CBMMSs are also real-time systems, whose correctness not only depends on the correctness of the computing result, but also on the time when the result is delivered [57]. A classic example of real-time systems is the airbag of a car. When a collision occurs, the airbag must be inflated at the proper time. Either too early or too late inflation of the airbag is considered as a failure. Therefore, a real-time system must meet both the functional and timing requirements. The satisfaction of the timing constraints is typically verified by timing analysis. The timing analysis of a multi-mode system is even more complex as the system must be schedulable both in each stable mode and during a mode switch. The schedulability assurance in a stable mode can be analyzed by taking the advantage of the schedulability of a single-mode system, since the timing analysis in different modes is independent. Such analysis is a well-researched problem outside the scope of this thesis. However, the timing analysis during a mode switch is inevitably related to the mode switch handling. The worst-case mode switch time must be derived so as to guarantee schedulability during a mode switch.

The timing analysis of a CBMMS is equally important, yet requires a dif-

ferent methodology. In this chapter, we present a timing analysis for the composable mode switch of CBMMSs which implement our MSL.

7.1 The mode switch timing analysis for MSL

According to the mode switch runtime mechanism of MSL, a mode switch starts from the mode switch propagation, after which the MSDM decides whether to trigger a mode switch or not. Once a mode switch is triggered, all components follow the mode switch dependency rule. It has been proven that both the MSP protocol and the mode switch dependency rule guarantee bounded mode switch propagation time, and bounded mode switch time since the MSDM issues an **`MSI`**. Hence the complete mode switch time must also be bounded. Next we shall ascertain that this mode switch time is not only bounded but also composable and analyzable. Let α denote an MSS triggering a specific mode switch scenario and β denote the corresponding MSDM, with C_M as the set of vertically intermediate components between α and β, while $\forall c_k \in C_M$, $c_k \in A_\alpha \cap D_\beta$. The following lists some key timing factors and notations:

- $c_i.t_{msr}$, $c_i.t_{msq}$, $c_i.t_{ok}$, $c_i.t_{nok}$, $c_i.t_{msi}$, $c_i.t_{msd}$, $c_i.t_{msc}$: the transmission time of each primitive between c_i and SC_{c_i}. We assume that the transmission time of the same primitive between c_i and its different subcomponents is the same.

- $\alpha.msdt$: the Mode Switch Detecting Time (MSDT) of the MSS α, i.e. the time required to issue an **`MSR`** since α detects a mode switch event. The major activities include analyzing the mode switch event and deriving the new mode of α.

- $c_i.pht$: the Primitive Handling Time (PHT) of $c_i \in CC$, i.e. the time required for a composite component to make a decision upon receiving a primitive. Mode mapping should be the most time-consuming activity, thus $c_i.pht$ is only considered for an **`MSR`** and **`MSQ`** which evokes the mode mapping of c_i.

- $c_i.sct$: the State Checking Time (SCT) of c_i, i.e. the time required to check the current state and respond to an **`MSQ`**.

- 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** with **MSOK** or **MSNOK**. If $c_i \in PC$, $c_i.qrt = c_i.sct$; if $c_i \in CC$, $c_i.qrt \geq c_i.sct$, because c_i may also wait for the reply from SC_{c_i}. Nevertheless, if $c_i \in CC$ replies with an **MSNOK**, then $c_i.qrt = c_i.sct$ because c_i will not propagate the **MSQ** further and wait for any feedback.

- $c_i.rct$: the reconfiguration time of c_i.

- $c_i.mst$: the Mode Switch Time (MST) of c_i since the start of its reconfiguration. If $c_i \in PC$, $c_i.mst = c_i.rct$; if $c_i \in CC$, $c_i.mst \geq c_i.rct$, because c_i may also wait for the **MSC** from SC_{c_i}.

All the timing factors above are for the worst case and can be mode-dependent. A CBMMS is guaranteed to satisfy all the timing constraints for its mode switch only when the worst-case mode switch times of the system itself and all its components meet the specified requirements. These timing factors can be considered as mode-dependent or mode-independent EFPs of a component. If a timing factor is a mode-dependent EFP, its value must be different in different modes. For instance, the atomic execution time of the same AEG may be changed after a mode switch. Sometimes, a mode-dependent EFP may also depend on the current mode and the target mode. For instance, suppose c_i supports three modes: $m^1_{c_i}$, $m^2_{c_i}$ and $m^3_{c_i}$. The reconfiguration time from $m^1_{c_i}$ to $m^2_{c_i}$ can be different from the reconfiguration time from $m^1_{c_i}$ to $m^3_{c_i}$.

In addition, all scheduling effects are assumed to be included in the values of these timing factors. Since our focus is on the mode switch coordination of different components, the consideration of scheduling will compromise the mode switch timing analysis, making it much less comprehensible. Issues such as the hardware platform, the scheduling policy, the possibility of parallel execution, shared resources and execution dependencies between different components inevitably expel our mode switch timing analysis from being understandable. For example, in a fully parallel system, the reconfiguration of different components can be taken without interfering with each other. Yet for a single-processor system, the reconfiguration of different components must be taken by turns, following a specific scheduling policy. As a consequence, the reconfiguration time of some components can be longer due to the preemption of other components. For that reason, we resort to abstract all the scheduling effects and integrate them in the values of the timing factors. After that, the system can be treated as a fully parallel system. Still, a more detailed timing analysis, also including scheduling and other aspects, would be of interest in

obtaining tighter (and possibly also safer) worst-case timing estimates. Such analysis is however outside the scope of this thesis.

In Chapter 3, three scenarios have been identified in the mode switch propagation process (see figures 3.5-3.7). Among these scenarios, only Scenario 2 triggers a mode switch. Our mode switch timing analysis is thus based on Scenario 2. The mode switch propagation process in Scenario 2 has been divided into two phases. The first phase starts when an MSS issues an **MSR** and ends after the **MSOK/MSNOK** collection of the MSDM. The second phase starts when the MSDM issues an **MSI** and ends when the **MSI** is propagated to all Type A components. This phase division idea is also adopted in the mode switch timing analysis. The complete mode switch process of a CBMMS starts when an MSS detects a mode switch event and ends after the MSDM completes its mode switch or after the **MSC** collection of the MSDM. We divide the complete mode switch process into three phases:

- Phase 1: **MSR** propagation. Phase 1 starts when an MSS detects a mode switch event and ends when the MSDM receives the **MSR** originally issued by this MSS.
- Phase 2: **MSQ** propagation. Phase 2 starts when the MSDM decides to issue an **MSQ** to its Type A subcomponents and ends when the MSDM receives all the replies, which must be **MSOK** in order to trigger a mode switch.
- Phase 3: **MSI** propagation and mode switch. Phase 3 starts when the MSDM decides to issue an **MSI** to its Type A subcomponents. If the MSDM is both the top component and a Type A component, Phase 3 ends when the MSDM completes its mode switch. Otherwise, Phase 3 ends after the **MSC** collection of the MSDM.

The timing analysis in these three phases can be performed separately without disturbing each other.

7.1.1 The timing analysis in Phase 1—MSR propagation

Phase 1 only exists when the MSS is not the top component. For a CBMMS with α as an MSS and β as the MSDM, together with C_M as the set of the vertically intermediate components between α and β, Phase 1 is simply the upstream **MSR** propagation from α to β. Figure 7.1 illustrates Phase 1 by a small example, where *a* is an MSS (α); *d* is the MSDM (β); *b* and *c* are the

intermediate components (C_M). When a detects a mode switch event, it spends $a.msdt$ time units analyzing this event, deriving the proper new mode, and determining to issue an **MSR**. After $b.t_{msr}$, the **MSR** arrives at b, which spends another $b.pht$ referring to its mode mapping and deciding to forward the **MSR** to c. Since all components in C_M treat the **MSR** equally, $c.t_{msr}$ and $c.pht$ can be explained in the same manner. When d receives the **MSR**, Phase 1 is over. Let T_1 denote the duration of Phase 1, then T_1 in Figure 7.1 can be easily calculated as

$$T_1 = a.msdt + b.t_{msr} + b.pht + c.t_{msr} + c.pht + d.t_{msr} \tag{7.1}$$

Generalizing (7.1), we get

$$T_1 = \alpha.msdt + \sum_{c_k \in C_M} (c_k.t_{msr} + c_k.pht) + \beta.t_{msr} \tag{7.2}$$

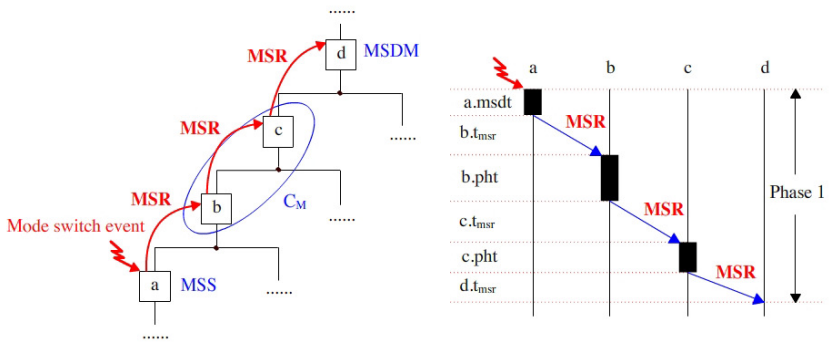

Figure 7.1: The mode switch timing analysis—Phase 1

7.1.2 The timing analysis in Phase 2—MSQ propagation

Phase 2 is dedicated to **MSQ** propagation and **MSOK/MSNOK** collection. Atomic component execution is also handled in this phase. The timing analysis in Phase 2 is essentially realized by the composition of the QRT (Query Response Time) of each Type A component. Figure 7.2 illustrates the behavior of a composite component in Phase 2 by a small example assuming a system with

no AEG. As an ordinary Type A composite component, a receives an **MSQ** from the parent and propagates the **MSQ** to its Type A subcomponents, i.e. b, c and e in Figure 7.2, where T_{c_i} tells whether c_i is of Type A or Type B. Yet two preliminary tasks (represented by the grey bar and black bar in Figure 7.2) must be fulfilled before its **MSQ** propagation. Component a must first check its current state and readiness for a mode switch (the required time is $a.sct$). If the current state of a indicates that a is not ready to switch mode, a will directly reply an **MSNOK** to the parent without further **MSQ** propagation. In Figure 7.2, a is ready to switch mode and then refers to its mode mapping to derive its Type A subcomponents (the required time is $a.pht$). When b, c and e receive the **MSQ**, they will respond after some time (denoted by linear gradient bars). In this example, they all reply with an **MSOK** and a finally sends an **MSOK** to its parent. Here the most interesting timing factor is the QRT of a, $a.qrt$, which on the one hand is used to determine the QRT of the parent of a, and on the other hand is dependent on the QRT of b, c and e. More precisely, $a.qrt$ depends on $a.sct$, $a.pht$ and the longest response from b, c and e:

$$\begin{aligned} a.qrt =& a.sct + a.pht + \max\{a.t_{msq} + b.qrt + a.t_{ok}, a.t_{msq} + c.qrt + a.t_{ok}, \\ & a.t_{msq} + e.qrt + a.t_{ok}\} \end{aligned} \tag{7.3}$$

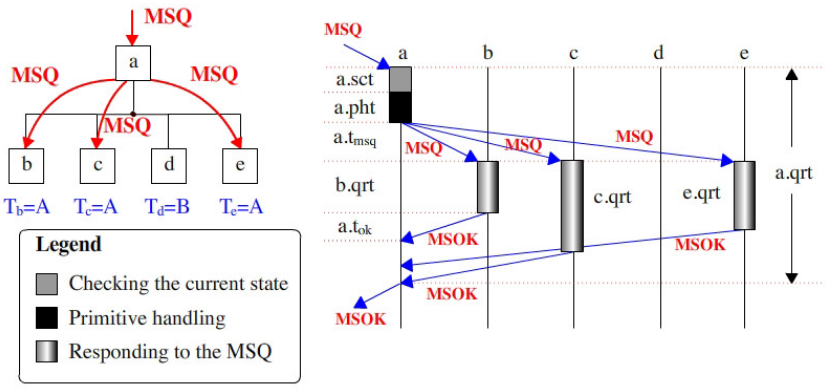

Figure 7.2: The mode switch timing analysis—Phase 2 (without AEG)

The example in Figure 7.2 assumes no atomic component execution. Figure 7.3 extends Figure 7.2 by defining b and c as an AEG G_1. As a result, the

MSQ propagation from a to b and c is delayed by AE_{G_1} (represented by the frameless grey bars), which is the atomic execution time of G_1. AE_{G_1} may vary depending on when a starts to check the DPS of G_1. AE_{G_1} is by default assumed to be the worst-case value here. In the worst case, the atomic execution of G_1 just starts when a starts to check its DPS. In the best case, there is no ongoing atomic execution in $G1$ when a starts to check its DPS and the **MSQ** propagation to b and c will not be delayed. Taking G_1 into account, $a.qrt$ can be calculated as

$$\begin{aligned} a.qrt =& a.sct + a.pht + \max\{AE_{G_1} + a.t_{msq} + b.qrt + a.t_{ok}, AE_{G_1} + \\ & a.t_{msq} + c.qrt + a.t_{ok}, a.t_{msq} + e.qrt + a.t_{ok}\} \end{aligned} \tag{7.4}$$

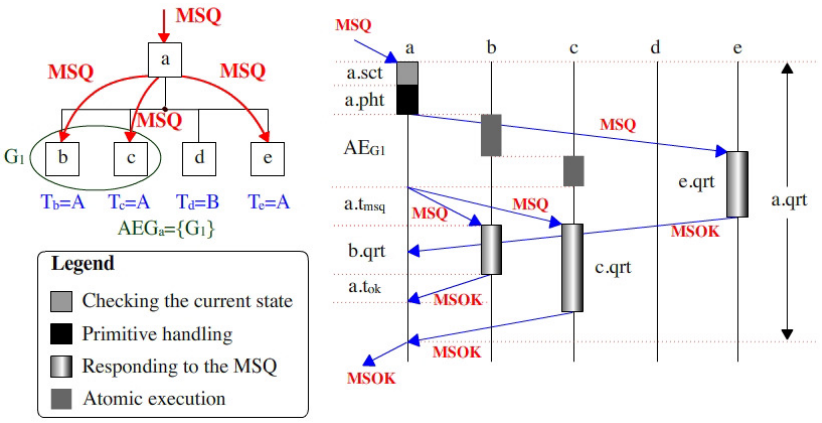

Figure 7.3: The mode switch timing analysis—Phase 2 (with AEG)

Combining both cases in figures 7.2 and 7.3, in general, $\forall c_i \in PC, T_{c_i} = A$, the QRT of c_i must be equal to its SCT as c_i does not propagate the **MSQ** further, thus

$$c_i.qrt = c_i.sct \tag{7.5}$$

and $\forall c_i \in CC, c_i \neq \beta, T_{c_i} = A, AEG_{c_i} = \varnothing$,

$$c_i.qrt = c_i.sct + c_i.pht + \max_{\substack{c_k \in SC_{c_i}, \\ T_{c_k}=A}} \{c_i.t_{msq} + c_k.qrt + c_i.t_{ok}\} \tag{7.6}$$

and $\forall c_i \in CC, c_i \neq \beta, T_{c_i} = A, AEG_{c_i} \neq \varnothing$,

$$\begin{aligned} c_i.qrt = & c_i.sct + c_i.pht + \max_{\substack{c_j \in H_{c_i}, \\ c_k \in G_k, \\ G_k \in AEG_{c_i}, \\ T_{c_j}=T_{c_k}=A}} \{c_i.t_{msq} + c_j.qrt + c_i.t_{ok}, \\ & AE_{G_k} + c_i.t_{msq} + c_k.qrt + c_i.t_{ok}\} \end{aligned} \tag{7.7}$$

