Linköping Studies in Science and Technology

Dissertation No. 1005

Developing Reusable and Reconfigurable Real-Time Software using Aspects and Components

by

Aleksandra Tešanović

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköping 2006

To Goran

Abstract

Our main focus in this thesis is on providing guidelines, methods, and tools for design, configuration, and analysis of configurable and reusable real-time software, developed using a combination of aspect-oriented and component-based software development. Specifically, we define a reconfigurable real-time component model (RTCOM) that describes how a real-time component, supporting aspects and enforcing information hiding, could efficiently be designed and implemented. In this context, we outline design guidelines for development of real-time systems using components and aspects, thereby facilitating static configuration of the system, which is preferred for hard real-time systems. For soft real-time systems with high availability requirements we provide a method for dynamic system reconfiguration that is especially suited for resource-constrained real-time systems and it ensures that components and aspects can be added, removed, or exchanged in a system at run-time. Satisfaction of real-time constraints is essential in the real-time domain and, for real-time systems built of aspects and components, analysis is ensured by: (i) a method for aspect-level worst-case execution time analysis; (ii) a method for formal verification of temporal properties of reconfigurable real-time components; and (iii) a method for maintaining quality of service, i.e., the specified level of performance, during normal system operation and after dynamic reconfiguration.

We have implemented a tool set with which the designer can efficiently configure a real-time system to meet functional requirements and analyze it to ensure that non-functional requirements in terms of temporal constraints and available memory are satisfied.

In this thesis we present a proof-of-concept implementation of a configurable embedded real-time database, called COMET. The implementation illustrates how our methods and tools can be applied, and demonstrates that the proposed solutions have a positive impact in facilitating efficient development of families of real-time systems.

Acknowledgements

With mixed feelings I approach the end of my PhD studies. On one hand I am very happy that I have finished my thesis and will be moving onto another stage in my life, and on the other hand I am really sad for not being a student anymore. I blame the sadness on Jörgen Hansson, my supervisor. He has not only provided guidance through technical issues, but also made me feel protected, supported, encouraged, and happy during my studies. Jörgen has, over the years, become one of my best friends and allies.

I was fortunate to do my research within the COMET project, a joint project between Linköping and Mälardalen University, working alongside Dag Nyström and Christer Norström. Collaboration with Dag had a catalytic effect on my research and has also showed me that great ideas are a product of spirited and witty discussions. Christer's feedback, questions, and suggestions have helped greatly in shaping my research.

ARTES, a network for real-time research and graduate education in Sweden, and CUGS, the Swedish computer science graduate school, have both provided financial support for my graduate studies, as well as given me the opportunity to meet new friends and exchange ideas. My last two years of studies would not be as interesting if I was not financed by the SAVE project. SAVE meetings and their members have shown me another, much more fun, dimension of research collaboration. ARTES and SAVE have been founded by the SSF, Swedish foundation for strategic research.

I extend my gratitude to past and present members of RTSLAB for providing an enjoyable and lively working environment. Simin Nadjm-Tehrani has supported me gracefully through the years of my PhD studies at RTSLAB. She is also responsible for the heaviness of formal analysis that, when I was not paying attention, sneaked into my thesis. The miracle woman and my friend, Anne Moe, has done wonders when it comes to helping me with administrative and other personal matters. Mehid Amirijoo and Thomas Gustafsson deserve praise for many gossip sessions and all the help they have unselfishly given me. United in supervision, we successfully traumatized many master's thesis students, of which a majority was working on the implementation of the COMET database platform. Calin Curescu has shown his real (soft) side and, in the last and most stressful days of thesis writing, helped out with teaching so I could focus on writing. Traveling

back and forth from the SAVE meetings would not be as fun without the company of the SAVE silver member, Jonas Elmqvist, and an RTSLAB fika would not be as enjoyable without the companionship of remaining RTSLAB members: Diana Szentiványi, Kalle Burbeck, Mikael Asplund, and Erik Kuiper.

I also thank the merry ESLAB bunch for their hospitality in the "ESLAB corridor" and, most importantly, for keeping me up to date with all spicy events that happen in their and neighboring labs.

My friends in Sweden and abroad have been great throughout all these years of my studies. I could always count on their kindness and encouragement.

Love and gratitude go to my parents. To my mother, for bringing up an ambitious and stubborn daughter in the world of men, and my father, who's patience and understanding has helped me in many situations. I also thank them for getting me a great younger sister. It is a bless having a younger sibling: you at the same time get a friend, admirer, and a follower, who is, for the greater part of her life, blind to your many flaws.

I think that we are all constructed out of two related opposites, positive pole and negative pole, and that the two need to be in balance for the person to be complete. I would have been incomplete without my husband, Goran. He has been and continues to be my positive pole. Without his unconditional love and encouragement I would not have been able to finish this thesis. He is Neo of my Matrix.

Hvala,
Aleksandra Tešanović

Linköping, February 2006

List of Publications

This work is done as a part of the COMET research project, a joint project between Linköping University and Mälardalen University. The principal investigators of the project are Jörgen Hansson (Linköping University) and Christer Norström (Mälardalen University). The doctoral students in the project are Aleksandra Tešanović (Linköping) and Dag Nyström (Mälardalen). The COMET project has over the years evolved and involved collaboration with additional senior researchers (Mikael Nolin, Mälardalen University, and Simin Nadjm-Tehrani, Linköping University), and doctoral students (Mehdi Amirijoo and Thomas Gustafsson, both at Linköping University). Furthermore, several master's students have completed their thesis work by implementing or investigating specific issues related to the COMET project, and thereby also contributed to the research presented in this thesis; their contributions are recognized in the author lists of the papers produced within the project.

The COMET project has resulted in the following published papers.

Publications Included in the Thesis

The contributions of this thesis are based on the following publications (the list contains descriptions of the content of each paper, relating it to the other papers, and the role of the thesis author in contributing to the paper).

Data Management Issues in Vehicle Control Systems: a Case Study, Dag Nyström, Aleksandra Tešanović, Christer Norström, Jörgen Hansson, and Nils-Erik Bånkestad, In Proceedings of the 14th IEEE Euromicro International Conference on Real-Time Systems (ECRTS'05), pages 249-256, IEEE Computer Society, June 2002.

This paper presents a case study of a class of embedded hard real-time control applications in the vehicular industry that, in addition to meeting transaction and task deadlines, emphasize data validity requirements. The paper also presents how a database could be integrated into the studied application and how the database management system could be designed to suit this particular class of systems.

The paper was written based on the industrial stay at Volvo Construction Equipment Components AB, Sweden. The industrial stay, and thereby the wri-

ting of this case study paper, was made possible by Nils-Erik Bånkestad. Aleksandra Tešanović and Dag Nyström investigated two different real-time systems, one each. Additionally, Tešanović studied the impact of current data management in both systems.

Integrating Symbolic Worst-Case Execution Time Analysis into Aspect-Oriented Software Development, Aleksandra Tešanović, Dag Nyström, Jörgen Hansson, and Christer Norström, OOPSLA 2002 Workshop on Tools for Aspect-Oriented Software Development, November 2002.

This workshop paper presents an initial proposal for providing support for predictable aspect-oriented software development by enabling symbolic worst-case execution time analysis of aspect-oriented software systems.

Aleksandra Tešanović developed a way of representing temporal information of aspects and an algorithm that enables worst-case execution time analysis of aspect-oriented systems.

Towards Aspectual Component-Based Development of Real-Time Systems, Aleksandra Tešanović, Dag Nyström, Jörgen Hansson, and Christer Norström, In Proceeding of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2003), Lecture Notes in Computer Science, vol. 2968, pp. 558-577, Springer-Verlag, 2003.

This paper introduces a novel concept of aspectual component-based real-time system development. The concept is based on a design method that assumes decomposition of real-time systems into components and aspects, and provides a real-time component model that supports the notion of time and temporal constraints, and space and resource management constraints.

The main ideas and contributions of the paper are developed by Aleksandra Tešanović.

Aspect-Level Worst-Case Execution Time Analysis of Real-Time Systems Compositioned Using Aspects and Components, Aleksandra Tešanović, Dag Nyström, Jörgen Hansson, and Christer Norström, In Proceeding of the 27th IFAC/IFIP Workshop on Real-Time Programming, Elsevier Science Ltd, May 2003.

This paper extends and refines the method for analyzing the worst-case execution time of a real-time system composed using aspects and components, introduced in the OOPSLA 2002 workshop paper. In addition of presenting the aspect-level WCET analysis of components, aspects and the composed real-time system, the paper also presents design guidelines for the implementation of components and aspects in a real-time environment.

The paper is a successor with extensions to the workshop paper presented at the OOPSLA 2002 Workshop on Tools for Aspect-Oriented Software Development.

Hence, in this paper as well, Aleksandra Tešanović developed the main ideas and contributions.

Aspect-Level WCET Analyzer: a Tool for Automated WCET Analysis of a Real-Time Software Composed Using Aspects and Components, Aleksandra Tešanović, Jörgen Hansson, Dag Nyström, Christer Norström, and P. Uhlin, In Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis (WCET 2003), July 2003.

This paper presents a tool that implements methods for aspect-level WCET analysis developed and presented in two previous papers on this topic.

The main ideas and contributions of the paper are developed by Aleksandra Tešanović.

Structuring Criteria for the Design of Component-Based Real-Time Systems, Aleksandra Tešanović and Jörgen Hansson, In Proceedings of the IADIS International Conference on Applied Computing 2004, pp. I401-I411, IADIS Press, March 2004.

In this paper we identify the criteria a design method for component-based real-time systems should fulfill to enable efficient, reuse-oriented, development of reliable and configurable real-time systems. The criteria include a real-time component model, separation of concerns in real-time systems via aspects, and support for configuration and analysis of real-time software.

The main ideas and contributions of the paper are developed by Aleksandra Tešanović.

Aspects and Components in Real-Time System Development: Towards Reconfigurable and Reusable Software, Aleksandra Tešanović, Dag Nyström, Jörgen Hansson, and Christer Norström, Journal of Embedded Computing, Volume 1, October 2004.

This paper is an extended version of the paper that was published at RTCSA 2003 conference.

The main ideas and contributions of this paper are developed by Aleksandra Tešanović.

Empowering Configurable QoS Management in Real-Time Systems, Aleksandra Tešanović, Mehdi Amirijoo, Mikael Björk, and Jörgen Hansson, In Proceedings of the 4th ACM SIG International Conference on Aspect-Oriented Software Development (AOSD'05), pp. 39-50, ACM Press, March 2005.

In this paper we present a method for developing reconfigurable feedback-based QoS management for real-time systems, denoted Re-QoS. The Re-QoS method ensures reconfiguration of QoS management by having components and aspects as constituents of the QoS management architecture and policies.

Aleksandra Tešanović contributed with the initial idea and the solution when it comes to reconfigurability of QoS via aspects and components.

Development Environment for Configuration and Analysis of Embedded Real-Time Systems, Aleksandra Tešanović, Peng Mu, and Jörgen Hansson, In Proceedings of the 4th International AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS'05), March 2005.

In this paper we present a tool set that provides developers of real-time systems with support for building and configuring a system out of components and aspects, as well as real-time analysis of the configured system.

The main ideas and contributions of this paper are developed by Aleksandra Tešanović.

Modular Verification of Reconfigurable Components, Aleksandra Tešanović, Simin Nadjm-Tehrani, and Jörgen Hansson, Book chapter in Component-Based Software Development for Embedded Systems - An Overview on Current Research Trends, Lecture Notes in Computer Science, vol. 3778, pp. 59-81, Springer-Verlag, 2006.

This book chapter presents a method for modular verification of reconfigurable real-time components. The method enables proving that the reconfiguration of components via aspect weaving provides expected functional and temporal behavior in the reconfigured component.

The main ideas and contributions of this book chapter are developed by Aleksandra Tešanović.

Ensuring Real-Time Performance Guarantees in Dynamically Reconfigurable Embedded Systems, Aleksandra Tešanović, Mehdi Amirijoo, Daniel Nilsson, Henrik Norin, and Jörgen Hansson, In Proceedings of the IFIP International Conference on Embedded And Ubiquitous Computing (EUC'05), Springer-Verlag, December 2005.

In this paper we present a method for dynamic, QoS-aware, reconfiguration of real-time systems. This work is an extension of the work on enabling development of real-time systems using aspects and components.

The main initial idea on dynamic reconfiguration was developed by Aleksandra Tešanović.

Publications by the Author, not Included in the Thesis

Finite Horizon QoS Prediction of Reconfigurable Soft Real-Time Systems, Mehdi Amirijoo, Aleksandra Tešanović, Torgny Andersson, Jörgen Hansson, and Sang H. Son, in submission.

Separating Active and On-Demand Behavior of Embedded Systems into Aspects, Aleksandra Tešanović, Thomas Gustafsson, and Jörgen Hansson, In Proceedings of the Workshop on Non-functional Properties of Embedded Systems, March 2006.

Application-Tailored Database Systems: a Case of Aspects in an Embedded Database, Aleksandra Tešanović, Ke Sheng, and Jörgen Hansson, In Proceedings of the 8th IEEE International Database Engineering and Applications Symposium (IDEAS'04), IEEE Computer Society, July 2004.

COMET: A Component-Based Real-Time Database for Automotive Systems, Dag Nyström, Aleksandra Tešanović, Mikael Nolin, Christer Norström, and Jörgen Hansson, In Proceedings of the Workshop on Software Engineering for Automotive Systems at 26th International Conference on Software engineering (ICSE'04), IEEE Computer Society Press, May 2004.

Pessimistic Concurrency-Control and Versioning to Support Database Pointers in Real-Time Databases, Dag Nyström, Mikael Nolin, Aleksandra Tešanović, Christer Norström, and Jörgen Hansson, In Proceedings of the 16th IEEE Euromicro Conference on Real-Time Systems (ECRTS'04), Sicily, Italy, IEEE Computer Society Press, June 2004.

Database Pointers: a Predictable Way of Manipulating Hot Data in Hard Real-Time Systems, Dag Nyström, Aleksandra Tešanović, Christer Norström, and Jörgen Hansson, In Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2003), Lecture Notes in Computer Science, vol. 2968, Springer-Verlag, 2003.

Contents

I Preliminaries **1**

1 Introduction **3**
1.1 Motivation . . . 3
1.2 Research Challenges . . . 7
1.3 Research Contributions . . . 8
1.4 Thesis Outline . . . 9

2 Basic Concepts in Software Engineering and Real-Time **11**
2.1 Basic Notions in Software Engineering . . . 11
2.2 Basic Notions in Real-Time Computing . . . 21
2.3 Ensuring Real-Time Performance Guarantees . . . 25
2.4 Embedded Real-Time Database Systems . . . 34

3 Developing Reconfigurable Real-Time Systems **41**
3.1 Requirements . . . 41
3.2 Existing Approaches . . . 51
3.3 Goals . . . 56

II Aspectual Component-Based Real-Time System Development **59**

4 Enabling System Reconfiguration with ACCORD **61**
4.1 ACCORD Overview . . . 61
4.2 Aspects in Real-Time Systems . . . 65
4.3 Real-Time Component Model . . . 68
4.4 Aspect Packages . . . 84
4.5 Static Reconfiguration . . . 89
4.6 Dynamic Reconfiguration . . . 89

5 Ensuring Real-Time Performance Guarantees **95**
5.1 Overview . . . 95

5.2 Aspect-Level WCET Analysis 97
5.3 Components and their Schedulability 105
5.4 Feedback-Based QoS Management 106
5.5 Formal Analysis of Aspects and Components 109

6 Tool Support and Evaluation **125**
6.1 Development Environment 125
6.2 ACCORD Evaluation 131

III Component-Based Embedded Real-Time Database System **135**

7 Data Management in Embedded Real-Time Systems **137**
7.1 A Case Study: Vehicle Control Systems 137
7.2 Data Management Requirements 142
7.3 Observations 145

8 COMET Database Platform **147**
8.1 COMET Decomposition 147
8.2 COMET Components 151
8.3 Concurrency Control Aspect Package 157
8.4 Index Aspect Package 163
8.5 QoS Aspect Package 165

9 COMET Configuration and Analysis **173**
9.1 COMET Configurations 173
9.2 Static and Dynamic COMET Configuration 178
9.3 Performance Evaluation 184
9.4 Experience Report 190

IV Epilogue **195**

10 Related Work **197**
10.1 Component-Based Real-Time Systems 197
10.2 Component-Based Database Systems 202
10.3 Aspect-Oriented Database Systems 207
10.4 Formal Approaches for Aspects and Components 208

11 Conclusions **211**
11.1 Summary 211

11.2 Future Work 213

A Abbreviations **215**

Bibliography **217**

Index **234**

List of Figures

1.1 Examples of real-time computing applications 4
1.2 System assembly with components and aspects 5

2.1 Components and interfaces 13
2.2 An example of an aspect definition 16
2.3 A typical pointcut syntax 16
2.4 An example of a pointcut definition 16
2.5 The behavior of a before, after, and around advice 17
2.6 An example of the advice definition 17
2.7 Illustration of aspect weaving (a) one aspect and one white box component and (b) multiple aspects woven into several white box components. 18
2.8 Classes of component-based systems 19
2.9 Typical temporal characteristics of a task in a real-time system . . 22
2.10 The `power` function 26
2.11 An architecture of the real-time system using feedback control structure 28
2.12 Formal analysis of real-time systems using model checking 30
2.13 An example of a simple timed automaton. 31
2.14 Expressing properties in TCTL logic. 33

3.1 An example of the component-to-task relationship 44

4.1 An overview of real-time system development via ACCORD constituents 62
4.2 Classification of aspects in real-time systems 66
4.3 A real-time component model (RTCOM) 68
4.4 The functional part of RTCOM 70
4.5 An example of (a) allowed, and (b) not allowed relationship among the operations 71
4.6 An example of the relationship of operations and mechanisms in a component 71

4.7 An illustration of the structure of an aspect that invasively changes the behavior of a component 72
4.8 An example of component reconfiguration via aspect weaving . . . 74
4.9 The functional part of the linked list component 75
4.10 The `listPriority` application aspect 76
4.11 The resulting woven component 77
4.12 A general specification template for run-time aspects 78
4.13 An example of the specification template for the run-time part of an application aspect 79
4.14 A specification of the run-time aspect of the linked list component 80
4.15 A specification of the run-time aspect of the priority list application aspect 80
4.16 Different types of interfaces defined within RTCOM 81
4.17 The functional provided and required interface supported by RTCOM 82
4.18 The configuration interface of RTCOM 83
4.19 The composition interface of RTCOM 83
4.20 A QoS aspect package 87
4.21 A real-time system where QoS aspect package is used for configuring QoS management 88
4.22 Static (re)configuration of a system 90
4.23 Dynamic configuration of a system 92
4.24 Reconfiguration of a component 93
4.25 An example of the relation between tasks and components. 93

5.1 An analysis part of the ACCORD development process 96
5.2 An overview of the automated aspect-level WCET analysis process 97
5.3 The aspect-level WCET specification of the linked list component . 99
5.4 The aspect-level WCET specification of the `listPriority` aspect 100
5.5 Main constituents of aspect-level WCET analysis 100
5.6 An overview of the aspect-level WCET analysis life cycle 104
5.7 Temporal schedulability analysis within ACCORD 105
5.8 Fluctuations of performance under reconfiguration 107
5.9 Dynamic system reconfiguration with support of the feedback control 108
5.10 Examples of timed automata specifying (a) the transaction manager component and (b) the locking advice 111
5.11 An example of preservation of clock constraints 113
5.12 An example of a reconfigured component 114
5.13 One possible execution trace of for the transaction manager component and the property $EF(end \wedge x < 11)$ 123

6.1 ACCORD Development Environment 126
6.2 The editing window in ACCORD-ME 128
6.3 An implementation of the run-time aspect of a component in XML 130

7.1 The overall architecture of a vehicle control system 137
7.2 The structure of an ECU 138
7.3 The architecture of the VECU 140
7.4 The architecture of the IECU 141

8.1 Decomposition of COMET into components 148
8.2 The outlook of the functional part of the COMET components . . 153
8.3 The execution steps of a transaction in COMET 156
8.4 The locking manager component in COMET 160
8.5 The structure of a locking-based concurrency control aspect 161
8.6 The GUARD policy aspect 164
8.7 The QAC connector aspect 168
8.8 The QAC utilization policy aspect 169
8.9 The missed deadline monitor aspect 170
8.10 The missed deadline control aspect 171
8.11 The scheduling strategy aspect 171

9.1 Creating a family of real-time systems from the COMET QoS aspect package 176
9.2 Requirement-based configuration of the real-time database system 180
9.3 The snapshot of ACCORD-ME when doing analysis on a real-time system configuration 181
9.4 Dynamic COMET configuration 184
9.5 Deadline miss ratio as a function of load 185
9.6 Deadline miss ratio when components are replaced 189

10.1 The 2K middleware architecture 199
10.2 Embedded system development in VEST 200
10.3 The Oracle extensibility architecture 203
10.4 The Universal Data Access (UDA) architecture 205
10.5 The KIDS subsystem architecture 206

List of Tables

2.1 Examples of quality attributes [180] 12
2.2 Concurrency control methods in real-time database systems 36
2.3 The index concurrency control lock compatibilities 39

3.1 Criteria for evaluation of design approaches 52

5.1 Aspect-level WCET specifications of aspects and components . . . 99

6.1 Evaluation criteria for ACCORD 132

7.1 Data management characteristics for the systems 143

8.1 Crosscutting effects of different application aspects on COMET components 151
8.2 The lock compatibility for HP-2PL concurrency control policy . . . 158
8.3 The components that are crosscut and used by lock-based concurrency control methods 162
8.4 The utilization transaction model 167

9.1 Relationship between different parts of the concurrency control and the index aspect package and various COMET configurations . 175
9.2 Relationship between different parts of the QoS package and various COMET QoS configurations 177
9.3 Measured attributes of transactions 183
9.4 Execution times for transactions 187
9.5 Reconfiguration times with one transactions running 187
9.6 Reconfiguration times in COMET with a 300% load 188

Part I

Preliminaries

Chapter 1

Introduction

In this chapter we motivate the need for using new software engineering techniques in development of real-time systems, and formulate our research goals. We also present the main contributions of the thesis, addressing the identified goals. Finally, we outline the structure of the thesis.

1.1 Motivation

A large majority of computational activities in modern computing systems are performed within embedded and real-time systems. Figure 1.1 shows some of the application areas where real-time systems can be found, e.g., vehicle systems, traffic control, and medical equipment.

Successful deployment of real-time systems greatly depends on low development costs, a short time to market, and high degree of configurability [161]. Component-based software development (CBSD) [169] is a modern software engineering technique that enables systems to be assembled from a pre-defined set of components explicitly developed for multiple usages (see left part of figure 1.2 for an illustration of system configuration out of components). Thus, the introduction of CBSD into real-time and embedded systems development offers significant benefits, namely:[1]

- ❏ configuration of software for a specific application using components from the component library, thus, reducing software complexity as components can be chosen to provide exactly the functionality needed by the application;

[1] We use the terms real-time software and real-time system interchangeably throughout the thesis. Both of these denote a real-time software program running on a run-time platform, i.e., an operating system and the hardware, and performing a specific task for an application, i.e., a user of the program.

Figure 1.1: *Examples of real-time computing applications*

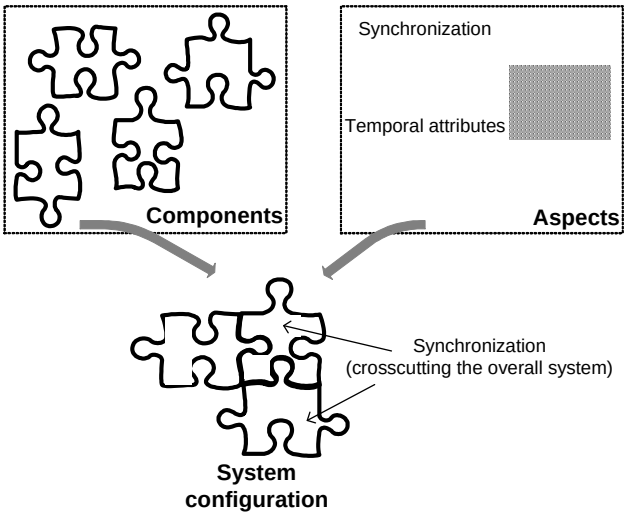


Figure 1.2: *System assembly with components and aspects*

- ❏ rapid development and deployment of real-time software as many software components, if properly designed and verified, can be reused in many applications; and
- ❏ evolutionary design as components can be replaced or added to the system, which is appropriate for complex embedded real-time systems that require continuous hardware and software upgrades.

However, there are features of real-time systems that crosscut the overall system, e.g., synchronization, memory optimization, and temporal attributes, and cannot be encapsulated in a component with a well-defined interface. This phenomenon is depicted in the system configuration in figure 1.2 using synchronization as an example. Aspect-oriented software development (AOSD) [81] has emerged as a new principle for software development that provides an efficient way of modularizing crosscutting concerns in software systems. AOSD allows encapsulating crosscutting concerns of a system in "modules", called aspects (see figure 1.2). Aspect are added to the system in a process called aspect weaving. Introducing AOSD into real-time system development would ensure that:

- ❏ crosscutting concerns of a real-time system and its constituting components can be developed independently with clear interfaces toward components with which aspects should be combined;
- ❏ a component or an entire software composed of components can be fine-tuned for a specific application with which the software is to be integrated; and

- ❏ a real-time software system can be extended with new functional and non-functional features by defining and integrating new aspects into an already existing system.

Consequently, using AOSD in the real-time and embedded system domain would reduce the complexity of the system design and development, and provide means for a structured and efficient way of designing and implementing crosscutting concerns in real-time software.

As argued, there are strong motivations for using the AOSD principle of separation of concerns, as well as the CBSD notion of the system assembly of pre-defined components in the real-time software development. The integration of the two disciplines, CBSD and AOSD, into the real-time domain would enable:

- ❏ efficient system configuration from components and aspects from the library based on the application requirements; and

- ❏ easy reconfiguration of components and/or a system for a specific application, i.e., reuse context, by changing the behavior (code) of the component via aspect weaving.

This results in enhanced flexibility and reusability of real-time software through the notion of system and component reconfigurability. However, applying main ideas from both AOSD and CBSD into the real-time domain is not straightforward, for several reasons.

Firstly, CBSD assumes a component to be a black box, where internals of the component are not visible or accessible to a component user, i.e., a system designer or other components in the system. In contrast, AOSD promotes white box components, where the code of a component is fully accessible and can be changed by the component user. Thus, to utilize benefits of both technologies, we need to provide support for aspect weaving into component code, while preserving information hiding of a component to the largest degree possible.

Secondly, systems developed based on the CBSD principles are normally dynamically reconfigurable, implying that components can be added, removed, or exchanged in a system on-line. AOSD, on the other hand, focuses primarily on combining aspects and components into a system configuration statically, i.e., off-line. In real-time systems residing in time-critical applications and requiring conformance to strict performance specifications, static configuration is necessary. In real-time applications where performance can be traded for availability, dynamic reconfiguration is preferable. Namely, reconfiguring a system on-line is desirable for embedded real-time systems that require continuous hardware and software upgrades in response to technological advancements, environmental change, or alteration of system goals during system operation [44, 166]. For example, small embedded real-time sensor-based control systems should be designed and developed such that software resources, e.g., controllers and device drivers, change on the fly. Hence, a reconfiguration mechanism, enabling adding, removing, and

exchanging components on-line, is needed to ensure that the software is updated without interrupting the execution of the system. It would, therefore, be beneficial to ensure support for both static and dynamic reconfiguration of a real-time system. Enabling dynamic reconfiguration implies providing mechanisms for on-line exchange of components and aspects especially suited for resource-constrained environments (as most real-time systems reside in such environments). To ensure static configuration, the process of system design and development has to be defined and preferably accompanied by tools to provide guidelines to real-time system developers and, thereby, support them in system development.

Finally, the development process of real-time systems has to be based on a software technology that supports predictability in the time domain. Hence, we need to provide methods and tools for satisfying real-time performance of the system composed of components and aspects. Namely, analysis of the static temporal behavior, e.g., worst-case execution time, of individual aspects and components, as well as resulting system configurations, is necessary for statically configured systems. Furthermore, to facilitate meaningful dynamic reconfiguration in a real-time system, methods for guaranteeing real-time performance before and after the reconfiguration are essential.

1.2 Research Challenges

Emergence of new requirements on the cost-effective development of real-time systems focusing on reuse and reconfiguration has given rise to new research challenges in software engineering. One promising approach, as we show in this thesis, is to merge CBSD and AOSD and apply them to real-time system development. This calls for novel solutions for design and development of reconfigurable real-time systems, including:

1. a real-time component model enforcing information hiding and component reuse, while enabling tailoring of components for a particular application;

2. support for static and dynamic (re)configuration of a real-time system assembled out of components and aspects;

3. methods for enforcing satisfaction of real-time performance requirements in both dynamically and statically reconfigurable real-time systems; and

4. tools for configuring and analyzing a system and its constituting components and aspects.

These solutions would be conducive to the reusability of components and aspects in a variety of real-time applications.

1.3 Research Contributions

The main research contributions of this thesis are as follows.

1. A reconfigurable real-time component model (RTCOM) that describes how a real-time component, supporting different aspects and enforcing information hiding, could efficiently be designed and implemented.
2. Support for static and dynamic reconfiguration of a real-time system in terms of:
   (a) Design guidelines for development of real-time systems using components and aspects, which prescribe that a real-time system design should be carried out in the following sequential phases: (i) decomposition of the real-time system into a set of components, followed by (ii) decomposition into a set of aspects, and (iii) implementation of components and aspects based on RTCOM.
   (b) A method for dynamic system reconfiguration suited for resource-constrained real-time applications ensuring that components and aspects can be added, removed, or exchanged in the system at runtime. Thus, in addition to traditional static reconfiguration, we support dynamic reconfiguration of a system.
3. Methods for ensuring satisfaction of real-time constraints, namely:
   (a) A method for aspect-level worst-case execution time analysis of real-time systems assembled using aspects and components, which is performed at a system composition time.
   (b) A method for formal verification of temporal properties of reconfigurable real-time components that enables (i) proving temporal properties of individual components and aspects, and (ii) proving that reconfiguration of a component via aspect weaving preserves expected temporal behavior in the reconfigured component.
   (c) A method for reconfigurable quality of service that enables configuring quality of service in real-time systems in terms of desired performance metric and performance level based on the underlying application requirements. The method ensures that the specified level of performance is maintained during system operation and after reconfiguration.

We have implemented a tool set that provides developers of real-time systems with support for configuration and analysis of a system assembled using components and aspects. This way a real-time system can efficiently be configured to meet functional requirements and analyzed to ensure that non-functional requirements are also fulfilled. The analysis tools represent an automation of the analysis methods from (3).

We use the term ACCORD, which stands for aspectual component-based real-time system development, to collectively denote the above mentioned research contributions and tools and to indicate that these solutions, in addition of being used in isolation, can be used together to facilitate efficient development of reconfigurable and reusable real-time software.

In this thesis we also present a proof-of-concept implementation of a component-based embedded real-time (COMET) database. Using the COMET example, we demonstrate the applicability of the proposed methods and tools for building reconfigurable and reusable real-time systems. Requirements for the COMET database have been extracted from requirements on data management identified in the vehicular industry [124]. As a result of implementing the database platform, we collected experiences in using available CBSD and AOSD programming technologies in the real-time domain and we report on these. The experiences are valuable for current and future implementors of real-time systems that would like to use components and aspects in the system development.

1.4 Thesis Outline

The thesis is organized into four parts as follows.

Part I: Preliminaries. In chapter 2 we present the main terminology used throughout the thesis and introduce advanced topics in real-time computing, including methods for ensuring real-time performance and efficient data management. In chapter 3 we provide an extensive problem description that serves as an in-depth motivation for the work presented in this thesis.

Part II: Aspectual Component-Based Real-Time System Development. In this part of the thesis we give a detailed description of ACCORD constituents. In chapter 4 we introduce aspects, components, and aspect packages as system constituents, and elaborate on the way static and dynamic system reconfiguration is performed. Methods for ensuring real-time performance guarantees of resulting configured systems are presented in chapter 5. These include the method for aspect-level worst-case execution time analysis, the method for ensuring quality of service under dynamic reconfiguration, and formal analysis of components woven with aspects. Finally, in chapter 6 we present automated tool support for development and analysis of reconfigurable real-time systems, and evaluate ACCORD against the requirements placed on development of reusable and reconfigurable real-time systems previously identified in chapter 3.

Part III: Component-Based Embedded Real-Time Database System In this part we present an example of utilizing ACCORD to develop COMET. The goal with the COMET platform is to enable development of various database configurations for specific embedded and real-time applications. The type of

requirements placed on COMET development is best illustrated by an example of one of the COMET targeting application areas: vehicle control systems. Therefore, in chapter 7 we present a study of data management in two different real-time systems developed at Volvo Construction Equipment Components AB, Sweden. We then elaborate in chapter 8 on how COMET was developed using ACCORD and discuss various components, aspects, and aspect packages in the COMET library. In chapter 9 we present possible ways of configuring COMET using the proposed tools, and evaluate performance of COMET. In this chapter we also present experiences gathered while implementing the platform.

Part IV: Epilogue This final part of the thesis contains the related work and conclusions. Namely, the related work is discussed in chapter 10, and the thesis is concluded with chapter 11 where we summarize the work, give conclusions, and outline possible directions for the future work.

Chapter 2

Basic Concepts in Software Engineering and Real-Time

This chapter introduces the terminology related to software engineering and real-time systems adopted in the thesis. Furthermore, we discuss how to ensure real-time performance of a software system and achieve efficient management of data.

2.1 Basic Notions in Software Engineering

Massive amounts of software have been developed since the development of the first software program. Moreover, the complexity of developed software has proportionally increased. As a result, different software engineering techniques devoted to assuring efficient design, development, and use of software have emerged [21] (see figure 2.8 for an evolutionary pyramid of the software engineering techniques). In this section we introduce software engineering terminology adopted in the thesis. We start with more general software engineering notions and then focus on the terminology of component-based software development and aspect-oriented software development.

Software architecture. Every software system has an architecture, which may or may not be explicitly modeled [114]. The software architecture represents a high level abstraction of a system, where a system is described as a collection of interacting components [5]. The definition of the software architecture adopted in this thesis is [22]:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software com-

Quality attribute	Description
Performance	The capacity of a system to handle data or events.
Reliability	The probability of a system working correctly over a given period of time.
Safety	The property of the system that it will not endanger human life or the environment.
Security	The ability of a software system to resist malicious intended actions.
Availability	The probability of a system functioning correctly at any given time.
Testability	The ability to easily prove correctness of a system by testing.
Reusability	The extent to which the architecture or its parts can be reused.
Portability	The ability to move a software system to a different hardware and/or software platform.
Maintainability	The ability of a system to undergo evolution and repair.
Modifiability	Sensitivity of the system to changes in one or several components.

Table 2.1: *Examples of quality attributes [180]*

ponents, the externally visible properties of those components, and the relationship among them.

Thus, the software architecture enables decomposition of the system into well-defined components and their interconnections, and consequently it provides means for building complex software systems [114]. The word component in this context denotes a generic building block of the system and no specific characteristics of a component are assumed.

Quality attributes. Those attributes of a system that are relevant from the software engineering perspective, e.g., maintainability and reusability, and those that represent quality of the system in operation, e.g., performance, reliability, robustness, and fault tolerance, are called quality attributes of a system. Quality attributes are often regarded as non-functional (or extra-functional) system properties. Table 2.1 presents examples of common quality attributes.

2.1.1 Component-Based Software Development

The need for transition from monolithic to configurable systems has emerged from problems in traditional software development, such as high development costs,

inadequate support for long-term maintenance and system evolution, and often unsatisfactory quality of software [37]. Component-based software development (CBSD) is a development paradigm that enables this transition by allowing systems to be assembled from a pre-defined set of components explicitly developed for multiple usages.

Software Components. The core of CBSD are software components. However, many definitions and interpretations of a component exist. In general, within software architecture, a component is considered to be a unit of composition with explicitly specified interfaces and quality attributes [37]. In systems where COM [48] is used as a component framework, a component is generally assumed to be a self-contained binary package with precisely defined standardized interfaces [121]. Similarly, in the CORBA component framework [127], a component is assumed to be a CORBA object with standardized interfaces. A component can be also viewed as a software artifact that models and implements a well-defined set of functions, and has well-defined, but not necessarily standardized, component interfaces [55]. Hence, there is no common definition of a component for every component-based system. The definition of a component depends on the architectural assumptions and the way it is to be reused in the system. However, all component-based systems have one common fact: *components are for composition* [169].