By composing the QRT of each Type A component, the QRT of the subcomponents of the MSDM β can be obtained. β does not receive any **MSQ**, hence $\beta.sct = 0$. What β needs to do in Phase 2 is to approve an **MSR**, issue an **MSQ** to SC_β based on the mode mapping and then wait for the reply. Let T_2 denote the duration of Phase 2. If $AEG_\beta = \varnothing$, then

$$T_2 = \beta.pht + \max_{\substack{c_k \in SC_\beta, \\ T_{c_k}=A}} \{\beta.t_{msq} + c_k.qrt + \beta.t_{ok}\} \tag{7.8}$$

If $AEG_\beta \neq \varnothing$, then

$$\begin{aligned} T_2 = & \beta.pht + \max_{\substack{c_j \in H_\beta, \\ c_k \in G_k, \\ G_k \in AEG_\beta, \\ T_{c_j}=T_{c_k}=A}} \{\beta.t_{msq} + c_j.qrt + \beta.t_{ok}, \\ & AE_{G_k} + \beta.t_{msq} + c_k.qrt + \beta.t_{ok}\} \end{aligned} \tag{7.9}$$

7.1.3 The timing analysis in Phase 3—MSI propagation and mode switch

The timing analysis in Phase 3 resembles Phase 2. β issues an **MSI** whose propagation trace is exactly the same as the **MSQ**, thereby Type A composite components do not need to refer to the mode mapping again. Atomic component execution, which is handled in Phase 2, can be ignored in Phase 3 as well. The **MSC** collection is also akin to the **MSOK/MSNOK** collection in Phase 2. Figure 7.4 illustrates Phase 3 based on the same example in Figure 7.2. No SCT is considered in this phase because all Type A components have already checked their current states. Also, the propagation of **MSI** implies all Type A

components are ready to switch mode. The MSP protocol specifies that a component must stop running when its current state allows a mode switch after it receives an **MSQ**. This prevents the anomaly that the same component keeps running and enters a new state which prohibits a mode switch when it receives an **MSI**. Since no mode mapping is required to propagate an **MSI**, $a.pht = 0$ and that is why it is excluded in Figure 7.4. As a Type A component, a must reconfigure itself after propagating the **MSI** to b, c, and e. Therefore,

$$\begin{aligned} a.mst = \max\{&a.rct, a.t_{msi} + b.mst + a.t_{msc}, a.t_{msi} + c.mst + a.t_{msc}, \\ &a.t_{msi} + e.mst + a.t_{msc}\} \end{aligned} \tag{7.10}$$

Generalizing Equation (7.10), $\forall c_i \in PC, T_{c_i} = A$, $c_i.mst$ must be equal to its reconfiguration time $c_i.rct$ in that c_i does not propagate the **MSI** further, thus

$$c_i.mst = c_i.rct \tag{7.11}$$

and $\forall c_i \in CC, c_i \neq \beta, T_{c_i} = A$,

$$c_i.mst = \max_{\substack{c_k \in SC_{c_i}, \\ T_{c_k} = A}} \{c_i.rct, c_i.t_{msi} + c_k.mst + c_i.t_{msc}\} \tag{7.12}$$

By composing the MST of each Type A component, the MST of the subcomponents of the MSDM β can be obtained. Let T_3 denote the duration of Phase 3. Then the calculation of T_3 depends on two conditions: (1) $T_\beta = B$, then β will not reconfigure itself after **MSI** propagation; (2) $T_\beta = A$, then β must be the top component and will reconfigure itself after **MSI** propagation. Based on the two conditions, T_3 is calculated as follows:

If $T_\beta = B$,

$$T_3 = \max_{\substack{c_k \in SC_\beta, \\ T_{c_k} = A}} \{\beta.t_{msi} + c_k.mst + \beta.t_{msc}\} \tag{7.13}$$

Otherwise, if $T_\beta = A$,

$$T_3 = \max_{\substack{c_k \in SC_\beta, \\ T_{c_k} = A}} \{\beta.rct, \beta.t_{msi} + c_k.mst + \beta.t_{msc}\} \tag{7.14}$$

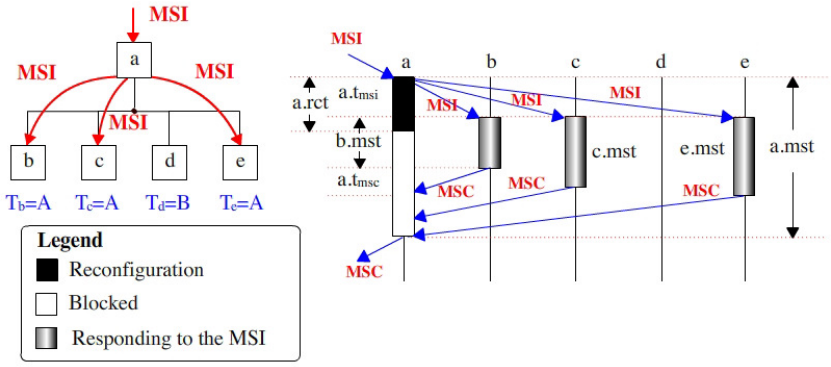

Figure 7.4: The mode switch timing analysis—Phase 3

7.1.4 The composition of three phases

Let T denote the complete mode switch time of a CBMMS. Due to the fact that the three identified phases are absolutely sequential without mutual interference, T is purely the summation of the duration of all phases:

$$T = T_1 + T_2 + T_3 \tag{7.15}$$

Our mode switch timing analysis for a CBMMS based on the MSL can be illustrated by the example introduced in Figure 3.4 together with the illustration of its component connections in Figure 5.3. The complete mode switch process of this system has been demonstrated in Figure 5.4. Additionally, a number of timing factors with values are assigned to this system, given in Table 7.1, including MSDT, PHT, SCT and RCT. The mode switch timing analysis based on Figure 5.4 and Table 7.1 is depicted in Figure 7.5, in which the transmission time of each primitive is constantly 1 for the sake of simplicity. Moreover, the atomic execution time of the AEG *b*, i.e. AE_b, is 8, We assume some data just enters *b* when *a* starts to check the DPS of *b*. The data is first processed by *e* for 4 units and then processed by *f* for another 4 units. The data transmission time from *e* to *f* is assumed to be 0. The three phases are lucidly highlighted on the rightmost part of Figure 7.5, which shows that the complete mode switch time is 40.

Next we shall calculate the complete mode switch time based on the given values of these timing factors. Since $\alpha = h$ and $\beta = a$, $C_M = \{c\}$. According

Component	MSDT	PHT	SCT	RCT
a	–	3	–	–
b	–	2	2	6
c	–	2	3	7
e	–	–	2	3
g	–	–	2	4
h	3	2	1	5
m	–	–	3	6

Table 7.1: Timing factors assigned to the system in Figure 5.4

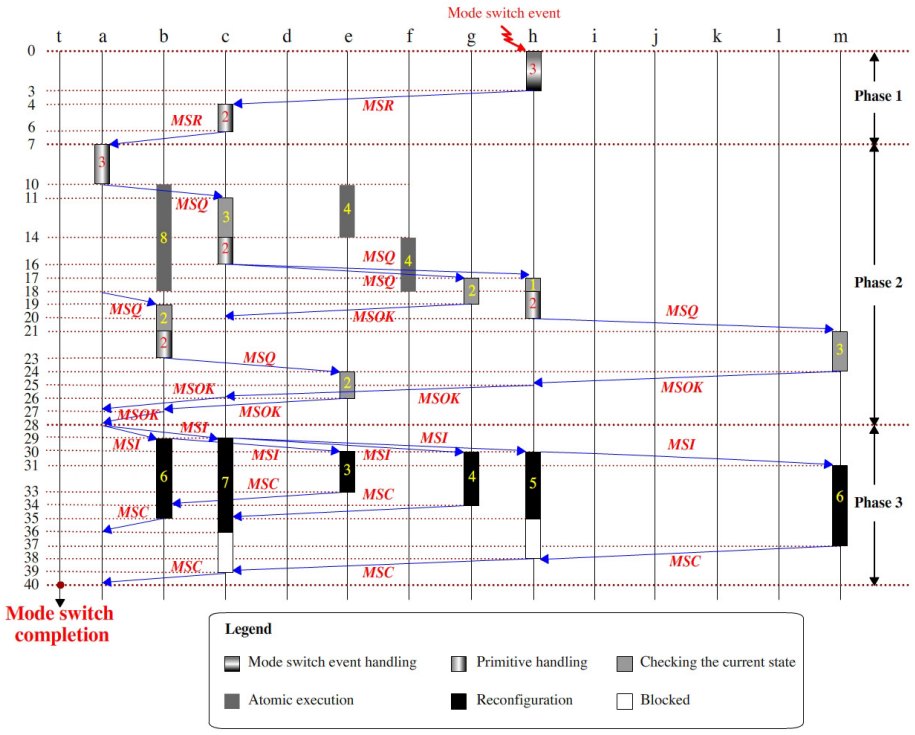

Figure 7.5: The timing analysis of a complete mode switch

to Equation (7.2),

$$T_1 = h.msdt + (c.t_{msr} + c.pht) + a.t_{msr} = 3 + (1 + 2) + 1 = 7 \quad (7.16)$$

In order to derive T_2, the QRT of Type A primitive components can be first calculated based on Equation (7.5):

$$e.qrt = e.sct = 2 \quad (7.17)$$

$$g.qrt = g.sct = 2 \quad (7.18)$$

$$m.qrt = m.sct = 3 \quad (7.19)$$

Once $e.qrt$ is known, guided by Equation (7.6), the QRT of *b* can also be worked out:

$$b.qrt = b.sct + b.pht + (b.t_{msq} + e.qrt + b.t_{ok}) = 2 + 2 + (1 + 2 + 1) = 8 \quad (7.20)$$

Likewise,

$$h.qrt = h.sct + h.pht + (h.t_{msq} + m.qrt + h.t_{ok}) = 1 + 2 + (1 + 3 + 1) = 8 \quad (7.21)$$

Using Equation (7.6), $g.qrt$ and $h.qrt$,

$$\begin{aligned} c.qrt &= c.sct + c.pht + \max\{c.t_{msq} + g.qrt + c.t_{ok}, c.t_{msq} + h.qrt + c.t_{ok}\} \\ &= 3 + 2 + \max\{1 + 2 + 1, 1 + 8 + 1\} = 15 \end{aligned} \quad (7.22)$$

By composing $b.qrt$ and $c.qrt$, guided by Equation (7.9), T_2 can be calculated. Due to the atomic execution of the AEG *b*, the **`MSQ`** propagation from *a* to *b* is delayed by $AE_b = 8$. Therefore,

$$\begin{aligned} T_2 &= a.pht + \max\{AE_b + a.t_{msq} + b.qrt + a.t_{ok}, a.t_{msq} + c.qrt + a.t_{ok}\} \\ &= 3 + \max\{8 + 1 + 8 + 1, 1 + 15 + 1\} = 21 \end{aligned} \quad (7.23)$$

The calculation of T_3 starts with the MST of primitive Type A components by using Equation (7.11):

$$e.mst = e.rct = 3 \tag{7.24}$$

$$g.mst = g.rct = 4 \tag{7.25}$$

$$m.mst = m.rct = 6 \tag{7.26}$$

Then the MST of the parent of *e* and *m* can be computed by Equation (7.12):

$$b.mst = \max\{b.rct, b.t_{msi} + e.mst + b.t_{msc}\} = \max\{6, 1+3+1\} = 6 \tag{7.27}$$

$$h.mst = \max\{h.rct, h.t_{msi} + m.mst + h.t_{msc}\} = \max\{5, 1+6+1\} = 8 \tag{7.28}$$

After using Equation (7.12) one more time,

$$\begin{aligned} c.mst &= \max\{c.rct, c.t_{msi} + g.mst + h.t_{msc}, c.t_{msi} + h.mst + c.t_{msc}\} \\ &= \max\{7, 1+4+1, 1+8+1\} = 10 \end{aligned} \tag{7.29}$$

After that, T_3 can be computed by Equation (7.13):

$$\begin{aligned} T_3 &= \max\{a.t_{msi} + b.mst + a.t_{msc}, a.t_{msi} + c.mst + a.t_{msc}\} \\ &= \max\{1+6+1, 1+10+1\} = 12 \end{aligned} \tag{7.30}$$

Finally, applying Equation (7.15), the complete mode switch time T is:

$$T = T_1 + T_2 + T_3 = 7 + 21 + 12 = 40 \tag{7.31}$$

This result is exactly the same as the complete mode switch time provided in Figure 7.5.

As a final remark, our mode switch timing analysis does not assume the global knowledge of all timing factors, thanks to the compositional feature of the QRT and MST of each component, which can be explicitly exposed as EFPs during the composition. However, for a given mode switch scenario, the MSS and MSDM must be known and the set of vertically intermediate components C_M between the MSS and the MSDM should also be known for deriving T_1.

7.2 Deriving the worst-case atomic execution time of an AEG

From the mode switch timing analysis of the example in the last section, one may notice that AE_b, i.e. the worst-case atomic execution time of the AEG *b*, is the only one timing factor that is related to component connections and the functional behavior of a system which the other timing factors have no direct relation to. Since *b* has only two subcomponents which are sequentially connected, AE_b can be easily computed by summing the worst-case execution time of *e* and *f*. However, for a general AEG G_k, there is a need for a systematic approach to the derivation of AE_{G_k}, which can be affected by many contributing factors such as the component hierarchy, component connections, and data flow. In this section, we provide a model-checking approach as the solution to calculate the worst-case atomic execution time of an AEG.

7.2.1 The AEG model

The modeling of an AEG requires a clear specification of the AEG execution semantics. Our target is a pipe-and-filter CBMMS, whose execution semantics is consistent with the pure data flow component described in [39]. Each AEG of such a system can also be considered as a smaller CBMMS with the same execution semantics.

In Section 5, it has been mentioned that a component has a number of input and output ports in a pipe-and-filter CBMMS. The input data is first read by a component, which will process the data and then produce the output data. Furthermore,

- We assume that primitive components have data going through all its input and output ports, i.e. input data has to be available at all input ports before processing can start and output data must be sent via all output ports. Whenever a primitive component receives new data at an input port, the data is first queued in a corresponding input buffer. While a primitive component is processing data, new arriving data must wait in its input buffers and cannot be processed until the component completes its current data processing.

- As opposed to a primitive component, a composite component does not buffer its input data. Whenever it receives new data, it will simply propagate the data to its subcomponents. Similarly, whenever it receives out-

put data from its subcomponents, it will immediately forward the data via its corresponding output port(s).

We also make the following restrictions (assumptions) on an AEG composed by components that follow the execution semantics above:

- An AEG should have only one input port. The reason is that the parent of an AEG is responsible for freezing its input before checking its DPS. If an AEG has multiple input ports, it may receive data from different input ports at different times. Consequently, input freezing becomes a complex issue, which is out of the scope of this thesis.
- There is no cyclic connection, i.e. feedback loop, in an AEG. Feedback loops complicate the data flow within an AEG, thus making it more challenging to derive AE_{G_k}.
- 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 can be considered negligible (or included in the component execution time).

7.2.2 An illustrative example of AEG

Our model-checking approach can be demonstrated by an illustrative example. Figure 7.6 depicts an AEG G_k consisting of primitive components *a-f*. The ports of G_k and its composing primitive components are marked in red. Composite components or deactivated components in the AEG are not shown because they do not affect AE_{G_k}. We assume that the data processing time of each component $c_i \in PC$ in G_k is bounded by a timing interval $[C_{c_i}^{min}, C_{c_i}^{max}]$, and that the incoming data rate of the AEG is within the interval $[R_{min}, R_{max}]$. To ensure that AE_{G_k} is bounded and that our calculations terminate, we will enforce a maximum number of data elements in G_k. Depending on the incoming 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 (further details in Section 7.2.4), but could also be a mechanism in the real system. The timing factors and their values in G_k are as follows:

- Incoming data rate $R = [7, 8]$.
- Data processing time C of components *a-f*: $C_a = [4, 5]$, $C_b = [7, 8]$, $C_c = [6, 7]$, $C_d = [5, 6]$, $C_e = 5$, $C_f = [7, 8]$.

- Maximum number of data elements in G_k $N = 5$.

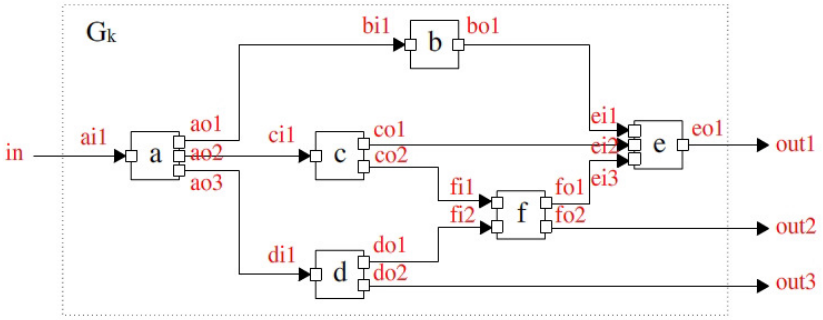

Figure 7.6: An illustrative example of AEG

To calculate AE_{G_k}, we propose a model-checking approach based on UPPAAL [38]. UPPAAL is a model-checking tool which is widely used to model, simulate and verify real-time systems. In particular, since version UPPAAL 4.1.3, there is a "sup" operator able to find out the maximal value of a variable or clock. If an AEG G_k is properly modeled by UPPAAL, we can use "sup" to obtain AE_{G_k}. In this way, the focus is moved from AE_{G_k} derivation to the UPPAAL modeling of G_k. Please note that we do not consider any scheduling effect on AE_{G_k} which will be included in our future work.

7.2.3 UPPAAL modeling

By using UPPAAL, we first model the execution and mode switch behavior of G_k and then derive AE_{G_k} via its property verification. No matter how complex an AEG G_k is, we can divide its UPPAAL model into four parts:

1. *Data source*: generates data at a flexible rate.
2. *AEG*: receives data from *Data source*, processes it 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 G_k is not involved in any mode switch, *Data source* is turned on again when n decreases. When the AEG receives an **`MSQ`**, *Data source* will also be turned off as the parent of G_k freezes its input. AE_{G_k} is equivalent 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 component connection definitions referred to by *Data forwarder* need to be updated.

4. *Primitive components*: modeled by a UPPAAL template for each of them. Though the number of components can be arbitrary, a parameterized generic model applying to all components can be used.

The full UPPAAL model of the example in Figure 7.6 is available in [24]. Here we will only focus on modeling the AEG G_k itself and its enclosed primitive components.

Figure 7.7 models the AEG G_k in Figure 7.6. When a new data enters G_k, G_k will propagate this data to its enclosed components through the *dataOut!* channel. *dataCounter* counts the number of data within G_k. In the function *shutDS()*, G_k compares *dataCounter* with the threshold N. If the threshold is reached, the data source will be turned off in *shutDS()*. When *dataCounter* decreases and no **MSQ** arrives, the data source is turned on in function *dataNControl()*. The model has two major states: **Waiting** and **Processing**. In state **Waiting**, no data is within G_k. The **MSQ** from the parent of G_k will not be delayed because there is no ongoing atomic execution in G_k. In state **Processing**, at least one data is in G_k. The **MSQ** is modeled by the channel *MSQ?*. Clock z plays a significant role in that the maximal possible value of z in its State **Processing** corresponds to AE_{G_k}.

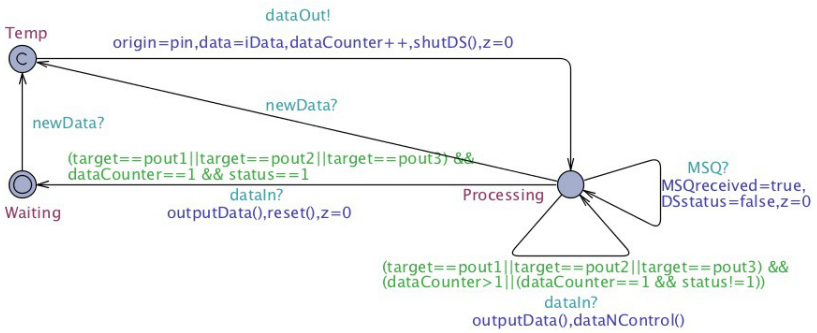

Figure 7.7: UPPAAL model: AEG G_k

The primitive components in G_k can be categorized into four types: (1) a component with single input and output; (2) a component with single input and multiple outputs; (3) a component with multiple inputs and single output; (4) a component with multiple inputs and outputs. Figure 7.8 illustrates the model of Component *f*, which has multiple inputs and outputs. Actually the model of f is generic in the sense that all the other components in G_k from Figure 7.6 can be modeled in the same way. When not processing any data, f is in state **nonProcessing**. When processing data, it is in state **Processing**. The invariant $x <= 8$ and guard $x > 7$ define the interval of its data processing time, i.e. $C_f = [7, 8]$. Component f receives data through the channel *dataIn?* and sends output data through the channel *dataOut!*. Component f recognizes new data by the guard $target == fi1 \mid\mid target == fi2$ where $fi1$ and $fi2$ are its input ports. When all input buffers are non-empty, the boolean variable *readyToProcess* is set to true and f will switch to location **Processing** by the urgent channel *Go!*. Data is processed by the function *processData()*, thus representing mode-specific behavior of a primitive component. After processing the data, f immediately sends its output data through all its output ports. This is modeled by the sequential and atomic output data generation from its output ports. The two committed states **Temp1** and **Temp2** guarantee atomicity. *outputCounter* records how many output ports have sent out the data. When the output data is sent through all its output ports, f goes back to state **nonProcessing** and checks its buffer status again.

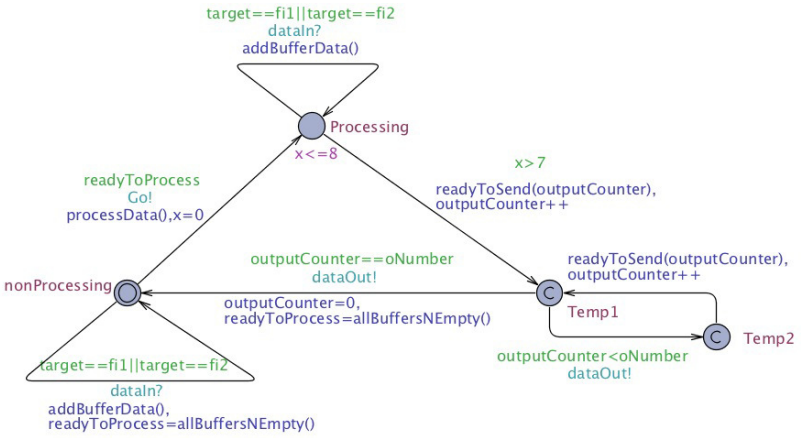

Figure 7.8: UPPAAL model: Component *f*

Component f belongs to Type (4) in terms of its ports. Similarly, components of the other three types can be modeled by following the same pattern. Figures 7.9-7.11 illustrate the models of *a*(Type (2)), *b*(Type (1)) and *e*(Type (3)), respectively. The model structure of all these components are the same, despite the variation of some parameters. If a component has only one output port, the model can be simplified by removing state **Temp2** and *outputCounter*.

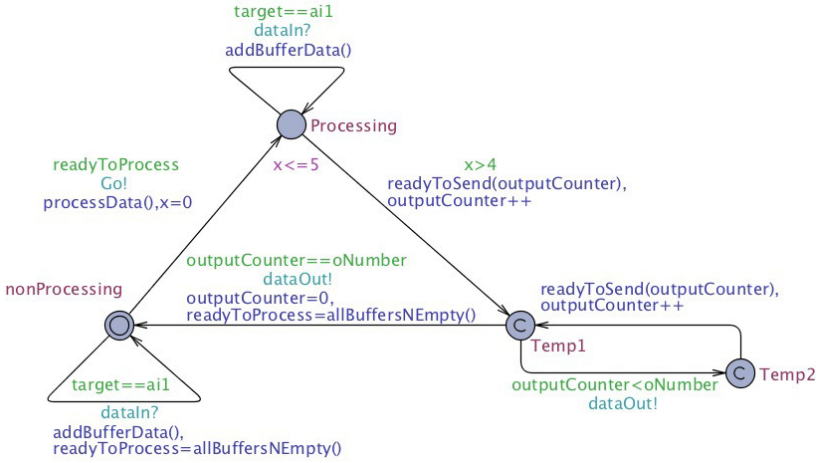

Figure 7.9: UPPAAL model: Component *a*

7.2.4 Verification and evaluation

Some interesting results including AE_{G_k} can be obtained by verifying the following properties of the UPPAAL model:

- *A[] not deadlock*: no deadlock will occur in the model.
- *sup{AEG.Processing}: AEG.z*: returns the maximal value of the clock z of *AEG* in state **Processing**. This equals AE_{G_k}.
- *E<> AEG.Processing && AEG.z*==AE_{G_k}: there is a scenario in which the clock z reaches AE_{G_k} when *AEG* is in state **Processing**. Once AE_{G_k} is derived, this property searches the worst-case scenario, and using the "Diagnostic Trace" function of UPPAAL, the worst-case scenario can be displayed as an execution trace.

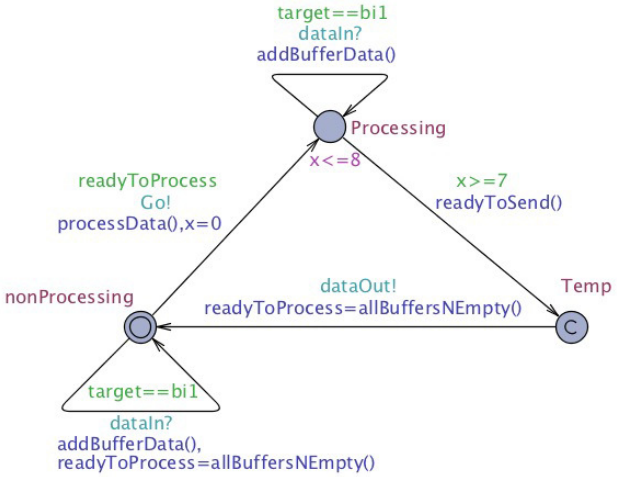

Figure 7.10: UPPAAL model: Component *b*

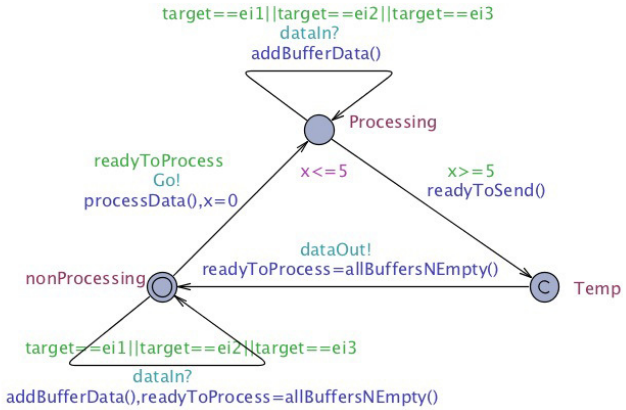

Figure 7.11: UPPAAL model: Component *e*

- *sup: dataCounter*: returns the maximal number of data items that can be simultaneously processed in G_k. If N is only a modeling artifact, then for the validity of the calculated AE_{G_k}, 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.

- *sup: Component.bufferN[Index]*: returns the maximal number of elements in one buffer of a component.

Using UPPAAL, all properties have been verified. In addition, the verification results show that for $R = [7, 8]$, the maximal number (i.e. n) of data items in G_k is 5, meaning that the threshold N can be reached. In the worst-case, $AE_{G_k} = 40$.

Moreover, the verification results show that R and C_{c_i} ($c_i \in PC$ is activated in G_k) have substantial influence on the property verification time. The differences are related to variations in the number and length of executions leading up to the worst-case scenario. We repeated the verification of the same set of properties for different data rates. The most important results are summarized in Table 7.2[1].

An interesting side effect of our modeling is that we can use the last property above to obtain the maximal buffer usage (i.e. required buffer sizes) for the component input buffers. These values are for the considered data rates presented in Table 7.3. For instance, the maximal usage of the buffer associated with port $ei2$ in Figure 7.6 is 1 when $R = [10, 12]$, 2 when $R = [8, 10]$, and 4 when $R = [6, 8]$.

7.2.5 Generalization

Apart from R, verification time also depends on the number of components in the AEG, the number of connections and output ports of an AEG, the threshold N and the data processing time of each component. Regardless of the verification time, the way that we model the system does not change. Although we have only demonstrated how to derive AE_{G_k} for a simple example, our UPPAAL models are generic. We conjecture that for any AEG that is in line with our system and component models, we are able to make transformation rules, based on which the corresponding UPPAAL models can be automatically generated. Since the UPPAAL verification is based on generating and exploring the global state space, it is subject to state explosion while modeling a too complex system. However, we do not expect this to be a limitation in practice, since the complexity of an AEG typically is rather low.

[1] Verification is performed on MacBook Pro, with 2.66GHz Intel Core 2 Duo CPU and 8GB 1067 MHz DDR3 memory.

Property/Value	$R = [6, 8]$	$R = [7, 8]$	$R = [8, 10]$	$R = [10, 12]$
No deadlock	28.64s	5.617s	0.139s	0.108s
Maximal n	5	5	4	3
Deriving AE	45.667s	4.36s	0.1s	0.069s
AE	40	40	25	25
Worst-case scenario	36.716s	4.576s	0.013s	0.016s

Table 7.2: Property verification results for different data rates

Buffer Index	$R = [6, 8]$	$R = [7, 8]$	$R = [8, 10]$	$R = [10, 12]$
$ai1$	1	1	1	1
$bi1$	3	3	1	1
$ci1$	2	1	1	1
$di1$	1	1	1	1
$ei1$	3	3	2	1
$ei2$	4	4	2	1
$ei3$	3	3	1	1
$fi1$	3	3	1	1
$fi2$	3	3	1	1

Table 7.3: Maximal buffer usage for different data rates

7.3 Summary

In this chapter, the mode switch time of a CBMMS is analyzed based on MSL. The entire mode switch process of a system is divided into three non-overlapping phases: (1) **`MSR`** propagation; (2) **`MSQ`** propagation; and (3) **`MSI`** propagation and mode switch. We provide the calculation of the time spent in each phase and the total mode switch time is the summation of all three phases. Furthermore, a model-checking approach is presented to derive the worst-case atomic execution time of an AEG, which may prolong the mode switch time. This model-checking approach is explained based on an example modeled in the model-checker UPPAAL. We also demonstrate how the worst-case atomic execution time of an AEG can be derived from the property verification in UPPAAL.