While frameworks and standards for components today primarily focus on CORBA, COM, or JavaBeans, the need for component-based development has also been identified in the area of operating systems (OSs). The aim is to facilitate OS evolution without endangering legacy applications and provide better support for distributed applications [63, 115].

Component interfaces. Common for all types of components, independent of their definition, is that they communicate with its environment through well-

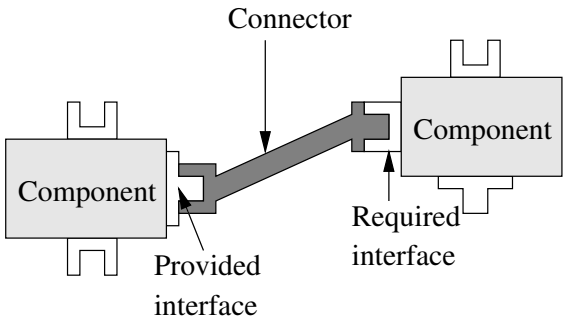


Figure 2.1: *Components and interfaces*

defined interfaces, e.g., interfaces in COM and CORBA are defined in an interface definition language (IDL), Microsoft IDL and CORBA IDL, respectively. Note that in an interface, only a collection of operations implemented in a component is normally listed together with the descriptions and the protocols of these operations [51]. Having well-defined interfaces ensures that an implementation part of a component can be replaced in the system without changing the interface.

Components typically have more than one interface. For example, a component may have three types of interfaces: provided, required, and configuration interfaces [37]. Provided and required interfaces are intended for the interaction with other components, whereas configuration interfaces are intended for use by the user of the component, i.e., a software engineer (developer) who is constructing a system using reusable components. Each interface provided by a component is, upon instantiation of the component, bound to one or more interfaces required by other components. The component providing an interface may service multiple components, i.e., there can be a one-to-many relation between provided and required interfaces.

Component connectors. When using components in system composition there might be a syntactic mismatch between provided and required interfaces, even when the semantics of the interfaces match. This requires adaptation of one or both of the components or an adapting connector to be used between components to perform the translation between them (see figure 2.1).

Black box component. Independently of an application area, a software component is normally considered to have the *black box* property [61, 55], i.e., each component sees only interfaces of other components, thus, internal state and attributes of the component are strongly encapsulated.

Component domain. Every component implements some field of functionality, i.e., a domain [37]. Domains can be hierarchically decomposed into lower level domains, e.g., the domain of communication protocols can be decomposed into several layers of protocol domains as in the OSI model. This means that components can also be organized hierarchically, i.e., a component can be composed out of subcomponents. In this context, two conflicting forces need to be balanced when designing a component. First, small components cover small domains and are likely to be reused, as it is likely that such component would not contain large parts of functionality not needed by the system. Second, large components give more leverage than small components when reused, since choosing the large component for the software system would reduce the cost associated with the effort required to find the component, and analyze its suitability for a certain software product [37]. Hence, when designing a component, a designer should find the balance between these two conflicting forces, as well as actual demands of the system in the area of component application.

System configuration (assembly). A system assembled using components is referred to as a system configuration or an assembly. Often, an assembly of components can be viewed as one composite component implementing a large domain.

2.1.2 Aspect-Oriented Software Development

Aspect-oriented software development (AOSD) has emerged as a new principle for software development and it is based on the notion of separation of concerns [81]. Typically, an aspect-oriented implementation of a software system consists of components and aspects, which are combined into a resulting system using an aspect weaver.

White box components. The AOSD community uses the term component to denote a traditional software module, e.g., program, function, or method, completely accessible by the users. Hence, components in AOSD do not enforce information hiding and are fully open to changes and modifications of their internal structure. This is in contrast to black box components used for configuration of component-based software. To preserve original terminology from the AOSD community, and at the same time distinguish between the components used in component-based software and the traditional modules used in aspect-oriented software, we refer to the latter as components the with white box property or simply white box components. A white box component is typically written in a standard programming language, such as C, C++, or Java.

Aspects. A property of a system that affects its performance or semantics, and that crosscuts the functionality of the system, is commonly considered to be an aspect [81]. Aspects of software such as persistence and debugging can be described separately and exchanged independently of each other without disturbing the modular structure of the system.

Aspect language. Aspects are written in an aspect language that corresponds to the language in which the white box components are written, e.g., AspectC [47] for components written in C, AspectC++ [158] for components written in C++ or C, and AspectJ [182] for Java-based components.[1] The way an aspect can be defined in AspectC++ is illustrated in figure 2.2.

Pointcuts and join points. As can be seen from figure 2.2, the aspect declaration consists of advices and pointcuts. A *pointcut* in an aspect language consists of one or more join points, and is described by a pointcut expression. A *join point* refers to a point in the white box component code where aspects should be weaved, e.g., a method, a type (struct or union). The syntax of the pointcut

[1] All existing aspect languages are conceptually very similar to AspectJ.

```
aspect printID{
  pointcut getLockCall(int lockId)=
    call("void getLock(int)")&&args(lockId);
  advice getLockCall(lockId):
    void after (int lockId){
      cout<<"Lock requested is"<<lockId<<endl;}
}
```

Figure 2.2: *An example of an aspect definition*

```
p ∈ {pointcuts}
m ∈ {function|method_signatures}
v ∈ {identifiers_with_types}
p::=call(m)|execute(m)|target(v)|args(v)|p&&p|p||p|!p
```

Figure 2.3: *A typical pointcut syntax*

is illustrated in figure 2.3. The first two pointcuts (`call` and `execute`) match join points that have the same signature[2] as the join point m. While the `call` pointcut refers to the point in code where some function/method is called, the `execute` pointcut refers to the execution of the join point (i.e., after the call has been made and a function started to execute). The pointcuts `target` and `args` match any join point that has values of a specified type; in this case v. Operators &&, ||, and ! logically combine or negate pointcuts.

Figure 2.4 shows the definition of a named pointcut `getLockCall`, which refers to all calls to the function `getLock()` and exposes a single integer argument to that call.

```
pointcut getLockCall(int lockId)=
  call("void getLock(int)"&&args(lockId);
```

Figure 2.4: *An example of a pointcut definition*

Advices. A declaration used to specify code that should run when the join points are reached is referred to as an *advice*. Different kinds of advices can be declared, as follows:

- *before advice*, which is executed before the join point,

[2]A function or a method signature refers to a description that consist of the function name, its type, and input/output parameters.

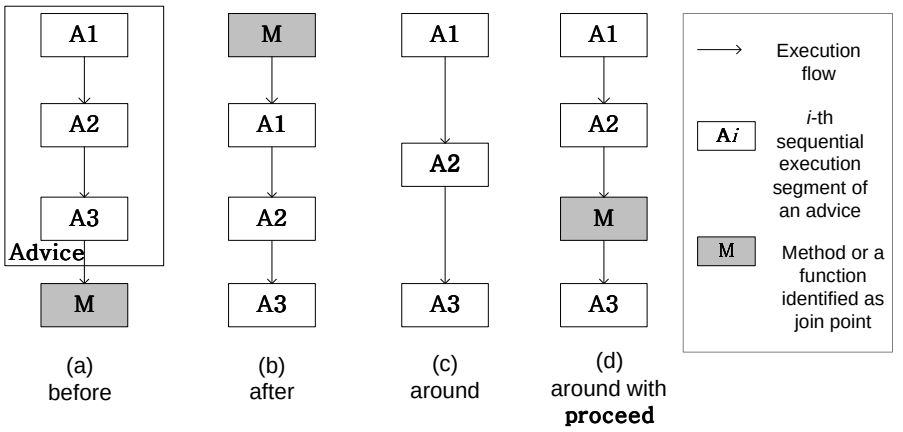


Figure 2.5: *The behavior of a before, after, and around advice*

```
advice getLockCall(lockId):
  void after (int lockId)
  {
    cout<<"Lock requested is"<<lockId<<endl;
  }
```

Figure 2.6: *An example of the advice definition*

- *after advice*, which is executed immediately after the join point, and
- *around advice*, which is executed instead of the join point.

A special type of an advice, called *inter-type advice* or an introduction can be used for adding members to the structures or classes.

When implementing an around advice, one can choose to execute the code of a function or a method of the join point at a suitable place, or suppress the join point execution completely. Figure 2.5 shows simplified flow of execution of a before, after, and around advice. In the example, we assume that an advice can be divided into three sequential execution segments, denoted $A1$, $A2$, and $A3$. M represents the execution of the join point function that triggers the advice execution. In figure 2.5(c) the code of the join point, i.e., the execution of M is completely suppressed. In the case (d), M still is executed after execution step A2. This is achieved by using the `proceed` construct of the aspect language to indicate that the execution of the join point function should be resumed. Conceptually, the execution behavior of the around advice using `proceed` can be viewed as an execution of two advices, before and after, at the same join point.

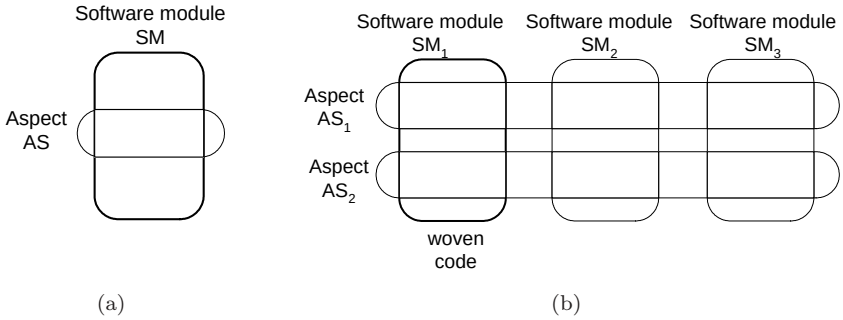


Figure 2.7: *Illustration of aspect weaving (a) one aspect and one white box component and (b) multiple aspects woven into several white box components.*

Figure 2.6 shows an example implementation of an after advice. With this advice each call to `getLock()` is followed by the execution of the advice code, i.e., printing of the lock id.

Aspect weaver. Aspects are combined with white box components using a special compiler called aspect weaver, which performs code-to-code transformation, transforming the source code of components and aspects into weaved source code of the system that is expressed in the same language as components. Figure 2.7(a) depicts the effects of aspect weaving on a single white box component, where weaving an aspect changes the internal behavior of the component. Figure 2.7(b) depicts a case where two aspects are woven into multiple white box components, thereby, modifying the internal behavior of a system.

2.1.3 From Components to Composition

Research in the software engineering community increasingly emphasizes composition of the system as a way to enable development of reliable systems and improve reuse of components. In this section we give an overview of the software engineering techniques that primarily focus on system composition. Figure 2.8 provides a hierarchical classification of composition-oriented approaches [21].

Component-based systems. The first level in figure 2.8 represents component-based systems, e.g., CORBA, COM, and JavaBeans. These systems are referred to as "classical" component-based systems [21]. Frameworks and standards for components of today in industry primarily focus on classical component-based systems. In these systems components are black boxes and communicate through standard interfaces, providing standard services to clients, i.e., components are standardized. Standardization eases adding or exchanging of components in the software system, and improves reuse of components. However, classical

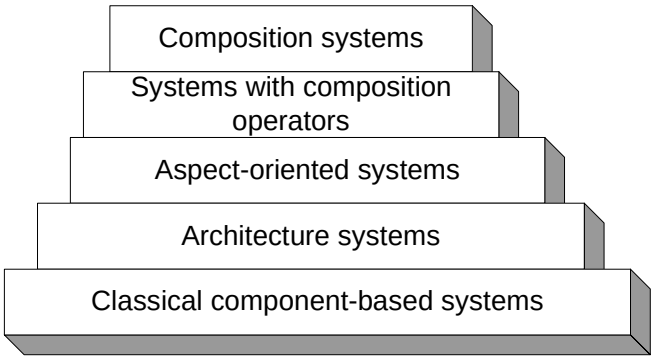


Figure 2.8: *Classes of component-based systems*

component-based systems lack rules for the system composition, i.e., there is no composition recipe.

Architecture systems. The next level represents architecture systems, e.g., RADL [151], RAPIDE [111], and UNICON [185]. These systems provide an architectural description language (ADL) used to specify the architecture of the software system. In an architecture system, components encapsulate application-specific functionality and are also black boxes. Components communicate through connectors, which encapsulate the communication between application-specific components [5]. This is a significant advancement in the composition compared to classical component-based systems, since communication and the architecture can be varied independently of each other. Thus, architecture systems separate three major aspects of the software system: architecture, communication, and application-specific functionality. One important benefit of an architecture system is the possibility of early system testing. Tests of the architecture can be performed with "dummy" components leading to the system validation in the early stage of the development. This also enables the developer to reason about the software system at an abstract level. Classical component-based systems can be viewed as a subset of architecture systems as they are in fact simple architecture systems with fixed communication [21].

Aspect-oriented systems. The third level represents systems that are developed using the AOSD principles [81]. Aspect-oriented systems separate more concerns of the software system than architecture systems. Beside architecture, application, and communication, aspects of the system can be separated further: representation of data, control flow, memory management, etc. Temporal constraints can also be viewed as an aspect of the software system, implying that a real-time system could be developed using AOSD [33]. Notably, several pro-

jects sponsored by DARPA (Defense Advanced Research Projects Agency) have been established with the aim to investigate possibilities of reliable composition of embedded real-time systems using AOSD [136], e.g., ARIES [6], ISIS PCES [78], and FACET [59]. In aspect-oriented systems, aspects are separated from modules constituting the system functionality; they are combined automatically through weaving. Weaving breaks the modules of the system at joint points and crosscuts then with aspects. Hence, the weaving process results in an integrated system. Here, modules constituting the system functionality are refereed to as white box components as they can be fully accessed and crosscut with aspects. However, aspect weavers can be viewed as black boxes since they are written for a specific language, and for each new language a new aspect weaver needs to be written. The process of writing an aspect weaver is not trivial, thus, introducing additional complexity in the development of aspect-oriented systems and usability of aspects. Compared to architecture systems, aspect systems are more general and allow separation of various additional aspects, thus, architecture systems can be viewed as a subset of the class of aspect-oriented systems. Having different aspects improves reusability since various aspects can be combined (reused) with different white box components. The main drawback of aspect systems is that they depend on special languages for aspects, requiring system developers to learn these languages.

Composition operators. At the fourth level are systems that provide composition operators by which components can be composed. Composition operators are comparable to component-based weaver, i.e., a weaver that is no longer a black box, but is also composed out of components, further improving the reuse. Subject-oriented programming (SOP) [129], an example of systems with composition operators, provides composition operators for classes, e.g., merge (merges two views of a class) and equate (merges two definition of classes into one). SOP is a powerful technique for compositional system development since it provides a simple set of operators for weaving aspects or views, and SOP programs support the process of system composition. However, SOP focuses on composition and does not provide a well-defined component model. Instead, SOP treats C++ classes as components.

Composition language. Finally, the last level includes systems that contain a full-fledged composition language, and are called composition systems. A composition language should contain basic composition operators to compose, glue, adopt, combine, extend, and merge components. The composition language should also be tailorable, i.e., component-based, and provide support for composing (different) systems in the large. Invasive software composition [21] is one approach that aims to provide a system composition language, and here components may consist of a set of arbitrary program elements (also known as boxes) [21]. Boxes are connected to the environment through very general connection points, called

hooks, and can be considered grey box components. Composition of the system is encapsulated in composition operators (composers), which transform a component with hooks into the component with code. The process of system composition using composers is more general than aspect weaving and composition operators, since invasive composition allows composition operators to be collected in libraries and to be invoked by the composition programs (recipes) in a composition language. Composers can be realized in any programming or specification language. Invasive composition supports software architecture, separation of aspects, and provides composition receipts, allowing production of families of variant systems. Reuse is improved, as compared to systems in the lower levels, since composition recipes can also be reused, leading to easy reuse of components and architectures. An example of the system that supports invasive composition is COMPOST [177]. However, COMPOST is not suitable for systems that have limited amount of resources and enforce real-time behavior, since it does not provide support for representing temporal properties of the software components.

2.2 Basic Notions in Real-Time Computing

In this section we define the basic real-time terminology that is used in the thesis as follows.

Embedded systems. Digital systems can be classified in two categories: general-purpose systems and application-specific systems [69]. General-purpose systems can be programmed to run a variety of different applications, i.e., they are not designed for any special application, as opposed to application-specific systems. Application-specific systems are typically a part of a larger host system and perform specific functions within the host system [41]. Such systems are usually referred to as embedded systems, and they are implemented partly in software and partly in hardware. Frequently, when standard microprocessors, micro-controllers, or DSP processors are used, specialization of an embedded system for a particular application concerns primarily specialization of software. An embedded system is required to be operational during the lifetime of the host system, which may range from a few years, e.g., a low-end audio component, to decades, e.g., an avionic system. The nature of embedded systems also requires them to interact with the external world, as these systems need to monitor sensors and control actuators for a wide variety of real-world devices.

Real-time systems. Most embedded systems are also real-time systems. In a real-time system the correctness of the system depends both on the logical result of the computation and the time when the results are produced [159]. Hence, enforcing timeliness is essential to the overall correctness of a real-time system. We use terms real-time software and real-time system interchangeably throughout the thesis. Both of these denote a software program running on a run-time platform

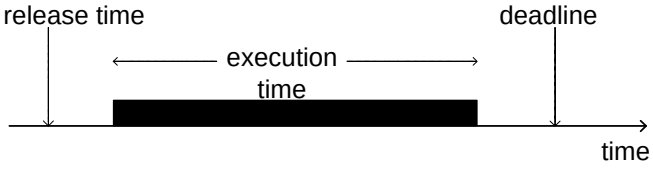


Figure 2.9: *Typical temporal characteristics of a task in a real-time system*

and performing a specific task for an application, i.e., a user of the program. We denote the underlying hardware and the operating system on which a real-time system operates as the run-time environment of the system.

Although the majority of real-time systems are embedded and vice versa, there are embedded systems that do not enforce real-time behavior, e.g., electronic toys, and there are real-time systems that are not embedded, e.g., stock exchange. The focus of this thesis are embedded real-time systems, i.e., the computing systems that are both embedded and real-time. If not otherwise specified, in the remainder of the thesis we use the term a real-time system to refer to an embedded real-time system.

Real-time task model. Real-time systems are typically constructed out of concurrent programs called tasks. A task in a real-time system is characterized by a number of temporal constraints amalgamating a task model. Figure 2.9 illustrates typical temporal constraints a task has to satisfy:

- ❏ release time, representing the point in time after which the task can be executed,
- ❏ execution time, representing the time needed for the task to execute, and
- ❏ deadline, representing the time point by which the task needs to be completed.

Execution times of tasks. Ensuring timeliness in a real-time system greatly depends on the ability to measure or estimate the amount of the CPU time tasks require for execution [38]. To maintain real-time performance, the execution time of a task in terms of the worst-case execution time (WCET) is required. The execution needs of a task can be obtained either by [38, page 59] (i) testing the task set on a hardware with appropriate test data, (ii) analyzing the task set by simulating the target system, or (iii) estimating the execution time by analyzing the programs at the high language level, or possibly assembler language level. The first method has a disadvantage that test data usually does not completely cover the domain of interest, while the second method heavily relies on the model of the underlying hardware. The hardware model typically represents an approximation

of the actual system and therefore might not accurately represent the worst-case behavior of tasks. In this thesis we adopt the third method, i.e., estimating bounds of the execution times of the tasks in the system by means of WCET analysis of programs [137]. The estimated WCET should be as tight as possible in order to asses the temporal behavior of a real-time system as close to the real behavior that will be exposed during run-time. A real-time system is considered to be predictable if its temporal behavior can be estimated (closely to the actual behavior exposed during run-time) before running the system.

Periodic, aperiodic, and sporadic tasks. Tasks can be associated with additional temporal properties, including periods and minimum inter-arrival times. A period specifies the time interval between two consecutive executions of a task in a real-time system. Tasks associated with periods are referred to as periodic tasks. Aperiodic and sporadic tasks are typically associated with arrival (or inter-arrival) times, where the arrival time is used to denote the release time of an aperiodic (or sporadic) task. While aperiodic tasks have no constraints on inter-arrival times, sporadic tasks are characterized by the minimum inter-arrival time, i.e., a minimum time interval that must elapse between arrivals of two consecutive tasks in the system.

Hard and soft real-time systems. Depending on the consequence of missing a deadline, real-time systems can be classified as hard or soft. In a hard real-time system consequences of missing a deadline can be catastrophic, e.g., aircraft and train control, while in a soft real-time system missing a deadline does not cause catastrophic damage to the system but may affect performance negatively, e.g., mobile computing systems, web services, video streaming, and e-commerce.

Open and closed real-time environments. Traditionally, the focal point of real-time computing has been hard real-time application areas and they are typically considered to be closed environments for real-time system operation since the workload characteristics of a real-time system are required to be known and do not change during the operational lifetime of the system.

Although real-time systems operating in closed environments are an essential part of the real-time research domain, the application of real-time computing to soft real-time application areas has gained momentum in recent years. The soft application areas, for which the workload is generally unknown and not predictable before or during the system execution, are considered to be open environments for real-time system operation.

Scheduling techniques. Regardless of the type of a real-time system or an environment in which the system operates, it is necessary to ensure and maintain real-time performance. This includes determining the order in which tasks should

execute to ensure that tasks meet their respective deadlines. The process of determining the order of task execution is known as scheduling [39, 89], and it enables real-time system designers to predict the behavior of a real-time system by ensuring that all tasks fulfill their execution requirements and meet their deadlines. A number of scheduling techniques exist and each is developed for a particular task model or an environment in which a real-time system operates.

Static scheduling techniques. Static scheduling is primarily used in hard real-time systems where temporal attributes of a task are known and fixed throughout the operational lifetime of the system. Rate-monotonic scheduling (RMS) is a typical representative of static scheduling techniques as the algorithm assumes that priorities of tasks are assigned based on periods [104, 89]. The priorities assigned to tasks based on their periods are invariant during run-time.

Dynamic scheduling techniques. Dynamic scheduling can also be used in closed hard real-time environments where tasks have known temporal attributes. These scheduling schemes are more flexible than static ones as they allow the task order to be determined at run-time. Namely, a scheduling algorithm does not have to posses full knowledge of the task set or its time constraints. New task activations, not known to the algorithm when scheduling the current task set, can arrive at a future unknown time [108]. The earliest deadline first (EDF) algorithm is a representative of dynamic scheduling techniques as it assigns priorities to tasks at run-time based on task deadlines [104, 89]. Hence, priorities assigned to tasks based on their deadlines are changing dynamically at run-time.

Feedback control scheduling. Traditional dynamic approaches, such as EDF, providing hard real-time guarantees rely on worst-case execution times and worst-case arrival patterns of tasks, are not effective for a large class of soft real-time systems that operate in unpredictable environments as they result in highly underutilized systems. Scheduling based on feedback control theory has been identified as a promising foundation for performance control of real-time systems that are both resource insufficient and exhibit unpredictable workloads [15, 11, 108, 110, 133, 45, 99]. Feedback control scheduling enables the designer or system operator to explicitly specify the performance of the system in terms of the desired steady state and transient state system performance. We return to feedback control scheduling in the context of maintaining real-time performance in section 2.3.2.

Quality of Service. In the context of a real-time system, quality of service (QoS) refers to system's performance. QoS can be quantitatively captured by a real-time system's performance parameters, such as utilization of the system or a number of task deadline misses in the system [108, 140]. Since feedback control scheduling methods enable specifying the desired performance level of a

real-time system, they are considered to be efficient as methods for real-time QoS management [73, 108, 110, 133].

2.3 Ensuring Real-Time Performance Guarantees

As mentioned previously, the essential difference between general-purpose software and real-time software system is that real-time performance needs to be satisfied and maintained in real-time software. Different techniques exist for ensuring and maintaining real-time performance, depending on the type of a real-time system, the environment in which the system operates, and the phase in the life cycle of the system.

For hard real-time systems operating in closed environments it is essential to determine the WCETs of tasks before the system is deployed into a run-time environment. To that end, various WCET analysis techniques have been developed [38]. In this thesis we employ symbolic WCET analysis (discussed in section 2.3.1) for determining the WCET of a task in the pre-run-time phase of the system's life cycle.

For soft real-time systems operating in open environments it is important to maintain real-time performance in terms of a specified level of QoS during run-time, i.e., in a post-deployment phase of the system's life cycle. A number of feedback control-based techniques have been developed to tackle this challenge. In section 2.3.2 we discuss the feedback-based QoS management techniques adopted in this thesis.

For both soft and hard real-time systems it is valuable to check temporal and functional behavior of a system already in the early design phase of the system development to ensure that the system is going to behave as required and prescribed by the design and/or requirement specification. A number of techniques with a strong mathematical foundation can be used for analyzing the system behavior against the specification. These techniques are referred to as formal analysis methods and are used in this thesis as the foundation for formal analysis of the functional and temporal real-time system behavior (section 2.3.3).

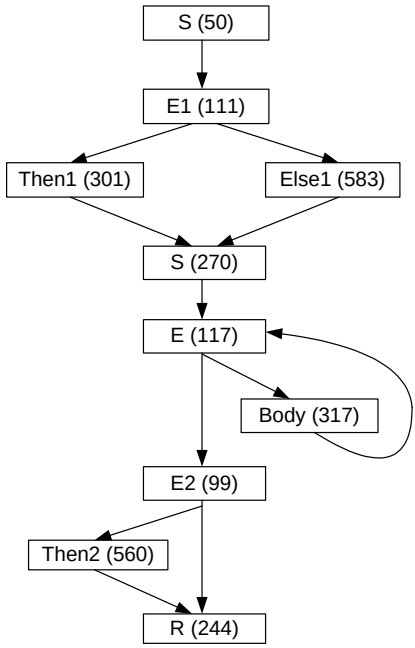
2.3.1 Symbolic Worst-Case Execution Time Analysis

As mentioned, one of the most important elements in real-time system development is temporal analysis of real-time software. Determining the WCET of the code guarantees that the execution time does not exceed the WCET bound. WCET analysis is usually done on two levels [137]: (i) low level, analyzing the object code and the effects of hardware-level features, and (ii) high level, analyzing the source code and characterizing the possible execution paths.

Symbolic WCET addresses the problem of obtaining the high level tight estimate of the WCET by characterizing the context in which code is executed [27].

```
1  double power(double f, int n) {
2      double res = 0;
3      int times = 0;
4      bool pos = true;
5
6      if n < 0 then
7              times = -n;
8      else
9              pos = false;
10             times = n;
11     end if
12     res = 1.0;
13
14     for (i in 1 ··· times) loop
15             res = res * f;
16         end loop
17
18     if (pos) then
19             res = 1 / res;
20     end if
21     return res;
22 end power
```

(a) The code of the `power` function



(b) Control-flow graph of the `power` function (WCETs in parenthesis)

Figure 2.10: *The `power` function*

Hence, the symbolic WCET technique describes the WCET as a symbolic expression, rather than a fixed constant.

To illustrate the main idea and benefits of the symbolic WCET technique, we provide an example of the WCET calculations for code of the `power` function given in figure 2.10(a) (the example is adopted from [27]). The function computes the `n`-th power of a float number `f`. If `n` is negative the function computes $\frac{1}{f^{abs(n)}}$.

Using traditional techniques to calculate WCET of the `power` function one first needs to estimate the maximum range of the exponent `n`, thus, determining the maximum number of iterations of the loop in the function (lines 14-16 in figure 2.10(a)). Then, the WCET of the function is determined by adding the execution times of all sections of code, looking for the longest path in conditional branches. Figure 2.10(b) shows the control flow of the `power` function, with an example of the execution times of each of the sections in the code (numbers in parenthesis). Consider a case where `n` is in the range [-10,10], implying that 10 is the maximum number of loop iterations. In this case, adding the execution times of code sections (given in figure 2.10(b)) results in the following WCET of the

`power` function:

$$WCET_{power} = 50 + 111 + max(301, 583) + 270 + 10(117 + 317) + 99 + max(560, 0) + 244 = 6374$$

The obtained result is pessimistic as the two if-statement branches with the maximum WCET can never be taken together. Furthermore, this WCET calculation takes a pessimistic approach to calculations of loop iterations as it uses 10 as the maximum number of iterations.

The symbolic WCET technique allows expressing the WCETs of the code as an algebraic expression. In this case, the WCET of the `power` function can be formulated as a function of the exponent `n` as follows:

$$WCET_{power}(\mathtt{n}) = 50 + 111 + [\mathtt{n} < 0]301 + [\mathtt{n} \geq 0]583 + 270 + 117 + \sum_{i=1}^{abs(\mathtt{n})} (117 + 317) + 99 + [\mathtt{n} < 0]560 + [\mathtt{n} \geq 0]0 + 244$$

The above expression can be further simplified, e.g., by using Maple V, into:

$$WCET_{power} = \begin{cases} 1752 - 434\mathtt{n} & \text{if } \mathtt{n} < 0 \\ 1474 & \text{if } \mathtt{n} = 0 \\ 1474 + 434\mathtt{n} & \text{if } \mathtt{n} > 0 \end{cases}$$

The WCET of the `power` function is maximal for $\mathtt{n} = -10$. The value of $WCET_{power}$ for $\mathtt{n} = -10$ is $WCET_{power} = 6092$. The maximal value of the WCET obtained by symbolic analysis (6092) is tighter than the value of WCET obtained by traditional analysis (6374). It is worth noting that the symbolic expression is left parameterized until the actual call to the function is made, in which case, based on the passed value of the exponent e, the symbolic expression is evaluated and the tight bound on the execution time is obtained.

Now, the obtained WCET estimation can be used by scheduling algorithms to ensure that all task deadlines are met and, thereby, guarantee the performance of the real-time system.

2.3.2 Feedback-Based QoS Management

Recall that for soft real-time systems operating in open environments it is essential to guarantee a specific level of QoS. Assuring QoS guarantees is typically done by employing feedback control. Hereafter we refer to real-time QoS management based on feedback control as feedback-based QoS management. A typical structure of a feedback control system is given in figure 2.11. Input to the controller is a performance error, $y_r(k) - y(k)$. The performance error is computed as the difference between the reference $y_r(k)$, representing the desired state of the

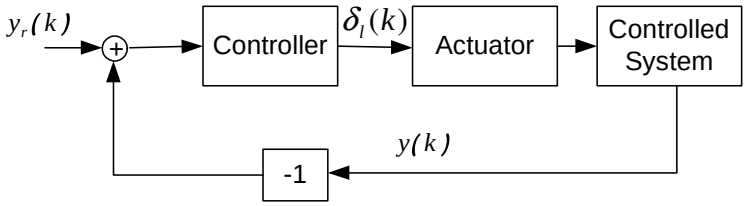


Figure 2.11: *An architecture of the real-time system using feedback control structure*

controlled system at time kT, where T is the sampling period and k is the sampling instant, and the actual system state measured using the sensor and given by the controlled variable $y(k)$.

Based on the performance error the controller changes the behavior of the controlled system via the manipulated variable $u(k)$ and the actuator. The objective of the control is to compute $u(k)$ such that the difference between the desired state and the actual state is minimized, i.e., we want to minimize $(y_r(k) - y(k))^2$. This minimization results in a more reliable performance and system adaptability as the actual system performance is closer to the desired system performance.

Next we review the main feedback-based QoS management policies used in various real-time applications. The goal is to illustrate the variety of sensors, actuators, and controllers used in QoS management of real-time systems.

Controlling the utilization using feedback control has been important in applications where meeting deadlines is vital [3, 11, 45, 108, 88]. In controlling utilization, a sensor periodically measures the utilization of the system. A controller compares the utilization reference with the actual measured utilization and computes a change to the manipulated variable. The actuator then carries out the change in utilization in one of the following ways:

- ❏ The precision or the quality of the result of the tasks can be modified by applying the imprecise computation technique [105]. Here, the utilization is decreased by reducing the execution time of the tasks. This in turn implies that the precision of the task results is lowered. Conversely, the result precision increases as the utilization increases.

- ❏ The utilization is directly related to the the inter-arrival times of the tasks, i.e., the utilization increases with decreasing inter-arrival times. Hence, the utilization can easily be adjusted by changing the inter-arrival times of the tasks [108, 45].

- ❏ Since the utilization of the system increases with the number of admitted tasks, the utilization can be changed by enforcing admission control, where a subset of the tasks are allowed to be executed [16].

In a number of real-time applications, e.g., video streaming and signal processing, tasks can deliver results of less precision in exchange for timely delivery of

results. For example, during overloads alternative filters with less output quality and execution time may be used, ensuring that task deadlines are met. In general, the precision of the results increases with the execution time given to a task, calling for the use of a feedback structure to control the precision [14, 15, 99]. In such a structure, the sensor is used to estimate the output quality of the tasks, while a controller forces the tasks to maintain an output precision equal to the reference. The execution time given to individual tasks is controlled by the actuator, thereby, ensuring that the output precision is maintained at the desired level.

In telecommunication and web servers applications, arriving packets and requests are inserted into queues, where the requests wait to be processed. The time it takes to process requests once they arrive at a server is proportional to the length of the queue, i.e., the processing time increases with the length of the queue. Controlling the queue length is the key to guarantee timely processing of requests. If the queue length is too long, then the time it takes to process a request may be unacceptable as there are time constraints on the requests. A feedback controller can be used to adjust the queue length such that the length equals its reference [133, 4, 153, 2, 148]. Ways of manipulating the queue length include changing the admission rate of the requests. Namely, by admitting more arriving requests the queue length increases, thus, increasing the latency time of the requests.

Controlling the execution times of tasks is important in real-time systems with constraints on energy consumption. Efforts have been carried out trying to reduce energy consumption in real-time systems, while preserving timely completion of tasks [186]. In this case execution times are monitored and the voltage and, thus, frequency of the CPU is varied such that the power consumption is reduced and tasks are executed in a timely manner. Hence, the sensor is used to measure the execution time of the tasks and the actuator is used to carry out the change in the voltage or the frequency of the CPU.

By studying the examples above, we note that there are many ways to implement sensors and actuators. This shows that the choice of a sensor and an actuator depends highly on the type of an application and its constraints. Control-related variables also vary depending on the application being controlled. Furthermore, controllers are implemented using a particular metric and a controlling algorithm, explicitly tuned for a specific application.

In the feedback control-based policies mentioned in this section, traditional control algorithms such as PID, state-feedback, and Lead-Lag are used (details on these algorithms can be found in [62]). These algorithms assume that the controlled system does not change during run-time, i.e., the behavior of the controlled system is time-invariant. Hence, the control algorithm parameters are tuned off-line and may not be altered during run-time. However, the behavior of some real-time systems is time-varying due to significant changes in load and/or execution times. This is addressed by employing adaptive control algorithms [146] where the behavior of the controlled system is monitored at run-time and

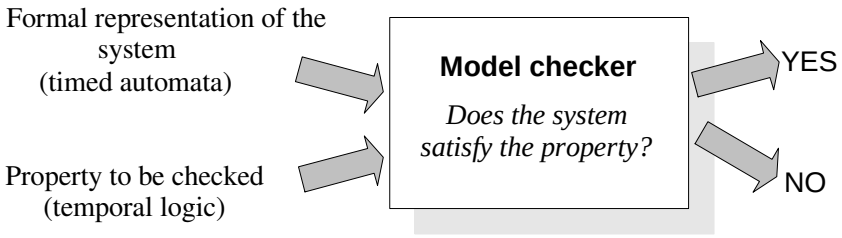


Figure 2.12: *Formal analysis of real-time systems using model checking*

the control algorithm parameters are adapted accordingly. Hence, different types of controllers may be employed and, as such, there is a need for configurability of the controllers.

As can be observed, there are many different types of control algorithms that are applied to specific QoS management policies. The choice of a control algorithm depends on the control performance, run-time complexity, memory footprint, and adaptiveness.

2.3.3 Formal Analysis of Real-Time Systems

Several techniques for formal analysis have been developed of which model checking has been found to be most beneficial in verification of real-time systems behavior [74, 96, 8]. Model checking of real-time systems is based on a formal representation of the system, logics for checking relevant system properties, and algorithms that check whether the formal representation of the system satisfies the property (see figure 2.12).

Finite state real-time systems are frequently formally represented using timed automata. A timed automaton is a timed extension of a finite state automaton with a finite set of real-valued clock variables (clocks) [9]. Constraints on the clocks, referred to as guards, are used to restrict the behavior of an automaton, and accepting conditions are used to enforce progress properties. A transition, represented by an edge in a graph of the automaton, can be taken when the clocks satisfy the guard (condition) that labels the edge.