Chapter 8

Case study–An Adaptive Cruise Control system

In this chapter, the main principles of our MSL is applied to a case study: an Adaptive Cruise Control (ACC) system [2]. An ACC system is typically used as a subsystem of a car to automatically maintain both the desired speed and the safe distance from 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.

8.1 The system description

An ACC system can be developed as a CBMMS. Figure 8.1 shows the component hierarchy of the ACC system:

- Speed Limit (SL): deriving the desired speed based on the speed-limit regulations.
- Object Recognition (OR): detecting the presence of a preceding car and calculating the relative speed between two cars.
- ACC Controller (ACC): controlling the speed by adjusting the throttle lever in order to maintain both the speed and distance, as the core of the ACC system. It consists of two subcomponents: Distance Controller and Speed Controller.
- Brake Assist (BA): Assisting the driver to brake in extreme situations.
- HMI Output (HMI): Displaying information related to the ACC system to the driver.
- Distance Controller (DC): providing the distance information to the Speed Controller.
- Speed Controller (SC): controlling the vehicle speed.

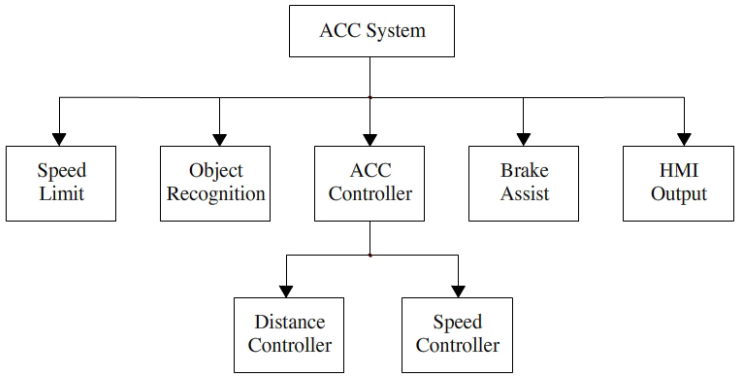

Figure 8.1: The component hierarchy of an ACC system

Furthermore, the ACC system can run in three different modes:

- *CC* mode: the traditional cruise control mode, in which the ACC system only maintains the desired speed regardless of the preceding vehicle. In

this mode, components OR and BA are deactivated and the ACC Controller equals a CC Controller, i.e. it maintains the desired speed regardless of obstacles and speed limits. The connections between the subcomponents of the ACC system in this mode are depicted in Figure 8.2. The ACC system can be manually controlled via two input ports: "Pedals" and "CC/ACC switch". The driver can regulate the vehicle speed through "Pedals", and manually switch its running mode between CC mode and ACC mode through "CC/ACC switch".

- *ACC* mode: the normal adaptive cruise control mode, in which the ACC system maintains both the desired speed and distance. Compared with CC mode, this mode activates Component OR, however, Component BA is still deactivated. The connections between the subcomponents of the ACC system in this mode are depicted in Figure 8.3.

- *EMERGENCY* mode: a mode in which Component BA is activated to help the driver with the braking process. The connections between the subcomponents of the ACC system in this mode are depicted in Figure 8.4.

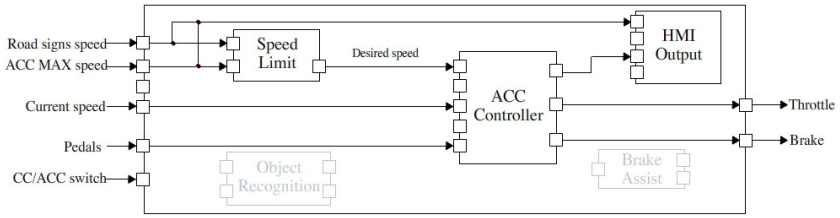

Figure 8.2: The ACC system in *CC* mode

The ACC Controller supports the same modes as the ACC system: *Cc*, *Acc* and *Emergency* (lowercase letters are used to distinguish them from the modes supported by the ACC system). The component connections within the ACC Controller are depicted in Figure 8.5 where the port names of ACC and its subcomponents SC and DC are marked in red. Component DC is deactivated when the ACC Controller is in *Cc* mode, and activated when the ACC Controller is in *Acc* or *Emergency* mode. As is indicated by white, black and grey colors, Component SC has three different behaviors which correspond to its supported modes *Basic*, *Advance* and *Brake*.

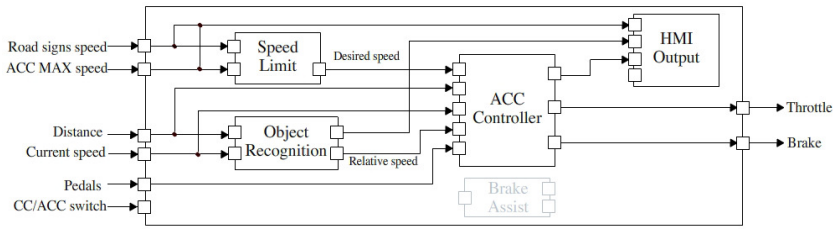

Figure 8.3: The ACC system in *ACC* mode

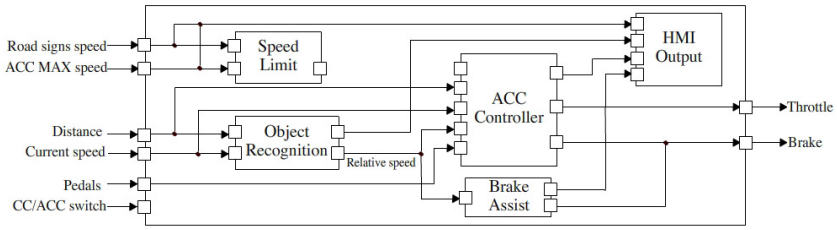

Figure 8.4: The ACC system in *EMERGENCY* mode

Let α, β and γ denote the mode-specific behaviors of SC in *Basic*, *Advance* and *Brake*, respectively. Suppose SC has a mode-dependent EFP: WCET, denoted as $wcet$ such that SC.$wcet = 50$ when SC is in *Basic*; SC.$wcet = 75$ when SC is in *Advance*; and SC.$wcet = 100$ when SC is in *Brake*. Also suppose SC has a mode-independent EFP: memory consumption, denoted as $mem = 40$ in all modes. As a primitive component, SC can be formally defined by the tuple,

$$<\text{SC}.IP, \text{SC}.OP, \text{SC}.p^{MSX}, \text{SC}.B, \text{SC}.MIP, \text{SC}.MDEFP, \text{SC}.m,$$
$$\text{SC}.MB, \text{SC}.AIP, \text{SC}.AOP, \text{SC}.MS, \text{SC}.MP, \text{SC}.MSRM >$$

where SC.*MSRM* can be implemented by the algorithms described in Chapter 6 and

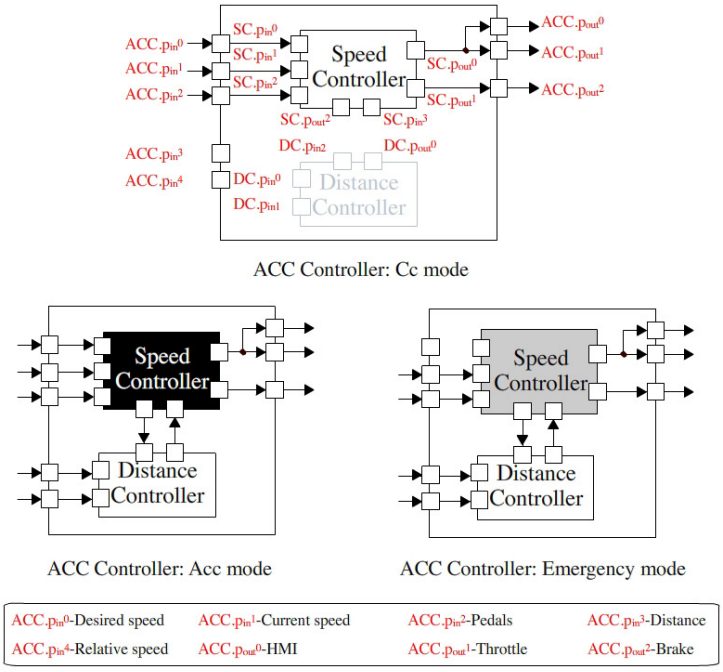

Figure 8.5: ACC Controller in different modes

$$
\begin{aligned}
\text{SC}.IP &= \{\text{SC}.p^0_{in}, \text{SC}.p^1_{in}, \text{SC}.p^2_{in}, \text{SC}.p^3_{in}\}\\
\text{SC}.OP &= \{\text{SC}.p^0_{out}, \text{SC}.p^1_{out}, \text{SC}.p^2_{out}\}\\
\text{SC}.B &= \{\alpha, \beta, \gamma\}\\
\text{SC}.MDEFP &= \{mem\}\\
\text{SC}.m &= \text{Basic}\\
\text{SC}.MB &= \{\text{Basic} \rightarrow \alpha, \text{Advance} \rightarrow \beta, \text{Brake} \rightarrow \gamma\}\\
\text{SC}.AIP &= \{\text{Basic} \rightarrow \{\text{SC}.p^0_{in}, \text{SC}.p^1_{in}, \text{SC}.p^2_{in}\},\\
&\quad \text{Advance} \rightarrow \{\text{SC}.p^0_{in}, \text{SC}.p^1_{in}, \text{SC}.p^2_{in}, \text{SC}.p^3_{in}\},\\
&\quad \text{Brake} \rightarrow \{\text{SC}.p^1_{in}, \text{SC}.p^2_{in}, \text{SC}.p^3_{in}\}\}\\
\text{SC}.AOP &= \{\text{Basic} \rightarrow \{\text{SC}.p^0_{out}, \text{SC}.p^1_{out}\},\\
&\quad \text{Advance} \rightarrow \{\text{SC}.p^0_{out}, \text{SC}.p^1_{out}, \text{SC}.p^2_{out}\},\\
&\quad \text{Brake} \rightarrow \{\text{SC}.p^0_{out}, \text{SC}.p^1_{out}, \text{SC}.p^2_{out}\}\}\\
\text{SC}.MS &= \{\text{Basic} \rightarrow Activated, \text{Advance} \rightarrow Activated,\\
&\quad \text{Brake} \rightarrow Activated\}\\
\text{SC}.MP &= \{(\text{Basic}, wcet) \rightarrow 50, (\text{Advance}, wcet) \rightarrow 75,\\
&\quad (\text{Brake}, wcet) \rightarrow 100\}
\end{aligned}
$$

In addition, SC.*MIP* is another tuple,

$$< \text{SC}.M, \text{SC}.m^0, \text{SC}.MIEFP >$$

where

$$\begin{array}{lcl} \text{SC}.M & = & \{\text{Basic}, \text{Advance}, \text{Brake}\} \\ \text{SC}.m^0 & = & \text{Basic} \\ \text{SC}.MIEFP & = & \{mem = 40\} \end{array}$$

Furthermore, ACC has a mode-dependent EFP: CPU consumption, denoted as cpu such that ACC.$cpu = 60$ when ACC is in *Cc*; ACC.$cpu = 120$ when ACC is in *Acc*; and ACC.$cpu = 150$ when ACC is in *Emergency*. ACC also has a mode-independent EFP: activation period, denoted as T such that ACC.$T = 200$ in all modes. As a composite component, ACC can be formally defined by the tuple,

$$\begin{array}{l} <\text{ACC}.IP, \text{ACC}.OP, \text{ACC}.p^{MSX}, \text{ACC}.p_{in}^{MSX}, \text{ACC}.SC, \text{ACC}.Con, \text{ACC}.MIP, \\ \quad \text{ACC}.MDEFP, \text{ACC}.m, \text{ACC}.AIP, \text{ACC}.AOP, \text{ACC}.MS, \text{ACC}.ASC, \text{ACC}.DSC, \\ \quad \text{ACC}.ACon, \text{ACC}.MP, \text{ACC}.MSRM > \end{array}$$

where ACC.*MSRM* can be implemented by the algorithms described in Chapter 6 and

$$\begin{array}{lcl} \text{ACC}.IP & = & \{\text{ACC}.p_{in}^0, \text{ACC}.p_{in}^1, \text{ACC}.p_{in}^2, \text{ACC}.p_{in}^3, \text{ACC}.p_{in}^4\} \\ \text{ACC}.OP & = & \{\text{ACC}.p_{out}^0, \text{ACC}.p_{out}^1, \text{ACC}.p_{out}^2\} \\ \text{ACC}.SC & = & \{\text{SC}, \text{DC}\} \\ \text{ACC}.Con & = & \{(\text{ACC}.p_{in}^0, \text{SC}.p_{in}^0), (\text{ACC}.p_{in}^1, \text{SC}.p_{in}^1), \\ & & (\text{ACC}.p_{in}^2, \text{SC}.p_{in}^2), (\text{SC}.p_{out}^0, \text{ACC}.p_{out}^0), \\ & & (\text{SC}.p_{out}^0, \text{ACC}.p_{out}^1), (\text{SC}.p_{out}^1, \text{ACC}.p_{out}^2), \\ & & (\text{SC}.p_{out}^2, \text{DC}.p_{in}^2), (\text{DC}.p_{out}^0, \text{SC}.p_{in}^3), \\ & & (\text{ACC}.p_{in}^3, \text{DC}.p_{in}^0), (\text{ACC}.p_{in}^4, \text{DC}.p_{in}^1)\} \\ \text{ACC}.MDEFP & = & \{cpu\} \\ \text{ACC}.m & = & \text{Cc} \\ \text{ACC}.AIP & = & \{\text{Cc} \rightarrow \{\text{ACC}.p_{in}^0, ACC.p_{in}^1, \text{ACC}.p_{in}^2\}, \\ & & \text{Acc} \rightarrow \{\text{ACC}.p_{in}^0, \text{ACC}.p_{in}^1, \text{ACC}.p_{in}^2, \text{ACC}.p_{in}^3, \\ & & \text{ACC}.p_{in}^4\}, \text{Emergency} \rightarrow \{\text{ACC}.p_{in}^1, \text{ACC}.p_{in}^2, \\ & & \text{ACC}.p_{in}^3, \text{ACC}.p_{in}^4\}\} \end{array}$$