A simplified version of the original timed automata, called timed safety automata [74], are introduced to specify progress properties using local invariant conditions. An automaton, in this case, can remain in a location as long as the clock values satisfy the invariant conditions of the location. Given that timed safety automata are, due to their simplicity, increasingly used in verification tools and model-checking theory [25], we focus on timed safety automata and hereafter refer to them as timed automata.

Figure 2.13 shows an example of a simple timed automaton that has three locations, *start*, *loc*1, and *loc*2 and one clock x. The automaton can stay in the

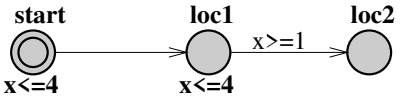


Figure 2.13: *An example of a simple timed automaton.*

location *start* only while the value of the clock is less than or equal to four, i.e., while the location invariant is satisfied. However, due to the "empty" guard, equivalent to the condition *true*, the system may move from *start* to *loc*1 at any time point. The transition from location *loc*1 to *loc*2 is taken when the guard is satisfied, i.e., clock x is equal to one or higher. Note that edges are taken instantaneously, but the time elapses in a location. Additionally, clocks can be reset to zero at any edge of a timed automaton. In the example from figure 2.13, clocks are not reset.

Formally, a clock is defined as a variable ranging over $\mathbf{R}^+$. For a set C of clocks with $x, y \in C$, the set of clock constraints over C, $\psi(C)$, is defined by

$$\alpha ::= x \prec c \mid x - y \prec c \mid \neg\alpha \mid (\alpha \wedge \alpha),$$

where $c \in \mathbf{N}$, and $\prec \in \{<, \leq\}$. A formal definition of timed automata is given as follows [25].

Definition 1 (Timed automaton) *A timed automaton $\mathcal{A}$ is a tuple $\langle L, l_0, E, C, r, g, Inv\rangle$, where L is a non-empty finite set of (named) locations, $l_0 \in L$ is an initial location, $E \subseteq L \times L$ is a set of edges, C is a finite set of clocks, $r : E \mapsto 2^C$ is a function that assigns to each edge $e \in E$ a set of clocks $r(e)$ to be reset, $g : E \mapsto \psi(C)$ is a function that labels each edge $e \in E$ with a clock constraint $g(e)$ over C, and $Inv : L \mapsto \psi(C)$, a function that assigns to each location $l \in L$ an invariant $Inv(l)$.*

The values of clocks are formally defined by *clock valuations*. A clock valuation v is a function that assigns a value $v(x)$ to each clock $x \in C$. In the remainder of the thesis we consider guards, clocks to be reset, and invariants as sets of clock valuations. We use notation $v \in g$ to denote that v satisfies a guard g, and $[r \mapsto 0]v$ to denote the clock valuation that maps all clocks in r to 0 and agrees with v for other clocks $C \backslash r$. Similarly, $v+d$ denotes the clock valuation that maps all $x \in C$ to $v(x) + d$, $d \in \mathbf{R}^+$. The operational semantics of a timed automaton is defined as a transition system where a state consists of a current location and current values of clocks. Two types of transitions can be taken between the states: an action transition and a delay transition.

Definition 2 (Operational semantics) *Let $\mathbf{R}^C$ be the set of all clock valuations, and $v_0 = v_0(x) = 0$, for all $x \in C$. The semantics of a timed automaton*

$\mathcal{A}$=$\langle L,\ l_0,\ E,\ C,\ r,\ g,\ Inv\rangle$ is a transition system $(S, s_0, \mapsto)$, where $S = L \times \mathbf{R}^C$ is the set of states, $s_0 = \langle l_0, v_0\rangle$ is the initial state, and $\mapsto \subseteq S \times S$ is the transition relation defined as:

- ❑ $\langle l, v\rangle \mapsto \langle l, v + d\rangle$ *if* $v + d \in Inv(l)$, *for* $d \in \mathbf{R}^+$ *(delay transition); and*
- ❑ $\langle l, v\rangle \mapsto \langle l', v'\rangle$ *if there exists* $e = \langle l, l'\rangle \in E$ *such that* $v \in g(e)$, $v' = [r(e) \mapsto 0]v$ *and* $v' \in Inv(l')$ *(action transition).*

The foundation for decidability results in verification of timed automata is based on the notion of *region equivalence* over clock assignments [10]. A more efficient representation of the state-space for the timed automata is based on the notion of *clock zones* and *zone graphs* [74]. A zone Z represents a solution set of a clock constraint, i.e., the maximal set of clock assignments satisfying the constraint. Zones can be represented as conjunctions in $\psi(C)$ and, therefore, $\psi(C)$ denotes a set of zones. The symbolic semantics of timed automata is defined by a transition system where a symbolic state consists of a current location and a current zone.

Definition 3 (Symbolic semantics) *Let* $Z_0 = \bigwedge_{x \in C} x \geq 0$ *be the initial zone. The symbolic semantics of a timed automaton* $\mathcal{A}$=$\langle L,\ l_0,\ E,\ C,\ r,\ g,\ Inv\rangle$ *is a transition system* $(S, s_0, \rightsquigarrow)$ *called the* zone graph *where* $S = L \times \psi(C)$ *is the set of symbolic states,* $s_0 = \langle l_0, Z_0 \wedge Inv(l_0)\rangle$ *is the initial state, and* $\rightsquigarrow \subseteq S \times S$ *is a symbolic transition defined as:*

- ❑ $\langle l, Z\rangle \rightsquigarrow \langle l, Z'\rangle$, $Z' = Z^{\uparrow} \wedge Inv(l)$; *and*
- ❑ $\langle l, Z\rangle \rightsquigarrow \langle l', Z'\rangle$, $Z' = r_e(Z \wedge g(e)) \wedge Inv(l')$ *if* $e = \langle l, l'\rangle \in E$;

where

$Z^{\uparrow}$=$\{v + d | v \in Z \wedge d \in \mathbf{R}^+\}$ *is the future operation, and* $r_e(Z)$=$\{[r(e) \mapsto 0]v | v \in Z\}$ *is the reset operation.*

The set of zones is closed under reset and future operations. That is, the result of a reset operation on a zone results in a new zone in which adequate clocks are reset. The symbolic semantics is a full and correct characterization of the operational semantics of timed automata [25]. The symbolic semantics can be extended to cover networks of communicating timed automata, where a location vector is used instead of a location.

Verification of real-time systems emphasizes checking of safety and bounded liveness properties of real-time systems using reachability analysis [25]. For an automaton with symbolic semantics described by definition 3, a state $\langle l, Z\rangle$ is reachable if there is a sequence of symbolic transitions (i.e., a path) from the initial state $\langle l_0, Z_0\rangle$ to the state $\langle l, Z\rangle$. A number of tools exist, e.g., UPPAAL [178] and Kronos [60], that use reachability analysis based on symbolic semantics of timed automata for checking properties of real-time systems. The algorithms utilize

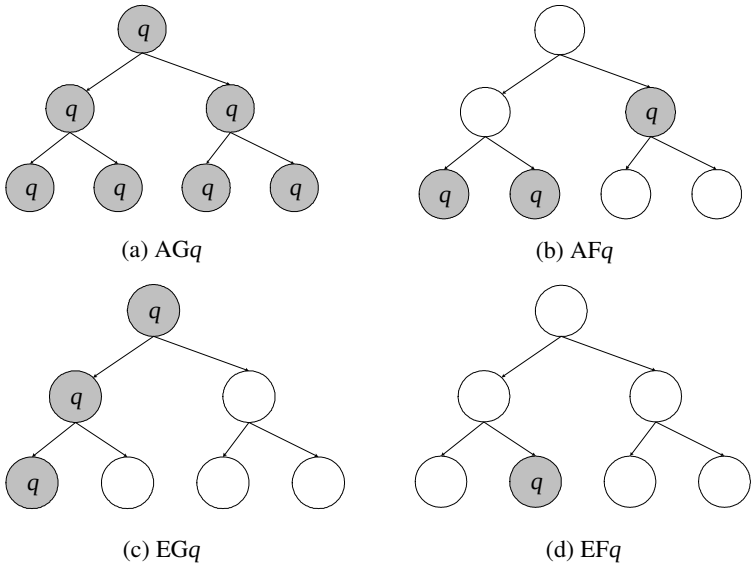


Figure 2.14: *Expressing properties in TCTL logic.*

the property of the inclusion operation $\subseteq$ on the zones, where clock constraints satisfied in a zone Z' are also satisfied in zone Z if $Z' \subseteq Z$.

Properties in the real-time model checking tools are expressed in timed computational tree logic (TCTL) [9] using temporal operators G and F, and path quantifiers A and E. The semantics of the operators and quantifiers is as follows:

- Aq means that the property q holds for every path;
- Eq means that property q holds for some path;
- Gq means that the property q holds globally for every state on the path; and
- Fq means that the property q holds eventually for some state on the path.

By combining temporal operators and path quantifiers we can obtain expressions for checking behavior of the system, namely (see figure 2.14):

- AGq means that the property q holds for every state in every path;
- AFq means that the property q holds for some state on every path;
- EGq means that property q holds for every state on some path; and

- ❏ EFq means that property q holds for some state on some path.

For example, we can express the reachability property that some state satisfying proposition q will eventually be reached in the system on some path as EFq. Invariant properties, e.g., $\phi = AGq$ checking if all states in the system at every path satisfy q, are checked using the negation of reachability properties (AGq is equivalent to $\neg EF\neg q$). In this thesis we utilize the verification of real-time systems using symbolic techniques and representation of timed automata using zones.

2.4 Embedded Real-Time Database Systems

The amount of data that needs to be managed by real-time and embedded systems is increasing [43]. Hence, a database management system (DBMS) functionality suitable for embedded and real-time systems is needed to provide efficient support for storage and manipulation of data. In the remainder of the thesis, were there is no risk of ambiguity, we use the term database to denote a DBMS. Embedding databases into real-time systems offers significant gains:

- ❏ reduction of development costs due to reuse of database systems;
- ❏ improvement of quality in the design of embedded systems since the database provides support for consistent and safe manipulation of data, which makes the task of the programmer simpler; and
- ❏ increased maintainability as the software evolves.

Furthermore, embedded databases provide mechanisms that support porting of data to other embedded systems or large central databases. Naturally, requirements placed on such a database originate both from characteristics of embedded and of real-time systems.

Namely, most embedded systems need to be able to run without human presence, which means that a database in such a system must be able to recover from a failure without external intervention [125]. Also, the resource load the database imposes on the embedded system, e.g., memory footprint and power consumption, should be carefully balanced. For example, in embedded systems used to control a vehicle, the minimization of the hardware cost is of utmost importance. This usually implies that memory capacity must be kept as low as possible and, consequently, databases used in such systems must have a small memory footprint. Embedded systems can be implemented in different hardware environments supporting different operating system platforms; this requires the embedded database to be portable to different operating system platforms.

On the other hand, real-time systems impose a different set of demands on a database system. The data in the database used in real-time systems must be logically consistent, as well as temporally consistent [139]. Temporal consistency of

data is needed in order to maintain consistency between the actual state of the environment that is being controlled by the real-time system, and the state reflected by the content of the database. Temporal consistency has two constituents:

- *absolute consistency*, between the state of the environment and its reflection in the database, and
- *relative consistency*, among the data used to derive other data.

A temporally constrained data element, d, is characterized by three attributes [139]:

- value $v(d)$, which is the current state (value) of data element d in the database,
- time-stamp $ts(d)$, which is the time when the observation relating to d was made, and
- absolute validity interval $avi(d)$, i.e., the length of the time interval following $ts(d)$ during which d is considered to be absolute consistent.

A set of data items used to derive a new data item forms a relative consistency set, denoted R, and each such set is associated with a relative validity interval, R_{rvi}. Data in the database, such that $d \in R$, has a correct state if and only if [139]

1. $v(d)$ is logically consistent, i.e., satisfies all integrity constraints, and
2. d is temporally consistent, both
   - absolute, i.e., $(t - ts(d)) \leq avi(d)$, and
   - relative, i.e., $\forall d' \in R, |ts(d) - ts(d')| \leq R_{rvi}$.

A transaction, i.e., a sequence of read and write operations on data items, in conventional databases must satisfy the following properties: atomicity, consistency, isolation, and durability, normally called ACID properties [155]. In addition, transactions that process real-time data must satisfy temporal constraints. Some of the temporal constraints on transactions in a real-time database come from the temporal consistency requirement, and some from requirements imposed on the system reaction time (typically, periodicity requirements) [139]. These constraints require time-cognizant transaction processing so that transactions can be processed to meet their deadlines, both with respect to completion of the transaction as well as satisfying the temporal correctness of the data [107]. Also, these requirements result in different temporal attributes attached to transactions that make a transaction model. Note that a transaction in a database system corresponds to a task in a real-time system.

Method	Controls Access to	Suitable for		
		Hard	Soft	Mixed
HP-2PL	Data		√	
OCC	Data		√	
Priority Inheritance	Data		√	
RWPCP	Data	√		
Timestamp-Based	Data		√	
Versioning	Data	√	√	
2V-DBP	Data			√
IM	Data			√
Similarity-Based	Data	√	√	√
Epsilon-Based	Data	√	√	√
ARIES/KVL	Data/(index)	(√)	√	(√)
B-Tree ICC	Index	√	√	√
B-Tree ICC w. GUARD	Index		√	

Table 2.2: *Concurrency control methods in real-time database systems*

2.4.1 Concurrent Access to Resources

It is frequently the case in a real-time database that there is a number of concurrently executing transactions. Concurrent execution of transactions brings several benefits in terms of effectively increasing performance. However, concurrency also introduces problems as violating consistency becomes a risk. We illustrate this with a simple example as follows. Consider two transactions, τ_1 and τ_2, accessing the same data item x. We denote the value of a data item x as $v(x)$. The operation of writing a new value of x performed by a transaction τ_i is denoted $W(\tau_i, v(x), x)$. Reading of a value of a data item $v(x)$ and storing this value into a new data item x_i is denoted $R(\tau_i, v(x), x_i)$. Assume that transactions τ_1 and τ_2 perform the following operations:

$$R(\tau_1, v(x), x_1) \rightarrow R(\tau_2, v(x), x_2) \rightarrow W(\tau_2, x_2 + 1000, x) \rightarrow W(\tau_1, x_1 + 1, x)$$

Assume that data item x had the value $v(x) = 0$ before the transactions started. When both transactions have committed, x has the value 1. Hence, the update that τ_2 made has disappeared.

To ensure concurrent access to data in the real-time database a number of concurrency control (CC) methods have been developed. Most of them aim at preserving serializability, i.e., the property that transactions should be isolated from each other in the sense that the effect of executing a concurrent transaction should be the same as if each transaction is executed in a system without concurrency [183].

Table 2.2 lists well-known CC methods together with resources they control access to. Whether algorithms are suitable for hard, soft, or mixed real-time

systems is also indicated in the table. Here follows a brief explanation of the listed methods.

High priority 2-phase locking (HP-2PL) [1] is a locking scheme based on regular 2-phase locking (2PL). HP-2PL takes priorities into account, while 2PL does not. The main idea of the HP-2PL algorithm is to relinquish resources to the high priority transaction by aborting the low priority transactions. Variants of HP-2PL exist, using different conflict resolution methods. HP-2PL suffers from unbounded number of transaction restarts and unbounded waiting times. Thus, it is primarily suitable for soft real-time systems.

Optimistic concurrency control (OCC) [91] assumes that the execution consists of three phases: read, validation and write phases. During the read phase a transaction reads from the database and updates data in the local space. In the validation phase, the system correctness is checked. If the system is correct, the transaction enters its write phase and writes the locally updated data to the database. Transactions in OCC may be restarted an arbitrary number of times, which makes the method suitable for soft real-time systems.

Priority inheritance [165] is an approach where a low priority transaction blocking a high priority one inherits the higher priority. The idea is to allow the low priority transaction to finish quicker, which results in decreased blocking times. Priority inheritance must be used together with a locking protocol, e.g., 2PL, and is only suitable for soft real-time systems as waiting for the completion of a low priority transaction (even though its priority is raised) can make the blocking time unbounded.

Read/write priority ceiling protocol (RWPCP) [154] is an extension of the well-known priority ceiling protocol. RWPCP is designed for hard real-time systems and differentiates between read and write access to data items. RWPCP requires that the transactions are scheduled using fixed priority scheduling, e.g., RMS.

Timestamp-based CC [29] is a lock-free scheme feasible in soft real-time systems. Each transaction τ_i is assigned a timestamp, $ts(\tau_i)$, when it starts. Timestamp-based CC is based on enforcing the following rule. If $p_i(x)$ and $q_j(x)$ are conflicting operations, then $p_i(x)$ is processed before $q_j(x)$ iff $ts(\tau_i) < ts(\tau_j)$. Variants exist, but they all use this rule as a basis.

Versioning [30] maintains several versions of data at the same time. Typically, every update on the data creates a new version. Read-only transactions are then allowed to run without being blocked, by only reading data versions that committed before the read-only transaction started. The concept of versioning

can be used in both hard and soft systems. The specific suitability is dependent on how updates are handled.

2-version database pointer CC (2V-DBP) [122] is a method for use in mixed hard and soft systems. It uses HP-2PL for the soft transactions and 2-versioning for the hard ones. It allows both classes of transactions to execute with minimal interference.

Integrated method (IM) [94] is a method designed for mixed soft and non-real-time systems. It uses OCC for soft transactions and 2PL for non-real-time transactions. Some variants exist depending on how inter-class conflicts are handled.

Similarity-based CC [92] is a class of CC methods that relaxes the serializability correctness criterion with the motivation that it is often too strict for a real-time database. The idea is that by specifying what values are similar, non-serializable schedules are allowed as long as the result becomes similar to what a serializable schedule would produce. Similarity-based CC is suitable for both soft and hard real-time systems, depending on the specific algorithm chosen from this class.

Epsilon-based CC [181] uses a generalization of classic serializability, called epsilon-serializability (ESR). ESR allows some inconsistency, e.g., read-only queries may read data that is concurrently being updated. This is realized by specifying a limit on how much inconsistency these queries may import. Transactions performing queries like this are called epsilon transactions (ETs), and may fit in all types of real-time systems.

ARIES/KVL [117] is applicable in systems that use a B-tree index structure. One variant of the algorithm can be used to enforce transaction serializability only. This is done by placing locks on the index nodes of the accessed data items. By using locks the algorithm experiences the same problems as HP-2PL regarding unbounded blocking times. The algorithm can also be used to enforce index consistency at the same time as transaction serializability.

B-tree index CC is used in many conventional databases. There are three well-known classes of B-tree index CC protocols [72]: Bayer-Schkolnick [24], Top-Down [119], and B-link [97]. Each class also has several flavors. Common to all classes and flavors are that they use some of, or all, four types of locks: intention share (IS), intention exclusive (IX), share and intention exclusive (SIX), and exclusive (X). The compabilities among these locks are shown in table 2.3. The symbol $\surd$ in table 2.3 means that the locks are compatible with each other, while an empty space indicates that the locks are in conflicting modes. Index

Lock	IS	IX	SIX	X
IS	√	√	√	
IX	√	√		
SIX	√			
X				

Table 2.3: *The index concurrency control lock compatibilities*

operations lock tree nodes and subtrees in a variety of ways, depending on the type of the ICC algorithm. Nodes are typically locked on descent, i.e., going from the top of the three to the leaves, and the locks are kept if modification of the tree is needed, otherwise they are released. In the real-time variants of these methods, the locks are preempted and index operations aborted if a higher priority transaction tries to lock a node in a conflicting mode.

B-Tree index CC with GUARD [71] is designed specifically for real-time systems. The method introduces an admission control called gatekeeping using adaptive earliest deadline (GUARD). The admission control decides what transactions are allowed execute. By introducing this control, the contention for the index is significantly decreased under high system load, which in turn decreases the blocking times. GUARD is tailored to be used together with EDF scheduling.

2.4.2 Emerging Requirements

There are several embedded databases on the market, e.g., Polyhedra [135], RDM and Velocis [113], Pervasive.SQL [134], Berkeley DB [26], and TimesTen [176]. They all have different characteristics and are designed with a specific application in mind. They support different data models, e.g., relational vs. object-relational model, and different operating system platforms [171]. Moreover, they have different memory requirements and provide various types of interfaces for users to access data in the database. Application developers must carefully choose the embedded database their application requires, and find the balance between required and offered database functionality. Hence, finding the right embedded database is a time-consuming, costly, and difficult process, often with a lot of compromises. Additionally, the designer is faced with the problem of database evolution, i.e., the database must be able to evolve during the life-time of an embedded system, with respect to new functionality. However, traditional database systems are hard to modify or extend with new required functionality, mainly because of their monolithic structure and the fact that adding functionality results in additional system complexity.

Although a significant amount of research in real-time databases has been done in the past years, it has mainly focused on various schemes for concurrency control, transaction scheduling, and logging and recovery, and less on configu-

rability of software architectures. Research projects that are building real-time database platforms, such as ART-RTDB [82], BeeHive [163], DeeDS [17] and RODAIN [102], have monolithic structure, and are targeted for a specific real-time application. Hence, the issue of how to enable development of an embedded database system that can be tailored for different embedded and real-time applications arises.

Chapter 3

Developing Reconfigurable Real-Time Systems

Enabling development of reconfigurable and reusable software is essential for efficient and cost-effective development of modern real-time systems. In this chapter we first identify key requirements that facilitate development of such systems. We then explore to what degree these requirements are fulfilled by existing approaches in the software engineering and real-time community. Finally, based on the identified requirements and shortcomings of existing approaches, we form a list of research challenges that are addressed in this thesis.

3.1 Requirements

The requirements are classified into three main categories and include issues concerning the component model for real-time systems, separation of concerns through the support for aspects and aspect weaving, and system composability encompassing the support for configuration and analysis of an assembly.

3.1.1 Component Model

One of the most fundamental reasons behind the need to divide software into modules is to guarantee exchange of parts [21]. In mature industries, e.g., mechanical engineering, an engineer constructs a product by decomposition; the problem is decomposed into sub-problems until one arrives to the basic problem for which basic solution elements can be chosen. Software engineering is not different from other engineering disciplines in the effort to mature, i.e., enable software decomposition into components and use of already developed components to build software systems.

To be able to build software systems using reusable components, a way of specifying what a component should look like, i.e., a component model, is needed. A component model supports modularity of software in the sense that it defines what should be hidden in the component, i.e., it enforces information hiding, such that it can be exchanged and reused in several systems [21, 51, 169].

While information hiding in terms of black box components is assumed in CBSD, AOSD uses the white box component metaphor to emphasize that all details of the component implementation should be revealed. Both black box and white box component abstractions have their advantages and disadvantages. For example, hiding all details of a component implementation in a black box manner has the advantage that a component user does not have to deal with the component internals. In contrast, having all details revealed in a white box manner allows the component user to freely optimize and tailor the component for a particular software system.

A component model that enforces information hiding for ensuring easy exchange and reuse and, at the same time, provides controlled access to its internals in well-defined places of the component structure is the balance between the two extremes; a component exhibiting this property is referred to as a grey box component. A component with the grey box property is particularly attractive in applications where optimization of components for each particular application is necessary, e.g., real-time, database, and sensor network applications [167, 23, 75]. Thus, we obtain the following requirement for a component model (CM).

Requirement

CM1 **Information hiding**. A component has to hide details of its internal design and implementation from the environment and the component users. To ensure reusability in a wide domain of applications a component should exhibit the grey box property enabling the users to fine-tune it for individual applications.

Interfaces of components should be well-defined by the component model to provide necessary information to the component user, e.g., reuse context and performance attributes. The interfaces that have a constructive role in the system and are used for component communication and system configuration are frequently referred to as functional or constructive interfaces. Provided and required interface of components are typical representative of the class of constructive interfaces. Beside these, it is found to be advantageous to have other types of interfaces that can be used for various analysis of the resulting system, e.g., latency of the system [76]. These interfaces are denoted non-functional or analytical interfaces [51, 76].

Requirement

CM2 **Interfaces**. A component should have well-defined interfaces to the environment. The component can be accessed only through these interfaces, thus, complementing the information hiding criterion. Moreover, the interfaces provided by the component should enable construction as well as analysis, e.g., WCET and formal analysis, of an assembled system.

A transparent system evolution and configuration requires support for defining and implementing connectors. This is to ensure that a newly developed component can be added to a system even if the system is developed not aware of the possible new extensions.

Requirement

CM3 **Connectors**. There should be support for integrating a component, developed independently of other components, into a system even if the system was not initially developed aware of this particular component, or there are syntactical mismatches between the interfaces of the components in the system.

We observe that ensuring reusability of a software system or its parts necessitates a well-defined component model supporting information hiding (preferably in terms of grey box components) and providing interfaces for construction and, possibly analysis of the system, as well as ways of gluing non-matching components that have been developed independently.

Approaches to real-time system development sharing the component vision are faced with additional requirements on the component model as components in these environments should provide mechanisms for handling temporal constraints. Moreover, since the traditional view of real-time systems implies tasks as building elements of a system, the relationship between tasks and components needs to be clarified. We argue that the relationship between a real-time software component and a task should not be fixed for several reasons. First, we would like to reuse any software components applicable to a real-time system under construction, i.e., we do not want to limit reusability of real-time software only to tasks. Second, perfect mapping of one component to one task (for all applications) is normally hard to determine. We illustrate this with a simple example as follows. Assume that we have a set of components $c_1,\ldots,\ c_6$ in a component library (see figure 3.1). If components are developed to be reusable that implies that the same set or a subset of components $c_1,\ldots,\ c_6$ can be used in different run-time environments (denoted RT_1-RT_3 in figure 3.1). Since different run-time environments typically

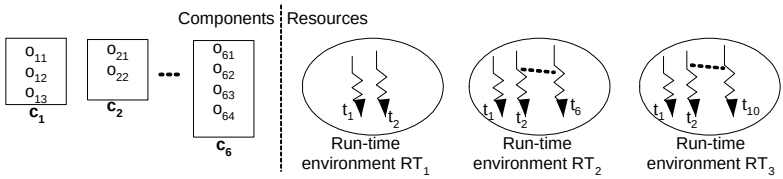


Figure 3.1: *An example of the component-to-task relationship*

have different resources available, we can observe that operations o_{ij} offered by component c_i to the environment via interfaces, could be allocated differently to tasks in different real-time environments, depending on the actual available resources of the underlying run-time environment.

The challenge now is to perform appropriate mapping of components to tasks. One way to achieve this is to distinguish between a temporal and a structural dimension of a component. The structural dimension (or the structural view) of a component represents a traditional software component as perceived by the software engineering community, i.e., a software artifact with well-defined interfaces that enforces information hiding. The temporal dimension should reflect real-time attributes needed to map components to tasks on the target platform and perform temporal analysis of the resulting real-time system. This is especially important for components used in hard real-time systems. Hence, in a component-based real-time system each component should have both a structural and a temporal dimension.

As shown, components can be mapped in numerous ways onto a run-time environment. To ensure suitable mapping for each environment guidelines or algorithms, possibly supported by tools, for component to task mapping need to be provided.

Requirements

CM4 **Component views.** A design method should support decomposition of a real-time software system into components as basic building blocks and, further, components should support both a structural and a temporal view. In the structural view real-time software is composed out of software components, and in the temporal view it is composed out of tasks.

CM5 **Task mapping.** There should be clear guidelines and tools to support mapping of components into tasks.

Having two different views of a real-time software component yields the need for explicit support of the relevant temporal properties in the component model.

Each real-time software component in its structural view should carry enough temporal attributes so that, when mapped to the temporal view, schedulability analysis of the system can easily be performed.

Requirement

CM6 **Temporal attributes.** A component model should provide mechanisms for handling temporal attributes of components, e.g., worst-case execution time, to support temporal and structural views of a real-time system, and enable static and dynamic temporal analysis of the overall real-time system.

We conclude that in the real-time domain a software component model should ensure that the relationship of a component and a task is not fixed and that components can be mapped to variable number of tasks in diverse real-time environments. Therefore, it is valuable to provide temporal and structural views of components and systems, and guidelines for mapping components to tasks. Furthermore, to facilitate component-to-task mapping, the component model should provide support for specification of temporal attributes.

3.1.2 Aspect Separation

While modularity helps to functionally decompose a system, designers would like to have modular exchange in several dimensions so that different features of components can be exchanged separately [21]. As already mentioned, the separation of concerns is the main idea behind AOSD [81]. The benefits of using aspect-orientation for developing software systems are as follows [21, 175]:

- ❏ independent development of crosscutting concerns of a system, implying that aspects of a software system can be developed independently with clear interfaces toward the system with which aspects should be woven;
- ❏ localized changes, implying that a change in an arbitrary number of places in a system can easily be carried out by simply modifying the code of an aspect;
- ❏ extensibility, implying that a software system can be extended with new functional and non-functional features by defining and weaving new aspects;
- ❏ improved comprehensibility, implying that having different features of a software system encapsulated into aspects allows reasoning about different parts of the software and their interaction separately;
- ❏ tailorability, allowing a software system to be tailored toward a target application;

- improved testability, implying that a system functionality developed independently of additional, typically non-functional, features introduced by aspects can be more efficiently tested as less software should be tested; and

- improved maintainability, implying that aspects encapsulated into modules and separated from the main software functionality enable more efficient maintainability of software as less software needs to be maintained - this, combined with the localized changes in software, allows the entire software systems (with aspects) to be more efficiently maintained.

These are the reasons why an increasing number of approaches in software engineering provide support for aspects and aspect weaving (see [19] for an overview of current aspect-oriented technology). A restricted form of aspect separation is also realized in COM and CORBA through defining multiple interfaces as access points for the component, since each of the interfaces can be viewed as one aspect [21]. Additionally, tools to support aspect specification and weaving are essential in assuring successful utilization of separation of concerns in software systems via aspects.

It is clear that the software engineering community increasingly emphasizes support for aspects and aspect weaving, giving us the following criteria with respect to aspect separation (AS) [7, 21, 81].

Requirements

AS1 **Aspect support.** To enable design and development of highly reusable and reconfigurable software systems support for identifying and specifying aspects in the software system should be ensured.

AS2 **Aspect weaving.** Tools that weave aspect specifications into the final product are needed in order to increase (re)usability of software systems.

AS3 **Multiple interfaces.** A component should have multiple interfaces through which it can be accessed to ensure that the component can be used for system composition via aspect weaving as well as system composition via component assembling.

The frontier where aspects and aspect weaving meet real-time has not been explored fully, although there is a strong motivation for using aspects in real-time system development. Namely, some of the core real-time concerns such as synchronization, memory optimization, power consumption, temporal attributes, etc., are, in typical implementations of real-time systems, crosscutting the overall system. Moreover, these concerns cannot easily be encapsulated in a component

with well-defined interfaces. In a real-time environment, not only is it desirable to support aspects that crosscut the code of the components, but also aspects that crosscut the structure of the system and describe the behavior of the components and the system [164]. This implies that for designing and developing reconfigurable and reusable real-time systems, one should provide support for multiple aspect types. We express this in the following criterion.

Requirement

AS4 **Multiple aspect types.** The notion of separation of concerns in real-time systems is influenced by the nature of real-time systems, e.g., temporal constraints and run-time environment. Thus, a real-time system design should support aspects that invasively change the code of the components, as well as additional aspect types that enable specification of properties determining the behavior of the component in the run-time environment, e.g., WCET and memory consumption.

3.1.3 System Composability

Software for real-time systems should be produced quickly, reliably, and should be optimized for a particular application or a product. We already argued that adopting CBSD and AOSD paradigms in real-time system development provides means for fulfilling these needs. It is a challenge, however, to produce real-time systems using various artifacts from a library such that the resulting system has the required functionality and exposes necessary temporal behavior. Hence, a support for system configuration in terms of tools or guidelines is necessary. A system developer should be aided in choosing a relevant subset of artifacts from the possible choices that might exist in a library. This is to ensure correct system composition and reduce the time it takes to choose exactly the components needed for a particular configuration. Hence, the following should be considered with respect to system composability (SC).

Requirement

SM1 **Static configuration.** Recipes for combining different components into system configurations that satisfy desired functional requirements are required.

Most real-time component-based software systems are pre-compiled. This implies that the resulting running system is monolithic and not dynamically reconfi-

gurable. Consequently, when updating or maintaining these systems, they need to be shut down for recompilation. However, reconfiguring a system on-line is desirable for embedded real-time systems that require continuous hardware and software upgrades in response to technological advancements, environmental change, or alteration of system goals during system operation [44, 120, 168]. For example, small embedded real-time sensor-based control systems should be designed and developed such that software resources, e.g., controllers and device drivers, can change on the fly [44, 120, 168]. Furthermore, systems like telecommunication, e-commerce, and radar applications are required to have a high degree of availability and, hence, it is not feasible to stop the execution of the system due to maintenance. Instead, on-line reconfiguration mechanisms that enable software to be updated without interrupting the execution of the system are needed.

To achieve dynamic reconfigurability in a real-time system, a number of requirements for facilitating dynamic reconfiguration should be fulfilled.

- ❏ A reconfiguration mechanisms should be light-weighted, implying that in normal operation a dynamically reconfigurable system must not introduce significant overhead in the task execution and memory consumption. This is to ensure that the system is usable environments with sparse recourses in terms of CPU and memory.

- ❏ There should not be a restriction on the number of components that could be exchanged or added/removed as the system needs to be able to evolve, e.g., due to changes in the application goals it might be necessary to include new functionality by adding one or several new components to an existing system.

- ❏ Reconfiguration may be requested at any time, and the system has no a priori knowledge of the possible components that are to be added, removed,or exchanged in the system.

- ❏ Reconfiguration must be carried out as efficiently as possible in terms of the granularity of exchangeable parts, implying that flexibility for fine-tuning of an embedded real-time application should be ensured. One should be able to exchange both functionality that is encapsulated into components and real-time algorithms that typically crosscut several components.

Accurate temporal characteristics of software components that are being added, removed, or exchanged in the system are not always available. Therefore, we need to ensure that, when reconfiguring a real-time system, the reconfiguration does not affect the performance of the system negatively, i.e., a real-time system under reconfiguration should consider the varying temporal behavior of the software components being used in reconfiguration, and adapt accordingly. The adaptation should be such that specified performance requirements expressed in terms of desired QoS, bounded worst-case QoS, and timely adaptation are satis-

fied. Hence, the following additional requirements on dynamically reconfigurable system arise.

- ❏ The system user must be able to specify the desired system QoS during steady state, i.e., the state in which no reconfiguration is applied.
- ❏ In the face of a transient state during which reconfiguration is applied, the worst-case system QoS and temporal adaptation must comply with the specified requirements. More specifically, the QoS must satisfy worst-case system QoS requirements and the QoS must converge toward and reach the desired QoS within a certain specified time interval (also known as settling time).

In summary, reconfigurable real-time systems need to provide support for the following.

Requirement

SM2 **Dynamic reconfiguration**. The system needs to be able to undergo adding, removing, or exchanging of components at any point in its operational lifetime. The overhead of performing the reconfiguration, e.g., an exchange of a components, should not introduce a significant overhead in the task execution and memory consumption of the system to ensure usability in resource-constrained embedded environments. Moreover, real-time performance needs to be maintained in the system before, during, and after reconfiguration, especially when exact temporal characteristics of components are unavailable.

To ensure that once developed, a real-time component can be (re)used for building various real-time systems one needs to ensure that appropriate techniques for satisfying real-time performance in every phase of the system life cycle (under development and run-time operation) exist.

For hard real-time systems operating in closed environments it is important to ensure that static temporal analysis of the system configuration can be done before the system is deployed into a run-time environment and, thus, enable schedulability analysis. Thereby, predictable behavior of the system can be guaranteed. Soft real-time systems operating in open environments need to maintain real-time performance in terms of assuring QoS guarantees, e.g., via the feedback-based QoS management techniques, in order to reduce the CPU overhead and efficiently handle overload situations.

Regardless of the type of the environment or an application in which the component or the resulting system configuration is going to be used, it is valuable to check temporal and functional behavior of the system in the early design phase

of the system development against the system specification. Applying formal reasoning and formal analysis techniques ensures that some errors in the design can be detected and corrected before the actual implementation of the system takes place. We epitomize this in the following requirements.

Requirements

SM3 **Temporal analysis.** Support for temporal analysis of the composed real-time system should be provided to enable reuse of components in real-time applications. Tools to achieve predictable temporal software behavior are preferable.

SM4 **QoS assurance.** Methods for maintaining real-time performance in terms of desirable QoS levels are necessary as they increase applicability of components and component-based real-time systems in open environments where exact workload characteristics are generally unknown.

SM5 **Formal verification.** Formal methods for verification of functional and temporal behavior of components and the composed system are needed to assist in detecting the shortcomings and errors in the temporal and functional design of a real-time system.