$$
\begin{array}{lcl}
\text{ACC.}\mathit{AOP} & = & \{\text{Cc} \rightarrow \{\text{ACC}.p^0_{out}, \text{ACC}.p^1_{out}, \text{ACC}.p^2_{out}\}, \\
 & & \text{Acc} \rightarrow \{\text{ACC}.p^0_{out}, \text{ACC}.p^1_{out}, \text{ACC}.p^2_{out}\}, \\
 & & \text{Emergency} \rightarrow \{\text{ACC}.p^0_{out}, \text{ACC}.p^1_{out}, \text{ACC}.p^2_{out}\}\} \\
\text{ACC.}\mathit{MS} & = & \{\text{Cc} \rightarrow \mathit{Activated}, \text{Acc} \rightarrow \mathit{Activated}, \\
 & & \text{Emergency} \rightarrow \mathit{Activated}\} \\
\text{ACC.}\mathit{ASC} & = & \{\text{Cc} \rightarrow \{\text{SC}\}, \text{Acc} \rightarrow \{\text{SC}, \text{DC}\}, \text{Emergency} \rightarrow \{\text{SC}, \text{DC}\}\} \\
\text{ACC.}\mathit{DSC} & = & \{\text{Cc} \rightarrow \{\text{DC}\}, \text{Acc} \rightarrow \varnothing, \text{Emergency} \rightarrow \varnothing\} \\
\text{ACC.}\mathit{ACon} & = & \{\text{Cc} \rightarrow \{(\text{ACC}.p^0_{in}, \text{SC}.p^0_{in}), (\text{ACC}.p^1_{in}, \text{SC}.p^1_{in}), \\
 & & (\text{ACC}.p^2_{in}, \text{SC}.p^2_{in}), (\text{SC}.p^0_{out}, \text{ACC}.p^0_{out}), \\
 & & (\text{SC}.p^0_{out}, \text{ACC}.p^1_{out}), (\text{SC}.p^1_{out}, \text{ACC}.p^2_{out})\}, \\
 & & \text{Acc} \rightarrow \{(\text{ACC}.p^0_{in}, \text{SC}.p^0_{in}), (\text{ACC}.p^1_{in}, \text{SC}.p^1_{in}), \\
 & & (\text{ACC}.p^2_{in}, \text{SC}.p^2_{in}), (\text{SC}.p^0_{out}, \text{ACC}.p^0_{out}), \\
 & & (\text{SC}.p^0_{out}, \text{ACC}.p^1_{out}), (\text{SC}.p^1_{out}, \text{ACC}.p^2_{out}), \\
 & & (\text{SC}.p^2_{out}, \text{DC}.p^2_{in}), (\text{DC}.p^0_{out}, \text{SC}.p^3_{in}), \\
 & & (\text{ACC}.p^3_{in}, \text{DC}.p^0_{in}), (\text{ACC}.p^4_{in}, \text{DC}.p^1_{in})\}, \\
 & & \text{Emergency} \rightarrow \{(\text{ACC}.p^1_{in}, \text{SC}.p^1_{in}), (\text{ACC}.p^2_{in}, \text{SC}.p^2_{in}), \\
 & & (\text{SC}.p^0_{out}, \text{ACC}.p^0_{out}), (\text{SC}.p^0_{out}, \text{ACC}.p^1_{out}), \\
 & & (\text{SC}.p^1_{out}, \text{ACC}.p^2_{out}), (\text{SC}.p^2_{out}, \text{DC}.p^2_{in}), \\
 & & (\text{DC}.p^0_{out}, \text{SC}.p^3_{in}), (\text{ACC}.p^3_{in}, \text{DC}.p^0_{in}), \\
 & & (\text{ACC}.p^4_{in}, \text{DC}.p^1_{in})\}\} \\
\text{ACC.}\mathit{MP} & = & \{(\text{Cc}, \mathit{cpu}) \rightarrow 60, (\text{Acc}, \mathit{cpu}) \rightarrow 120, \\
 & & (\text{Emergency}, \mathit{cpu}) \rightarrow 150\}
\end{array}
$$

In addition, ACC.*MIP* is another tuple,

$$< \text{ACC}.M, \text{ACC}.m^0, \text{ACC}.M_{\text{SC}}, \text{ACC}.m^0_{\text{SC}}, \text{ACC}.m_{\text{SC}}, \text{ACC}.\mathit{MM}, \text{ACC}.\mathit{MIEFP} >$$

where ACC.*MM* will be presented as a set of MMAs in the next section. Figure 8.5 indicates that DC is running in the same mode when ACC is in *Acc* and *Emergency* while it is deactivated otherwise. Let *ON_DC* be the mode supported by DC, and then

$$
\begin{array}{lcl}
\text{ACC}.M & = & \{\text{Cc}, \text{Acc}, \text{Emergency}\} \\
\text{ACC}.m^0 & = & \text{Cc} \\
\text{ACC}.M_{\text{SC}} & = & \{\text{SC} \rightarrow \{\text{Basic}, \text{Advance}, \text{Brake}\}, \text{DC} \rightarrow \{\text{ON\_DC}\}\} \\
\text{ACC}.m^0_{\text{SC}} & = & \{\text{SC} \rightarrow \text{Basic}, e \rightarrow \text{Deactivated}\} \\
\text{ACC}.m_{\text{SC}} & = & \{\text{SC} \rightarrow \text{Basic}, e \rightarrow \text{Deactivated}\} \\
\text{ACC}.\mathit{MIEFP} & = & \{T = 200\}
\end{array}
$$

8.2 Mode mapping for the ACC system

Since the supported modes of the components in Figure 8.1 are different, mode mapping is required for the composition of these components. Table 8.1 and 8.2 present the mode mapping tables for the ACC system (also called Top here) and the ACC Controller (ACC). It can be noticed that SL, OR, HMI, BA and DC are all single-mode components. SL and HMI are mode-independent components which always run in the same mode, while OR, BA and DC can be deactivated in certain circumstances. The modes of Component SC, i.e. *Basic*, *Advance* and *Brake* are mapped to *Cc*, *Acc* and *Emergency* of ACC.

Component	Supported modes		
ACC System (Top)	CC	ACC	EMERGENCY
Speed Limit (SL)	ON_SL		
Object Recognition (OR)	Deactivated	ON_OR	
HMI	ON_HMI		
ACC Controller (ACC)	Cc	Acc	Emergency
Brake Assist (BA)	Deactivated		ON_BA

Table 8.1: The mode mapping table of the ACC system

Component	Supported modes		
ACC Controller (ACC)	Cc	Acc	Emergency
Distance Controller (DC)	Deactivated	ON_DC	
Speed Controller (SC)	Basic	Advance	Brake

Table 8.2: The mode mapping table of the ACC Controller

The system has two MSSs. One is Top which can request to switch between *CC* and *ACC* based on the driver's command. When Top is in *EMERGENCY*, it can only switch to *CC* manually. The other MSS is ACC which requests to switch between *Acc* and *Emergency* by sending an **`MSR`** to its parent Top which will decide whether to approve it or not. When ACC is in *Acc*, if the distance from the preceding vehicle or an obstacle in front is closer than a threshold, ACC will request to switch to *Emergency*. Similarly, when this distance returns to normal after a successful braking process, ACC will request to switch back to *Acc*. It should be noted that both Top and ACC may trigger a mode switch

when Top is in *EMERGENCY*, potentially giving rise to a mode switch conflict. In this thesis we assume such a conflict will never occur; we intend to address this issue in our future work.

Figures 8.6 and 8.7 illustrate the parent MMA of Top and the child MMAs of OR, ACC, BA, SL and HMI, which jointly define the mode mapping of Top. At one level down, figures 8.8 and 8.9 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 in Section 4.2. For complexity reasons, these MMAs only consider application-specific mode switch scenarios where only Top and ACC are the MSSs.

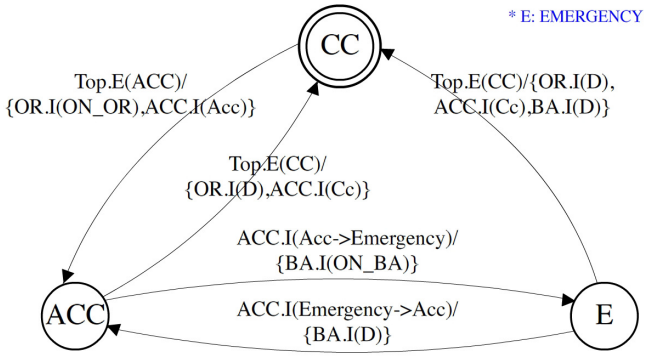

Figure 8.6: The parent MMA of Top

8.3 Mode switch at runtime

To demonstrate our mode switch runtime mechanism in the ACC system, a mode switch scenario is illustrated in Figure 8.10. It is assumed that ACC is in *Acc* and requests a mode switch to *Emergency* as an MSS by sending an **`MSR`** to Top, which approves the **`MSR`** as the MSDM by issuing an **`MSQ`** to its Type A subcomponents, i.e. ACC and BA, which are asked to switch to *Emergency* and *ON_BA* respectively. ACC checks its current state which allows a mode switch and then propagates the **`MSQ`** to its Type A subcomponent SC that is asked to switch to *Brake*. When Top receives all replies which are all **`MSOK`**, it will trigger a mode switch by issuing an **`MSI`** that follows the propagation trace of the **`MSQ`**. Each component will start its reconfiguration upon receiving an **`MSI`**

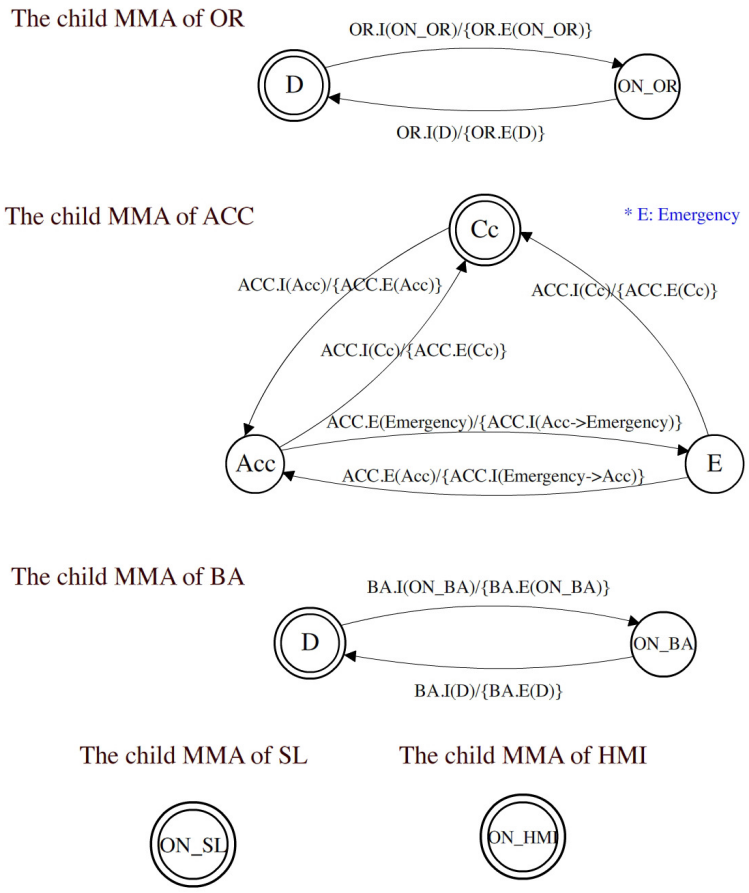

Figure 8.7: The child MMAs of OR, ACC, BA, SL and HMI

and is expected to send an **MSC** upon mode switch completion. This procedure strictly follows the mode switch dependency rule described in Section 3.3.

In fact, the ACC system has been previously studied by many existing works. What is most interesting to compare our MSL with is the ACC case study designed as a CBMMS in the SaveCCM component model [2]. 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

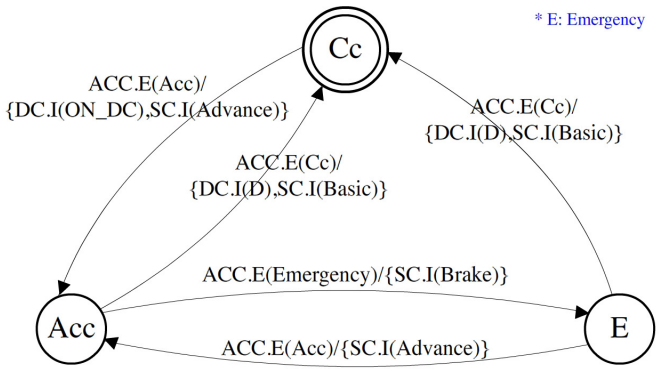

Figure 8.8: The parent MMA of ACC

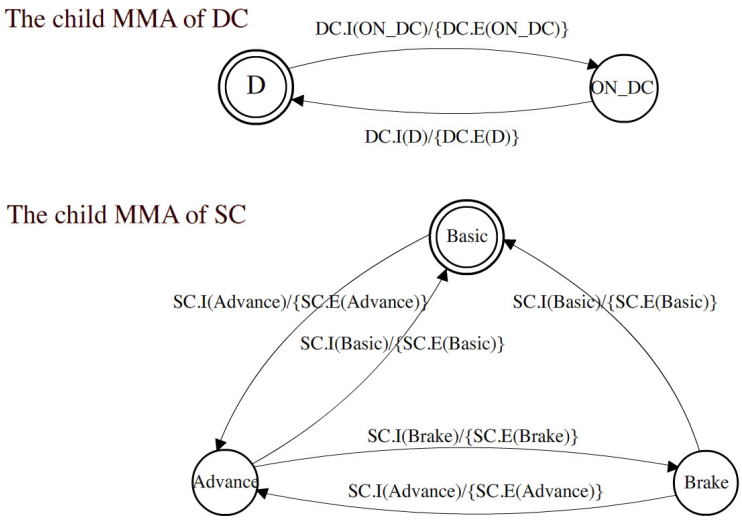

Figure 8.9: The child MMAs of DC and SC

with mode switch in a component-based framework, 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

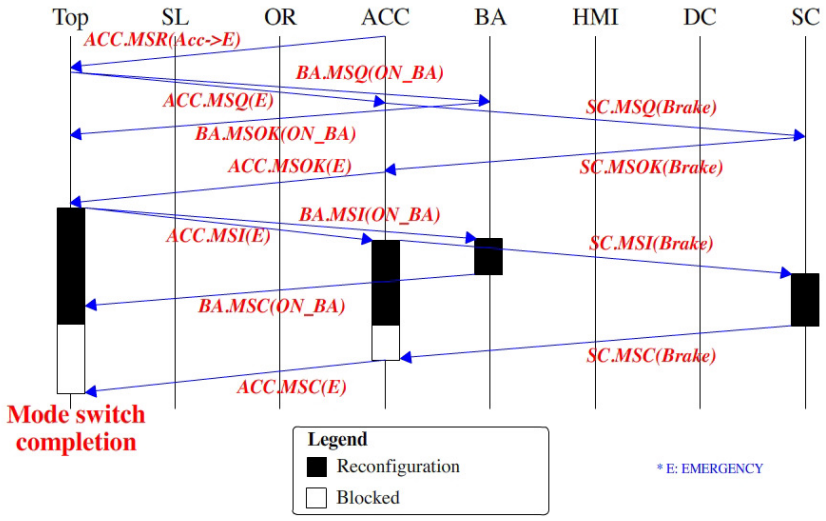

Figure 8.10: A mode switch scenario of the ACC system

consequence, it becomes an arduous task for complex component connections. Moreover, many "switch" connectors must be revised when the component hierarchy or component connection is changed. By contrast, our MSL can efficiently manage the complex procedures of a composable mode switch. A major part of our MSL can be automated while being implemented. For example, the external connection change of a component during a mode switch is simply taken care of by its parent during the reconfiguration. Only the mode mapping needs to be manually designed and it can be easily updated. In comparison with existing approaches to the handling of composable mode switch, the advantage of our MSL becomes substantial as the complexity of a CBMMS rapidly grows.