It can be concluded that it is important to provide means for static and dynamic reconfiguration of real-time systems accompanied by methods and tools for real-time performance assurance.

3.1.4 Notes on Requirements

The requirements identified in this section are not exhaustive in the sense that we identified all possible issues that development efforts for building reconfigurable and reusable real-time systems need to satisfy. Moreover, each of the identified requirements could possibly be disintegrated into smaller (or greater number of) constituents. However, the given set of requirements contains the issues that are considered by the software engineering community to be central to the development reconfigurable and reusable systems, and the issues that are considered by the real-time community to be central to development of a system enforcing real-time constraints.

The requirements discussed are both complemental and interdependent. Namely, the requirement for interfaces of a component model is complemental to the requirement for the support of multiple interface types in requirements for aspect separation. Necessity of ensuring temporal view in the component model is interdependent with the requirement for support for temporal constraints, as well as support for temporal analysis of the configured system.

It is often difficult to distinguish a clear boundary between the requirements. For example, providing the support for formal analysis requires defining formal models of components, which in turn can be viewed as analytical interfaces, or a specific type of an aspect, in a real-time system. However, to establish the consistent evaluation criteria for existing and future development efforts for reconfigurable real-time systems it is beneficial to adopt a view as perceived by a community that deals with the particular topic. In the case of formal analysis, the real-time community dealing with formal techniques considers a particular formal method to be a self-contained research effort that could be applied to a number of real-time systems, provided that they comply to a specific set of requirements identified for that particular verification method.

3.2 Existing Approaches

In this section we discuss the extent to which different approaches for building reusable and reconfigurable systems in software engineering and real-time community satisfy the identified requirements. The goal of this discussion is not to give a detailed survey of the related work[1], rather to highlight that each of the approaches is developed with a very specific and limited subset of requirements in mind. Table 3.1 illustrates the requirements and their fulfillment by different approaches.

3.2.1 Software Engineering Design Methods

The design methods for general-purpose software systems in the software engineering community are mostly targeted toward defining a component model. This is a component view taken by the first generation of CBSD systems, e.g., COM and CORBA. Both COM and CORBA provide a standardized component model with an interface definition languages (see table 3.1). Hence, once developed components in these types of systems can be reused in many applications. The need for facilitating dynamic reconfiguration of the system is a requirement stressed in the development of CBSD systems. Hover, these systems (COM and CORBA) lack support in configuration in terms of assisting a system developer in proposing adequate components for configuration, and in analysis of the assemblies [56, 131].

To enable support for analysis of modern component-based systems, a way of specifying and analyzing the correctness of the system configuration has been developed within the RADL approach [151]. RADL belongs to the class of architecture systems as it provides an ADL for describing a system configuration such that its extra-functional behavior, e.g., safety (liveness properties) and execution times, can formally be analyzed [152]. The formal analysis of assemblies in RADL is founded on finite state machines and Petri net models. The analysis process is automated with the TrustME tool suite [147].

[1]Detailed survey of the related work is presented in chapter 10.

		Real-time				Software engineering			
Design approaches / **Criteria**		DARTS	TRSD	VEST	HRT-HOOD	ISC	COM CORBA	RADL	AOP (AspectJ)
Component model									
CM1	Information hiding	◐	◐	◐	●	●	●	●	◐
CM2	Interfaces	◐	◐	◐	●	●	●	●	●
CM3	Connectors	-	-	◐	-	●	◐	●	◐
CM4	Component Views	◐	◐	●	◐	◐	-	◐	-
CM5	Task mapping	●	◐	◐	-	-	-	◐	-
CM6	Temporal attributes	◐	●	●	●	-	-	-	-
Aspect separation									
AS1	Aspect support	-	-	◐	-	●	-	-	●
AS2	Aspect weaving	-	-	◐	-	●	-	-	●
AS3	Multiple interfaces	-	-	-	◐	●	●	●	◐
AS4	Multiple aspect types	-	-	◐	-	-	-	-	-
System composability									
SC1	Static configuration	◐	◐	●	◐	●	◐	●	●
SC2	Dynamic reconfiguration	-	-	◐	-	◐	●	●	◐
SC3	Temporal analysis	-	●	●	●	-	-	◐	-
SC4	QoS assurance	-	-	-	-	-	-	-	-
SC5	Formal verification	-	-	-	-	-	-	◐	-

LEGEND: ● supported ◐ partially supported - not supported

DARTS: design approach for real-time systems
TRSD: transactional real-time system design
VEST: Virginia embedded systems toolkit
HRT-HOOD: a hard real-time hierarchical object-oriented design
ISC: invasive software composition
AOP: aspect-oriented programming
RADL: reliable architecture description language

Table 3.1: *Criteria for evaluation of design approaches*

Some design approaches [182, 21, 7, 130] have taken one step further in software configurability by providing support for separation of crosscutting concerns in the system. Normally, the support is provided for aspects and aspect weaving, thus adopting the main ideas of AOSD in general and aspect-oriented programming in particular. A typical representative of programming languages that explicitly provide ways for specifying aspects is AspectJ [182]. It is accompanied by powerful configuration tools for development of software systems using aspects written in AspectJ and components written in Java. Observe that AspectJ is a representative of aspect-oriented programming (AOP) languages and as such it provides mechanisms for implementing the system according to principles of the AOSD community. Also, pure AOSD systems, e.g., [112, 46], support only the notion of white box components, thus not exploiting the full power of information hiding. Invasive software composition (ISC) [21] overcomes the drawbacks of pure aspect language approaches, and enforces information hiding by having a well-defined component model, called the box, which is a general component model supporting two types of interfaces: explicit interfaces and implicit interfaces. Explicit interfaces are used for inter-component communication; implicit interfaces are used for aspect weaving into the code of components. Here, components can be viewed as grey boxes in the sense that they are encapsulated, but still their behavior can be changed by aspect weaving.

3.2.2 Real-Time Design Methods

There are several established design methods developed by the real-time community, and we focus on a few representative approaches [66, 67, 87, 57, 38] to demonstrate the types of requirements addressed.

DARTS

DARTS [66] developed by Gomaa is a well-established design approach for real-time systems. An Ada-based extension of DARTS is called ADARTS [67]. Gomaa has also extended DARTS to use UML for real-time system design in an approach called COMET/UML [68]. In DARTS and its variants, a real-time system is first decomposed into tasks that are then grouped into software modules (see table 3.1). Modules in DARTS typically represent traditional functions, implying white box properties and lack of enforcement of information hiding. DARTS emphasizes the need for having task structuring criteria that could help the designer to make a transition from tasks to modules. Hence, the method accentuates two views on a real-time system: (i) a temporal view where the system is composed out of tasks, and (ii) a structural view where the system is composed out of modules performing a specific function. The temporal view here only refers to representation of the system modules as tasks, but there is no support for specification of task attributes within the method. Configuration of DARTS-based systems is done off-line using configuration guidelines that have been refined through their use in industry. In

COMET/UML, the design of real-time systems is done using the UML tool environment and object-oriented technology, thereby improving information hiding as objects provide better information hiding than traditional modules (however, still lacking well-defined interfaces). As in DARTS, components in COMET/UML are not reusable since they are developed for a designated system. Although it is developed for all classes of real-time systems, DARTS does not provide means for real-time performance assurance either in system development or under its operation [87, 38]. Similarly, COMET/UML, the most recent variant of DARTS, does not provide support for temporal analysis and configurability of the system under construction.

TRSD

TRSD, a transactional real-time system design, is an approach to real-time system development introduced by Kopetz et al. [87]. Building blocks of a real-time system are transactions consisting of one or several tasks. A transaction is associated with real-time attributes, e.g., deadline and criticality. TRSD, in addition to rules for decomposition of real-time systems into tasks, provides temporal analysis of the overall real-time system. The extension of TRSD discussed in [86] complements the pure temporal view of TRSD with the structural view by establishing the notion of a component in a distributed real-time system as a distributed node (with software and the underlying hardware). To facilitate temporal analysis of the composed distributed real-time system components are equipped with temporal interfaces. However, TRSD and its recent extensions focus on system-level components that are large-grained as compared to the traditional software components as they include both software and hardware. As shown in table 3.1, the support for separation of concerns is not provided.

HRT-HOOD

HRT-HOOD [38], a hard real-time hierarchical object oriented design is an extension of the well-defined HOOD design method to the real-time domain. As such, it utilizes the HOOD tools to support the real-time design process. Building blocks of a real-time system in HRT-HOOD are HRT-HOOD objects, supplied with two different types of interfaces, namely required interface and provided interface. Having been based on the object-oriented technology and supporting different types of interfaces, HRT-HOOD enforces information hiding in terms of objects as entities that hide the information. HRT-HOOD makes a distinction between the logical and physical architectural design. The logical design results in a collection of terminal objects (these do not require further decomposition) with a fully defined interaction. At the physical design stage, the logical architecture is mapped to the physical resources on the target system. The physical design stage is primarily concerned with object allocation, network scheduling, processor scheduling, and dependability. Additionally, HRT-HOOD provides support for static priority

analysis of the overall real-time system. Although the HRT-HOOD design process is well-defined and supported by tools, it does not facilitate component reuse.

VEST

VEST [161, 164], a Virginia embedded systems toolkit, is a configuration tool for development of component-based real-time systems. VEST provides a graphical environment in which temporal behavior of the building blocks of a real-time system can be specified and analyzed, e.g., WCETs, deadlines, and periods. VEST supports two views of real-time components, temporal and structural, and assumes that components making the system configuration are later mapped to tasks. However, the actual process of mapping between a component and a task is not defined. VEST recognizes the need for having separation of concerns in the real-time system design. Hence, VEST provides support for analysis of the component memory consumption, which is a concern that crosscuts the structure of the overall component-based real-time system.

In its recent edition [164], the tool has been extended to support design-level crosscutting concerns by providing a description language for design-level aspects of a real-time system. The VEST configuration tool allows tool plug-ins, thus enabling temporal analysis of the composed system by enabling plugging off-the-shelf analysis tools into the VEST environment.

In its first version, VEST did not have an explicit component model, implying that components could be pieces of code, classes, and objects [161]. Currently VEST uses the well-defined CORBA component model [164].

RT-UML

RT-UML [57], real-time unified modelling language, provides stereotypes for specifying real-time notions. Namely, it provides support for modeling concurrency in a real-time system, i.e., identifying threads, assigning objects to threads, as well as defining thread rendezvous and assigning priorities to threads. RT-UML allows specifying and visualizing real-time properties of a component, and, thus, supports both structural and temporal dimension of a software artifact constituting a real-time system. However, we omit the detailed description of RT-UML and its evaluation since RT-UML provides syntactical notation for modeling real-time systems but still lacks well-established semantics for the real-time system design. However, there is initial work aiming at providing semantics for RT-UML for the purposes of analysis of real-time systems using timed automata [70]. Therefore, currently RT-UML cannot be considered a design method, rather it is an infrastructure for a design method as it provides a visual language as a basis for enforcing design methods, e.g., its expressive power could be used by a design method as means of specifying real-time software components [68, 50].

3.2.3 Lessons Learned

From early '80s till now real-time design methods have mostly focused on task structuring and two different views on the system and only with moderate emphasis on information hiding. The analysis of the real-time system under design, although missing from early design approaches, has been highlighted as important for real-time system development (see table 3.1). Furthermore, configuration guidelines and tools for system decomposition and configuration have been an essential part of all design methods for real-time systems so far and have, more or less, been enforced by all design methods. On the other hand, modern software engineering design methods primarily focus on the component model, strong information hiding, and interfaces as means of component communication. Also, the notion of separation of concerns is considered to be fundamental in software engineering as it captures aspects of the software system early in the system design.

Hence, most of the approaches discussed in this section both in the software engineering and the real-time community are developed with a specific subset of requirements in mind. However, to fully exploit the benefits of modern software engineering techniques for development of reconfigurable and reusable real-time systems, we need an approach that would address requirements we have found so far.

Specifically, a component model for real-time systems with two views, temporal and structural, is required to facilitate easy system composition and mapping of the components and the composed real-time system to a particular run-time environment. Separation of concerns in real-time systems through the support for aspects and aspect weaving is a valuable feature as it allows efficient component and system tailoring; this has not been fully addressed by existing real-time design approaches. Hence, a approach that would fully support aspects in the real-time system development should provide support for aspect weaving into the code of the components. Moreover, to satisfy the traditionally strong requirement for temporal analysis of the overall real-time system, a real-time development should be accompanied by methods and tools for temporal analysis of software composed using components and aspects. This requirement is essential if component-based real-time systems are used in hard real-time environments. In summary, facilitating development of reconfigurable and reusable real-time systems calls for new approaches that meet the requirements for the real-time component model, separation of concerns, and system composability.

3.3 Goals

Emergence of the new set of requirements on cost-effective development of real-time systems focusing on reuse and reconfiguration, and the limitations of existing approaches, have given rise to new research challenges in software engineering

for real-time systems. Resolving the identified issues would enable successful application of the ideas and notions from modern software engineering approaches, namely CBSD and AOSD, to real-time system development. Therefore, the goal of the work presented in this thesis is to provide:

- a component model supporting temporal and structural view of the system, and enforcing information hiding, component reuse and connection, while enabling tailoring of components for a particular application (via aspect weaving);
- support for static and dynamic reconfiguration of a real-time system assembled using components and aspects;
- mechanisms for enforcing satisfaction of real-time performance requirements of the configured system both off-line and during run-time; and
- tools for analysis and configuration of the reconfigurable system under development.

By providing these we facilitate design and development of reconfigurable and reusable real-time systems operating in open and closed environments.

Part II

Aspectual Component-Based Real-Time System Development

Chapter 4

Enabling System Reconfiguration with ACCORD

We have already argued that the growing need for enabling development of reconfigurable and reusable real-time systems calls for an introduction of new software engineering solutions for real-time system development. In that context, we have developed methods for design, configuration, and analysis of real-time software built using aspects and components. We refer to these methods collectively as the ACCORD framework to indicate that, in addition of being used in isolation, the solutions can be used together to further alleviate efficient development of reconfigurable and reusable real-time software. Therefore, in this chapter we first present how our solutions fit together in ACCORD and discuss what they offer to system developers in various phases of system development. Then, we introduce aspects, components, and aspect packages as real-time system constituents. Finally, we elaborate on the way static and dynamic system reconfiguration is done using these constituents.

4.1 ACCORD Overview

Ideally, the development process should cover all the phases of system design and development, from requirements specification to system analysis, composition, and deployment. The reconfigurable real-time system development process using ACCORD constituents is depicted in figure 4.1. As can be seen, development of a real-time system can be done both when

- components and aspects are not available in the library (steps ❶-❽), and
- there is a pre-existing library of aspects and components developed for a family of real-time systems (steps ❼-❽).

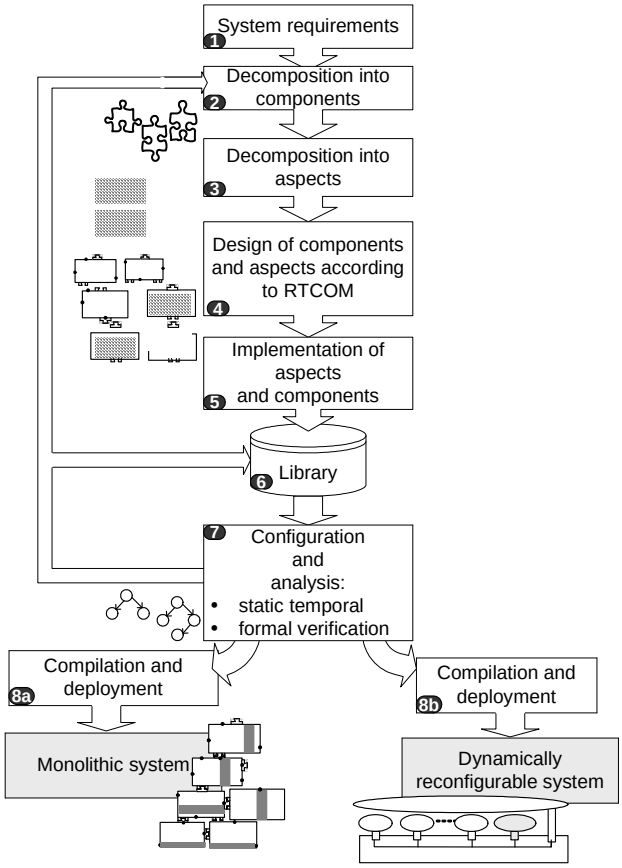


Figure 4.1: *An overview of real-time system development via ACCORD constituents*

When components and aspects are not available in a library, the design of a real-time system starts with phase ❶, the requirements specification. Our focus has been on the phases of system development that succeed requirements specification. Hence, the solutions we propose are primarily for phases ❷-❽.

System design is performed according to our design guidelines prescribing that the initial system design should be done by first decomposing a real-time system into a set of components, and then decomposing the system into a set of aspects (steps ❷ and ❸ in figure 4.1). A component encapsulates a well-defined functionality of a system that can be straightforwardly decoupled from other functionality the system carries out. Hence, decomposition into components is guided by the need to have functionally exchangeable units that are loosely coupled, but with strong cohesion. An aspect is a functional or non-functional feature of a system affecting its performance or semantics, and crosscutting the functionality or the structure of the system. Aspects enable designing, developing, and maintaining feature (or code) segments that are spread over multiple components, functions, or modules in the system, but collectively perform a particular functionality in the system. Therefore, the decomposition into aspects is guided by the need to encapsulate crosscutting features of the system into aspects. For example, an aspect can be an algorithm, e.g., QoS, scheduling, or concurrency control, as the algorithm implementation is typically entangled with the overall functionality of the system.

In step ❹, identified components and aspects are designed according to the reconfigurable real-time component model (RTCOM) that describes how a real-time component supporting aspects and enforcing information hiding should be designed and implemented. In step ❺ the actual implementation of aspects and components according to the previously made design takes place. Once implemented, components and aspects are stored in a library, grouped into so-called aspect packages (step ❻). An aspect package represents a number of components and aspects that provide a specific functionality to the system, grouped together for facilitating reuse and efficient system evolution. As we explain in detail in section 4.4, aspect packages also enable development of families of real-time systems with variations in their functionality, where variations (aspects from the package) are injected into the existing system. The library also contains models of components and aspects resulting from step ❹.

In step ❼, aspects and components are assembled to form a system configuration and analyzed to check if the configuration satisfies timeliness, e.g., allowed WCETs. In this phase a system is configured and analyzed using models of existing components and aspects from the library. Observe that this step could just as easily be performed before the implementation of aspects and components takes place (i.e., before step ❺), in which case the analysis is done on models of components and aspects, and the implementation of these is done only for the case of a positive outcome of the analysis process. The development tool set we implemented provides developers of real-time systems with automated support for this step of the development process. The analysis tools enable automatiza-

tion of the theoretical method we developed for static temporal (WCET) analysis [173, 172] of different configurations of real-time systems assembled out of aspects and components. In this step, the system configuration can also be checked using a method for modular verification of reconfigurable components. If analysis of a system configuration shows that the system does not satisfy desired properties, configuration or a design of components and aspects can be refined until a satisfactory system configuration is obtained.

Once the appropriate configuration is obtained, the system is compiled and deployed into the run-time environment in step ❽. Note that depending on the type of an application, the system can either be compiled into (a) a monolithic system, or (b) a system that is reconfigurable on-line. For example, in the case of hard real-time applications the monolithic compilation is preferable, while in applications requiring high availability at the cost of performance the reconfigurable solution could be optimal.

In the case when the system is developed from pre-existing library of aspects and components, system development starts (after the requirements are specified) in step ❽ by choosing appropriate aspects and components from the library and forming a system configuration based on application requirements.

The following is a recap of our main contributions presented earlier in section 1.3 with a relation to the development procedure in figure 4.1.

1. RTCOM is developed to support step ❹.

2. Support for static and dynamic reconfiguration of a real-time system is provided by the following.

   (a) Design guidelines defining steps ❷-❹ enable system designers to develop real-time systems using components and aspects.

   (b) A method for dynamic system reconfiguration suited for resource-constrained, time-critical, environments is represented in step ❽(b). Dynamic system reconfiguration is also supported in step ❹ by the extended guidelines for RTCOM, specifically developed to enable dynamic reconfiguration of RTCOM components and aspects.

3. Methods for ensuring satisfaction of real-time constraints as follows.

   (a) A method for static temporal analysis of real-time systems assembled using aspects and components is contained in step ❼.

   (b) A method for formal verification temporal properties of reconfigurable real-time components is also embodied in step ❼.

   (c) A method for reconfigurable quality of service that ensures that the specified level of performance is maintained during system operation and after reconfiguration is covered by steps ❸-❻ and step ❽(b).

4. A development tool set that provides developers of real-time systems automated support for configuring and analyzing a system built of components and aspects, is represented by step ❼.

In summary, ACCORD facilitates development of highly reconfigurable and analyzable real-time systems. Moreover, the developed techniques ensure that real-time systems operating both in closed and open environments can efficiently be reconfigured and evolved when the need for a new system functionality arises.

The contributions of the thesis and their relation to the overall development process are clarified further in this part of the thesis where ACCORD constituents are discussed in detail. In the remainder of the chapter, we explain guidelines for decomposition of a real-time system into aspects in section 4.2, and then present the RTCOM model in section 4.3. The concept of an aspect package is introduced in section 4.4. Finally, in sections 4.5 and 4.6, we elaborate how static and dynamic reconfiguration is performed, respectively.

4.2 Aspects in Real-Time Systems

We classify aspects in a real-time system as follows (see figure 4.2):

- ❏ application aspects (section 4.2.1),
- ❏ run-time aspects (section 4.2.2), and
- ❏ composition aspects (section 4.2.3).

Having separation of aspects in a number of categories eases reasoning about various application-related requirements, as well as the composition of a system and its integration into a run-time environment. For example, one could define what (run-time) aspects a real-time system configuration should fulfill so that appropriate components and application aspects could be chosen from the library. Aspect separation and classification offers a flexibility since additional aspect types can be added to components, and therefore, to the overall real-time system, further improving reconfigurability of the system and its integration with the run-time environment.

4.2.1 Application Aspects

Application aspects are programming (aspect) language-level constructs encapsulating crosscutting concerns that invasively change the code of a component, thus, (re)configuring a component or a system according to specific needs of an application. Hence, application aspects can change the internal behavior of components as they crosscut component code. An application in this context refers to the application toward which a real-time and an embedded system should be configured. The application aspects include (see figure 4.2): memory optimization, synchronization, security, task model, and real-time policies.

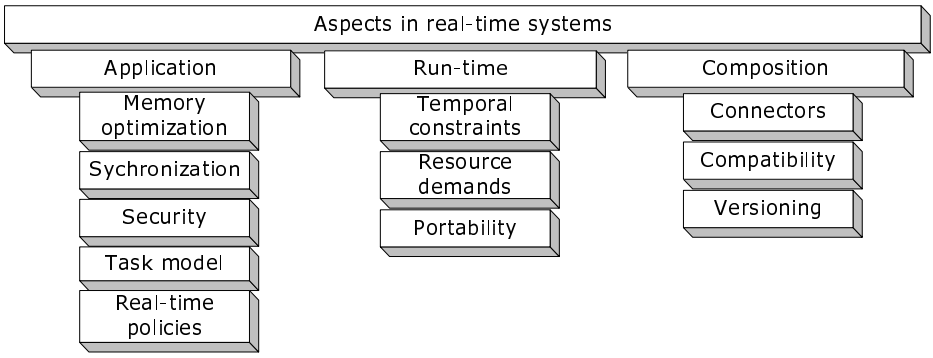


Figure 4.2: *Classification of aspects in real-time systems*

We view memory optimization as an application aspect since: (i) code for optimizing memory in an embedded system crosscuts the overall system, (ii) efficient memory usage is one of the key issues for embedded systems, and (iii) different ways of memory optimization are appropriate depending on the type of an application with which the system is to be embedded. Security is another application aspect that influences the behavior and the structure of a system. Distinct security methods might be appropriate for various types of applications, e.g., a system must distinguish users with distinct security clearances. It is also beneficial to encapsulate synchronization in a real-time and embedded system into a synchronization aspect [175]. Namely, in a typical real-time system there exists many data areas spread over the entire system that should be protected by semaphores, and the placement of semaphores can depend on the specific needs of an application. Memory optimization, synchronization, and security aspects are commonly mentioned aspects in AOSD [81]. A number of other aspects frequently referred to by the AOSD community, e.g., failure detection, logging, and recovery, can also be included in the category of application aspects.

Application aspects specific to real-time systems, and normally not found among mentioned aspects in the AOSD community, include the task model and real-time policies aspects. The task model aspects enrich the task model of a real-time system with additional properties, e.g., period, deadline, priority, WCET, and CPU utilization, to accommodate the needs of the underlying application (concrete examples of the task model are given in chapter 8).

The real-time policy aspects adapt a real-time system to a target application as they include implementation of various real-time algorithms that crosscut the overall real-time system. For example, if a real-time system is used in an open and unpredictable environment an appropriate QoS algorithm should be added to assure real-time performance guarantees. Since the QoS algorithm is highly dependent on the target application, and it crosscuts the structure of the ove-

rall system, it is beneficial to encapsulate it into an aspect. Concurrency control algorithms in the domain of real-time database systems or distributed real-time systems exhibit similar properties as QoS algorithms, and therefore can be considered as aspects in the category of real-time policy aspects. Depending on the requirements of an application, task model and real-time policies aspects could further be refined as we show in detail in the example of the COMET system (see section 8.1).

4.2.2 Run-Time Aspects

Run-time aspects are language-independent design-level constructs encapsulating crosscutting concerns that contain the parameters that determine run-time behavior of a component, e.g., WCET and memory footprint. This implies that the run-time aspects do not invasively change the code of the component.

Run-time aspects are critical as they refer to aspects of a monolithic real-time system that need to be considered when integrating the system into the run-time environment. Thus, run-time aspects give information needed by the run-time environment to ensure that integrating a real-time system would not compromise timeliness or available memory consumption. Therefore, each component should have declared resource demands in its resource demand aspect, and should have information of its temporal properties, e.g., WCET, contained in the temporal constraints aspect. The temporal constraints aspect enables a component to be mapped to a task (or a group of tasks) with specific temporal requirements.

Additionally, each component should contain information of the platform with which it is compatible, e.g., real-time operating system supported, and other hardware related information. This information is contained in the portability aspect. It is imperative that the information contained in the run-time aspects is provided to ensure predictability of the composed system and ease the integration into a run-time environment.

4.2.3 Composition Aspects

Composition aspects are twofold in their purpose as they are used to ensure functionally correct system composition, and facilitate system evolution. Version and compatibility aspects in the category of composition aspects can be viewed as language-independent design-level constructs encapsulating crosscutting concerns that describe the composition needs of each component. In contrast, connector aspects can invasively change the behavior of a component to adapt the component for communication with other, newly developed components.

Version aspects and compatibility aspects therefore describe the version of components and aspects, and compositional constraints when assembling a system using both components and aspects. Namely, the compatibility aspect eases correct functional composition of a system by providing information with which aspects and components a component can be combined.

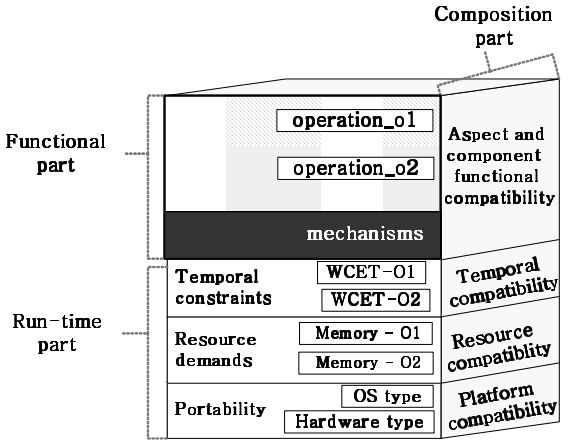


Figure 4.3: *A real-time component model (RTCOM)*

The connector aspects are used for facilitating system evolution. They represent an implementation of connectors between syntactically incompatible components. These aspects are useful when extending an existing system with a component that is not envisioned when the system was first developed.

Note that in the remainder of this thesis, when discussing aspects without explicitly naming their type, we are discussing application aspects. Also, when talking about aspect weaving we exclusively refer to aspects that can invasively change the code of the components.

4.3 Real-Time Component Model

In this section we present RTCOM, a component model that enables facile and predictable weaving of aspects, while preserving information hiding, thereby reflecting decomposition of the system into components and aspects. RTCOM consists of the following parts (see figure 4.3):

- ❏ functional part,
- ❏ run-time system dependent part, and
- ❏ composition part.

RTCOM represents a coarse-granule component model as it provides a broad infrastructure within its functional part. This broad infrastructure enables reconfiguring of a component through aspect weaving, thereby changing the functionality and the behavior of the component to suit the needs of a specific application.

RTCOM components have the grey box property as they are encapsulated in interfaces, but changes to their behavior can be performed in well-defined places of the component structure via aspect weaving. In contrast, traditional component models are typically black box, fine-grained, and allow only limited configuration of a component (see [51] for an overview of component models). Although a fine-grained component is often more optimal for one particular application in terms of code size, it does not allow component tailoring for various applications, but merely fine-tuning of the restricted set of parameters in the component [51].

For each component designed and implemented based on RTCOM, the functional part of a component is first implemented together with application aspects. Then, the run-time system dependent part is defined, followed by the composition part and rules for composing different components and application aspects.

In the remainder of this section we give details on RTCOM constituents, including functional, run-time, and composition parts, as well as supported interfaces. Using a simple example of a linked list component we exemplify how implementing RTCOM can be done, showing how weaving is performed, and illustrate the result of the weaving process. While the linked list example is presented here for clarification purposes, in chapter 7 we present the COMET system implementation where complex components and aspects are implemented using RTCOM.

4.3.1 Functional Part

The functional part of RTCOM implements the behavior of a component, i.e., represents the actual code of the component. It consists of two parts:

- ❏ mechanism part representing the invariant part of a component that exposes the black box property toward the component users and the environment; and

- ❏ policy part representing the changeable part of a component that extends the mechanism part and exhibits the grey box property toward the user of a component and the component environment.

Figure 4.4 depicts the RTCOM functional part including the mechanism and policy parts, which we describe next.

Mechanism Part

As depicted in figure 4.4, the mechanism part in itself is essentially a black box component contained within a larger (overall RTCOM) component. The mechanism part consists of a number of mechanisms, which are methods, or function calls, used by the policy part of a component to implement the component behavior. Mechanisms are private to the component and their implementation is, therefore, inaccessible to the component environment, i.e., component users and

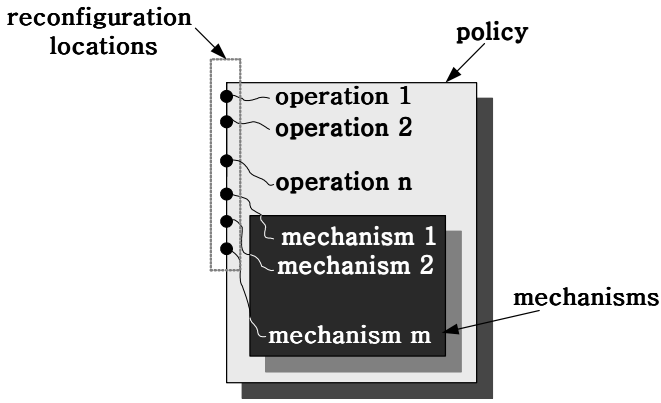


Figure 4.4: *The functional part of RTCOM*

other components. All mechanisms within a component are mutually independent, i.e., one mechanism cannot be used for implementing another component mechanism. We consider mechanisms to be basic building blocks of a component.

Policy Part

The policy part consists of a number of public methods, or function calls, used for component composition and communication. We refer to these methods as operations. An operation in a component represents the functionality or a service that a component can provide to other components or the overall system. Operations, therefore, compose the internal behavior, i.e., policy, of a component.

The relationship between the operations in a component, and in the overall system, should be such that the control flow that determines component communication forms a directed acyclic graphs (DAGs), as shown in figure 4.5 (a). That is, it is not allowed that operation o_1 is implemented using operation o_2, which in turn is implemented using operation o_3, and operation o_3 gives a recursive cycle by being implemented using operation o_1 (see figure 4.5(b)). Having the recursive calls to operations in the component, and between different components, makes temporal analysis of the system composed out of components inherently difficult [138]. Hence, RTCOM in its current form only supports operations that are implemented such that the control flow between them forms a DAG.

An operation is implemented using the underlying component mechanisms. Hence, operations can be viewed as coarse methods (or function calls) as they are implemented using finer methods, namely component mechanisms. An example of how operations and mechanisms of a component could be related is given in figure 4.6, where operation o_1 is implemented using component mechanisms m_1 and m_3, while operation o_2 is implemented using mechanism m_2. Furthermore,

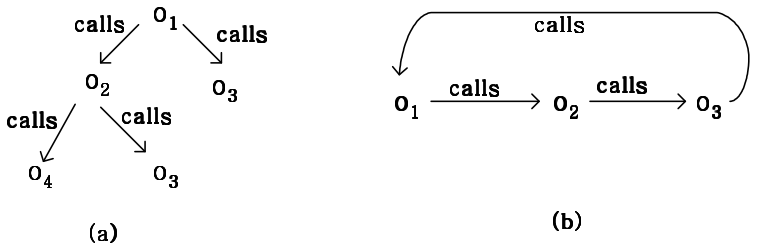


Figure 4.5: *An example of (a) allowed, and (b) not allowed relationship among the operations*

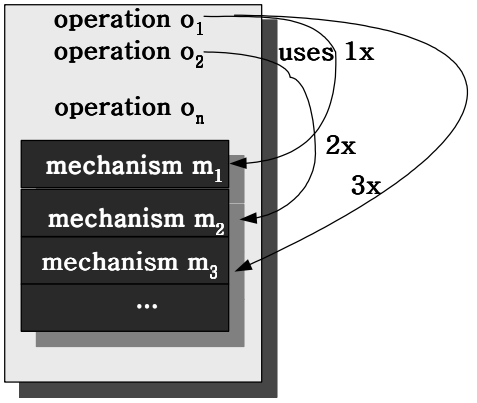


Figure 4.6: *An example of the relationship of operations and mechanisms in a component*

each operation in the component can use a mechanism in its implementation one or several times. In the example in figure 4.6 operation o_1 uses mechanism m_1 once and mechanism m_3 three times. The "uses relation" of operations and mechanisms reflects that the flow of control in operation o_1 is transferred to m_1 once and m_3 three times.

To facilitate weaving of aspects into a component, design of the functional part of the component is performed in the following manner. First, mechanisms as basic blocks of the component are identified and designed. Here, particular attention should be given to the previously identified application aspects, and the table that reflects the crosscutting effects of application aspects to components could be made to help the designer in the remaining steps of RTCOM design and implementation. Next, the operations of a component are designed using component mechanisms. Note that the operations provide an initial component policy,

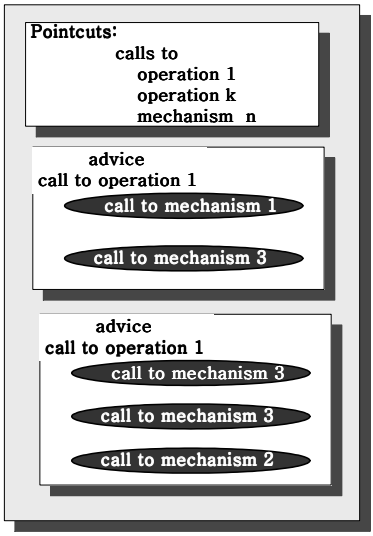


Figure 4.7: *An illustration of the structure of an aspect that invasively changes the behavior of a component*

i.e., basic and somewhat generic component functionality. This initial policy can be modified by weaving various application aspects. If identified crosscutting application aspects are considered when designing and implementing operations and mechanisms, then the resulting functional part of a component is highly reconfigurable. All the places where changes to the component via aspect weaving can be done, i.e., operations of the policy part and mechanisms of the mechanism part, are simply denoted reconfiguration locations (represented with black circles in figure 4.4).

Ensuring aspect weaving such that the grey box property of a component is preserved, and weaving is done in a predictable manner, is a primary concern of RTCOM. Given that the functional part of a component can be changed via aspect weaving, in the remainder of this section, we discuss the way aspects should be designed to support predictable weaving. By predictable weaving we mean the integration of aspects into component code such that further analysis of designs and implementations based on RTCOM is enabled. In the description of aspects that follows, we focus on aspects that can invasively change the code of the component. Hence, we focus on design and implementation of application aspects, and, as mentioned before, for presentation reasons we refer to these simply as aspects.