8.4 Mode switch timing analysis

Now let's perform the mode switch timing analysis for the ACC system based on the mode switch scenario illustrated in Figure 8.10. To simplify the problem, we assume no atomic component execution and that the transmission time of each primitive is always 1 time unit. Besides, Table 8.3 shows the mode

switch timing factors of the ACC system. Since ACC is the MSS and Top is the MSDM, the time spent in the first phase is

$$T_1 = ACC.msdt + Top.t_{msr} = 2 + 1 = 3 \tag{8.1}$$

Component	MSDT	PHT	SCT	RCT
Top	–	3	–	6
ACC	2	2	2	5
BA	–	–	3	2
SC	–	–	3	4

Table 8.3: Mode switch timing factors of the ACC system

In order to derive T_2, the QRT of Type A primitive components, i.e. BA and SC, can be calculated first:

$$BA.qrt = BA.sct = 3 \tag{8.2}$$

$$SC.qrt = SC.sct = 3 \tag{8.3}$$

Then $BA.qrt$ can be used to derive the QRT of its parent ACC:

$$\begin{aligned} ACC.qrt &= ACC.sct + ACC.pht + (ACC.t_{msq} + SC.qrt + ACC.t_{ok}) \\ &= 2 + 2 + (1 + 3 + 1) = 9 \end{aligned} \tag{8.4}$$

Using the results from equations (8.2) and (8.4), we get

$$\begin{aligned} T_2 = Top.pht + \max\{&Top.t_{msq} + ACC.qrt + Top.t_{ok}, Top.t_{msq} + \\ &BA.qrt + Top.t_{ok}\} = 3 + \max\{1 + 9 + 1, 1 + 3 + 1\} = 14 \end{aligned} \tag{8.5}$$

The calculation of T_3 starts with the MST of BA and SC:

$$BA.mst = BA.rct = 2 \tag{8.6}$$

$$SC.mst = SC.rct = 4 \tag{8.7}$$

Then the MST of ACC can be obtained:

$$\begin{aligned} ACC.mst &= \max\{ACC.rct, ACC.t_{msi} + SC.mst + ACC.t_{msc}\} \\ &= \max\{5, 1+4+1\} = 6 \end{aligned} \tag{8.8}$$

Using the results from equations (8.6) and (8.8), we get

$$\begin{aligned} T_3 = \max\{&Top.rct, Top.t_{msi} + ACC.mst + Top.t_{msc}, Top.t_{msi} + \\ &BA.mst + Top.t_{msc}\} = \max\{6, 1+6+1, 1+2+1\} = 8 \end{aligned} \tag{8.9}$$

Finally, the complete mode switch time of the ACC system for this mode switch scenario is

$$T = T_1 + T_2 + T_3 = 3 + 14 + 8 = 25 \tag{8.10}$$

The mode switch timing analysis above is also illustrated in Figure 8.11 which also indicates that the total mode switch time is 25.

8.5 Summary

In this chapter, we design an Adaptive Cruise Control (ACC) system by following the principles of MSL presented in previous chapters. As an exemplary CBMMS, the ACC system is composed of both single-mode and multi-mode components. All its multi-mode components comply with the mode-aware component model. The mode mapping of the ACC system has been presented by MMAs. The MSP protocol and the mode switch dependency rule have been demonstrated by a specific mode switch scenario in the system. Furthermore, the mode switch timing analysis is performed based on this mode switch scenario.

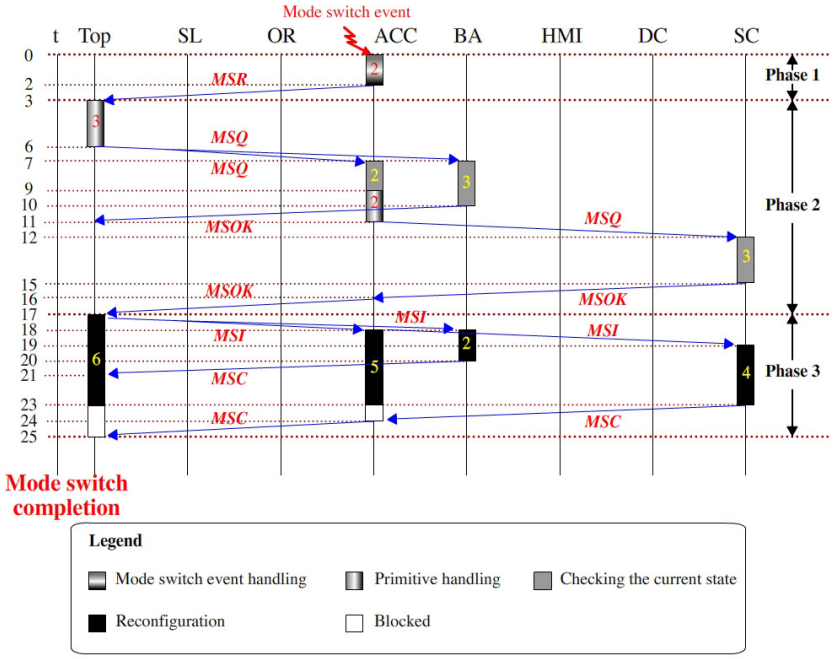

Figure 8.11: The mode switch timing analysis of the ACC system

Chapter 9

Related work

This chapter presents and discusses some existing work related to mode switch. Mode switch has been widely studied in many different research areas. What we have investigated most is mode switch in component-based systems and real-time systems. Due to the lack of standard terminology and the limitation of our survey, we may fail to cite some valuable mode switch publications. Nonetheless, we believe that the most well-known and influential works on mode switch are covered here.

9.1 The extended MechatronicUML for structural reconfiguration

To the best of our knowledge, the extended MechatronicUML [27] by Christian et al. is currently the most closely related work to our MSL. Inspired by UML 2, the original MechatronicUML [6] is introduced to model the software of mechatronic systems. It focuses on the development of distributed systems and the ability of dynamic reconfiguration. The extended MechatronicUML aims at enabling the reconfiguration of hierarchical components. Although operational modes are not considered, the structural reconfiguration in the extended MechatronicUML is similar to our composable mode switch.

Both the extended MechatronicUML and our MSL intend to separate reconfiguration behavior and functional behavior without breaking component encapsulation. Just like our model-aware component model, in the extended MechatronicUML, each component has an additional reconfiguration port,

which is bidirectional and resembles p^{MSX} and p_{in}^{MSX} in our mode-aware component model. According to our mode-aware component model, the configuration in each mode is pre-defined. In contrast, the extended MECHATRONICUML does not explicitly address what the target configuration looks like after reconfiguration.

Within each composite MECHATRONICUML component, reconfiguration is locally managed and executed by two dedicated components: the *Manager* and *Executor*. Figure 9.1 [27] illustrates such a composite MECHATRONICUML component. The *Manager* is connected to both the parent and its subcomponents via their reconfiguration ports (represented by "R" in Figure 9.1). Also, the *Manager* and *Executor* are connected to each other via their reconfiguration ports. Reconfiguration messages can be propagated between a composite component and its *Manager*, or between its *Manager* and its subcomponents, or from its *Manager* to its *Executor* when this composite component decides to perform reconfiguration. A reconfiguration message plays the same role as the MSI in our MSL. However, the information conveyed by a reconfiguration message highly relies on the system functionality while an MSI is only related to mode, thus enabling better separation of reconfiguration and functional behaviors. The *Manager* controls the reconfiguration process, e.g. reconfiguration propagation and decision to make reconfiguration. The *Executor* encapsulates reconfiguration rules and execute reconfiguration upon receiving the reconfiguration command from the *Manager* at the same nested level. The pair of *Manager* and *Executor* is comparable with the mode switch runtime mechanism of our MSL, particularly the MSP protocol. Nonetheless, the propagation criteria of a *Manager* is still not clear due to the fact that the propagation rules of different *Managers* can be different and they are highly related to the system functionality. In contrast, thanks to the mode mapping mechanism, in our MSL, the propagation of any primitive is predictable and all composite components have the same propagation criteria.

The extended MECHATRONICUML is also aware of the real-time properties by providing a preliminary analysis of the reconfiguration time. The essential idea of this reconfiguration timing analysis is similar to our mode switch timing analysis, yet still rather preliminary due to the simple reconfiguration propagation scheme. Besides, the mode switch timing analysis of MSL considers atomic component execution which has not been addressed by the extended MECHATRONICUML.

Moreover, the extended MECHATRONICUML suggests that system reconfiguration should be performed bottom-up, which is in line with our mode switch dependency rule. Both the extended MECHATRONICUML and our MSL agree on

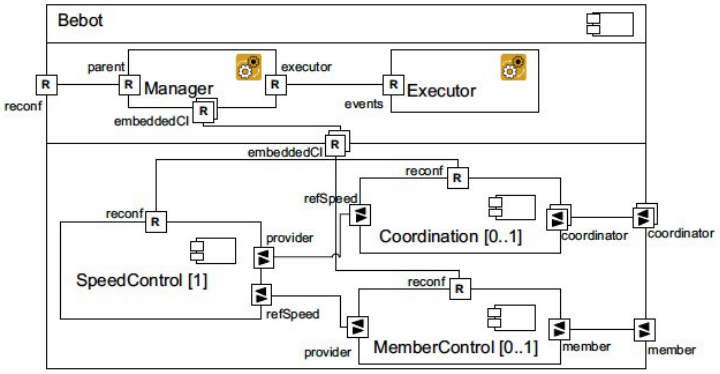

Figure 9.1: A reconfigurable component in the extended MECHATRONICUML

the correctness of reconfiguration (or mode switch), i.e. the configuration (or mode) consistency between different components.

To a large extent, the development of our MSL has been done independently of MECHATRONICUML, although we did get inspiration from MECHATRONICUML in defining the voting for reconfiguration: an idea originating from the 2-phase commit protocol for distributed databases [7]. Upon receiving a reconfiguration voting message, if the current condition of a component does not allow reconfiguration, it will report this to its parent and reconfiguration will be cancelled. In our MSL, the **`MSQ`** propagation implements the voting phase. A positive voting result entails an **`MSI`** propagation while a negative voting result leads to an **`MSD`** propagation. In addition, the extended MECHATRONICUML provides the formal modeling of reconfiguration based on component story diagrams [61], which is a formal specification of structural transformations. In contrast, MSL focuses on the mode switch synchronization of different components rather than the mode switch of a single component.

9.2 The oracle-based mode change approach with property networks

Tomas et al. propose an oracle-based approach [53] to handle composable mode switch. The basic idea is to abstract the behavior of each component into a local property network, constructed by a set of properties and the transitions

between different properties, associated with property functions. A property represents a particular feature of a component and can be either functional or non-functional. The component mode can be mapped from a specific set of its properties and their valuations. In fact, the component mode itself is also modeled as a property. The local property networks of different components at various levels can be interconnected and jointly become a global property network. A mode switch event is initially detected by the valuation change of a property in the global property network. Just like our mode switch propagation, a subsequent network re-calculation will be discharged in a transitive fashion. The network is stabilized when there is no property that changes its valuation. Mode switch is treated as an additional step, possibly after each global property network update. The update of component modes is achieved by a top-down inspection that calls the mapping function of each component from properties to component mode.

The oracle-based approach is also aware of real-time issues such as the time required to update the property network. Due to the strong dependency upon the number of properties and the number of their possible valuations, the network stabilizing time is unpredictable. The solution presented in [53] is to offline construct an oracle, i.e. a finite-state machine. Each state in the oracle represents the stable state of the property network, including the valuation of all properties. Since all intermediate states (when the property network is not stabilized) are excluded from the oracle, the property network update can always be referred to as a single transition of the oracle, with constant time. However, the downside of using an oracle is that it breaks component encapsulation by assuming that the global property networks are given. When a component provided by a third party is reused, its local property network is most likely not visible. Consequently, the oracle construction becomes impossible to use.

Furthermore, a multi-mode component model is informally defined. For instance, in each mode, a composite component is identified by a set of running subcomponents, their bindings (i.e. connections) and a set of attributes. All of these elements are already included in the component configuration of our mode-aware component model, which also considers other aspects such as the mode-specific behavior. Additionally, the oracle-based approach allows communication between a parent and its children during the propagation of a triggering event in the property network. Each component must have a dedicated interface for this vertical propagation. This dedicated interface is not explicitly specified in [53], whereas in [48] two such dedicated interfaces are defined: **setModeProperty** and **getModeProperty**.

In [48], Matěj et al. discuss the mode hierarchy and represent it by Hier-

archical Mode Automaton (HMA), which resembles our Mode Mapping Automata (MMA). HMA provides the static mode mapping the modes of a parent and its subcomponents which are determined by the mapping from the local property network. Nevertheless, to the best of our knowledge, HMA has not taken the dynamic mode mapping into account. MMA allows the specification of dynamic mode mapping rules by defining the DDMs of each component for all possible mode switch scenarios.

The runtime architecture of the oracle-base approach is presented in [48]. The property network update and mode switch is realized by three specific components: *Mode Property Storage*, *Mode Property Reactor* and *Mode Reconfigurator*. *Mode Property Storage* stores the current values of all properties and it is connected to the dedicated interfaces **setModeProperty** and **getModeProperty** of each common component. A new event can be triggered after the value change of a property and then this event is sent to *Mode Property Reactor* where the pre-computed oracle resides. *Mode Property Reactor* produces new property values and a new configuration of the system. Finally, based on the new configuration, *Mode Reconfigurator* carries out the reconfiguration of each component, e.g. enabling/disabling components or connections. This centralized runtime mechanism is easy to implement, however, two drawbacks are exposed. One is that the oracle breaks component encapsulation. The other is a scalability problem. The complexity of the three particular components is in direct proportion to the number of components in the system. In contrast, in our MSL, encapsulation is a key property and no component has the global knowledge of the entire system, hence it scales quite well.

9.3 Component models supporting mode switch

There are numerous component models, some of which have already been analyzed, compared and classified [13] [32]. Among existing component models, very few consider the mode switch problem. Here we make a non-exhaustive but representative presentation of component models supporting mode switch, including BlueArx, COMDES-II, Koala, MyCCM-HI, Rubus and SaveCCM.

BlueArx

BlueArx [36] is a component model developed by Bosch[1] particularly for automotive applications with constrained resources. BlueArx supports multi-mode

[1] http://www.bosch.com/

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. An additional analytic interface is introduced to refer to modes for specifying semantic or other context dependencies such as hardware features or tool chains.

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 the *XGen* tool. Unlike our MSL, the mode of a BlueArx component is not explicitly defined until a multi-mode system is completely built. In our MSL, the supported modes of a multi-mode component are well-defined right after it is developed.

The major purpose of defining modes for the BlueArx component model is more precise prediction of system properties, such as WCET. The mode switch problem is barely addressed.

COMDES-II

COMDES-II (COMponent-based design of software for Distributed Embedded Systems-version II) employs a hierarchical model to specify system architecture [35]. A multi-mode component is treated as a pair of state machine Function Block (FB) and 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) by avoiding non-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.

Koala and SaveCCM

Koala [47], a component model developed by Philips[2], is dedicated to managing software complexity in consumer electronics. Koala does not explicitly deal with mode switch. Instead, it proposes some primitive but practical solutions to efficiently handle 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. The motivation comes from the frequent occurrence of this design pattern. A switch is able to divert an incoming data flow to different outgoing branches according to different conditions being 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. In our MSL, the system functionality and the mode switch handling are completely orthogonal issues.

The SaveCCM component model [26] is designed specifically for the vehicular domain, with its 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. Just like Koala, SaveCCM integrates the mode switch handling mechanism into the functional behaviors of each component, consequently preventing clear separation between system functionality and mode switch management.