Aspects and Aspect Weaving

Within RTCOM we take the traditional view of programming language level aspects, and adapt it to the real-time domain by specifying pointcuts and advices in terms of mechanisms and operations (see figure 4.7). This enables performing temporal analysis[1] of the weaved system, and thereby use of aspects in real-time environments. This also enables existing aspect languages to be used for implementing aspects in real-time systems, and enables existing weavers to be used to integrate aspects into components while maintaining predictability of a real-time system, i.e., ensuring that temporal behavior of a composed system can be estimated closely to the actual behavior (exposed during run-time) before running the system. Hence, aspect are designed such that pointcuts refer to reconfiguration locations (operations and mechanisms) of available components. This implies that a pointcut in an aspect can point to one or several operations or mechanisms of a component, indicating these as places where modifications of the component code are allowed, as depicted in figure 4.7. To facilitate temporal analysis, the advices of an aspect are also implemented using component mechanisms as basic building blocks. Furthermore, the implementation of an entire aspect is not limited only to mechanisms of one component, since an aspect can contain any finite number of advices that can precede, succeed, or replace reconfiguration locations throughout the system configuration. Hence, aspects can be implemented using a number of mechanisms from several components.

Now, by defining adequate advices within an aspect, weaving can be done before, after, or around a reconfiguration location. Note that when weaving a mechanism, operations using this mechanism are affected by weaving, while the mechanism implementation remains intact and its black box behavior uncompromised. This is because weaving a mechanism adds code to the operation (before/after advices) that is going to be executed before or after the call to the mechanism is made. In the case of an around advice, the call to the mechanism within the operation is replaced. From this it follows that operations are flexible parts of the component as their implementation can change by aspect weaving, while mechanisms are fixed parts of the component infrastructure. Each advice within an aspect can change the behavior of a component by changing one or more operations in the component.

Once the component functional part is designed and implemented, and an aspect that invasively changes it is developed, the weaving can take place. By using an appropriate aspect weaver, the resulting woven component is obtained. As depicted in figure 4.8, weaving is adding code defined within advices to the original component in the places defined by reconfiguration locations and in the manner prescribed by pointcut expressions and advice types. Therefore, one can consider that the development process of the functional part of a component eventually results in a component woven with aspects.

[1]Temporal analysis refers both to static WCET analysis of the code and dynamic schedulability analysis of the tasks.

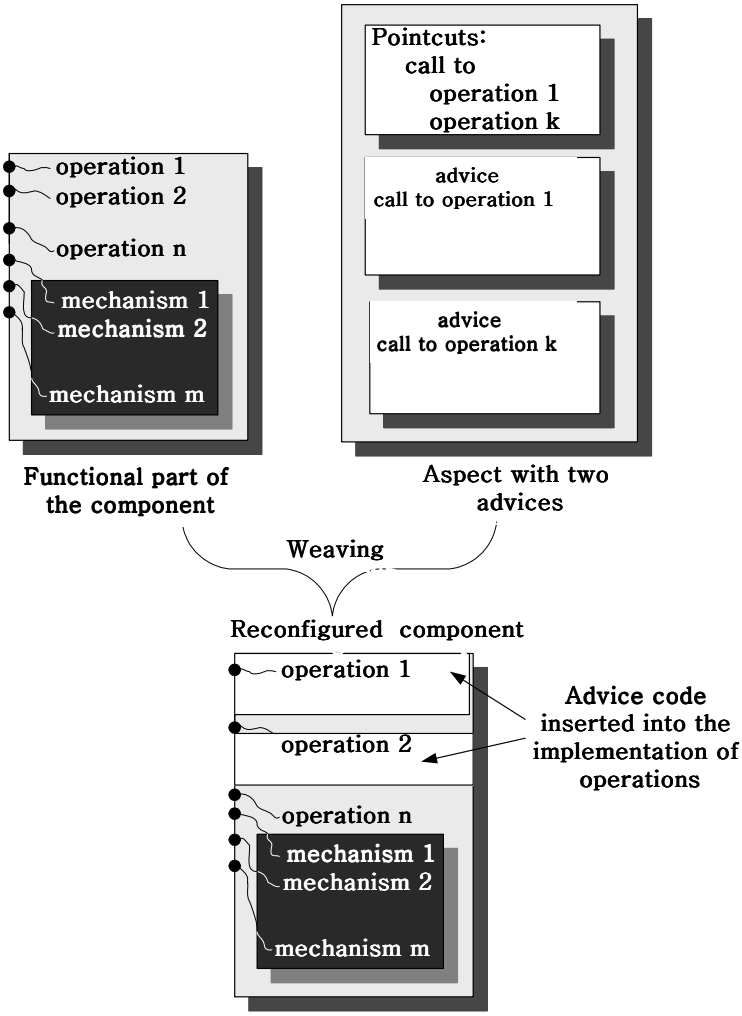


Figure 4.8: *An example of component reconfiguration via aspect weaving*

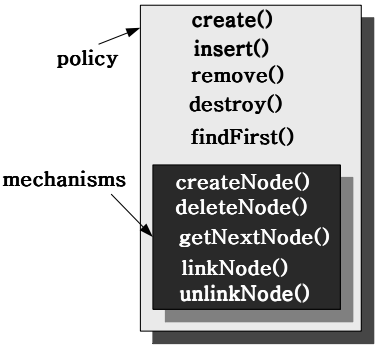


Figure 4.9: *The functional part of the linked list component*

Example

Consider an example of an ordinary linked list implemented based on RTCOM. The functional part of the component consists of the mechanism and the policy part. The mechanisms needed are the ones for the manipulation of nodes in the list, i.e., `createNode`, `deleteNode`, `getNextNode`, `linkNode`, and `unlinkNode` (see figure 4.9). Operations implementing the initial policy, e.g., `create`, `insert`, `remove`, and `findFirst`, define the behavior of the component, and are implemented using the underlying mechanisms. In this example, `insert` uses the mechanisms `createNode` and `linkNode` to create and link a new node into the list in first-in-first-out (FIFO) order. Hence, the initial policy of the component is FIFO.

Assume that we want to change the policy of the component from FIFO to priority-based ordering of the nodes. This can be achieved by weaving an appropriate application aspect. Figure 4.10 shows the `listPriority` application aspect, which consists of one pointcut `insertCall`, identifying `insert` as a join point in the component code (lines 2-3). When this join point is reached, the code in the `before` advice `insertCall` is executed. Hence, the application aspect `listPriority` intercepts the operation `insert` and, using the component mechanisms (`getNextNode`), determines the position of the node based on its priority (lines 5-14). The outlook of the woven component is depicted in figure 4.11.

4.3.2 Run-Time System Dependent Part

The run-time part of RTCOM accounts for run-time aspects of a component, including resource consumption and temporal behavior of both the original component (without application aspects) and the reconfigured component (when application aspects are woven into the component). Thus, a run-time aspect should

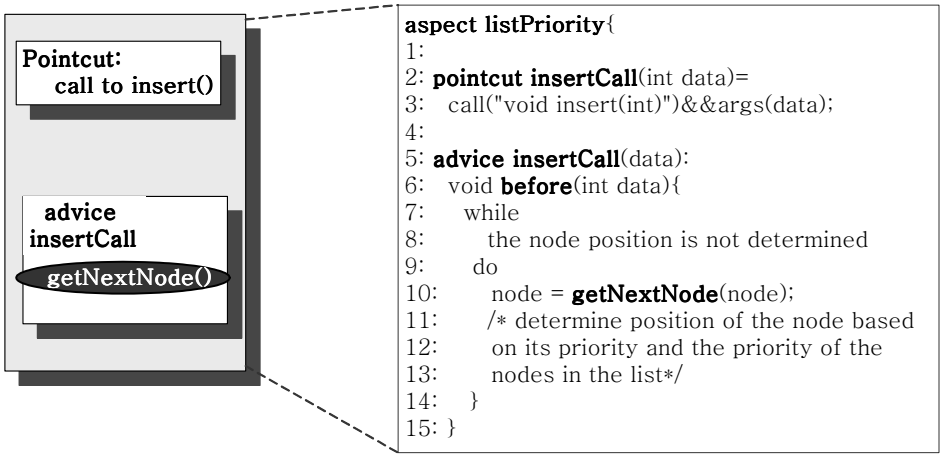


Figure 4.10: *The `listPriority` application aspect*

be specified both for the functional part of a component and for the application aspects. In the run-time part of a component, the constituents of run-time aspects are expressed as attributes of operations, mechanisms, and application aspects since they are the elements of the functional part of the component, and thereby influence the behavior of the component with respect to timeliness and available memory consumption.

Next, we illustrate how run-time aspects are represented and handled in RTCOM using two representative run-time aspects, one for illustrating temporal constraints (WCET) and one for illustrating resource demands (memory footprint). We already established that knowing WCETs is imperative for enabling schedulability analysis of hard real-time systems. Additionally, prediction of the static memory usage, i.e., memory footprint, is important in embedded systems having small amount of available memory. We exemplify the overall run-time aspect specification on the running example of the linked list component and `listPriority` application aspect.

The specification of static memory and WCET needs of components is done based on the following observations related to aspect weaving in the code of a component.

- ❏ Aspect weaving does not change static temporal behavior of a mechanism within a component nor its memory requirements since implementation of the mechanism is not changed by aspect weaving.

- ❏ Aspect weaving changes operations by changing the number of mechanisms that an operation uses, thus, changing static temporal behavior and memory consumption of the operation.

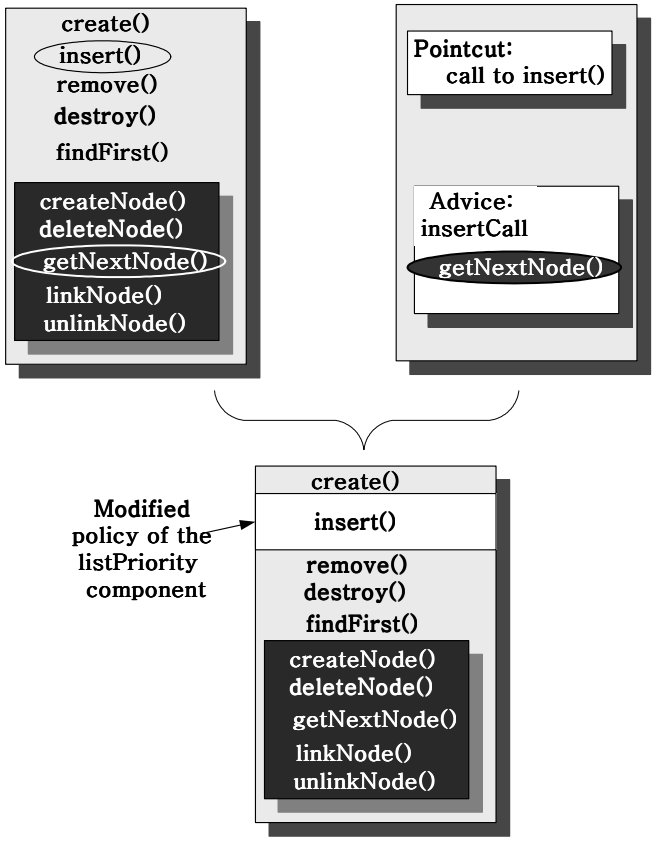


Figure 4.11: *The resulting woven component*

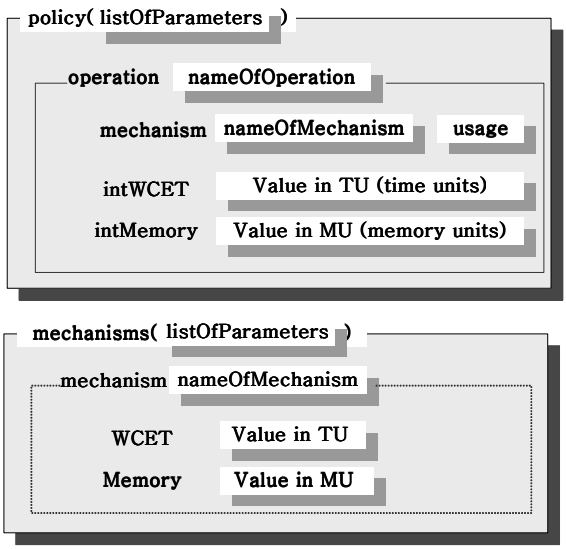


Figure 4.12: *A general specification template for run-time aspects*

- ❏ Aspect weaving augments the code of an operation in the case of weaving the before or after advice, and replaces the code of an operation in the case of the around advice.

To facilitate efficient WCET and memory footprint analysis of different configurations of aspects and components, the specification of WCETs and memory footprint within the run-time part of RTCOM should satisfy the following.

- ❏ WCETs and memory consumption of mechanisms are known and declared in the specification (illustrated in the lower part of figure 4.12).

- ❏ The WCET and memory footprint of an operation is determined based on two elements: (i) the WCETs and memory footprints of the mechanisms, and (ii) the internal values of WCET and memory footprint of the operation (see the upper part of figure 4.12). The internal WCET and memory footprint of an operation refer to the code of the operation not associated with the invocation of the mechanisms; we also refer to this code as the body of the operation.

- ❏ WCETs and memory requirements of every advice is based on: (i) the WCET and memory requirements of the mechanisms used for implementing the advice, and (ii) the internal WCET and memory footprint of the body of the advice, i.e., code that manages the mechanisms (see figure 4.13).

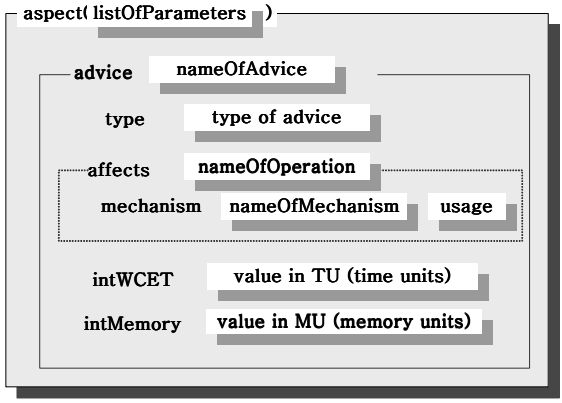


Figure 4.13: *An example of the specification template for the run-time part of an application aspect*

Figure 4.12 shows the template for specification of a run-time aspect within RTCOM. As can be seen from the bottom part of the figure, for each mechanism the WCET and memory needs are declared and assumed to be known. WCET values are expressed in time units, e.g., seconds, milliseconds, and microseconds. Memory footprint values are expressed in memory units, e.g., bytes and kilobytes. In the run-time aspect specification of a component, each operation defining the policy of the component declares what mechanisms it uses, and how many times it uses a specific mechanism. Figure 4.13 shows the run-time specification of an application aspect. For each advice type (before, around, after) that modifies an operation, the operation it modifies is declared together with the mechanisms used for implementing the advice, and the number of times the advice uses these mechanisms. Run-time specifications of aspects and components can also have a list of parameters used for expressing the values of WCETs and memory requirements.

Figure 4.14 presents an instantiation of a run-time specification for the linked list component. Each operation in the component is named and its internal WCET `intWcet` and memory consumption `intMemory` are declared. Additionally, the number of times an operation uses a particular mechanism is declared. Since the maximum number of elements in the linked list can vary, the specifications are parameterized with parameter `N` representing the number of nodes in the list. Figure 4.15 shows the run-time specification of the `listPriority` application aspect. The run-time properties of advice `insertCall` are specified by declaring that the advice is of type before and that it modifies operation `insert` by invoking mechanism `getNextNode N` times. The advice also has the internal WCET and memory footprint, accounting for run-time properties of the code that handles mechanism invocations.

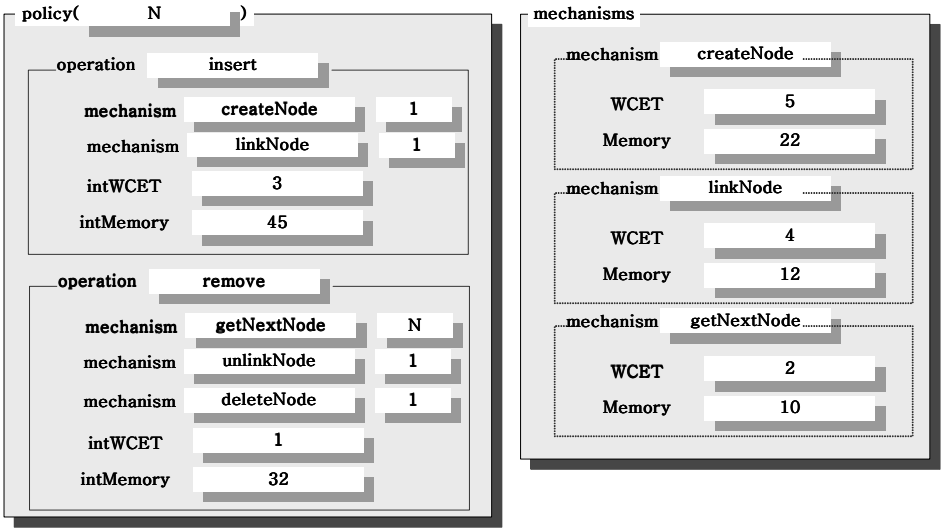


Figure 4.14: *A specification of the run-time aspect of the linked list component*

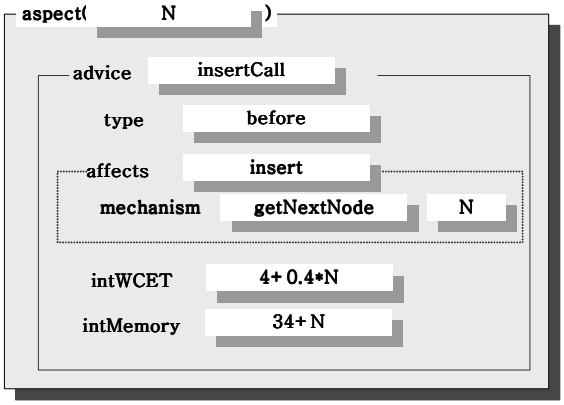


Figure 4.15: *A specification of the run-time aspect of the priority list application aspect*

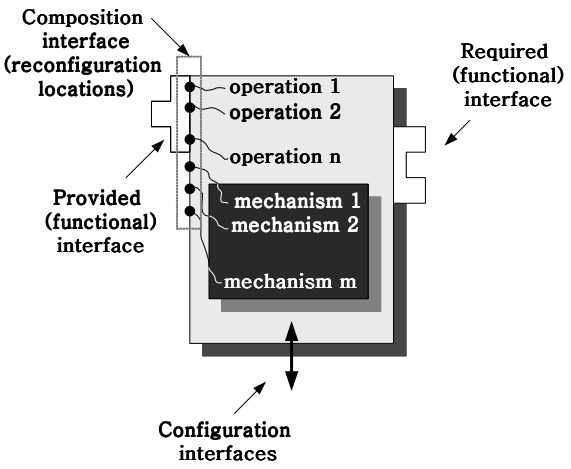


Figure 4.16: *Different types of interfaces defined within RTCOM*

We continue this example of run-time specifications of the linked list component in section 5.2 and use it to illustrate the way static analysis is performed within ACCORD.

4.3.3 Composition Part of RTCOM

The composition part refers both to the functional part and the run-time part of a component, and is graphically represented as a third dimension of the component model (see figure 4.3). Given that there are different application aspects that can be woven into a component, composition aspects represented in the composition part of RTCOM should contain information about component compatibility with respect to different application aspects, as well as with respect to different components. This part of RTCOM has not been a main focus of our work so far, and in its current form only supports, for a given component, simple composition rules listing (by name) compatible aspects and components.

4.3.4 RTCOM Interfaces

RTCOM supports three different types of interfaces (see figure 4.16):

- ❏ functional interface,
- ❏ configuration interface, and
- ❏ composition interface.

Typically, in a component-based software system, a component functional interface specification reflects operations of the component. Namely, the interface

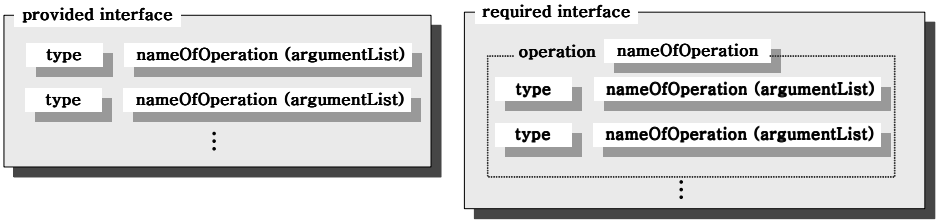


Figure 4.17: *The functional provided and required interface supported by RTCOM*

specification provides the name, type, and parameters of the operation, e.g., input, output, or input/output parameters [50].

Functional Interface

Functional interfaces of components are used for component communication and system configuration. Functional interfaces are classified in two categories:

- ❏ provided functional interfaces, and
- ❏ required functional interfaces.

The provided interface reflects a set of operations that a component provides to other components, as shown in figure 4.17. The required interface reflects operations that a component requires from other components. Namely, in this interface each operation provided by a component declares which operations from other components it needs (see figure 4.17). Having explicit separation into provided and required interfaces eases component exchange and addition of new components into the system [37].

Configuration Interface

The configuration interface lists changeable parameters for each operation that can be instantiated with different values depending on the target run-time environment. As such, the configuration interface is intended for supporting integration of a real-time system with its targeted run-time environment.

As illustrated in figure 4.18, this interface provides a set of parameters that can be changed so that the mapping of the component on the target run-time environment is eased, e.g., the parameter can be a number of tasks that are supported in the underlying run-time environment. Combining multiple components results in a system that also has a configuration interface (the collection of configuration interfaces of components), which enables the designer to further tune the behavior of the system toward the run-time environment.

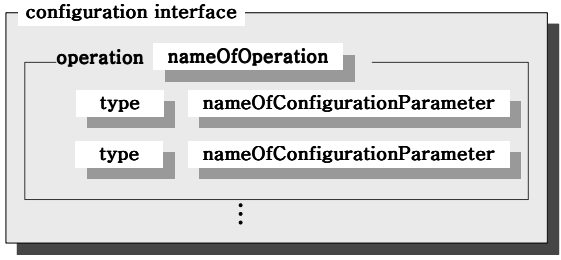


Figure 4.18: *The configuration interface of RTCOM*

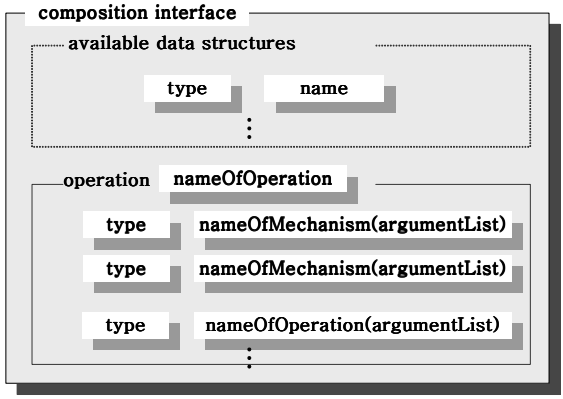


Figure 4.19: *The composition interface of RTCOM*

Composition Interface

The composition interface of a component represents a set of mechanisms, operations, and data structures of the component that can be used for aspect weaving (see figure 4.19). Hence, the composition interface is a precise declaration of reconfiguration locations of a component. The user of the component, e.g., system designer, uses this interface when extending the component or a system with new aspects or reconfiguring the component/system for new reuse contexts. Composition interfaces are ignored at component/system compile time if they are not needed; they are "activated" only when certain aspects are woven into the system. Thus, the composition interface eases integration of the component and aspectual part of the system.

4.4 Aspect Packages

The most common way of ensuring that real-time performance is guaranteed in a software system is to employ a certain real-time policy, e.g., scheduling policies are employed in the system to ensure that tasks meet their respective deadlines, and QoS policies are used for maintaining performance in open unpredictable environments. Numerous approaches developing new real-time policies or fine-tuning existing ones have been developed over the years, e.g., [15, 11, 40, 108, 110, 133, 45, 99]. However, these have focused primarily on providing a specific real-time policy, not addressing software engineering issues relating to configurability of policies and their reuse across application areas. Traditionally, real-time policies are implemented as an integral part of a real-time system, crosscutting its structure and cannot easily be modified, exchanged, or reused.

To ensure that real-time policies can be reused and reconfigured in newly developed and existing systems across applications, we define and use the notion of an aspect package, which represents a way of packaging specification and implementation of real-time policies for reuse and configuration. An aspect package consists of a number of components and aspects implementing a variety of real-time policies.

For example, when studying existing QoS management approaches we observed that a majority of approaches typically assume that a real-time system has a QoS management infrastructure upon which algorithms implementing specific QoS policies are implemented. The infrastructure can consist of services, functions, or methods provided by the system for adjusting the system load and is implemented as an integral part of the system. Algorithms implementing QoS policies use services provided by the infrastructure to ensure performance guarantees, and they often crosscut other parts of the system as well. Notable, many QoS management approaches have a similar QoS infrastructure but provide distinct QoS policies, i.e., have distinct QoS algorithms implemented on top of the infrastructure. For these reasons, a concept of an aspect package is useful as it provides a way of decomposing and implementing QoS management for a family of real-time applications. The decomposition is done such that the main functional infrastructure of QoS management is encapsulated into a set of components and various QoS policies crosscutting the QoS management infrastructure and the structure of a system are encapsulated into aspects. Hence, for a family of real-time applications a QoS aspect package has a unique set of components (as in this case all QoS policies in the family use the same infrastructure), and possibly a great number of aspects implementing various QoS policies.

Similarly, a concurrency control aspect package could be defined as a collection of aspects and components that implement a number of different concurrency control policies. The concurrency control aspect package can be used for development of a family of real-time systems with variations in their concurrency control.

Using an aspect package enables a system designer to develop several app-

lications with similar characteristics with respect to a certain real-time policy. Therefore, each family of applications would have its unique aspect package. By adding aspects and components from an aspect package to an existing system, we are able to use the same system in applications with distinct needs. For example, using aspects and components from the QoS aspect package, it is possible to configure QoS management of a real-time system based on the application requirements. Similarly, using aspects and components from the concurrency control aspect package, it is possible to configure the system to support different policies for resolving conflicts on data items, which are simultaneously accessed by multiple tasks.

Since we found that the concept of an aspect package is especially useful in for enabling reconfiguration and reuse of real-time policies in existing real-time systems, we first outline the main requirements an existing real-time system should fulfil to be able to employ the concept of an aspect package. We then present an example of a QoS aspect package. The purpose of this example is to provide guidelines for development of aspect packages as the described way of determining constituents of a QoS aspect package can be used for determining the constituents of most aspect packages.

4.4.1 Requirements

The concept of an aspect package as the means of configuring and extending an existing system applies both to the class of traditional (monolithic) real-time systems and to the class of component-based real-time systems, provided that they conform to the following requirements.

Traditional real-time systems should: (i) be written in a language that has a corresponding aspect language; (ii) have the source code of the system available; and (iii) have well-structured code such that the code is structured in fine-grained pieces that perform well-defined functions, i.e., good coding practice is employed.

Configurable, component-based, real-time systems should be built using "glass box" or "grey box" component models. These models imply that components have well-defined interfaces, but also internals are accessible for manipulation by the software developer, e.g., Koala [179], RTCOM [174], PBO [166], AutoComp [150], and Rubus-based component models [79].

In addition, both monolithic and configurable real-time systems should have functions upon which components and aspects from the aspect package can be added. For example, in the case of the QoS aspect package a real-time system should have functions for controlling the system load. Namely, there are multiple ways of controlling the load in the system, e.g., by changing the output quality of the tasks [105], modifying the period of the tasks [108, 45], admission control [16], and changing the frequency of the CPU [186]. This implies that the tasks must be scheduled by an on-line scheduler [40], e.g., EDF or RMS [104]. In the case of a concurrency control aspect package, the system should have functions for data access.

Given that a system conforms to the named requirements, an aspect package can be used for adding and configuring the majority of real-time policies. When a real-time systems is built from scratch, then the system can be optimized during design and development using ACCORD.

4.4.2 An Aspect Package Example

In this section we show how an aspect package can be identified and defined on an example of the QoS aspect package. The details of the concrete implementation of aspect packages for COMET database, including QoS and concurrency control, are discussed later in chapter 7.

Any aspect package essentially consists of two types of entities: components and aspects (see figure 4.20). The top level of the aspect package can be used for any policy. At the lower level, the components are refined for a specific set of policies. In the remainder of this section we discuss feedback-based QoS management, where components include a feedback controller component (FCC), a QoS actuator component (QAC), and a sensor component. The aspects are constructed to embrace the following (see figure 4.20):

- ❏ QoS policy aspects,
- ❏ QoS task model aspects, and
- ❏ QoS connector aspects.

The components should conform to the RTCOM model. As such, a component has an interface that contains: (i) functionality in terms of functions, procedures, or methods that a component requires (uses) from the system, (ii) functionality that a component provides to the system, and (iii) the list of reconfiguration locations where the changes of the component policy can be done. Reconfiguration locations are especially useful in the context of producing families of systems with variations in their QoS management.

The QAC is a component that its simplest form acts as a simple admission controller. It publishes a list of reconfiguration locations in its interfaces where different actuator policies can be woven. Similarly, the FCC is by default designed with a simple control functionality and appropriate reconfiguration locations such that the FCC can be extended to support more sophisticated control algorithms, e.g., adaptive control [146]. The sensor component collects necessary data and possibly aggregates it to form the metric representing the controlled variable. In its simplest form the sensor component measures utilization, which is commonly used as a controlled variable [108]. The sensor component publishes a set of reconfiguration locations, where it is possible to change the measured metric.

The QoS management policy aspects, namely actuator policy, controller policy, and sensor policy, adapt the system to provide different QoS management policies. Depending on the target application, the aspects modify the FCC to

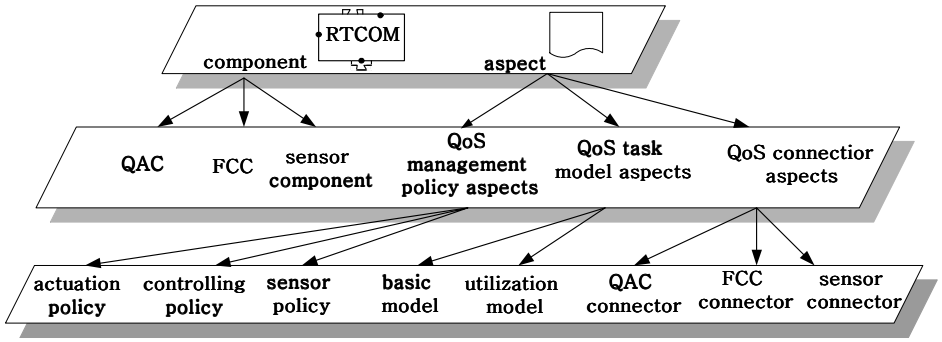


Figure 4.20: *A QoS aspect package*

support an appropriate QoS controller, and also change the QAC and the sensor component according to the choice of manipulated variable and controlled variable, respectively. For example, if deadline miss ratio is to be controlled, then a QoS policy aspect measuring deadline miss ratio is chosen. The QAC is modified by the aspect, exchanging the admission policy for an actuation mechanism where the quality of the task results are modified.

The QoS task model aspects adapt the task model of a real-time system to the model used by QoS policies, e.g., a utilization task model needs to be used for a feedback-based QoS policy where utilization of the system is controlled. There can be a number of aspects defined to ensure enrichments or modifications of the task model, e.g., by adding various attributes to tasks, so that the resulting task model is suitable for distinct QoS or applications needs. Concrete examples of task models are given in chapter 8.

The QoS connector aspects facilitate the composition of a real-time system with the QoS-related components FCC, QAC, and the sensor component.

Once a QoS aspect package is specified as described above, i.e., components and aspects identified, it needs to be populated with actual implementations of aspects and components for a particular application or a family of applications. An existing system that complies with requirements from section 4.4.1 is configured for the specific real-time QoS management as follows.

- ❏ The architecture of the existing system is inspected and reconfiguration locations are identified.
- ❏ The new context in which the system is going to be used, i.e., the application domain, is determined.
- ❏ Given the new usage context of the system, a suitable QoS policy consisting of the sensor policy, controlling policy, and the actuation policy, is identified.
- ❏ If the aspects implementing the QoS policy do not exist in a QoS aspect package, then the corresponding aspects are defined, developed, and added

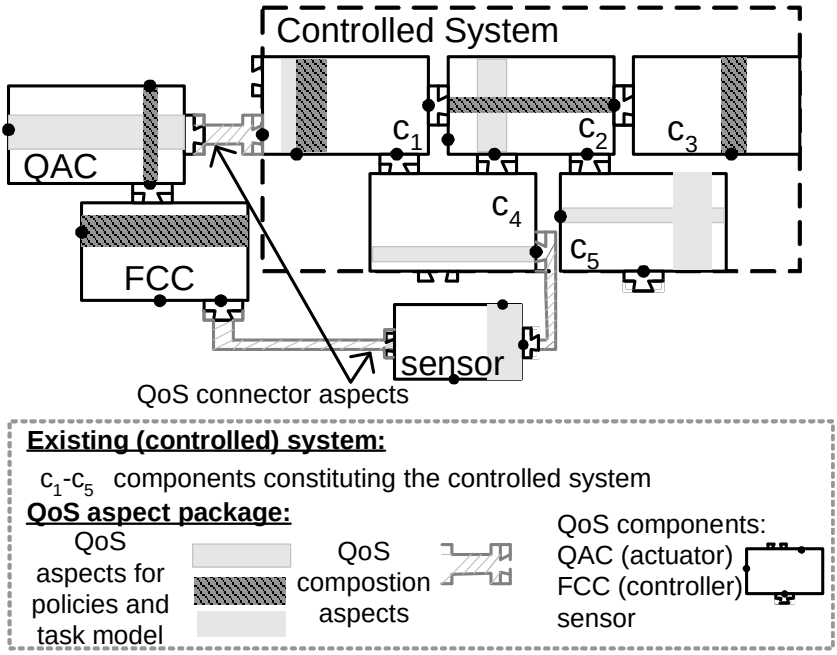


Figure 4.21: *A real-time system where QoS aspect package is used for configuring QoS management*

to the package. Aspects use reconfiguration locations in the existing system to inject the QoS policy. Similarly, the QoS aspect package has to be populated with a sensor, FCC and QAC components, in case these do not already exist in the package.

- ❑ Aspects and components for a specific QoS configuration are chosen from the aspect package and woven with the existing system.

Now, a real-time system can be developed so that it fulfills certain functionality (without QoS guarantees), and then QoS management can be added to the system using the QoS package in the previously described procedure.

Note also that aspects implemented within a QoS aspect package can easily be reused in different applications. Moreover, the QoS aspect package enables closed systems to be efficiently used in open environments. It is indeed possible to design a real-time system without QoS management and then add the QoS dimension to the system using the QoS aspect package. We prove these claims in chapter 8 on the example of the COMET database.

A real-time system where a QoS aspect package is applied is shown in figure 4.21. In this figure it is depicted that a controlled (existing) system is a

component-based system that consists of RTCOM components $c_1, \ldots, c_5$, which have well defined interfaces and internals accessible for modification.[2]

QoS connector aspects from a QoS aspect package are used to add the sensor, FCC, and QAC to the parts/components of the system or a system itself, where needed; these aspects are represented with grey dashed lines between component connections in figure 4.21. Additionally, QoS connector aspects offer significant flexibility in the system design as the feedback loop can easily be placed "outside" the controlled system and between any components in the system by simply adding QoS connector aspects. QoS management policies are added to the system using appropriate aspects from the package. Moreover, QoS policies can easily be exchanged by adding or changing aspects within the QoS management policy type. Hence, a QoS aspect package ensures that QoS management policies are modifiable and configurable, depending on the application requirements.

4.5 Static Reconfiguration

In this section we elaborate on how static real-time system reconfiguration is done from components and aspects that are previously designed and implemented based on RTCOM, and placed in a library. Static system configuration is the preferred type of system composition for hard real-time systems operating in closed environments.

To facilitate system evolution and its reconfiguration, we utilize the concept of aspect packages. Hence, we group components and aspects constituting real-time policies of a system into aspect packages. Now, a library can be populated with components, aspects, and possibly aspect packages that can be used for developing a real-time system or a family of real-time systems. The overview of the composition process from software artifacts from the library is illustrated in figure 4.22.

The developer chooses appropriate aspects and components, and configures a system based on application requirements. The resulting system configuration is monolithic and is deployed on the target run-time environment as a monolithic system. If the system needs to be modified during its lifetime to support new functionality this can be achieved by static reconfiguration. Namely, upgrades of the system via aspect packages that provide needed functionality are done by re-compiling the system off-line.

4.6 Dynamic Reconfiguration

We already established (in chapter 3) that reconfiguring a system on-line is necessary for embedded real-time systems that require continuous hardware and soft-

[2]The explanation and the main points are the same as if the depicted controlled system would be a traditional monolithic real-time system conforming to the requirements listed in section 4.4.1.

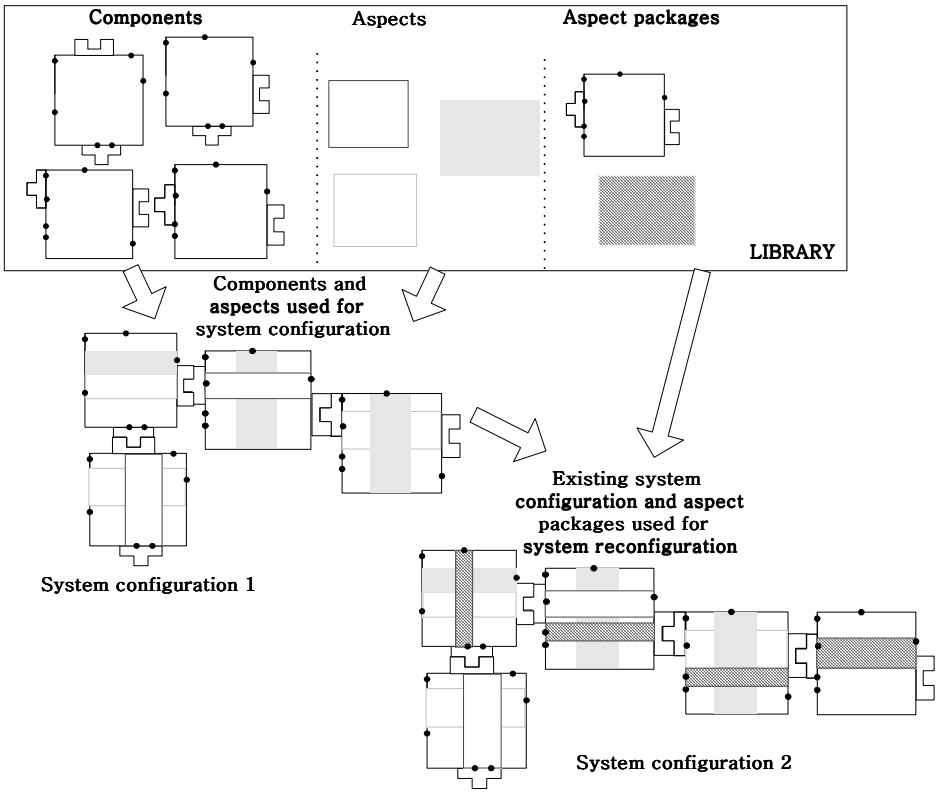


Figure 4.22: *Static (re)configuration of a system*

ware upgrades in response to technological advancements, environmental change, or alteration of system goals during system operation [44, 168]. Hence, an on-line reconfiguration mechanism is needed ensuring that software can be updated without interrupting the execution of the system.

To enable dynamic reconfiguration we extend the RTCOM component model to ensure preservation of component and aspect states during reconfiguration. Furthermore, we introduce a middleware layer, which handles communication among components and aspects.

Note that, in the context of ensuring real-time performance, an exchange of a component encompasses all the dimensions of the dynamic reconfiguration as it includes also removal of the old version of a component and the addition of a new version. Therefore, in the following, we focus primarily on explaining the component exchange[3] part of the dynamic reconfiguration.

4.6.1 Extensions to RTCOM

To preserve the internal state of components and aspects under reconfiguration, RTCOM is extended as follows. The provided interface of a component is extended with two mandatory operations, `export` and `import`. The `export` operation enables a component under reconfiguration to export its state, i.e., to store its state outside its data space (in the middleware layer). The `import` operation ensures that the exported state of the component, which is to be replaced, is correctly imported into the new version of the component.

Enabling aspect exchange implies changing the way aspects are implemented within RTCOM. To conform to the dynamic aspect reconfiguration, a feature not supported by current aspect languages, and still enable general applicability of our dynamic reconfiguration method with any of the aspect and/or component languages, we augment aspects with provided and required interfaces. For dynamic reconfiguration purposes, the code of advices is encapsulated into methods, or function calls, and these are declared in the provided interface of the aspect. The required interface lists all the reconfiguration locations of components that are used in the pointcut expressions of the aspect. Provided and required interfaces play an important role in communication and reconfiguration as we describe later. Observe also that having interfaces defined for each aspect provides better encapsulation of aspect functionality and still preserves the crosscutting nature of an aspect.

System composition out of components and aspects now consists of the following steps. First, aspects are woven off-line into components they affect. Then, using information stored in interfaces of components and aspects, these are translated into run-time entities (shared objects) recognized by the middleware layer. The run-time entities are then deployed onto the middleware layer (see figure 4.23). Conceptually, this means that weaving and run-time entity gene-

[3] A component exchange is also referred to as a live update of a system.

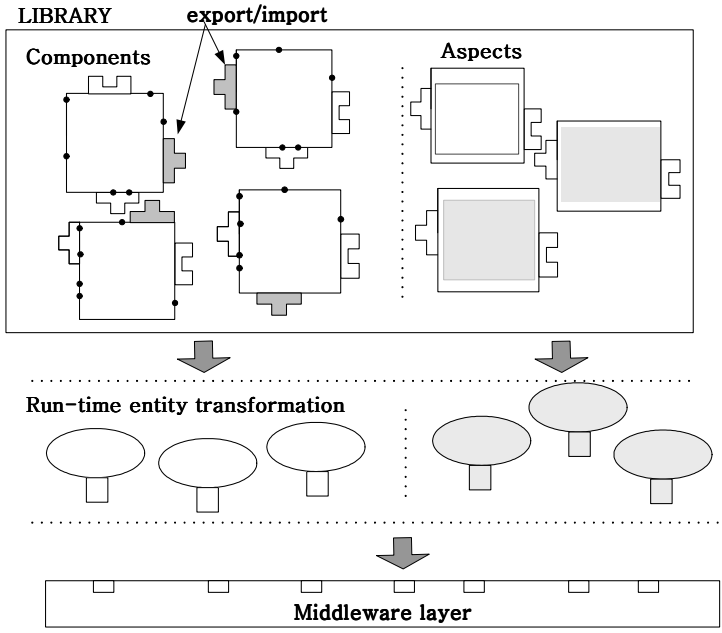


Figure 4.23: *Dynamic configuration of a system*

ration are done on a separate computer (e.g., development platform), and the obtained run-time entities are uploaded into the system running on the target computer platform. Having components and aspects mapped into run-time entities that communicate via the middleware layer enables easy reconfiguration of both components and aspects.

4.6.2 Middleware Layer

We introduce a middleware layer between a run-time environment and components, which handles communication among components and aspects using jump tables (inspired by [149]). The jump table contains the list of pointers to provided and required interfaces of components and aspects. When exchanging a component or an aspect, the jump table entries for the functional interface of a component or an aspect are re-pointed to the new version of the component/aspect. As mentioned, during reconfiguration the middleware layer temporarily stores component and aspect internal states, using the `export` and `import` operations.

The middleware layer provides a user interface for reconfiguration. This reconfiguration user interface consists of operations that enable adding, removing, or exchanging components (see figure 4.24). During the system lifetime, reconfiguration can be requested at any time using these operations.

```
1 exchange(c1,c2){
2   makeSystemReady();
3   remove(c1);
4   redirect(c1,c2);
5   add(c2);
6 }
```

(a) exchange

```
1 remove(c){
2   makeSystemReady();
3   if (c.hasState(){
4     state=c.export();
5     statePresent=true;
6   }
7 }
```

(b) remove

```
1 add(c){
2   makeSystemReady();
3   if (statePresent){
4     c.import(state);
5     statePresent=false;
6   }
7 }
```

(c) add

Figure 4.24: *Reconfiguration of a component*

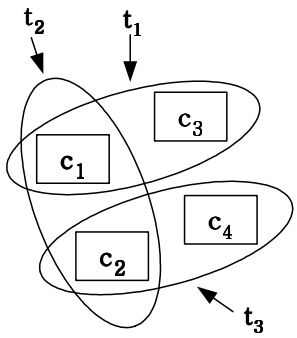


Figure 4.25: *An example of the relation between tasks and components.*

When reconfiguration is requested, the new version of the component is loaded into the memory. At this point, the new version is not added to the system, but we load it into the memory to carry out the reconfiguration as fast possible by minimizing the delay of removing the old version and adding the new version of the component. Furthermore, before allowing reconfiguration to take place, the middleware layer ensures that the system can undergo reconfiguration without interrupting the task execution (line 2 in figures 4.24(a-c)). Namely, tasks that rely on operations of components that are to be reconfigured, have to be completed before the reconfiguration is carried out, i.e., the task queue needs to be emptied of all tasks using the component. This is exemplified in figure 4.25, where there are four components denoted by $c_1, \ldots, c_4$, and three tasks denoted by $t_1, \ldots, t_3$. Task t_1 uses operations of component c_1 and c_3, while t_2 uses c_1 and c_2. Both t_1 and t_2 have to complete executing before c_1 can be exchanged. However, only t_1 needs to be completed before c_3 can be exchanged. Once the system can undergo reconfiguration, the actual exchange of components is initiated.

For example, the exchange of an old version of a component c, denoted c^1, with a new version, c^2, is carried out as follows. Note that we use notation c_i^j to refer to component i and its version j. First, a system is prepared for reconfiguration by emptying the task queue of all tasks using the component c^1. Hence, at the beginning of the actual reconfiguration there are no tasks calling the operations

of c^1, thus, its state will not change since it changes only through operation calls. Therefore, we can now exchange the old component c^1 with the new version c^2. The exchange is carried out by first removing the component from the system (line 3 in 4.24(a)). The removal is carried out by exporting the state of (an old version) of a component into the middleware layer(lines 3-5 in figure 4.24(a)). Then the jump table is re-pointed to the functional interfaces of the new version c^2 of the component (line 4 in figure 4.24(b)). Finally, the new component is added to the system (line 5 in figure 4.24(a)). The operation `add` restores states in the new version of the component using the `import` operation (line 4 in figure 4.24(c)). The role of the `statePresent` variable in `remove` and `add` operations is to ensure that the state of a component is restored only if it is needed. The described reconfiguration mechanism is hidden in the middleware layer and it enables fast and light-weight reconfiguration (we confirm this in experimental evaluations on the COMET database in chapter 9).

The reconfiguration of any number of components can be done by a user, which can either be the system user (human) or an application. The middleware layer is aware of the version of a component, via the version number of a component, in the current configuration. When a new version is compiled into the run-time directory (the component is loaded in the memory by the middleware layer), this can detected during application self-inspection and the component is exchanged with another version using the `exchange` operation of the middleware layer (see figure 4.24(a)). If a number of components needs to be exchanged this can be done by invoking `exchange` operation for each of the components. This also implies that components with dependencies can be exchanged safely, in a sequence of appropriate `exchange` calls, since the tasks using these components will be completed before the exchange takes place.

When wanting to exchange one of the algorithms that crosscut the overall system, the reconfiguration can be done by exchanging aspects. Exchange is done by first weaving the desired new aspect into affected components off-line, and then employing the reconfiguration of affected components as described above.[4] Note that it is not possible to exchange aspects if they are not encapsulated into interfaces and translated into corresponding real-time entities. Although the dynamic reconfiguration of aspects is done via component exchange, the benefits of aspect-orientation are still retained as changes to the code that crosscuts many components are still done in an automated, efficient, and modular way via aspect weaving.

[4] An aspect exchange can be, from the dynamic reconfiguration perspective, regarded as exchange of a number of components with new (woven) versions.

Chapter 5

Ensuring Real-Time Performance Guarantees

In this chapter we present methods that enable maintaining real-time performance guarantees in reconfigurable real-time systems. The methods include (i) analysis of WCETs needs of components woven with aspects, followed by (ii) maintenance of specified QoS levels under dynamic system reconfiguration, and (iii) formal verification of components reconfigured with aspects. The developed analysis techniques ensure that real-time performance can be guaranteed in hard real-time systems operating in closed environments, and in soft real-time systems operating in open environments. Hence, this chapter starts with an overview of techniques where we briefly elaborate the domain of applicability of each of the technique, e.g., open vs. closed environment. Then, the proposed techniques are discussed in detail.

5.1 Overview

We already established that maintaining real-time performance is essential for real-time systems. In section 2.3 we reviewed a number of techniques for ensuring and maintaining real-time performance of software operating in both open and closed environments. We use these techniques as a foundation for analysis of reconfigurable real-time systems.

To be able to deploy a statically configured hard real-time system into a closed run-time environment we need to ensure that it is possible to estimate WCETs of tasks and perform schedulability analysis. This is also important in the context of ensuring that the system can undergo reconfiguration, especially if there are, e.g., multiple candidate components or aspects for adding to the system. To that end, we have developed an approach for WCET analysis of real-time systems configu-

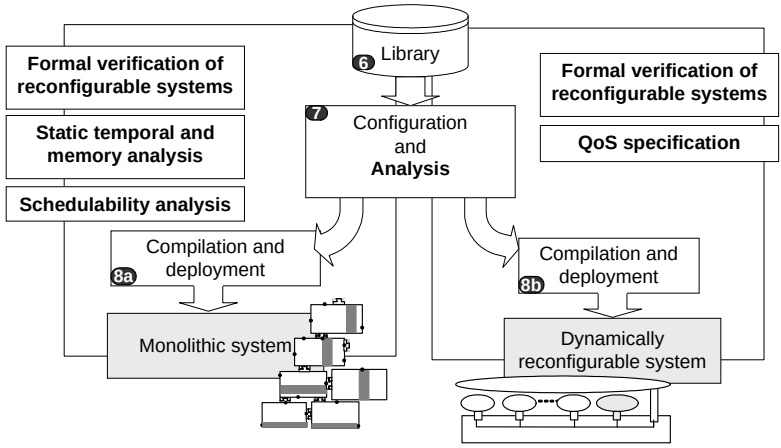


Figure 5.1: *An analysis part of the ACCORD development process*

red using aspects and components (discussed in section 5.2), which is founded on symbolic WCET analysis [27]. As we show in this chapter the obtained WCETs can be used for schedulability analysis of the system in the pre-run-time phase of system life cycle (see figure 5.1). For statically reconfigurable hard real-time systems it is important to perform schedulability analysis prior to system execution to ensure that deadline misses do not occur; thus, utilization of the system is also fixed beforehand.

A dynamically reconfigurable real-time system is typically deployed into an open and unpredictable environment. Moreover, when a system undergoes a dynamic reconfiguration, it is not always possible to predict the workload submitted to the system beforehand, e.g., due to the unknown number of components that are being exchanged. This can cause the system to be overloaded in the worst case. This means that it is difficult to adjust the utilization to a certain level beforehand (since admitted workload and utilization are related) and missing deadlines is inevitable. Rather than striving for achieving a certain utilization or meeting deadlines, the focus in performance maintenance of dynamically reconfigurable system should lie in providing mechanisms for ensuring QoS predictability, i.e., guaranteeing that the utilization does not exceed a certain threshold and no more than a certain number of tasks miss their deadlines during a period of time. To provide these performance guarantees we (in section 5.4) integrate feedback-based QoS management techniques into the method for dynamic real-time system reconfiguration. Thereby, we ensure that real-time performance in terms of a specified level of QoS is maintained during run-time even if the system undergoes reconfiguration.

When composing systems out of reusable artifacts (both in open and closed

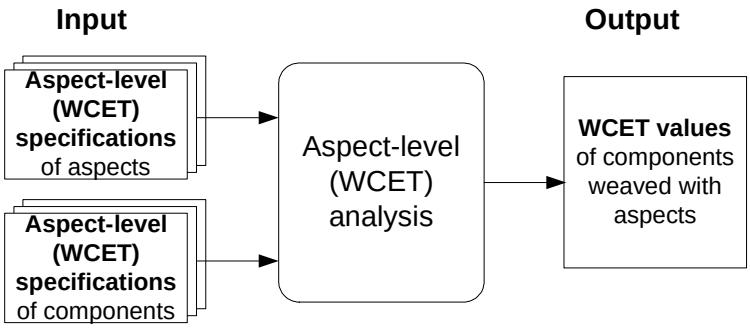


Figure 5.2: *An overview of the automated aspect-level WCET analysis process*

environment) it is desirable to be able to formally prove temporal and functional properties of components as well as the composed system. When doing formal analysis of component-based systems, one option is to first compose a system out of components and then do the verification of the overall system. However, this approach has drawbacks [100]. Namely, the possible number of configurations can be too large even if components are available in the library; which is a typical scenario in product line architectures. Hence, for verification to be tractable and usable in component-based development, it should apply to components with implications for the overall composed system [100, 101].

The verification challenge for reconfigurable real-time systems is great as the verification methodology needs to ensure that components are verified only once and the verification of configured systems is done on aspects. This is to overcome the possible state explosion that might happen in cases where verification is done on woven designs. To address this challenge, we provide a method for formal verification (in section 5.5) where we ensure that components are verified only once for a particular property, and where the property satisfaction of a configuration is checked only on aspects. We formally represent components and aspects as augmentations of timed automata, thereby building upon already established formalisms in the real-time domain.

5.2 Aspect-Level WCET Analysis

In this section we present an approach for determining the WCET needs of a real-time system composed using aspects[1] and components. We denote this aspect-level WCET analysis, and it is based on the concept of symbolic WCET analysis [27]. The main goal of aspect-level WCET analysis is determining WCETs of different real-time system configurations consisting of aspects and components

[1]Here aspects refer exclusively to application aspects, i.e., language-dependent aspects that invasively change the code of the components.

before any actual aspect weaving (system configuration) is performed. This enables the designer of a configurable real-time system to choose a system configuration fitting the WCET needs of the underlying real-time environment without paying the price of first performing aspect weaving for each individual candidate configuration.

Figure 5.2 presents an overview of the main constituents of analysis, namely:

- ❏ aspect-level WCET specifications of aspects and components,
- ❏ aspect-level WCET algorithm, which gives rules for the actual computation of the WCET of components woven with aspects, and
- ❏ resulting specification of WCETs of components woven with aspects.

The aspect-level WCET analysis gives WCET estimates of components woven with aspects to determine if the system configuration can be integrated into the target run-time environment. If very precise WCET estimates are needed, aspect-level WCET analysis can be followed by further analysis of the resulting woven code using a WCET tool that performs both low level and high level WCET analysis.

The following sections provide a detailed description of each of the elements involved in aspect-level WCET analysis.

5.2.1 Aspect-Level WCET Specification

Aspect-level specifications of components and aspects correspond to the run-time part of the RTCOM describing WCET needs. The reason we introduce the notion of aspect-level specifications is to emphasize that the approach to aspect-level WCET analysis could be generalized beyond ACCORD, if aspects and components are implemented in conformance with the guidelines presented in section 4.3. Based on the discussion in section 4.3.2 aspect-level WCET specification of an aspect and a component can be viewed as consisting of the internal WCET specification and the external WCET specification. *The internal WCET specification* is a fixed part of the aspect-level WCET specification and it is obtained by symbolic WCET analysis. It represents the WCET of the code that cannot be changed by aspect weaving. *The external WCET specification* is a variable part of the aspect-level WCET specification as it represents the WCET of the code that can be modified by aspect weaving, i.e., the temporal behavior can be changed by "external" influence.

Table 5.1 presents the relationship between components, aspects, and the aspect-level WCET specification. The temporal behavior of mechanisms, being fixed parts of a component, does not change by aspect weaving. Hence, the WCETs of mechanisms in a component are determined by the internal WCETs, specified as symbolic expressions. As operations can be modified by aspect weaving, their aspect-level WCET specifications consist both of fixed internal WCET

Components/Aspects		Aspect-level WCET	
		Internal WCET (symbolic expression)	External WCET (function of mechanisms)
Component	Mechanism	x	
	Operation	x	x
Aspect	Before advice	x	x
	After advice	x	x
	Around advice	x	x

Table 5.1: *Aspect-level WCET specifications of aspects and components*

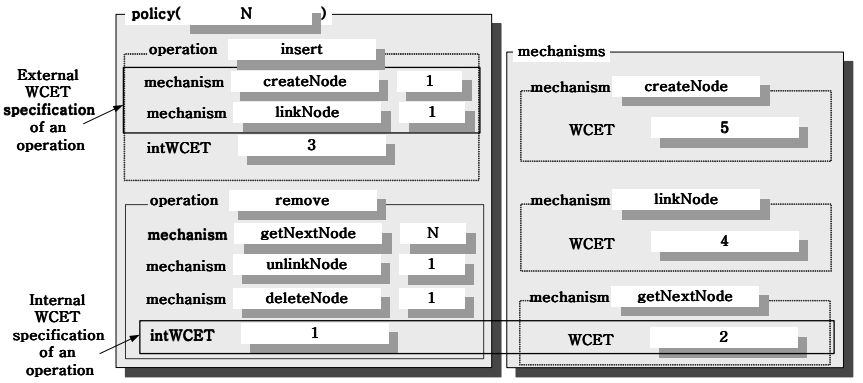


Figure 5.3: *The aspect-level WCET specification of the linked list component*

specifications and variable external WCET specifications (see table 5.1). The external WCET of an operation depends on the number of mechanisms called by this operation. This dependency exists since aspect weaving can change the operation implementation by changing the number of mechanisms used by the operation. Similarly, the WCET specification of an advice also consists of the fixed internal WCET specification and the variable external WCET specification.

Figure 5.3 represents the aspect-level WCET specification for the linked list component. The aspect-level specification for the aspect `listPriority` changing the code of the linked list component is shown in figure 5.4.

Aspect-level WCET specifications of aspects and components are input to the analysis process.

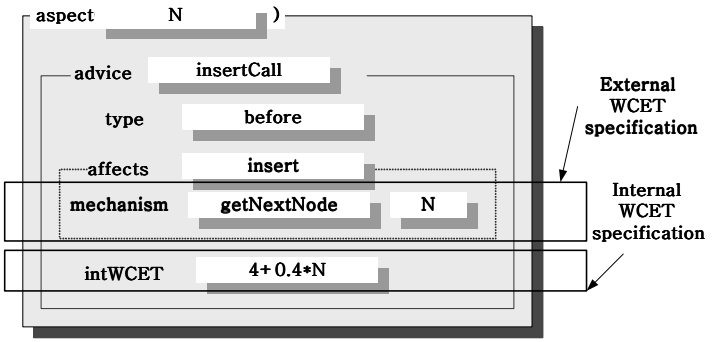


Figure 5.4: *The aspect-level WCET specification of the* `listPriority` *aspect*

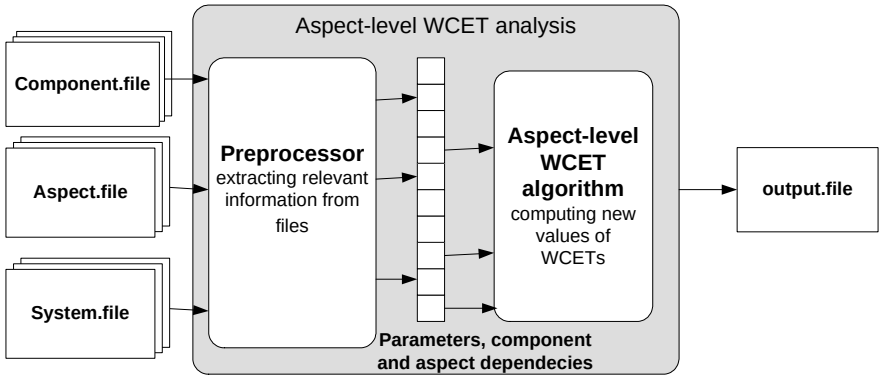


Figure 5.5: *Main constituents of aspect-level WCET analysis*

5.2.2 Aspect-Level WCET Analysis

The aspect-level WCET analysis consists of two main steps, preprocessing and applying the aspect-level WCET algorithm, as shown in figure 5.5.

The task of the preprocessor is to transform the information contained in aspect-level WCET specifications into a form useful for the WCET algorithm. Note that for successful analysis the knowledge about dependencies among components in the configuration is also required. This information can be obtained by using functional interfaces of components where provided and required operations are stated. The preprocessor analyzes aspect-level WCET specifications, and produces data structures containing the WCET values and interdependency information for all components and aspects needed for the algorithm.

The aspect-level WCET algorithm computes WCETs of components woven with aspects. Since internal WCETs in the aspect-level specifications are symbolic

expressions and are a function of certain parameters, the values of these need to be determined. Therefore, the fist step of the analysis is to obtain the values of parameters in the expressions and, based on them, calculate the internal WCET values. This is done in the preprocessing step before applying the algorithm.

The resulting, parameterized aspect-level specifications are used as input to aspect-level WCET algorithm to calculate the WCETs of all the operations within the real-time system configuration under development.

Algorithm 1 used for calculating the total WCET of components woven with aspects consists of three interdependent parts (top-down description):

- `WCETanalyzer()` is the main program of the WCET analyzer that computes the WCETs of every operation in the chosen system configuration;
- `operationWCET()` is called from `WCETanalyzer()` to compute the WCET of an operation in the component; and
- `codeBlockWCET()` is called from the `operationWCET()` to compute the WCET of an advice or an operation that is not weaved with aspects (note that advices and operations use mechanisms as basic blocks).

`operationWCET()` computes the WCET of an operation by taking into account that the operation might be modified by aspect weaving, and if it is, the following is applicable. For every advice within the aspect that modifies an operation we need to recalculate the WCET of the operation, depending on the advice type. The WCET of an around advice is calculated directly by `codeBlockWCET()`, where around advice now is a code block. The WCETs of before and after advices are calculated by taking into account not only the WCET of an advice as a code block, but also the WCET of the operation since the advice runs before or after the operation. If the operation is not modified by aspect weaving, then the above described actions are ignored and the value of the WCET of the operation is obtained simply by calling `codeBlockWCET()`. Finally, if the operation for which we are calculating the WCET is implemented using operations from other components, then in the WCET of the operation we need to include all the WCETs of every other operation called (these are calculated by the same principle). Thus, we need to have a recursive call to the `operationWCET()` itself.

The function `codeBlockWCET()` is used for calculating the WCET of a code block (`codeBlock`), which can be either an advice or an operation. `codeBlockWCET()` does so by first calculating the value of the internal WCET of a given code block based on a symbolic expression. Then, to obtain an aspect-level WCET of a `codeBlock`, the internal value of the WCET is augmented with the value of the external WCET. The external WCET is computed using the values of WCET for each mechanism called by the `codeBlock` such that the value of WCET of a mechanism (a symbolic expression) is multiplied with the number of times the `codeBlock` uses the mechanism.

Algorithm 1: Aspect-level WCET estimation

Input:

- aspect-level WCET specifications of aspects and components,
- values N_i of parameters, and
- required and provided interfaces of components in the configuration.

Output:

- output file, specifying WCETs of operations woven with aspects.

```
WCETanalyzer()
    For every operation_i do
        newWCET=operationWCET(operation_i)
    end for
```

```
operationWCET()
    operationWCET= 0
    If an advice is modifying the operation then
        For every advice_i in aspect_k modifying the operation do
            If around advice then
                operationWCET=operationWCET+ codeBlockWCET(advice_i)
            else before or after advice
                operationWCET=operationWCET+ codeBlockWCET(advice_i)
                + codeBlockWCET(operation)
            end if
        end for
    else operationWCET = codeBlockWCET(operation)
    end if
    If operation requires other operations then
        For every operation_k required by the operation
            operationWCET=operationWCET+operationWCET(operation_k)
        end for
    end if
return operationWCET
```

codeBlockWCET(*codeBlock*)

codeBlockWCET=intcodeBlockWCET

For every mechanism$_i$ used by the codeBlock **do**

codeBlockWCET=codeBlockWCET+WCET$_i * N_i$

end for

5.2.3 Example

Consider that we want to develop a real-time system using aspect and components that has to conform to specific WCET requirements. Hence, we need to apply aspect-level WCET analysis to the chosen configuration. To simplify the explanations, we return to our example of a configuration consisting of one component and one aspect, namely a linked list component and `listPriority` aspect. The aspect-level WCET specification of the linked list component is given in figure 5.3, while the specification of the `listPriority` aspect is depicted in figure 5.4. These are used as input to the analysis.

In the preprocessing part of the analysis a parameter `N` is detected and the designer is prompted for its value. Let us assume that we set the value of `N` to 5. Now we can apply the algorithm to compute the WCET values of the operations in the linked list component. Recall that the operation `insert` is modified by the advice `insertCall` of the `listPriority` aspect. To calculate the WCET of the modified operation, the WCET analyzer applies the `operationWCET()` part of the aspect-level WCET algorithm. Hence, a new value of the WCET of the operation `insert` woven with the before advice `insertCall` is calculated as follows:

$$\texttt{operationWCET} = \texttt{operationWCET} + \texttt{codeBlockWCET(insertCall)} + \texttt{codeBlockWCET(insert)}.$$

After applying the `codeBlockWCET()` part of the algorithm this results in:

$$\begin{aligned} \texttt{operationInsertWCET} &= 0 + \\ (1 + \texttt{createNodeWCET} * 1 + \texttt{linkNodeWCET} * 1) &+ \\ (4 + 0.4 * \texttt{N} + \texttt{getNextNode} * \texttt{N}) &= \\ 0 + (1 + 5 * 1 + 4 * 1) + (4 + 0.4 * 5 + 2 * 5) &= 26. \end{aligned}$$

5.2.4 Discussion

Ideally, the complete process of aspect-level WCET analysis should have a life cycle as presented in figure 5.6. The process starts with the implementation files of components and aspects, which are fed into a tool that performs the symbolic WCET analysis on the code, i.e., computes symbolic expressions for WCETs,

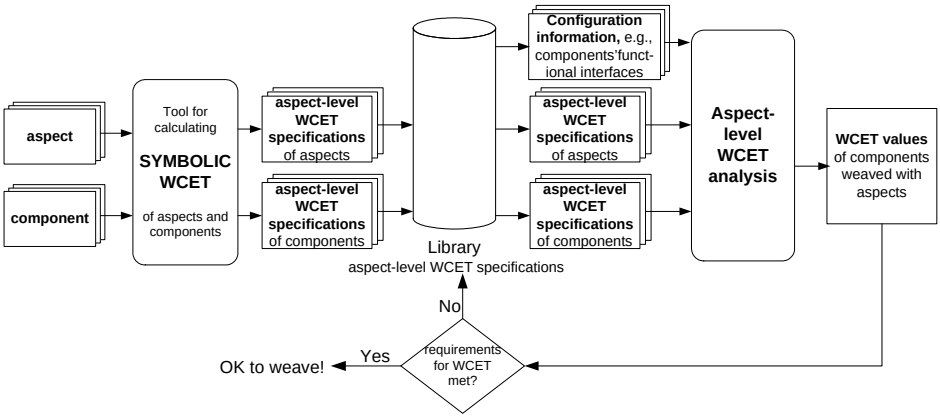


Figure 5.6: *An overview of the aspect-level WCET analysis life cycle*

and extracts these into aspect-level WCET specifications. These specifications are stored in a library and are used for the aspect-level WCET analysis, which computes the WCET of different system configurations so that the system designer can determine configuration eligibility for use in the underlying real-time environment (with respect to WCET constraints of the environment). If a given configuration does not fulfill the requirements with respect to the WCET, the designer can choose another configuration, i.e., another set of aspect-level WCET specifications, until the WCET requirements are met, and the actual weaving can be performed.

Figure 5.6 also illustrates limitations of current aspect-level WCET analysis. Namely, the tool that computes WCETs in the form of symbolic expressions and extracts these to aspect-level WCET specifications could be an adaptation of the pWCET tool for symbolic WCET analysis [28] to the aspect-level WCET analysis. This adaptation is currently not available. However, the aspect-level WCET analysis still provides benefits over traditional WCET analysis performed on weaved code as it enables calculations on WCET specifications, not on actual components and aspects. This way we reduce the overhead of performing the weaving and then WCET analysis for each potential configuration of aspects and components.

Note that the method for estimating WCET needs of components woven with aspects can also be applied for estimating static memory needs of components. In such a case memory needs are calculated using the aspect-level WCET algorithm and the aspect-level specification of memory needs, which are analogous to the specification of WCETs.

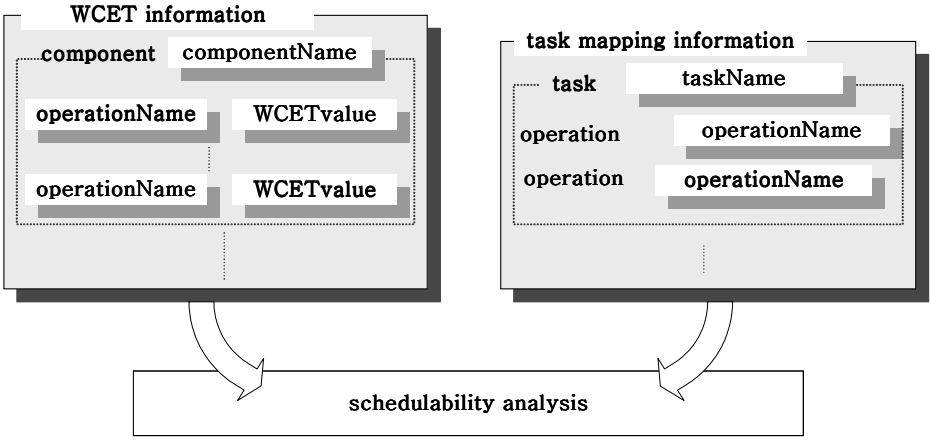


Figure 5.7: *Temporal schedulability analysis within ACCORD*

5.3 Components and their Schedulability

Schedulability analysis within ACCORD can be performed if the following is available (see figure 5.7):

- WCETs of operations in the system configuration, and
- task mapping information describing which operation of a component is executed by which task.

By combining the knowledge of WCETs of operations in the component with the task mapping information, we can obtain WCETs of each task in the system. This enables analysis of a given task set with respect to finding a feasible schedule on a specific platform, e.g., by using some off-the-shelf tool or a method for schedulability analysis such as RapidRMA [141] and TimeWiz [49].

The information about the WCET of each operation can be obtained by applying aspect-level WCET analysis. The task mapping information specifies on which task's thread of execution an operation of a component runs. Recall (from section 3.1) that the way operations of a component are mapped to a task depends on the target run-time environment and its available resources. For examples of how mapping is done in a specific run-time environment, see [150, 118]. When mapping components to tasks one should consider the characteristics of that relationship. The following list of characteristics is an adaptation of task structuring guidelines from DARTS [66] to component-based systems.

- Event dependency, which includes the following:
  - Aperiodic I/O device dependency. Operations of components dependent on the device input and output often need to be executed at the

speed of the I/O device with which they interact. In particular, if the device is aperiodic then a set of operations interacting with such a device could be structured into a separate I/O-dependent task.

- Periodic events. If operations provided by one or more components need to be executed at regular intervals of time, then these operations could be structured into a periodically activated task.

❑ Task cohesion, which includes the following:

- Sequential cohesion. Some operations need to be performed sequentially with respect to operations of other components. If the first operation (of the first component) in the sequence is triggered by an aperiodic or periodic event, these sequential components can be combined into one task.
- Temporal cohesion. Operations of components activated on the same event may be grouped into a task so that they are executed each time the task receives a stimuli.

❑ Task priority, which includes the following:

- Time criticality. Time-critical operations of the component need to run at a high priority and, thus, a component providing these operations could be structured as a separate high-priority task.
- Computational intensity. Non-time-critical but computationally intensive operations of components may run as a low priority task consuming spare CPU cycles.

Note that listed characteristics represent an initial effort in providing general characterization of the component-task relationship, and issues not discussed here, such as inter-process synchronization, are subject to further research.

5.4 Feedback-Based QoS Management

Recall requirements for maintaining performance under dynamic reconfiguration of soft real-time systems operating in open environments identified in section 3.1.3, stating that the system administrator must be able to specify desired system QoS and transient state system QoS in terms of worst-case QoS and how fast the QoS should converge toward the desired QoS. To provide QoS guarantees under system reconfiguration we employ feedback control [62].