MyCCM-HI

MyCCM-HI (Make Your Component-Container Model-High Integrity) is a component framework for critical, distributed, real time and embedded software [8]. Software components and their behaviors are described by the input architecture description language COAL (Component-Oriented Architecture Language), which supports enumerating different operational modes of the system and of each component. Each multi-mode component has an associated mode automaton that implements its mode switch mechanism. Mode automata 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 among different parts of the system. MyCCM-HI provides one of the most advanced frameworks among existing component models with

[2] http://www.philips.com/

respect to mode switch handling.

Rubus

Rubus [25] 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 a single node, i.e. Electronic Control Unit (ECU)) as different modes are typically defined at the top level through a mode/state transition diagram. In each mode, there is a system-wide static configuration of components. A mode switch of the system corresponds to the switch between different such configurations. Individual components have no sense of mode, i.e., are not mode-aware and it is up to the system integrator to integrate them into a multi-mode system. There is no published information about the Rubus mode switch handling at runtime.

The above presented component models use different mechanisms to handle the mode switch of CBMMSs, however, in contrast with our MSL, none of them provide any systematic strategy to cope with the coordination of the mode switches of different components. Besides, they also assume independent mode switches between different components. Our MSL circumvents the issues unexplored by these component models by focusing on the management of interdependent mode switches between components.

9.4 Mutli-mode real-time systems and mode switch

The research on mode switch in the real-time systems domain has been conducted for decades. In the real-time literature, mode switch is also called "mode change". There are miscellaneous topics studying multi-mode real-time systems and mode switch. Among them, topics that have been investigated most are the design of mode switch protocols and the schedulability analysis during a mode switch. For a multi-mode real-time system, a mode 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. Sometimes, the

[3] http://www.arcticus-systems.com/

parameters (e.g. the execution time or period) of tasks executing in both the old mode and the new mode are changed when changing modes. An extensive survey and classification of some basic mode switch protocols can be found in [54].

In general, five types of tasks have been distinguished: (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 (this can be compared to the component with atomic execution discussed in our MSL); (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* that is always running without being affected by any mode switch. A mode switch protocol can be classified driven by two criteria. With respect to the treatment of unchanged tasks, two categories can be identified: (1) *protocols with periodicity* which does not affect the execution of unchanged tasks during a mode switch; (2) *protocols without periodicity* which changes the activation of unchanged tasks during a mode switch. Currently most mode switch protocols belong to the second type, however, the first type has been proposed in recent years [44] [45]. 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; (2) *asynchronous protocols* where old and new mode tasks are allowed to be executed at the same time during a mode switch. Synchronous protocols, such as the *Idle Time protocol* [62], *Maximum-Period Offset protocol* [5] and *Minimum Single Offset protocol* [54], do not cause temporal overload during a mode switch. Therefore, no schedulability analysis is required during a mode switch. However, synchronous protocols are notorious for its long mode switch latency 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.

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. One of the earliest publications related to mode switch for real-time systems is by Sha et al. [56], who develop a simple mode switch protocol in a prioritized preemptive scheduling environment (Rate Monotonic (RM) scheduling) guaranteeing short and bounded mode switch latency. This protocol is improved by Tindell et al. [63] who consider Deadline Monotonic (DM) scheduling instead

of RM scheduling and introduce offsets for wholly new mode tasks to enhance schedulability. Pedro and Burns [49] further extend the protocol in [63] by introducing offsets for all new mode tasks. Real and Crespo [54] proposes a new asynchronous mode switch protocol together with the calculation of necessary offsets. M. Holenderski et al. [31] develop a mode switch protocol where mode switch is carried out by a dedicated mode switch task with highest priority. An old mode task can be aborted to perform a mode switch at pre-defined termination points to achieve swift mode switch. The above referenced works are based on static scheduling policies. In addition, the mode switch problem has also been considered in conjunction with dynamic scheduling policies such as Earliest Deadline First (EDF) [4] [58]. Moreover, Gerhard [17] provides a mode switch schedulability analysis in the context of offline scheduling. While all the aforementioned works deal with uniprocessor system, the focus is gradually being shifted onto multiprocessor platforms. For instance, two protocols [46] (synchronous and asynchronous) are presented for symmetrical multiprocessor platforms and they are further extended to uniform multiprocessor platforms [64]. Another mode switch protocol is developed for identical multiprocessor platforms [45]. Compared with [46], mode independent tasks are particularly considered in [45]. In [28], the mode switch timing analysis is performed for distributed systems. Also, M. Negrean et al. [44] extend the contribution in [28] by concerning communicating tasks and the recurrent effect of a mode switch in distributed systems.

Temporal isolation in real-time scheduling has also been intensively investigated in multi-mode real-time systems. The de facto approach for temporal isolation is the use of servers. The dynamic reconfiguration of servers at runtime can be regarded as a mode switch. Reconfigurable servers are suggested for adaptive resource reservation [59] [55]. A more recent work [37] replaces the TDMA server used in [59] by CBS server and finds that CBS is more suited to reconfigurations than TDMA. Another interesting work on temporal isolation and mode switch is by N. Fisher and M. Ahmed [16], who take both application-level and resource-level mode switches into account. Although [16] is based on Explicit-Deadline Periodic (EDP) resource model that does not support dynamic reconfiguration, it provides sufficient schedulability analysis of a sequence of mode switches with pseudo-polynomial time complexity and good scalability. In addition, Phan et al. extend the traditional Real-Time Calculus to handle multi-mode [50], and present a multi-mode automaton model for modeling multi-mode applications, together with an interface-based technique for compositional analysis [51] and a semantic framework for mode switch protocols [52].

The mode switch handling in the above approaches from the real-time literature is rather complementary to our MSL. On the one hand, in the real-time community, the top concern of a mode switch is real-time tasks and schedulability which is not really considered by MSL. On the other hand, MSL pays more attention to the mode switches of reusable components rather than tasks, which requires a different handling than the mode switch of a monolithic multi-mode real-time system.

9.5 Languages supporting mode switch

The mode switch problem has also been covered by some programming and specification languages, such as AADL, Giotto, TDL, the extended Darwin, Modechart and Mode-automata.

AADL

The Architecture Analysis & Design Language (AADL) [15], is a modeling language that supports analyses of a system's architecture with respect to performance-critical properties at the 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 mode switch. The initial mode of a component must be explicitly declared. 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 also be defined to distinguish its alternative behaviors in different modes. For instance, a component may have different worst-case execution time in different modes. AADL focuses on the mode switch of an individual component, thus the relation between the mode switches in different components is not explicitly specified.

Giotto and TDL

Apart from AADL, Giotto [29] and TDL [60] also support multi-mode and mode switch. Giotto is a time-triggered language for embedded programming. It considers a mode as the periodic invocation of a fixed set of tasks. The time-triggered nature of Giotto makes it suitable for safety-critical applications, yet at the sacrifice of flexibility. Compared with Giotto, TDL is different with

respect to the mode declaration and mode switch conditions, but it suffers from the same limitation as Giotto due to its time-triggered nature.

The extended Darwin

Dan et al. [30] 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 [40]. To be aware of modes, Darwin is extended by adding a mode attribute to each component indicating its current mode. They identify that the mode of a composite component is directly related to the modes of its subcomponents, i.e. the need of mode mapping. Yet they provide no solution that can be compared with our mode mapping mechanism. 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. This is rather consistent with our MSL. The mode switch handling is not considered in [30].

Modechart

Modechart [34] is a specification language for real-time systems. The semantics of Modechart is based on Real Time Logic (RTL) [33]. Modechart focuses on the specification of absolute timing properties as well as modes and mode transitions. Just like components, a mode can also 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. Actually, the mode concept in Modechart is more general than the mode in our MSL or most other related works, since sometimes even concurrent running threads are considered as parallel modes. Modechart assumes instantaneous mode transition and does not aim for component-based systems.

Mode-automata

Mode-automata [41] [42] is a programming model proposed as an extension of the synchronous language Lustre [11]. 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.

Compared with MSL, most of these languages regard mode switch as a system concept. AADL and the extended Darwin consider the mode switch of a component, however, AADL assumes independent mode switch between components. The extended Darwin is aware of the mode mapping between components, yet provides no mode switch handling at runtime. Actually, MSL can benefit from some new specification languages, which can, for instance, define primitive components and composite components according to the mode-aware component model and the mode mapping rules of a composite component that are represented by MMA.

Chapter 10

Conclusions and future work

Mode switch for Component-Based Multi-Mode Systems (CBMMSs) is to date an emerging research problem that has been barely explored. The main contribution of our work, Mode Switch Logic (MSL), provides a theoretical framework for the development of CBMMSs and the handling of its composable mode switch. As the closure of this thesis, this chapter summarizes our research results. Moreover, we shed some light on the evolution trace of MSL, which has been subject to frequent modification since its initial version in [18]. Actually, MSL is further revised so aggressively in this thesis that it looks quite different from the MSL presented in any of our previous publications. Finally, we discuss some open issues in our research area and our future work.

10.1 Summary

This thesis is originally motivated by two underlying technologies: mode switch and Component-Based Software Engineering (CBSE), both of which offer effective tactics for the development of complex embedded systems. As is introduced in Chapter 1, we, after combining both technologies, study the composable mode switch of CBMMSs. As the output of our research, an MSL has been developed to overcome the major challenges identified of such a composable mode switch. Our MSL comprises a mode-aware component model, a mode mapping mechanism and a mode switch runtime mechanism. Furthermore, the mode switch runtime mechanism consists of the MSP protocol and the mode switch dependency rule. Described in Chapter 2, the mode-aware

component model distinguishes itself from traditional component models in three aspects:

- It allows a multi-mode component to have unique configurations in different modes.
- It introduces dedicated mode switch ports to each multi-mode component for mode-dedicated communication between components.
- It distributes the mode switch runtime mechanism to each component to control its mode switch behavior.

Since the mode-aware component model only focuses on mode switch and it is defined in a generic fashion, it should be able to guide many existing component models to become mode-aware.

The mode switch runtime mechanism is spread over chapters 3 and 5. Chapter 3 explains the MSP protocol and the mode switch dependency rule assuming no atomic component execution. The MSP protocol is essentially a distributed algorithm which functions like a communication protocol. A couple of types of primitives have been defined. Then following the MSP protocol, each component emits and receives different types of primitives and behaves accordingly during the mode switch propagation. Once a mode switch is triggered as a consequence of the mode switch propagation, certain components will switch mode while following the mode switch dependency rule that enforces the mode switch completion order among different components.

The MSP protocol actually cannot fulfill its duty alone, as it requires the assistance of mode mapping which is elucidated in Chapter 4. Mode mapping is required for the composition of multi-mode components and for determining which components must switch mode as well as their new modes for each possible mode switch scenario. Chapter 4 presents the mode mapping mechanism of MSL. Each composite component defines a set of mode mapping rules presented by Mode Mapping Automata (MMA) for the mode mapping between the modes of itself and its subcomponents.

Chapter 5 uncovers a new problem for a composable mode switch: atomic component execution, which cannot be interrupted by the primitives transmitted during the mode switch propagation. This problem is tackled by slightly extending the MSP protocol for certain components, without altering the mode switch dependency rule and the mode mapping mechanism.

Integrating all the elements of MSL presented before, Chapter 6 provides algorithms implementing the mode switch runtime mechanism for primitive

components, non-top composite components and the top component, respectively. These algorithms establish the foundation for applying MSL to a real-world system in future.

We consider not only the correctness of MSL but also its timing effects. A key real-time metric of a multi-mode system is its mode switch time. Chapter 7 focuses on how to calculate the mode switch time of a CBMMS and its components for MSL. In addition, a model-checking approach is also proposed to obtain atomic component execution time.

Moreover, we also demonstrate the key elements of MSL in a case study: an Adaptive Cruise Control (ACC) system in Chapter 8, which contributes extra value to the pracaticability of MSL.

10.2 The evolution trace of MSL

By comparing the thesis and our previous publications, one shall witness the remarkable evolution of our MSL. The reason is that the lack of preceding works gives rise to the tentative development of MSL. As our understanding of this research problem goes deeper, the flaws and limitations of old ideas proposed before may be spotted and thus replaced by new ideas.

The initial version of MSL can be found in [18], which includes the mode-aware component model, the MSP protocol and the mode switch dependency rule. The mode-aware component model introduces a single dedicated mode switch port for both primitive components and composite components. A component can define a unique configuration in each mode, however, only a few factors included in a configuration are identified. The initial version of the MSP protocol is called *The MSR propagation mechanism*. **`MSR`** is the only one primitive used for mode switch propagation. No mode mapping is considered, therefore, it is assumed that the **`MSR`** issued by an MSS will lead to the mode switches of all the other components. Since there is no MSDM, an **`MSR`** can never be rejected and it immediately aborts the current execution of the receiver which will then start to switch mode. The original mode switch dependency rule in [18] is dependent on the data flow of a system. For two components c_i and c_j with the data flow from c_i to c_j, it requires that c_i must complete its mode switch before c_j to avoid the case that c_j running in the new mode still receives the data from c_i running in the old mode. In some situations, this is desired. Nevertheless, this may not be necessary for other types of systems. Besides, the bottom-up mode switch dependency is addressed in [18]. Thereafter, the mode switch timing analysis based on the initial version of MSL is

provided in [21], yet with the unrealistic assumption that there is no atomic component execution.

Mode mapping is first emphasized in [20], where it is addressed that a mode switch can be a local activity within some composite components instead of always being a global activity. The MSP protocol is further extended by only propagating an **MSR** to the components which need to switch mode. Both the mode mapping table and Mode Mapping Automata (MMA) are presented to express mode mapping rules. The original MMA must recognize an **MSR** that is treated as an external signal. Besides, deactivated components are not considered in any MMA.

The mode switch dependency rule is first extended in [22]. Compared with the original version, the extended version removes the mode switch completion dependency between sibling components. A composite component only needs to collect the mode switch completion signals from its subcomponents. **MSC** is called *ms_done* for this extended version. Apart from the extended mode switch dependency rule, its correctness and the correctness of the original MSR protocol are also proved in [22]. Another revision of MSL in [22] is the mode-aware component model where two dedicated mode switch ports are used for a composite component.

The atomic component execution problem is first addressed and handled in [23]. The MSP protocol is revised in the sense that an **MSR** can be either approved or rejected by the MSDM. If an **MSR** is approved, the MSDM will issue an **MSI** to trigger a mode switch. It is assumed in [23] that an Atomic Execution Group (AEG) (see Chapter 5) is an individual component. Under such an assumption, the handling of atomic component execution is achieved by delaying the **MSI** propagation from an AEG component to its subcomponents. The mode switch timing analysis is also updated in [23] based on the revised MSP protocol, including the model-checking approach used to obtain the worst-case atomic execution time of an AEG. Paper [23] also provides a preliminary approach in order to resolve the conflict due to multiple mode switch triggering. This is not included in the thesis and the in-depth solution will be included in our future work.

The latest version of MSL before the thesis is in [19]. The mode-aware component model is extended by further considering atomicity as an EFP in component configuration. The cooperation of the MSP protocol revised in [23] and mode mapping is highlighted in [19].