The desired nominal system QoS is expressed in terms of a reference QoS, as shown in figure 5.8. The reference gives the level of QoS that the system must provide when it is in the steady state, i.e., when no reconfiguration is currently taking place and any effects of previous reconfiguration have passed. When a

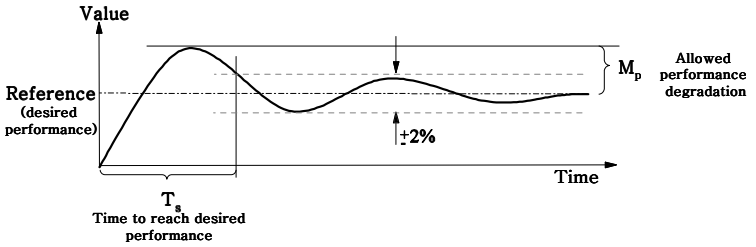


Figure 5.8: *Fluctuations of performance under reconfiguration*

reconfiguration is taking place the system alternates to a transient state, which is characterized by fluctuations in QoS depicted in figure 5.8. The desired behavior of the system under reconfiguration is expressed in terms of the maximum overshoot and the settling time [73, 62]. The maximum overshoot M_p is the worst-case system QoS in the transient system state and it is given in percentage. By defining M_p we bound the maximum allowed performance degradation of the system under reconfiguration. The settling time T_s is the time for the transient overshoot to decay and reach the steady state QoS. Hence, the settling time is a measure of system adaptability, i.e., how fast the system converges toward the desired QoS in the face of reconfiguration.

Typically, one is interested in controlling the performance of real-time systems using the deadline miss ratio metric [108], which gives the ratio of tasks that have missed their deadlines. Therefore, we employ a feedback-based QoS management method, referred to as FC-M [108], in the dynamic reconfiguration of real-time software. FC-M enables controlling deadline miss ratio by modifying the admitted load.

We say that a task is terminated when it has completed or missed its deadline. Let $missedTasks(k)$ be the number of tasks that have missed their deadline and $terminatedTasks(k)$ be the number of terminated admitted tasks in the time interval $[(k-1)T, kT]$. The deadline miss ratio is defined as follows:

$$m(k) = \frac{missedTasks(k)}{terminatedTasks(k)}$$

and denotes the ratio of tasks that have missed their deadlines.

The feedback loop has the static structure shown in figure 4.21. In the dynamic case, this means that QoS management related components and aspects are implemented according to the extended RTCOM model and deployed into the middleware layer. The structure of the dynamically reconfigurable system that guarantees QoS is depicted in figure 5.9. QoS guarantees are now satisfied by having the QAC that implements changes in the manipulated variable and the FCC, which implements the control loop by measuring the controlled variable and

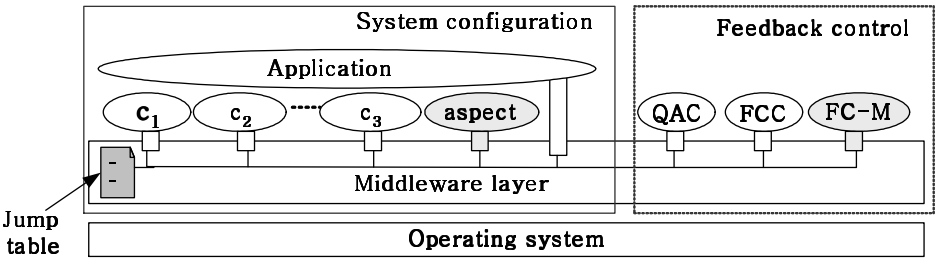


Figure 5.9: *Dynamic system reconfiguration with support of the feedback control*

computing the manipulated variable. The FC-M QoS algorithm is implemented as an aspect, crosscutting QAC and FCC.

The performance error, $e_m(k) = m_r(k) - m(k)$, is computed to quantize the difference between the desired deadline miss ratio, given by the reference $m_r(k)$, and the measured deadline miss ratio $m(k)$ (see figure 2.11). The change to load $\delta_l(k)$ is derived using a P-controller [62]. The load target $l(k)$ is the integration of $\delta_l(k)$, i.e., $l(k+1) = l(k) + \delta_l(k)$. The QAC is used to carry out the change in load as follows. A task is admitted into the system if the sum of the load of the task that is waiting to be admitted and the load of the already admitted tasks is less than the load target $l(k)$.

Consider the following example where the deadline miss ratio reference is set to 0.1 and the load threshold at the 10^{th} sampling instant is set to 0.9, i.e., $m_r = 0.1$ and $l(10) = 0.9$. A component exchange during the previous sampling interval has resulted in an increase in the execution time of the tasks, and consequently the deadline miss ratio has increased to $m(10) = 0.2$. Clearly, the component exchange has degraded the performance of the system. Therefore, we need to reduce the deadline miss ratio to the reference value $m_r = 0.1$. This is done by taking the measured value of the deadline miss ratio from the system and computing the performance error $e_m(10) = m_r(10) - m(10) = (0.1 - 0.2) = -0.1$ and then computing the change in load, i.e., $\delta_l(10) = -0.1K_P$; following the steps in the feedback structure from figure 2.11. The load threshold during the next sampling interval is changed to $l(11) = l(10) + \delta_l(10) = 0.9 - 0.1K_P$. The admitted load is reduced as a result of a decrease in the load threshold and, consequently, the deadline miss ratio for the 11^{th} sampling instant is reduced.

As we demonstrate experimentally on the COMET database in section 9.3 applying feedback-based QoS for guaranteeing the performance before and after reconfiguration is beneficial. As we show, the reconfigurability can indeed be achieved for real-time systems, even though the execution time of the tasks vary when adding, removing, or changing components.

However, even though QoS of the system reaches the reference in the steady state, overshoots and long settling times, i.e., violation of the QoS specification, could still occur for reconfiguration instances that are heavily affected by the

execution time and the arrival pattern of the tasks. Therefore, on-line QoS management of dynamically reconfigurable systems could further be strengthened by providing additional analysis of the system behavior off-line. Namely, one way to ensure that the QoS specification is not violated is to determine the overshoot and settling time before the actual reconfiguration is made, i.e., to predict the behavior of the system. That is, for a given system under reconfiguration and a given QoS specification one could determine if the reconfigured system can meet its QoS specification beforehand, and then carry out the actual reconfiguration only if it is possible to meet the given specification. By combining off-line analysis with the on-line mechanisms for QoS guarantees, one could ensure that the reconfiguration will not violate the QoS specification and that the performance of the system will be on the desired level after reconfiguration.

5.5 Formal Analysis of Aspects and Components

In this section we describe a formal method for verifying temporal properties of reconfigurable real-time components. This method enables: (i) proving temporal properties of individual components and aspects, and (ii) proving that reconfiguration of components via aspect weaving preserves expected temporal behavior in a reconfigured component. We primarily focus on the verification of one component reconfigured with one aspect because this is both the foundation and prerequisite for the successful verification of the overall composed real-time component-based system; we can then infer properties, functional and temporal, of the composed system based on the proven properties of individual reconfigurable components.

5.5.1 Modeling Reconfigurable Components

Our goal is to verify that properties proven for a component are preserved under component reconfiguration. More precisely, given a component $\mathcal{C}$, an aspect $\mathcal{AS}$, a weaving operation Υ, and a property ϕ, we want to prove that if ϕ is satisfied by component $\mathcal{C}$ (denoted $\mathcal{C} \models \phi$), then the reconfigured component $\mathcal{C}' = \mathcal{C} \Upsilon \mathcal{AS}$, obtained by weaving $\mathcal{AS}$ into $\mathcal{C}$ also preserves the property ($\mathcal{C}' \models \phi$). In order to be able to do this we need to formally model components, aspects, and reconfiguration, which is the focus of this section.

In section 4.3 we explained that aspects, or more precisely their constituents advices, can be woven into the code of a component such that they are executed before, after, or around (instead of) pre-defined reconfiguration locations. The information about the reconfiguration locations is stored in the composition interface of RTCOM. Thus, we augment a timed automaton, which represents a component, with the composition interface that fixes places, i.e., reconfiguration locations, in the component used for attaching advices of aspects.

Definition 4 (Component) *A reconfigurable component $\mathcal{C}$ is a tuple $\langle \mathcal{A}_c, I \rangle$, where $\mathcal{A}_c$ is a timed automaton $\langle L_c, l_{0c}, E_c, C_c, r_c, g_c, Inv_c \rangle$, and $I = \langle rl_1, \ldots, rl_k \rangle$, $rl_i \in L_c$, is a composition interface.*

Thus, each location rl_i from an interface I corresponds to a reconfiguration location in RTCOM. Note that the reconfiguration locations of a component are pre-defined and explicitly declared for each component and, thus, the number of these is known and fixed. Although in the structural description of components (see section 4.3) we only needed knowledge of the reconfiguration locations, for formal verification purposes, we also require this information about its predecessors and successors (definition 5). This information can easily be extracted from the timed automaton $\mathcal{A}_c$ of a component $\mathcal{C}$.

Definition 5 (Successor and predecessor) *Given a reconfigurable component $\mathcal{C} = \langle \mathcal{A}_c, I \rangle$, for each $rl_i \in L_c$ in the interface I of $\mathcal{C}$, we define two sets, a predecessor set of rl_i, pred, and a successor set of rl_i, succ, as*

- ❏ $pred(rl_i) = \{l \mid \langle l, rl_i \rangle \in E_c\}$
- ❏ $succ(rl_i) = \{l \mid \langle rl_i, l \rangle \in E_c\}$

To be able to place an advice of type before we use $pred(rl_i)$, since the advice is going to be placed between the $pred(rl_i)$ and rl_i location. An advice of type after is going to be placed between rl_i and $succ(rl_i)$. In the case of an around advice, the advice is going to be placed between locations $pred(rl_i)$ and $succ(rl_i)$ and will be executed instead of the code in the reconfiguration location.

Figure 5.10(a) illustrates an automaton that represents the behavior of a component. The example shows a simplified model of the component in charge of updating data times, e.g., a transaction manager in a database. For this component, the $\mathcal{A}_c$ part is shown as the timed automaton in figure 5.10(a), and the I part as the interface $I = \langle update \rangle$ associated with one location. The component is responsible for starting a transaction (*start*), performing operations defined within a transaction on data in the database (*update*) and ending the transaction (*end*). The interface of the component consists of one reconfiguration location *update*. The reconfiguration location is characterized also with its predecessor $pred(update) = \{start\}$ and successor $succ(update) = \{end\}$.

Next we define an aspect as a collection of timed automata.

Definition 6 (Aspect) *An aspect $\mathcal{AS}$ is a tuple $\langle \mathcal{AD}, PC, F \rangle$, where $AD = \langle \mathcal{A}d_1, \ldots, \mathcal{A}d_n \rangle$ is a collection of advices, $PC = \{pc_1, \ldots, pc_m\}$ is a set of pointcuts, and F is a function assigning a subset of pointcuts to each of the advices, $F(\mathcal{A}d_i) \subseteq PC$. Each $pc_i \in PC$ is some reconfiguration location rl from an interface I of some component $\mathcal{C} \in \mathfrak{C}$, where $\mathfrak{C}$ is a set of components. Each advice $\mathcal{A}d = \langle \mathcal{A}_a, in, out, t \rangle$ is a timed automaton $\mathcal{A}_a$ with two designated locations in and out and an associated type $t \in \{before, after, around\}$, such that the graph defined by $\langle L_a, E_a \rangle$ induced by $\mathcal{A}_a$ is a connected graph.*

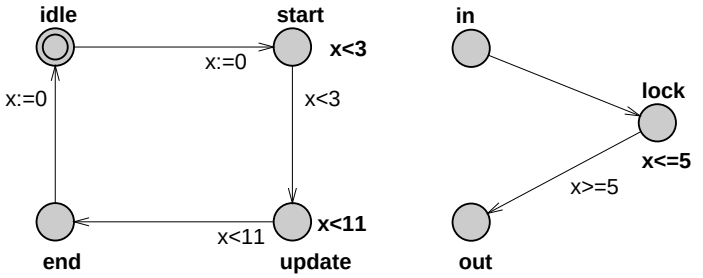


Figure 5.10: *Examples of timed automata specifying (a) the transaction manager component and (b) the locking advice*

In definition 6, following the traditional structure of aspects presented in section 4.3, an aspect consists of a number of advices and pointcuts. Pointcuts denote places where the advice should be woven into the component. Therefore, each pointcut corresponds to one or more reconfiguration locations from interfaces of components. Note that an advice of an aspect can be woven into multiple components, and that two and more advices in an aspect can share the same pointcuts. The *in* and *out* locations introduced in definition 6 are used as placeholders for the places at which the advice will be woven at reconfiguration time.

Following the previous example, assume that a given component should be extended such that it is ensured that data cannot be updated in the system unless the appropriate lock on a data item is obtained prior to updates. This can be done by defining an aspect $\mathcal{AS}_{lock}= \langle \mathcal{A}d, update\rangle$ with the advice $\mathcal{A}d = \langle \mathcal{A}_a, in, out, before\rangle$, where the automaton $\mathcal{A}_a$ describes the locking mechanism to be woven before the pointcut *update* in the component (see figure 5.10(b)).

Without loss of generality from now on we refer to aspects defined as $\mathcal{AS} = \langle \mathcal{A}d, pc\rangle$ consisting of only one advice $\mathcal{A}d$ and only one pointcut pc, and exclude function F. This is to simplify presentation and reduce the complexity of notation that otherwise would unnecessarily complicate the definition of a component reconfiguration.

Reconfiguration of a component $\mathcal{C} = \langle \mathcal{A}_c, I\rangle$ with an aspect $\langle \mathcal{A}d, pc\rangle$ is defined by the weaving operation Υ as follows.

Definition 7 (Component reconfiguration) *Given a component $\mathcal{C}=\langle \mathcal{A}_c,\ I\rangle$ and an aspect $\mathcal{AS}=\langle \mathcal{A}d,pc\rangle$, where $pc = rl_i \in L_c\backslash\{l_{0c}\}$ is a reconfiguration location from the interface I and $\mathcal{A}d = \langle \mathcal{A}_a, in, out, t\rangle$, the reconfigured component $\mathcal{C}'=\mathcal{C}\Upsilon\mathcal{AS}$ is defined as a tuple $\langle \mathcal{A}'_c,\ I\rangle$ where $\mathcal{A}'_c$ is defined as follows:*

- ❑ *the locations L'_c*

$$L'_c = \begin{cases} L_c \cup L_a\backslash\{in, out\} & \text{if } t \in \{before,\ after\} \\ L_c \cup L_a\backslash\{in, out, pc\} & \text{if } t{=}around \end{cases}$$

- $l'_{0c} = l_{0c}$
- $E'_c = \{\langle l, l' \rangle | \langle l, l' \rangle \in E_c \cup E_a,\ l, l' \notin \{in, out, pc\}\} \cup E_s$ *where* E_s *denotes the substituted transition set defined as follows:*

$$E_s = \begin{cases} \{\langle l, l' \rangle | \langle in, l' \rangle \in E_a, l \in pred(pc)\} \cup & \\ \{\langle l, l' \rangle | \langle l, out \rangle \in E_a, l' = pc\} & \textit{if } t\text{=}\textit{before} \\ \{\langle l, l' \rangle | \langle in, l' \rangle \in E_a, l = pc\} \cup E_{s_{succ}} & \textit{if } t\text{=}\textit{after} \\ \{\langle l, l' \rangle | \langle in, l' \rangle \in E_a, l \in pred(pc)\} \cup E_{s_{succ}} & \textit{if } t\text{=}\textit{around} \end{cases}$$

where $E_{s_{succ}}$ *is the substituted transition subset that depends on the content of* $succ(pc)$ *set as*

$$E_{s_{succ}} = \begin{cases} \{\langle l, l' \rangle | \langle l', out \rangle \in E_a, l = pc\} & \textit{if } succ(pc) = \emptyset \\ \{\langle l, l' \rangle | \langle l, out \rangle \in E_a, l' \in succ(pc)\} & \textit{otherwise} \end{cases}$$

- $C'_c = C_c \cup C_a$
- *the clocks to be reset* $r'_c(e)$

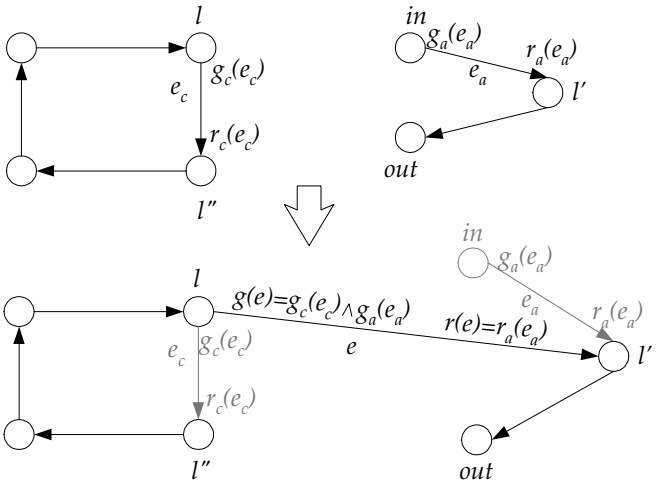
$$r'_c(e) = \begin{cases} r_c(e) & \textit{if } e \in (E'_c \backslash E_s) \cap E_c \\ r_a(e) & \textit{if } e \in (E'_c \backslash E_s) \cap E_a \\ r_a(e) & \textit{if } e = \langle l, l' \rangle \in E_s,\ e_a = \langle in, l' \rangle \in E_a, \textit{ and} \\ & \quad e_c = \langle l'', l' \rangle \in E_c \\ r_a(e_a) \cup r_c(e_c) & \textit{if } e = \langle l, l' \rangle \in E_s,\ e_a = \langle l, out \rangle \in E_a, \textit{ and} \\ & \quad e_c = \langle l'', l' \rangle \in E_c \end{cases}$$

- *the guards* $g'_c(e)$

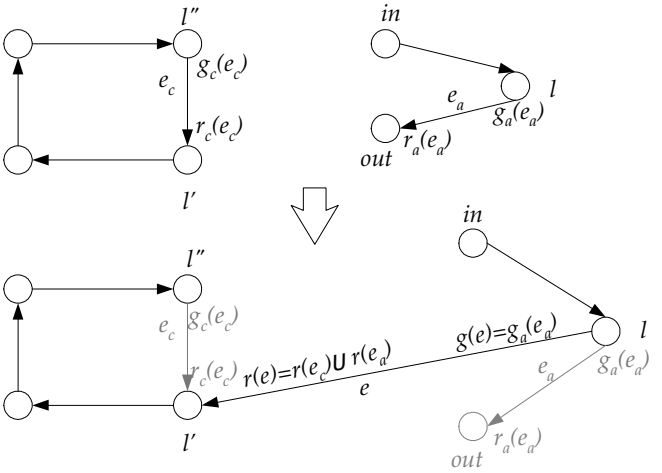
$$g'_c(e) = \begin{cases} g_c(e) & \textit{if } e \in (E'_c \backslash E_s) \cap E_c \\ g_a(e) & \textit{if } e \in (E'_c \backslash E_s) \cap E_a \\ g_c(e_c) \wedge g_a(e_a) & \textit{if } e = \langle l, l' \rangle \in E_s,\ e_c = \langle l, l'' \rangle \in E_c, \textit{ and} \\ & \quad e_a = \langle in, l' \rangle \in E_a \\ g_a(e_a) & \textit{if } e = \langle l, l' \rangle \in E_s,\ e_a = \langle l, out \rangle \in E_a \\ & \quad e_c = \langle l'', l' \rangle \in E_c \end{cases}$$

- $Inv'_c = \{Inv_c(l) | l \in L_c\} \cup \{Inv_a(l) | l \in L_a \backslash \{in, out\}\}$

Definition 7 of component reconfiguration is easily extendable to aspects consisting of many advices woven into multiple components. The definition describes the way edges are formed in a reconfigured component. Also, definition 7 ensures that aspect weaving into a component, when replacing an edge between



(a) Transfer of guards and clock constraints for an edge starting in a component and ending in an aspect.



(b) Transfer of guards and clock constraints for an edge starting in an aspect and ending in a component.

Figure 5.11: *An example of preservation of clock constraints*

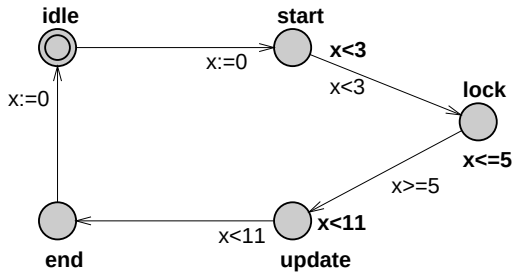


Figure 5.12: *An example of a reconfigured component*

the reconfiguration locations, preserves clock constraints that existed on the removed edge. That is, invariants, guards, and clock constraints of the replaced edges are transferred to the newly created edges. Figure 5.11 illustrates the way clock constraints are preserved on the removed edges. Grey colored edges and clock constraints depict elements substituted during component reconfiguration. Figure 5.11(a) shows the way guards and reset clocks are transferred to a newly created edge in the case when the newly created edge connects a component location to an aspect location in a reconfigured component. Similarly, figure 5.11(a) shows the way guards and reset clocks are transferred to a newly created edge in the case when the newly created edge connects an aspect location to a component location in a reconfigured component. To enable meaningful reachability analysis on the advices, preservation of clock constraints that existed on the transitions to/from the reconfiguration location is essential.

The reconfiguration of the component from figure 5.10(a) with the $\mathcal{AS}_{lock}$ aspect is illustrated in figure 5.12. Weaving the aspect $\mathcal{AS}_{lock}$ in the component at reconfiguration location $rl = pc = update$ results in the reconfigured component $\mathcal{C}' = \mathcal{C} \curlyvee \mathcal{AS}_{lock}$. The composition resulted in replacing the *in* location in the advice with $start = pred(rl)$, and the *out* location with the location $rl = update$. Now instead of taking a transition from location *start* to *update* directly, location *lock* is first visited to obtain the locks. The reconfigured component retains all clock constraints of the original components (which existed on the removed edge).

5.5.2 Formal Analysis of Reconfiguration

As mentioned, the goal of the formal analysis is to verify a reconfigurable component by verifying the aspect that is to be woven into the component. In our approach, when verifying a component, appropriate information about composition interfaces is extracted. The extracted information is used for verification of an aspect. Moreover, the same information can be reused for many aspects that might reconfigure the same component, e.g., in different reuse contexts.

To verify the timing and functional behavior of reconfigurable components via reachability analysis we employ the following three steps (S1)-(S3):

(S1) proving TCTL properties of components, which is performed using existing model checking techniques;

(S2) deriving constraints on composition interfaces of components to preserve a particular property upon reconfiguration; and

(S3) preserving properties of components upon reconfiguration, which is done by analyzing aspects, using the interface derived under (S2).

The above steps (S1)-(S3) constitute an extension to deal with timing properties in the verification approach for the non-real-time systems by Li et al. [100, 101]. While the untimed approach uses labeling on states with propositions, we use clock zones to represent timing information.

We provide a verification method that is of practical value as our method can be applied in existing model-checking tools for real-time systems. In UPPAAL, which is our candidate tool, verification of a real-time system is done on the subset of TCTL properties found essential for real-time system verification: EFq, AFq, and $AG(p \Rightarrow AFq)$, where q is a boolean expression over locations and clock constraints and can be directly checked on a state. Here, we primarily focus on reachability properties in the form $\phi{=}EFq$, where $q{=}l_q{\wedge}Z_q$ (l_q is a location and Z_q a zone represented by a clock constraint), and derive main results for these types of properties. This restriction to q may later be relaxed; in the relaxed form q can consist of an arbitrary number of boolean expressions over locations and clock constraints. An example of a standard EFq property is the property $\phi{=}EF(end{\wedge}x < 11)$ of the component from figure 5.10(a), ensuring that a transaction executed by this component will complete within 11 time units. Here, $l_q{=}end$ and $Z_q{=}\{x < 11\}$. The invariant properties AFq are dual to reachability properties EFq, and the results obtained for reachability properties are transparently applicable to the invariant properties. The property in the form $AG(p \Rightarrow AFq)$ can be derived from reachability properties as well.

The following section formally describes the verification steps.

5.5.3 Verifying a Reconfigured Component

Given component $\mathcal{C}{=}\langle \mathcal{A}_c, I\rangle$, TCTL reachability property ϕ in the form $EF(l{\wedge}Z)$, and aspect $\mathcal{AS}{=}\langle \mathcal{A}d, pc\rangle$, we illustrate steps (S1)-(S3) in detail.

Step 1 (S1)

Given property ϕ, and component $\mathcal{C}{=}\langle \mathcal{A}_c, I\rangle$, use the standard reachability algorithm [25] for model checking of timed automata $\mathcal{A}_c$. If the algorithm reports success then $\mathcal{C}$ satisfies property ϕ, denoted $\mathcal{C} \models \phi$.

The semantics of a component $\mathcal{C}=\langle \mathcal{A}_c,\ I\rangle$ is defined by semantics of a timed automaton $\mathcal{A}_c$, which is transition system $(S_c, s_{0c}, \mapsto)$ given by definition 2. Recall that the standard reachability algorithm works by traversing a zone graph $(S_c, s_{0c}, \rightsquigarrow)$ of $\mathcal{A}_c$ until the desired state is reached or there is nothing else to traverse. Hence, the following holds for a component $\mathcal{C}$.

Proposition 1 *Let $\mathcal{C}=\langle \mathcal{A}_c,\ I\rangle$ be a component, the semantics of which is defined by the transition system $(S_c, s_{0c}, \mapsto)$. Let $Z(\mathcal{C})=(S_c, s_{0c}, \rightsquigarrow)$ be a zone graph of $\mathcal{C}$, and $\phi = EFq$ be a reachability property where $q = l_q \wedge Z_q$. Let $\rightsquigarrow^*$ denote a sequence of action/delay transitions in the zone graph. If the component $\mathcal{C}$ satisfies property ϕ, then there exists at least one path σ in the zone graph $Z(\mathcal{C})$ such that $\sigma=\langle l_{0c}, Z_{0c}\rangle \rightsquigarrow^* \langle l_q, Z\rangle$, $Z \subseteq Z_q$.*

Proof Follows from the definition of reachability in the zone graph of a timed automaton, and the property of the inclusion operation on zones [25]. ∎

The path induced by a proof of reachability of ϕ in $\mathcal{C}$ is referred to as *reachability path of ϕ in $\mathcal{C}$*. To save notation, we use Z_q to denote zone Z associated with location l_q on the reachability path of ϕ in $\mathcal{C}$ where there is no risk for ambiguity. It follows from the proof of reachability that a component $\mathcal{C}$ can have several paths reaching the state $\langle l_q, Z_q\rangle$. We denote a set of rechablity paths of ϕ in $\mathcal{C}$ as Σ. Further, $l \in \sigma$ is used as a shorthand for stating that location l appears in some $\langle l, Z\rangle$ on the path σ; conversely, $l \notin \sigma$ denotes that location l does not appear on the path σ.

Proposition 2 *Let $\mathcal{AS}=\langle \mathcal{A}d,\ pc\rangle$ be an aspect with an advice $\mathcal{A}d=\langle \mathcal{A}_a,\ in,\ out,\ t\rangle$. Let $\mathcal{C}=\langle \mathcal{A}_c,\ I\rangle$ be a component, and $\mathcal{C}'=\mathcal{C}\curlyvee\mathcal{AS}$ be a reconfigured component at some reconfiguration location $rl = pc$. Let $\phi = EFq$ be a reachability property such that $\mathcal{C} \models \phi$, and Σ be the set of reachability paths of ϕ in $\mathcal{C}$. The following holds for $\mathcal{C}'$:*

(1) If for every reachability path $\sigma \in \Sigma$, $rl \notin \sigma$ then $\mathcal{C}' \models \phi$.

(2) If $rl = l_q$ and t=around, then $\mathcal{C}' \not\models \phi$.

Proof for both cases is as follows.

(1) If $rl \notin \sigma$ for all $\sigma \in \Sigma$ then property ϕ is always preserved under reconfiguration since none of the paths $\sigma \in \Sigma$ are altered by the reconfiguration.

(2) When $rl = l_q$ and t=*around* then by construction of a reconfigurable component (definition 7) it follows that the location l_q will not appear in $\mathcal{C}'$. Thus, $\langle l_q, Z_q\rangle$ cannot be reached in $Z(\mathcal{C})$, and $\mathcal{C}' \not\models \phi$. ∎

From proposition 2, it follows that: (1) the property is always preserved if the reconfiguration location does not exist in any of the reachability paths of the

property; and (2) the property is always violated when the advice of type *around* is reconfiguring the component at the reconfiguration location that exists in the property expression. Case (1) should be clearly identified in the verification of a component to avoid unnecessary aspect verification. Given that the advices of type *around* are rarely used, case (2) appears rarely. However, we include it here to provide adequate support for detecting these immediate property violations in the property preservation algorithm (presented later in this section) and, thus, enable designers to immediately see in which cases around advices are inappropriate for component reconfiguration. We deal with these cases by introducing a boolean variable, discussed further in (S2), that helps in determining if one of the two cases has occurred.

Step 2 (S2)

Step (S2) involves deriving constraints on the composition interfaces building up on the results of the verification from the previous step. These constraints include temporal information needed in the subsequent step (S3) of the verification. Since we need to store this information with the component, we augment component $\mathcal{C}$ with an appropriate interface, denoted verification interface, formally defined as follows.

Definition 8 (Verification interface) *Let $\mathcal{C}=\langle \mathcal{A}_c, I\rangle$ be a component with the zone graph $Z(\mathcal{C})=(S_c, s_{0c}, \rightsquigarrow)$ and an interface $I=\langle rl_1, \ldots, rl_k\rangle$. Let $\phi = EFq$ be a reachability property. Let $\mathcal{C} \models \phi$, inducing a set of reachability paths $\Sigma = \{\sigma_1, \ldots, \sigma_m\}$ of ϕ in $\mathcal{C}$, with $\sigma_j = \langle l_{0c}, Z_{0c}\rangle \rightsquigarrow^* \langle l_q, Z_q\rangle$. The verification interface I^ϕ of component $\mathcal{C}$ for the property ϕ is defined as a k-tuple $\langle rl_1^\phi, \ldots, rl_k^\phi\rangle$ where*

$$rl_i^\phi = \begin{cases} \{\langle IsPreserved : true\rangle\} & \text{if for all } \sigma_j \in \Sigma,\ rl_i \notin \sigma_j \\ \{rl_{ij}^\phi = \langle \langle IsPreserved : \bot\rangle, & \\ \quad \langle l, Z_{pred}\rangle, \langle rl_i, Z_{rl}\rangle \langle l', Z_{succ}\rangle \rangle | & \\ \quad rl_i \in \sigma_j, l \in pred(rl_i), l' \in succ(rl_i)\} & \text{otherwise} \end{cases}$$

All the states $\langle l, Z_{pred}\rangle$, $\langle rl_i, Z_{rl}\rangle$, $\langle l', Z_{succ}\rangle$, appearing in elements rl_{ij}^ϕ of the set rl_i^ϕ, appear in the reachability path σ_j of ϕ. The variable *IsPreserved* flags that case (1) identified in proposition 2 is trivially true, i.e., identifies those verification interfaces of components that are not affected by the reachability property ϕ no matter which aspect reconfigures them. When $\langle IsPreserved : \bot\rangle$ is stored in the verification interface for reconfiguration location rl_i, the reconfiguration of the component at this location potentially affects ϕ. $\bot$ denotes unknown as it is customary in 3-valued formalisms.

In algorithm 2 we describe step (S2), i.e., the extraction of a verification interface of a component, in algorithmic steps.

Algorithm 2: Verification interface extraction

Input:

- component $\mathcal{C}$ with composition interface $I{=}\langle rl_1, \ldots, rl_k\rangle$, and
- reachability property $\phi{=}\langle l_q, Z_q\rangle$.

Output:

- verification interface $I^\phi{=}\langle rl_1^\phi, \ldots, rl_k^\phi\rangle$ of component $\mathcal{C}$.

Construct the zone graph $Z(\mathcal{C})$ of $\mathcal{C}$
Compute the set $\Sigma = \{\sigma_1, \ldots, \sigma_m\}$ of reachability paths of ϕ in $\mathcal{C}$
$NotOnPath := true$
$I^\phi := \langle\rangle$
For $i = k$ downto 1 **do**
  $rl_i^\phi := \emptyset$
  For $j = 1$ to m **do**
    $rl_{ij}^\phi := \langle\rangle$
    If rl_i appears on path σ_j **then**
      rl_{ij}^ϕ:=append($\langle l', Z_{succ}\rangle$, rl_{ij}^ϕ) for location $l' \in succ(rl_i)$ on σ_j
      rl_{ij}^ϕ:=append($\langle rl_i, Z_{rl}\rangle$, rl_{ij}^ϕ) for state $\langle rl_i, Z_{rl}\rangle$ on σ_j
      rl_{ij}^ϕ:=append($\langle l, Z_{pred}\rangle$, rl_{ij}^ϕ) for location $l \in pred(rl_i)$ on σ_j
      rl_{ij}^ϕ:=append($\langle IsPreserved : \bot\rangle$, rl_{ij}^ϕ)
      $NotOnPath := false$
      rl_i^ϕ:=$rl_i^\phi \cup \{rl_{ij}^\phi\}$
    end if
  end for
  If $NotOnPath = true$ **then** rl_i^ϕ:=$rl_i^\phi \cup \{\langle IsPreserved : true\rangle\}$
  I^ϕ:=append(rl_i^ϕ, I^ϕ)
end for
Return verification interface I^ϕ

Step 3 (S3)

Before giving the algorithm that checks whether a property ϕ is preserved under reconfiguration of a component $\mathcal{C}$ with an aspect $\mathcal{AS}$, step (S3), we define the semantics and the satisfiability relation for the aspect reconfiguring the component. The semantics of an aspect $\mathcal{AS}{=}\langle \mathcal{A}d, pc\rangle$ with an advice $\mathcal{A}d{=}\langle \mathcal{A}_a, in, out, t\rangle$ is defined by the semantics of the timed automaton $\mathcal{A}_a$, which is a transition system $(S_a, s_{0a}, \mapsto)$ given by definition 2. The zone graph associated with $\mathcal{A}_a$ we denote by $Z(\mathcal{AS})$ and refer to it as the aspect zone graph.

Definition 9 (Enriched aspect zone graph) *Let $\mathcal{AS}{=}\langle \mathcal{A}d, pc\rangle$ be an aspect with an advice $\mathcal{A}d{=}\langle \mathcal{A}_a, in, out, t\rangle$, and the zone graph $Z(\mathcal{AS})$. Let $\mathcal{C}$ be a component under reconfiguration by aspect $\mathcal{AS}$ at some reconfiguration location $rl_i{=}pc$ from I. Let ϕ be a property under verification, with $\mathcal{C} \models \phi$. Let $I^{\phi}{=}\langle rl_1^{\phi}, \ldots, rl_k^{\phi}\rangle$ be the verification interface of the component for this property with $rl_{ij}^{\phi} \in rl_i^{\phi}$, $rl_{ij}^{\phi} = \langle\langle IsPreserved : \bot\rangle, \langle l, Z_{pred}\rangle, \langle rl_i, Z_{rl}\rangle, \langle l', Z_{succ}\rangle\rangle$. The enriched zone graph $Z(\mathcal{AS})'{=}(S'_a, s'_{0a}, \rightsquigarrow)$ for rl_{ij}^{ϕ} is computed as follows:*

- ❑ *the initial state*

$$s'_{0a} = \langle l_{0a}, Z'_{0a}\rangle = \begin{cases} \langle in, Z_{pred} \wedge Z_{in}\rangle & \text{if } t{\in}\{before,\ around\} \\ \langle in, Z_{rl} \wedge Z_{in}\rangle & \text{if } t{=}after \end{cases}$$

- ❑ $s'_{0a} \rightsquigarrow s'_{1a}$
  - $\langle l_{0a}, Z'_{0a}\rangle \rightsquigarrow \langle l_{0a}, Z''_{0a}\rangle$, $Z''_{0a} = Z'^{\uparrow}_{0a} \wedge Inv(l_{0a})$; *and*
  - $\langle l_{0a}, Z'_{0a}\rangle \rightsquigarrow \langle l_{1a}, Z'_{1a}\rangle$, $Z'_{1a} = r_e(Z'_{0a} \wedge g_a(e) \wedge g_c(e')) \wedge Inv(l_{1a})$ *for all* $e = \langle l_{0a}, l_{1a}\rangle \in E_a$ *and all* $e' \in E_c$ *such that*

$$e' = \begin{cases} \langle l, rl_i\rangle & \text{if } t{\in}\{before,\ around\} \\ \langle rl_i, l'\rangle & \text{if } t{=}after \end{cases}$$

- ❑ *for all* $s_{ia} \rightsquigarrow s_{(i+1)a}$ *in* $Z(\mathcal{AS})$, $i \geq 1$, *compute* $s'_{ia} \rightsquigarrow s'_{(i+1)a}$ *as follows:*
  - $\langle l_{ia}, Z'_{ia}\rangle \rightsquigarrow \langle l_{ia}, Z''_{ia}\rangle$, $Z''_{ia} = Z'^{\uparrow}_{ia} \wedge Inv(l_{ia})$; *and*
  - $\langle l_{ia}, Z'_{ia}\rangle \rightsquigarrow \langle l_{(i+1)a}, Z'_{(i+1)a}\rangle$, $Z'_{(i+1)a} = r_e(Z_{ia} \wedge g_a(e)) \wedge Inv(l_{(i+1)a})$ *for all* $e = \langle l_{ia}, l_{(i+1)a}\rangle \in E_a$, *where the reset operation is defined as*

$$r_e(Z'_{ia}) = \begin{cases} \{[r_a(e) \mapsto 0]v | v \in Z'_{ia}\} & \text{if } l_{(i+1)a} \neq out \\ \{[r_a(e) \mapsto 0]v | v \in Z'_{ia}\} \cup & \\ \{[r_c(e') \mapsto 0]v | v \in Z_{ia'}\} & \text{if } l_{(i+1)a} = out \end{cases}$$

for all $e' \in E_c$ *such that*

$$e' = \begin{cases} \langle l, rl_i\rangle & \text{if } t{\in}\{before,\ around\} \\ \langle rl_i, l'\rangle & \text{if } t{=}after \end{cases}$$

Before we formulate an algorithm that analyzes enriched zone graphs and instantiates the $IsPreserved$ variable for deducing satisfiability of a property ϕ, we introduce the notion of satisfiability for an aspect and prove the theorem that shows under which conditions the weaving operation Υ preserves a property by analyzing the enriched zone graph of an aspect.