Finally, we would like to outline the additional changes of MSL in this thesis:

- A more complete definition of the mode-aware component model. More items included in component configuration are identified, such as activated input and output ports, mode-dependent EFPs and mode-independent EFPs.
- The introduction of the voting phase in the MSP protocol. This prevents the case that a component is forced to switch mode even if its current state does not allow a mode switch.
- The formulation of key properties associated with the correctness of the revised MSP protocol and the mode switch dependency rule, as well as the proofs of the correctness of these properties.
- The updated MMA and the clear separation between the MSP protocol and the MMA. An MMA does not need to care about the primitive type of an external signal. And deactivated components are also considered in the MMA.
- A more general model of the AEG, which is not limited to an individual component as is assumed in [23] but can also include multiple components. The handling of atomic component execution is moved from an AEG component to a non-AEG component with AEG(s) among its subcomponents.
- The mode switch timing analysis based on the revised mode switch runtime mechanism. Besides, more timing factors are considered making the calculation result more reasonable.

Apart from the major changes above, the case study: the ACC system in Chapter 8 has never been published before. And myriads of new concepts and minor changes can also be found in the thesis.

10.3 Future work

Currently, the least attention in our work is paid to the verification and evaluation of MSL. When we come up with some new ideas, our initial attempt is to implement these ideas in UPPAAL by modeling a number of small but representative examples and check the correctness by verifying some properties formulated from the UPPAAL models. Since UPPAAL tends to succumb to state explosion, the notorious problem for all model-checking approaches,

the size of the system being modeled in UPPAAL cannot be too big. As a consequence, positive verification results from the examples modeled in UPPAAL is insufficient to be convincing in general. Even though the correctness of the mode switch runtime mechanism has been proved in the thesis, we are still in need of the extensive simulation and evaluation of MSL. Our goal is to develop a tool where a CBMMS can be automatically generated with regard to some pre-defined parameters such as the maximum number of components, the maximum number of children of a composite component and the maximum depth level in the system. The system should be visualized and the primitive transmission in the process of mode switch can be monitored. The algorithms described in Chapter 6 are implemented in the generated components. A mode switch scenario can be either automatically generated or manually defined as the stimuli of the system. Then the behavior of each component for this mode switch scenario can be observed. By repeating the simulation of a variety of systems with random parameters, we can assert if our MSL works properly for most CBMMSs. Then the positive evaluation result would make a closer step towards the implementation of MSL in a real-world CBMMS in future. Another alternative is to resort to theorem proof techniques to prove the correctness of MSL. Theorem proving is relatively less intuitive and more difficult to grasp, whereas it can produce a more convincing result.

In this thesis, we have made an unrealistic assumption that no mode switch is triggered when a system is switching mode. In practice, multiple MSSs can be defined and they may trigger two different mode switch scenarios that interfere with each other. According to the preliminary approach in [23], an arbitration mechanism can be applied to the top component to resolve such conflict. However, the top component cannot always be the MSDM and thus a more in-depth approach (one of our ongoing works) is required. Another ongoing work is to implement MSL in the ProCom [10] component model. It is our ambition to add mode switch support to ProCom and other well-developed component models.

According to the MSP protocol, a lot of primitives need to be transmitted during mode switch propagation. A primitive often has to be forwarded many times to reach the final destination. Conceptually, this is the right strategy. However, it does not have to be implemented in this way for systems with stringent timing constraints. For a statically configured system, it should be possible to pre-define all the possible mode switch scenarios and pre-calculate the recipients of each primitive. In this way, all the pre-calculated results before runtime can be stored in a particular component which has the global knowledge of the entire system and is able to communicate with all the other compo-

nents. The MSP protocol can be replaced by the multicast of primitives from that component. Once the system is built and ready to run, the overall performance of a system becomes more important than adhering to the principles of CBSE. Therefore, we may consider the practical optimization of MSL at runtime in future.

Bibliography

[1] Autosar GbR: Autosar-technical overview. Technical report, AUTOSAR GbR. http://www.autosar.org/index.php?p=3&up=1&uup=0.

[2] M. Åkerholm, A. Möller, H. Hansson, and M. Nolin. Towards a dependable component technology for embedded system applications. In *Proceedings of 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'05)*, January 2005.

[3] OSGi Alliance. *OSGi service platform: core specification*. aQute Publishing, 2009.

[4] B. Andersson. Uniprocessor EDF scheduling with mode change. In *Proceedings of 12th International Conference on Principles of Distributed Systems (OPODIS'08)*, pages 572–577, December 2008.

[5] C. M. Bailey. Hard real-time operating system kernel. investigation of mode change. Technical Report Task 14 Deliverable on ESTSEC Contract 9198/90/NL/SF, British Aerospace Systems Ltd., 1993.

[6] S. Becker, C. Brenner, S. Dziwok, T. Gewering, C. Heinzemann, U. Pohlmann, C. Priesterjahn, W. Schäfer, J. Suck, O. Sudmann, and M. Tichy. The MechatronicUML method – process, syntax, and semantics. Technical Report tr-ri-12-318, Software Engineering Group, Heinz Nixdorf Institute University of Paderborn, February 2012.

[7] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison Wesley, 1987.

[8] E. Borde, G. Haïk, and L. Pautet. Mode-based reconfiguration of critical software component architectures. In *Proceedings of Conference on Design, Automation and Test in Europe (DATE'09)*, pages 1160–1165, April 2009.

[9] D. Box. *Essential COM*. Addison-Wesley, 1997.

[10] T. Bureš, J. Carlson, I. Crnković, S. Sentilles, and A. Vulgarakis. ProCom - the Progress component model reference manual, version 1.0. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-230/2008-1-SE, Mälardalen University, June 2008.

[11] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. LUSTRE, a declarative language for programming synchronous systems. In *Proceedings of 14th Symposium on Principles of Programming Languages*, January 1987.

[12] I. Crnković and M. Larsson. *Building reliable component-based software systems*. Artech House, 2002.

[13] I. Crnković, S. Sentilles, A. Vulgarakis, and M. R. V. Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37(5):593–615, October 2011.

[14] Z. Deng, J. W. S. Liu, and J. Sun. A scheme for scheduling hard real-time applications in open system environment. In *Proceedings of 9th Euromicro Workshop on Real-Time Systems*, pages 191 – 199, June 1997.

[15] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical Report CMU/SEI-2006-TN-011, Software engineering institute, MA, 2006.

[16] N. Fisher and M. Ahmed. Tractable real-time schedulability analysis for mode changes under temporal isolation. In *Proceedings of 9th IEEE Symposium on Embedded Systems for Real-Time Multimedia (ESTImedia'11)*, pages 130–139, October 2011.

[17] G. Fohler. Changing operational modes in the context of pre run-time scheduling. *Special Issue on Responsive Computer Systems of the IEICE Transactions on Information and Systems*, E76-D(11):1333–1340, November 1993.

[18] Y. Hang, E. Borde, and H. Hansson. Composable mode switch for component-based systems. In *Proceedings of 3rd International Workshop on Adaptive and Reconfigurable Embedded Systems (APRES '11)*, pages 19–22, April 2011.

[19] Y. Hang, J. Carlson, and H. Hansson. Towards mode switch handling in component-based multi-mode systems. In *Proceedings of 15th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE'12)*, pages 183–188, June 2012.

[20] Y. Hang and H. Hansson. A mode mapping mechanism for component-based multi-mode systems. In *Proceedings of 4th Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'11)*, pages 38–45, November 2011.

[21] Y. Hang and H. Hansson. Timing analysis for a composable mode switch. In *Proceedings of the Work-in-Progress session of the 23rd Euromicro Conference on Real-Time Systems*, pages 15–18, July 2011.

[22] Y. Hang and H. Hansson. A mode switch logic for component-based multi-mode systems. Technical Report 261/2012, Mälardalen Real-Time Research Centre Mälardalen University, January 2012.

[23] Y. Hang and H. Hansson. Timing analysis for mode switch in component-based multi-mode systems. In *Proceedings of 24th Euromicro Conference on Real-Time Systems (ECRTS'12)*, pages 255–264, July 2012.

[24] Y. Hang and H. Hansson. A UPPAAL model for timing analysis of atomic execution in component-based multi-mode systems. Technical report, Mälardalen Real-Time Research Centre, Mälardalen University, February 2012.

[25] K. Hänninen, J. Mäki-Turja, M. Nolin, M. Lindberg, J. Lundbäck, and K. Lundbäck. The Rubus component model for resource constrained real-time systems. In *Proceedings of 3rd International Symposium on Industrial Embedded Systems (SIES'08)*, pages 177 –183, June 2008.

[26] H. Hansson, M. Åkerholm, I. Crnkovic, and M. Törngren. SaveCCM - a component model for safety-critical real-time systems. In *Proceedings of Euromicro Conference, Special Session on Component Models for Dependable Systems*, pages 627 – 635, September 2004.

[27] C. Heinzemann, C. Priesterjahn, and S. Becker. Towards modeling reconfiguration in hierarchical component architectures. In *Proceedings of 15th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE'12)*, pages 23–28, June 2012.

[28] R. Henia and R. Ernst. Scenario aware analysis for complex event models and distributed systems. In *Proceedings of 28th IEEE Real-Time Systems Symposium (RTSS'07)*, pages 171 –180, December 2007.

[29] T. A. Henzinger, B. Horowitz, and C. Meyer Kirsch. Giotto: A time-triggered language for embedded programming. In *Proceedings of the IEEE*, pages 166–184, 2001.

[30] D. Hirsch, J. Kramer, J. Magee, and S. Uchitel. Modes for software architectures. In *Proceedings of 3rd European conference on Software Architecture (EWSA'06)*, pages 113–126, September 2006.

[31] M. Holenderski, R. J. Bril, and J. J. Lukkien. Swift mode changes in memory constrained real-time systems. In *Proceedings of 12th IEEE International Conference on Computational Science and Engineering (CSE'09)*, pages 262–269, August 2009.

[32] P. Hošek, T. Pop, T. Bureš, P. Hnětynka, and M. Malohlava. Comparison of component frameworks for real-time embedded systems. In *Component-Based Software Engineering*, volume 6092 of *Lecture Notes in Computer Science*, pages 21–36. 2010.

[33] F. Jahanian and A. K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–904, September 1986.

[34] F. Jahanian and A. K. Mok. Modechart: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 20(12):933–947, 1994.

[35] X. Ke, K. Sierszecki, and C. Angelov. COMDES-II: A component-based framework for generative development of distributed real-time control systems. In *Proceedings of 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'07)*, August 2007.

[36] J. E. Kim, O. Rogalla, S. Kramer, and A. Hamann. Extracting, specifying and predicting software system properties in component based real-time embedded software development. In *Proceedings of ICSE Companion*, pages 28–38, 2009.

[37] P. Kumar, N. Stoimenov, and L. Thiele. An algorithm for online reconfiguration of resource reservations for hard real-time systems. In *Proceedings of 24th Euromicro Conference on Real-Time Systems (ECRTS'12)*, pages 245–254, July 2012.

[38] K. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *STTT-International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[39] K.-K. Lau, L. Safie, P. Štěpán, and C. Tran. A component model that is both control-driven and data-driven. In *Proceedings of 14th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE'11)*, pages 41–50, June 2011.

[40] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of 5th European Software Engineering Conference (ESEC'95)*, pages 137–153, September 1995.

[41] F. Maraninchi and Y. Rémond. Mode-automata: About modes and states for reactive systems. In *Proceedings of European Symposium On Programming*, March 1998.

[42] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, (46):219–254, 2003.

[43] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly Media, 2001.

[44] M. Negrean, M. Neukirchner, S. Stein, S. Schliecker, and R. Ernst. Bounding mode change transition latencies for multi-mode real-time distributed applications. In *Proceedings of 16th IEEE International Conference on Emerging Technologies Factory Automation (ETFA'11)*, pages 1 –10, September 2011.

[45] V. Nélis, B. Andersson, J. Marinho, and S. M. Petters. Global-EDF scheduling of multimode real-time systems considering mode independent tasks. In *Proceedings of 23rd Euromicro Conference on Real-Time Systems (ECRTS'11)*, pages 205–214, July 2011.

[46] V. Nélis, J. Goossens, and B. Andersson. Two protocols for scheduling multi-mode real-time systems upon identical multiprocessor platforms. In *Proceedings of 21st Euromicro Conference on Real-Time Systems (ECRTS'09)*, pages 151–160, July 2009.

[47] R. V. Ommering, F. V. D. Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *Computer*, 33(3):78 –85, March 2000.

[48] M. Outlý, T. Pop, M. Malohlava, and T. Bures. Mode change in real-time component systems-suitable form of run-time variability in resource constrained environments. Technical Report No. 2011/7, Charles University, 2011.

[49] P. Pedro and A. Burns. Schedulability analysis for mode changes in flexible real-time systems. In *Proceedings of 10th Euromicro Conference on Real-Time Systems (ECRTS'98)*, pages 172–179, June 1998.

[50] L. T. X. Phan, S. Chakraborty, and P. S. Thiagarajan. A multi-mode real-time calculus. In *Proceedings of 29th IEEE Real-Time Systems Symposium (RTSS'08)*, pages 59–69, December 2008.

[51] L. T. X. Phan, I. Lee, and O. Sokolsky. Compositional analysis of multi-mode systems. In *Proceedings of 22nd Euromicro Conference on Real-Time Systems (ECRTS'10)*, pages 197–206, July 2010.

[52] L. T. X. Phan, I. Lee, and O. Sokolsky. A semantic framework for mode change protocols. In *Proceedings of 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'11)*, pages 91–100, April 2011.

[53] T. Pop, F. Plasil, M. Outly, M. Malohlava, and T. Bures. Property networks allowing oracle-based mode-change propagation in hierarchical components. In *Proceedings of 15th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE'12)*, pages 93–102, June 2012.

[54] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26(2):161–197, 2004.

[55] L. Santinelli, G. Buttazzo, and E. Bini. Multi-moded resource reservations. In *Proceedings of 17th IEEE Real-Time and Embedded Technology*

and Applications Symposium (RTAS'11), RTAS '11, pages 37–46, April 2011.

[56] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems*, 1:243–264, 1989.

[57] J. A. Stankovic and K. Ramamritham, editors. *Tutorial: hard real-time systems*. IEEE Computer Society Press, 1989.

[58] N. Stoimenov, S. Perathoner, and L. Thiele. Reliable mode changes in real-time systems with fixed priority or EDF scheduling. In *Proceedings of Conference on Design, Automation and Test in Europe (DATE'09)*, pages 99–104, April 2009.

[59] N. Stoimenov, L. Thiele, L. Santinelli, and G. Buttazzo. Resource adaptations with servers for hard real-time systems. In *Proceedings of 10th ACM International Conference on Embedded software (EMSOFT'10)*, pages 269–278, October 2010.

[60] J. Templ. TDL specification and report. Technical report, Department of Computer Science, University of Salzburg, 2003.

[61] M. Tichy, S. Henkler, J. Holtmann, and S. Oberthür. Component story diagrams: A transformation language for component structures in mechatronic systems. In *Proceedings of 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER'08)*, pages 27–39, 2008.

[62] K. Tindell and A. Alonso. A very simple protocol for mode changes in priority preemptive systems. Technical report, Universidad Politécnica de Madrid, 1996.

[63] K. W. Tindell, A. Burns, and A. J. Wellings. Mode changes in priority pre-emptively scheduled systems. In *Proceedings of 13th IEEE Real-Time Systems Symposium (RTSS'92)*, pages 100–109, 1992.

[64] P. Meumeu Yomsi, V. Nelis, and J. Goossens. Scheduling multi-mode real-time systems upon uniform multiprocessor platforms. In *Proceedings of 15th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'10)*, September 2010.

Index