Definition 10 (Preserving ϕ under reconfiguration of $\mathcal{C}$) *Let $\mathcal{C}$ be a component under reconfiguration by the aspect $\mathcal{AS}=\langle \mathcal{A}d,\ pc\rangle$ at some reconfiguration location rl_i=pc from I. Let ϕ be a property under verification such that $\mathcal{C} \models \phi$, and I^{ϕ} verification interface of the component for this property with $rl_{ij}^{\phi} \in rl_i^{\phi}$, $rl_{ij}^{\phi}=\langle\langle IsPreserved : \bot\rangle, \langle l, Z_{pred}\rangle, \langle rl_i, Z_{rl}\rangle, \langle l', Z_{succ}\rangle\rangle$. Let $Z(\mathcal{AS})'$ be an enriched zone graph of $\mathcal{AS}$ for rl_{ij}^{ϕ}. We say that aspect $\mathcal{AS}$* preserves property ϕ under reconfiguration of $\mathcal{C}$, *denoted $\mathcal{AS} \models_{\Upsilon\mathcal{C}} \phi$, if there exists a path $\delta=\langle in, Z'_{oa}\rangle \rightsquigarrow^* \langle out, Z_{out}\rangle$, in the enriched zone graph $Z(\mathcal{AS})'$ such that the following holds*

- $Z_{out} \subseteq Z_{rl}$, *for t=before,*
- $Z_{out} \subseteq Z_{succ}$, *for t=after,*
- $Z_{out} \subseteq Z_{succ}$ *for t=around and* $rl \neq l_q$.

The following lemma is used in the proof of theorem 4.

Lemma 3 *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two timed automata, with $L_1 \subseteq L_2$, $l_{01} = l_{02}$, $E_1 \subseteq E_2$, $C_1 \subseteq C_2$, for all $e \in E_1$, $r_1(e) \subseteq r_2(e)$, $g_1(e) = g_2(e)$, and $Inv_1(l) = Inv_2(l)$ for all $l \in L_1$. Let σ_1 and σ_2 be paths in zone graphs $Z(\mathcal{A}_1)$ and $Z(\mathcal{A}_2)$, respectively, such that $\sigma_1=\langle l_1, Z_1\rangle \rightsquigarrow^* \langle l_n, Z_n\rangle$ and $\sigma_2=\langle l_1, Z'_1\rangle \rightsquigarrow^* \langle l_n, Z'_n\rangle$, where $l_i \in L_1$. If $Z_1 \subseteq Z'_1$ then $Z_n \subseteq Z'_n$.*

Proof by induction on transitions in the zone graph. ∎

Theorem 4 *Let $\mathcal{C}$ be a component, $\mathcal{AS}$ an aspect, Υ the weaving operation, and ϕ a reachability property. Let $\mathcal{C} \models \phi$ and $\mathcal{C}' = \mathcal{C} \Upsilon \mathcal{AS}$. If $\mathcal{AS} \models_{\Upsilon\mathcal{C}} \phi$ then $\mathcal{C}' \models \phi$.*

Proof We present a full proof for one advice type *t=before*. The proofs for other types of advices are analogous.

Since $\mathcal{C} \models \phi$, there exists a set of reachability paths Σ in zone graph $Z(\mathcal{C})$ of $\mathcal{C}$, where each reachability path $\sigma_j \in \Sigma$ of ϕ is in the form: $\sigma_j=\langle l_{0c}, Z_{0c}\rangle \rightsquigarrow^* \langle l_q, Z_q\rangle$.

Based on definition of $\mathcal{C}' = \mathcal{C} \Upsilon \mathcal{AS}$, we know that the reconfiguration of $\mathcal{C}$ with $\mathcal{AS}$ is done at some reconfiguration location $rl_i = pc$ from the composition interface I of $\mathcal{C}$.

If rl_i does not appear in any of the paths $\sigma_j \in \Sigma$ then $\mathcal{C}'$ is not affected in relation to satisfiability of ϕ (proposition 2), and the theorem is trivially true.

If rl_i appears in some $\sigma_j \in \Sigma$, then $\sigma_j=\langle l_{0c}, Z_{0c}\rangle \rightsquigarrow^* \langle l_{pred}, Z_{pred}\rangle \rightsquigarrow \langle rl, Z_{rl}\rangle \rightsquigarrow^* \langle l_q, Z_q\rangle$.

Since $\mathcal{C}' = \mathcal{C} \curlyvee \mathcal{AS}$, by the definition of component reconfiguration and the definition of zone graphs, it follows that there is a reachability path σ_{rl} for location rl_i in zone graph $Z(\mathcal{C}')$ of $\mathcal{C}'$, σ_{rl}=$\langle l_{0c}, Z'_{0c}\rangle \rightsquigarrow^* \langle l_{pred}, Z'_{pred}\rangle \rightsquigarrow^* \langle l_a, Z_a\rangle \rightsquigarrow \langle rl, Z'_{rl}\rangle$, where $l_a \in L_a$ such that $\langle l_a, out\rangle \in E_a$.

From $\mathcal{AS} \models_{\gamma\mathcal{C}} \phi$ we know that there exists an element rl_i^{ϕ} in the verification interface I^{ϕ} of $\mathcal{C}$, with some rl_{ij}^{ϕ}=$\langle\langle IsPreserved : \bot\rangle , \langle l, Z_{pred}\rangle, \langle rl_i, Z_{rl}\rangle, \langle l', Z_{succ}\rangle\rangle$. We further know that in the enriched zone graph of $\mathcal{AS}$ based on rl_{ij}^{ϕ} there exists a path δ=$\langle in, Z'_{0a}\rangle \rightsquigarrow^* \langle l_a, Z_a\rangle \rightsquigarrow \langle out, Z_{out}\rangle$ in which $Z_{out} \subseteq Z_{rl}$.

From the definition of component reconfiguration it follows that the zones associated with the state $\langle rl, Z'_{rl}\rangle$ from path σ_{rl} and the state $\langle out, Z_{out}\rangle$ from path δ are the same, i.e., Z'_{rl}=Z_{out}. Since $Z'_{rl} = Z_{out}$ and $Z_{out} \subseteq Z_{rl}$, then $Z'_{rl} \subseteq Z_{rl}$.

Based on definition 7 of component reconfiguration, for all locations $l, l' \in E'_c \cap E_c$, guards, invariants and reset clocks remain unchanged under reconfiguration and $\langle l, l'\rangle \in E_c \Rightarrow \langle l, l'\rangle \in E'_c$. This implies that, since there was a path reaching location l_q from location rl_i, $\langle rl_i, Z_{rl}\rangle \rightsquigarrow^* \langle l_q, Z_q\rangle$ in $Z(\mathcal{C})$, there will be a path between these two locations in $Z(\mathcal{C}')$, $\langle rl, Z'_{rl}\rangle \rightsquigarrow^* \langle l_q, Z'_q\rangle$. Given that location rl_i is reachable in $Z(\mathcal{C}')$ from location l_{0c} by path $\langle l_{0c}, Z'_{0c}\rangle \rightsquigarrow^* \langle rl, Z'_{rl}\rangle$, and l_q is reachable in $Z(\mathcal{C}')$ from rl_i by path $\langle rl, Z'_{rl}\rangle \rightsquigarrow^* \langle l_q, Z'_q\rangle$, then location l_q is reachable from l_{0c} in $Z(\mathcal{C}')$ by a path σ'_j= $\langle l_{0c}, Z'_{0c}\rangle \rightsquigarrow^* \langle rl, Z'_{rl}\rangle \rightsquigarrow^* \langle l_q, Z'_q\rangle$. Given that $Z'_{rl} \subseteq Z_{rl}$, by lemma 3, we have that $Z'_q \subseteq Z_q$. Hence, σ'_j is a reachability path of ϕ in $Z(\mathcal{C}')$. Thus, $\mathcal{C}' \models \phi$. ∎

Algorithm 3 checks for satisfiability of $\mathcal{AS} \models_{\gamma\mathcal{C}} \phi$, thus, giving algorithmic steps for performing step (S3) of the verification procedure.

We argue for the soundness of the algorithm as follows. If the algorithm returns $\langle true, rl_{ij}^{\phi'}\rangle$, then the aspect $\mathcal{AS}$ preserves property ϕ when reconfiguring $\mathcal{C}$. Here, $rl_{ij}^{\phi'}$ contains the path on which the property is preserved. In the first two steps of the algorithm, it trivially returns a correct value based on proposition 2. Namely, the algorithm returns $\langle true, \langle\rangle\rangle$ if the property is always preserved (case (1) of proposition 2). In this case, $\langle\rangle$ is returned together with *true* to indicate that the property is trivially satisfied and no further checks on an aspect need to be performed. The algorithm returns $\langle false, \emptyset\rangle$ if the property is trivially violated (case (2) of proposition 2). Otherwise, $\langle true, rl_{ij}^{\phi'}\rangle$ is returned if there exists a zone Z_{out} with properties named in definition 10. Thus, the aspect complies with definition 10 of preservation of ϕ. If the algorithm is not previously terminated, then the final step terminates it by returning $\langle false, rl_i^{\phi'}\rangle$, where $rl_i^{\phi'}$ gives all paths on which the property is not satisfied.

Algorithm 3: Property preservation

Input:

- aspect $\mathcal{AS} = \langle \mathcal{A}d, pc \rangle$ and an advice $\mathcal{A}d = \langle \mathcal{A}_a, in, out, t \rangle$,
- reachability property $\phi = \langle l_q, Z_q \rangle$, and
- component $\mathcal{C}$ with composition interface $I = \langle rl_1, \ldots, rl_k \rangle$ and verification interface $I^\phi = \langle rl_1^\phi, \ldots, rl_k^\phi \rangle$, such that $rl_i = pc$.

Output:

- $\langle true, rl_{ij}^{\phi'} \rangle$ if $\mathcal{AS} \models_{\gamma\mathcal{C}} \phi$, $\langle false, rl_i^{\phi'} \rangle$ otherwise.

If $rl_i^\phi = \langle IsPreserved : true \rangle$ **then return** $\langle true, \langle\rangle \rangle$

If $rl_i = l_q$ and $t =$*around* **then return** $\langle false, \emptyset \rangle$

$RemainingPaths := rl_i^\phi = \{rl_{i1}^\phi, \ldots, rl_{im}^\phi\}$

Repeat:

Choose an element $rl_{ij}^\phi = \langle \langle IsPreserved : \bot \rangle, \delta \rangle$ from $RemainingPaths$

$RemainingPaths := RemainingPaths \backslash \{rl_{ij}^\phi\}$

Construct $Z(\mathcal{AS})'$ for rl_{ij}^ϕ from definition 9

If ($Z_{out} \subseteq Z_{rl}$ and t=before) **or** ($Z_{out} \subseteq Z_{succ}$ and $t \in$ {after, around}) **then**

$rl_{ij}^{\phi'} := \langle \langle IsPreserved : true \rangle, \delta \rangle$

return $\langle true, rl_{ij}^{\phi'} \rangle$

end if

$rl_{ij}^{\phi'} := \langle \langle IsPreserved : false \rangle, \delta \rangle$

$rl_i^{\phi'} := rl_i^{\phi'} \cup \{rl_{ij}^{\phi'}\}$

Until $RemainingPaths = \emptyset$

Return $\langle false, rl_i^{\phi'} \rangle$

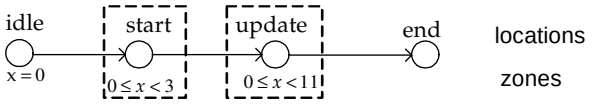


Figure 5.13: *One possible execution trace of for the transaction manager component and the property* $EF(end \wedge x < 11)$

5.5.4 An Example

We illustrate steps (S1)-(S3) of the verification procedure on the running example of the component $\mathcal{C} = \langle \mathcal{A}_c, I \rangle$ and the locking aspect $\mathcal{AS}_{lock}$=$\langle \mathcal{A}d, update \rangle$, where $\mathcal{A}d$=$\langle \mathcal{A}_a, in, out, \text{before} \rangle$. We want to check if the task executing this component will execute within 11 time units in the reconfigured component $\mathcal{C}' = \mathcal{C} \curlyvee \mathcal{AS}_{lock}$. That is, we need to check if the property ϕ=$EF(end \wedge x < 11)$ is preserved under reconfiguration, i.e., location *end* is reached within 11 time units.

(S1) Existing real-time model-checkers, e.g., UPPAAL, can be used in this step for model-checking timed automaton $\mathcal{A}_c$. This results in $\mathcal{C} \models \phi$. The tool generates the trace showing how the property is fulfilled. One such trace for the property ϕ=$EF(end \wedge x < 11)$ of the component TMC is illustrated in figure 5.13. This trace corresponds to the path σ=$\langle l_{0c}, Z_{0c} \rangle \rightsquigarrow^* \langle l_q, Z_q \rangle$, l_q=*end* and Z_q=$\{x < 11\}$, identified by proposition 1.

(S2) In this step the temporal information for the location *update* and it successor and predecessor locations *start* and *end* are extracted into the verification interface I^ϕ of $\mathcal{C}$ using algorithm 1. Following the steps of the algorithm we obtain the verification interface I^ϕ=rl^ϕ as follows. First, the state of the successor location *end*, $\langle end, Z_{succ} \rangle$, $Z_{succ} = Z_{end} = \{0 \leq x\}$ is appended to rl^ϕ. Next, the state corresponding to the reconfiguration location *update*, $\langle update, Z_{rl} \rangle$, where $Z_{rl} = Z_{update} = \{0 \leq x < 11\}$, is appended to rl^ϕ. Further, the state corresponding to the predecessor location *start*, $\langle start, Z_{pred} \rangle$, where $Z_{pred} = Z_{start} = \{0 \leq x < 3\}$ is appended to rl^ϕ. Finally, $< IsPreserved : \bot >$ is appended to rl^ϕ since *rl* appears on the path σ. Hence, algorithm 2 returns the verification interface:
I^ϕ=rl^ϕ=$\langle \langle IsPreserved : \bot \rangle, \langle start, \{0 \leq x < 3\} \rangle, \langle update, \{0 \leq x < 11\} \rangle, \langle end, \{0 \leq x\} \rangle \rangle$.

(S3) This step is done as prescribed by the property preservation algorithm 3. Following the steps in the algorithm 3, the checks for trivial satisfaction of $\mathcal{AS}_{lock} \models_{\curlyvee \mathcal{C}} \phi$ are passed without stopping the algorithm as the advice is neither of type around nor is the flag $IsPreserved$ set to true. In the next step, the enriched aspect zone graph is computed based on definition 9.

Then the zone Z_{out} corresponding to location *out* is checked for satisfiability. Since $Z_{out} = \{5 \leq x < 11\}$ then $Z_{out} \subseteq Z_{rl}$ and the algorithm returns $\langle true, rl^{\phi'} \rangle$. In this example, $rl^{\phi'} = rl^{\phi}$. Hence, $\mathcal{AS}_{lock} \models_{\Upsilon \mathcal{C}} \phi$.

5.5.5 Discussion

The presented verification method is beneficial as it ensures that components are verified only once for a particular property. The verification of reconfigured components is done on the aspects, in order not to waste the effort invested in verification of the components and to overcome the possible state explosion that might happen in cases where verification is done on woven designs.

So far we have primarily focused on the application of the method for the verification of an aspect with a single advice that is woven into a single component. This result is a foundation for further extension of the method to enable verification of reconfiguration of several components with aspects containing several advices running in parallel; this type of composition corresponds to a hybrid of sequential and parallel composition.

Chapter 6

Tool Support and Evaluation

In this chapter we present the tool set that provides real-time system developers support for configuration and analysis of a system built using components and aspects. Furthermore, we evaluate ACCORD against the previously identified requirements placed on development of reusable and reconfigurable real-time systems.

6.1 Development Environment

The ACCORD development environment is a tool set developed to provide the system designer tool support for assembling and analyzing a real-time system for a particular application. We assume that, for a particular family of real-time systems, components and aspects are already developed and placed in the library. The ACCORD development environment consists of (see figure 6.1):

- ACCORD library of pre-developed software artifacts,
- ACCORD modeling environment (ACCORD-ME), and
- configuration compiler.

In this section we describe each of the constituents of the ACCORD development environment and then discuss limitations and benefits of the environment.

6.1.1 ACCORD Library

The ACCORD library contains two types of artifacts, namely design-level artifacts and implementation artifacts (see figure 6.1). Design-level artifacts are:

- configuration paradigms for modeling of different system configurations,

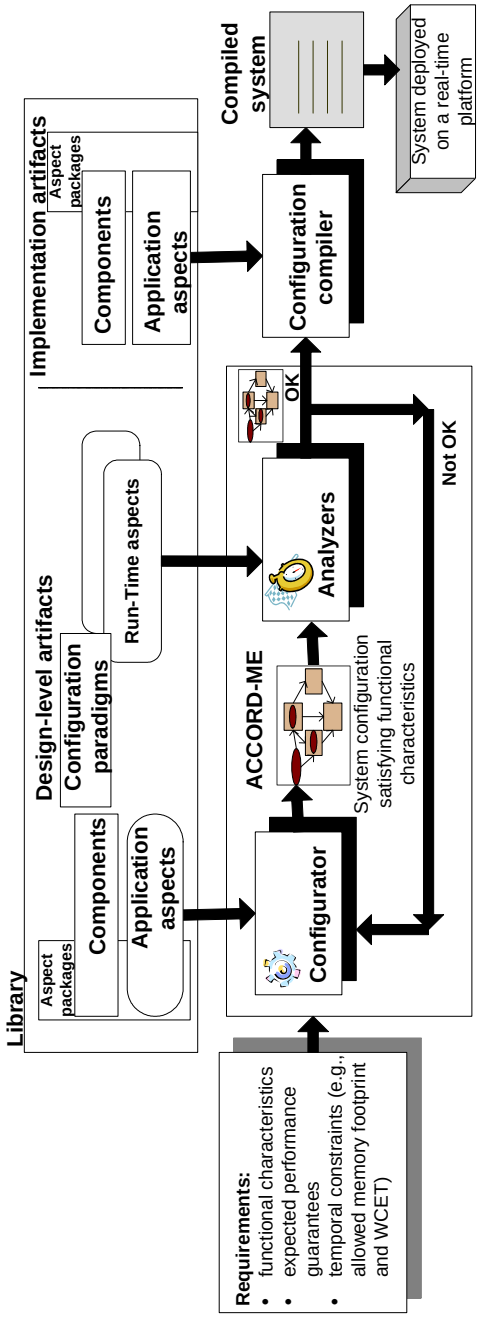


Figure 6.1: *ACCORD Development Environment*

- ❏ run-time aspects of components and application aspects (as prescribed by RTCOM), and
- ❏ formal representations of components and aspects.

Design-level artifacts are used when modeling, configuring, and analyzing the system (in ACCORD-ME). Implementation artifacts represent source code of components and aspects, and they are used when producing the final product, i.e., a compiled code of the system for deployment in a specific run-time environment.

6.1.2 ACCORD-ME

The ACCORD-ME part of the development environment is implemented using the generic modeling environment (GME), a tool kit for creating domain-specific modeling environments [65]. The creation of a GME-based tool is accomplished by defining meta-models that specify a modeling paradigm (modeling language) of the application domain. Modeling paradigms (given as UML diagrams) define the relationship between components and aspects in the library and their possible relationship to other components of the system, as well as possibilities of combining components and aspects into different configurations. The GME environment also enables specifying different tool plug-ins, which can be developed independently of the environment and then integrated with the GME for different modeling and/or analysis purposes.

The input to ACCORD-ME are requirements that are placed on the system. This includes functional and non-functional requirements a system should fulfill when used in a specific run-time environment, e.g., performance guarantees, and temporal and memory constraints. This tool uses the design-level artifacts for configuration and analysis of the system. The output of the tool is a configuration file containing information about the system configuration that fulfills the specified functional and non-functional requirements.

ACCORD-ME is developed with a number of sub-tools that are plugged into it: configurator, memory and WCET (M&W) analyzer, and formal verifier. Figure 6.2 shows a snapshot of the GME-based ACCORD-ME, with the tool plug-ins and the editing area.

The configurator helps the designer to assemble the system configuration by suggesting a subset of suitable aspects and components. The decision on what components and aspects are suitable is based on the pre-defined modeling paradigms and the requirements placed on the system. Modeling paradigms define the relationship between components and aspects in the library, and possibilities of combining those in different configurations. Modeling paradigms in GME-based tools are expressed by UML diagrams. This tool provides three levels of support to system configuration based on the developer's expertise and preferences.

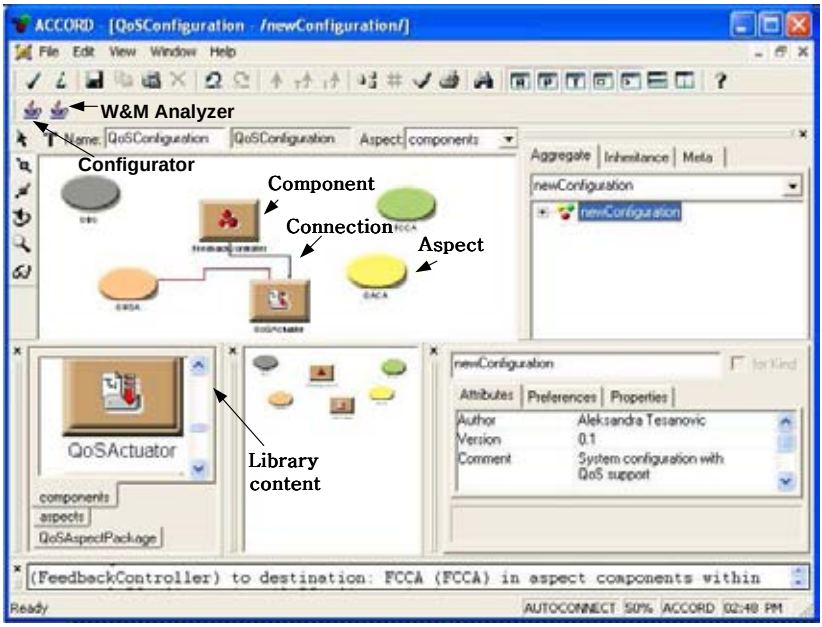


Figure 6.2: *The editing window in ACCORD-ME*

- *Expert option* is used by developers familiar with the library of components and aspects, and it enables the developers to create a number of custom made configurations of the system. This is useful in cases when a comparison of the performance of different configurations is of interest, e.g., WCET and memory footprint.

- *Configuration-based option* gives a list of possible configurations of the system, and it is intended for developers who can directly express the requirements of the system in terms of a desired system configuration.

- *Requirement-based option* provides the system developer with a list of requirements from which the developer can choose a relevant subset for a particular application. Thus, developers do not need to have knowledge of what components and aspects exist in the library.

The output of the tool is a configuration as an XML file of an assembled system that is functionally correct.

The M&W analyzer is a tool plug-in that analyzes a configuration with respect to WCET and memory requirements using the aspect-level WCET analysis presented earlier in section 5.2. This tool takes as input:

- an XML configuration file produced by the configurator; and

- run-time aspects, also expressed in XML, that contain the run-time information the tool needs to calculate the impact of aspect weaving and system composition with respect to WCET or static memory consumption.

Recall that WCETs in the run-time aspects are expressed in terms of symbolic expressions. Hence, they are a function of one or several parameters that in turn abstract the properties of the underlying run-time environment. The symbolic expressions need to be re-evaluated for each run-time environment, i.e., parameters in symbolic expressions should be instantiated in the analysis. Hence, the M&W analyzer provides a list of symbolic expressions and the list of necessary parameters that need to be configured. Since it is possible to assemble a number of configurations in ACCORD-ME, e.g., when using the expert option, the M&W analyzer detects the configurations and enables developers to choose which configuration she/he would like to analyze. Moreover, it is possible to choose whether WCET or memory analysis of the system should be performed. The way run-time aspects for a component are expressed in XML is illustrated in figure 6.3 for a running example of the linked list component. The XML description file for run-time aspect describing the WCET and memory needs of application aspects is analogous. The process of analysis using the M&W is identical to the process of aspect-level WCET analysis discussed in section 5.2.

```
<?xml version="1.0"?>
<!-- START INTERNAL DTD -->
<!DOCTYPE Component_Property (view source for a full document...)>
<!-- END INTERNAL DTD -->

<Component_Run_Time_Property>
<PARAMETERS number="1">
<PARA name="n"/>
</PARAMETERS>
<PROPERTY>
 <COMPONENT name="linkedList">
  <OPERATION name="insert" intWCET="3" intMEM="45">
   <MECHANISM name="createNode" WCET="5" MEM="22" num="1"/>
   <MECHANISM name="linknode" WCET="4" MEM="12" num="1"/>
  </OPERATION>
  <OPERATION name="remove" intWCET="1" intMEM="32">
   <MECHANISM name="getNextNode" WCET="2" MEM="10" num="n"/>
    .....
  </OPERATION>
    ....
 </COMPONENT>
 </PROPERTY>
</Component_Run_Time_Property>
```

Figure 6.3: *An implementation of the run-time aspect of a component in XML*

The formal verifier is a tool that performs formal verification of the composed real-time system based on the formal method for modular verification of reconfigurable components presented in section 5.5. The behavior of the system configuration is checked using formal models of aspects and components represented as augmented timed automata with reconfiguration and verification interfaces (stored in the ACCORD library).

6.1.3 Configuration Compiler

This tool is part of the development environment that aids the designer in compiling the final product. The configuration compiler takes as input:

- ❏ information obtained from ACCORD-ME about the created configuration that satisfied functional and non-functional requirements, and
- ❏ implementation level artifacts.

Based on this input the configuration compiler generates a compilation file, which is then used for compiling the source code of needed aspects and components into the final system. Hence, the output of this tool is the compiled system configuration, which is ready to be deployed in a specific run-time environment. The configuration compiler also provides necessary documentation about the generated configuration that can later be used for maintaining the system.

6.1.4 Discussion

The ACCORD development environment offers benefits for system developers in terms of automated support during the composition process (via the configurator tool), analysis of the system (via various analysis tools), and the compilation and deployment (via the configuration compiler tool). With this automation the overall development time for a real-time system decreases significantly.

The tool environment in general, and ACCORD-ME in particular, treats both components and aspects as first-class constituents of a real-time system. Moreover, the uniqueness of the environment is the set of tools encompassed by the ACCORD-ME for analysis of a real-time system woven with aspects.

For each family of real-time systems developed using ACCORD, i.e., each new application domain, specific implementation of components and aspects that constitute the domain should be made, and these should be placed in the ACCORD library. Moreover, parts of the ACCORD development environment should be extended to embrace a particular domain (section 5.5.4 shows the tool applied to the domain of real-time databases). Namely, the configurator should be extended with a set of requirements and composition rules for this particular domain, and the compilation rules in the configuration compiler tool need also to be updated to contain the rules for compilation of the newly developed components and aspects.

6.2 ACCORD Evaluation

This section provides an evaluation of ACCORD and its main constituents. The evaluation is performed by relating the requirements we identified in chapter 3 to ACCORD. The main requirements are epitomized in table 6.1, which compares ACCORD and the previously discussed design approaches (see chapter 3).

Component model - relation to ACCORD. ACCORD provides a component model for reconfigurable real-time systems, which enforces information hiding and supports three different types of interfaces, hence, fulfilling the requirements CM1 and CM2, respectively (see section 4.3). Moreover, within ACCORD aspects are used for implementation of connectors among components, ensuring efficient extensions to the system and fulfilling requirement CM3.

ACCORD provides a description language for defining temporal attributes of components (see section 4.3.2), corresponding to the fulfillment of CM6, and aims at providing tools that can automatically extract temporal information from any real-time software component built using RTCOM. The description language for temporal attributes of components and guidelines for mapping components to tasks, enable ACCORD to enforce fulfilling of the CM4 requirement by supporting both temporal and structural views of components and the overall real-time system.

We assume that a real-time system should first be decomposed into a set of

		Real-time				Software engineering				
Design approaches / **Criteria**		**DARTS**	**TRSD**	**VEST**	**HRT-HOOD**	**ISC**	**COM CORBA**	**RADL**	**AOP (AspectJ)**	**ACCORD**
Component model										
CM1	Information hiding	◦	◦	◦	●	●	●	●	◦	●
CM2	Interfaces	◦	◦	◦	●	●	●	●	●	●
CM3	Connectors	-	-	◦	-	●	◦	●	◦	●
CM4	Component Views	◦	◦	●	◦	◦	-	◦	-	●
CM5	Task mapping	●	◦	◦	-	-	-	◦	-	◦
CM6	Temporal attributes	◦	●	●	●	-	-	-	-	●
Aspect separation										
AS1	Aspect support	-	-	◦	-	●	-	-	●	●
AS2	Aspect weaving	-	-	◦	-	●	-	-	●	●
AS3	Multiple interfaces	-	-	-	◦	●	●	●	◦	●
AS4	Multiple aspect types	-	-	◦	-	-	-	-	-	●
System composability										
SC1	Static configuration	◦	◦	●	◦	●	◦	●	●	◦
SC2	Dynamic reconfiguration	-	-	◦	-	◦	●	●	◦	●
SC3	Temporal analysis	-	●	●	●	-	-	◦	-	◦
SC4	QoS assurance	-	-	-	-	-	-	-	-	●
SC5	Formal verification	-	-	-	-	-	-	◦	-	◦

LEGEND: ● supported ◦ partially supported - not supported

DARTS: design approach for real-time systems
TRSD: transactional real-time system design
VEST: Virginia embedded systems toolkit
HRT-HOOD: a hard real-time hierarchical object-oriented design
ISC: invasive software composition
AOP: aspect-oriented programming
RADL: reliable architecture description language
ACCORD: aspectual component-based real-time system development

Table 6.1: *Evaluation criteria for ACCORD*

components, which are later mapped to tasks. Hence, the relationship between tasks and components is not fixed. Within ACCORD we provide initial characterization of component-task relationship (see section 5.3). Hence, ACCORD partially fulfills CM5 (task mapping).

Aspect separation - relation to ACCORD. ACCORD meets the requirements AS1 and AS4 by supporting three different types of aspects: application, run-time, and composition aspects (see section 4.2). It also fulfills the requirement AS2 as application aspects can change the code of components by aspect weaving. Run-time aspects describe the behavior of components, e.g., temporal properties and resource consumption. Composition aspects refer to composability issues and relate both to the functional and the temporal compatibility of components in the real-time system. The requirement AS3 is fulfilled as RTCOM provides three types of interfaces: functional, configuration, and composition interfaces (see section 4.3.4).

System composability - relation to ACCORD. ACCORD provides configuration support for the reconfigurable real-time system design by providing design guidelines and tool support within ACCORD development tool environment, hence, meeting the requirement SC1; note that the configuration support has its limitation as discussed in section 6.1.4.

With its component middleware layer and design guidelines for extended RTCOM, ACCORD design is extended to handle dynamically reconfigurable systems (section 4.6). Thereby, requirement SC2 is fulfilled.

ACCORD partially meets the SC3 requirement as it supports static WCET analysis of composed system (see section 5.2); the dynamic schedulability analysis is not automated and relies on the guidelines for the task mapping, and, hence, requires additional refinements (see section 5.3).

Given that ACCORD with its notion of aspect packages enables adding QoS to the system, as well as maintaining the QoS in dynamically reconfigurable systems, requirement SC4 is fulfilled. Formal verification method is developed within ACCORD to support verification of temporal and functional behavior of components woven with aspects, as explained in section 5.5. Since we have primarily focused on the application of the method on the verification of an aspect with a single advice woven into a single component, further generalization of results to complex systems, consisting of many aspects and components, is needed. This implies that ACCORD partially fulfills requirement SC5.

Part III

Component-Based Embedded Real-Time Database System

Chapter 7

Data Management in Embedded Real-Time Systems

In this chapter we present a case study of data management in vehicle control systems developed at Volvo Construction Equipment Components AB, Sweden. We study two different hard real-time systems and observe that the variability of data management requirements in these systems calls for distinct database configurations specially suited for the particular system. The case study, thus, serves as a motivation for developing a reconfigurable COMET database.

7.1 A Case Study: Vehicle Control Systems

In this section we present the case study of two different hard real-time systems developed at Volvo Construction Equipment Components AB, Sweden, with respect to data management [124]. These systems are embedded into two different vehicles (an articulated hauler and a wheel loader), and are typical representative real-time systems for the class of vehicular systems. Both of these vehicle control systems consist of several subsystems called electronic control units (ECUs), connected through two serial communication links: the fast CAN link and the slow

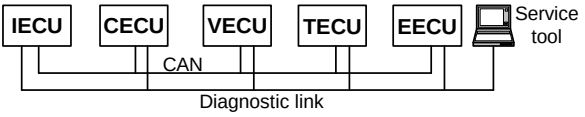


Figure 7.1: *The overall architecture of a vehicle control system*

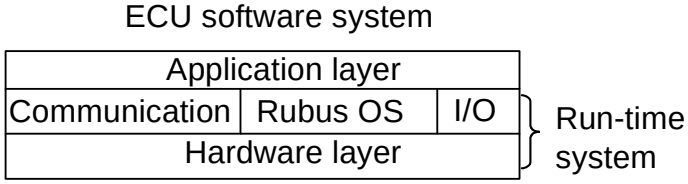


Figure 7.2: *The structure of an ECU*

diagnostic link, as shown in the figure 7.1. Both the CAN link and the diagnostic link are used for data exchange between different ECUs. Additionally, the diagnostic link is used by diagnostic (service) tools. The number of ECUs can vary depending on the way functionality is divided between ECUs for a particular type of the vehicle. For example, the articulated hauler consists of five ECUs: instrumental, cabin, vehicle, transmission, and engine ECU, denoted IECU, CECU, VECU, TECU, and EECU, respectively. In contrast, the wheel loader control system consists of three ECUs, namely IECU, VECU, and EECU.

We have studied the architecture and data management of the VECU in the articulated hauler, and the IECU in the wheel loader. The VECU and the IECU are implemented on hardware platforms supporting three different storage types: EEPROM, Flash, and RAM. The memory in an ECU is limited, normally 64Kb RAM, 512Kb Flash, and 32Kb EEPROM. Processors are chosen such that power consumption and the cost of ECUs are minimized. Thus, processors run at 20MHz (VECU) and 16MHz (IECU) depending on the workload.

Both VECU and IECU software systems consist of two layers: a run-time system layer and an application layer (see figure 7.2). The run-time system layer on the lower level contains all hardware-related functionality. The higher level of the run-time system layer contains an operating system, a communication system, and an I/O manager. Every ECU uses the Rubus [20] real-time operating system. The communication system handles transfer and reception of messages on different networks, e.g., CAN. The application is implemented on top of the run-time system layer. The focus of our case study is data management in the application layer. In the following section we briefly discuss the Rubus operating system. This is followed by sections where functionality and a structure of the application layer of both VECU and IECU are discussed in more detail. For presentation purposes, in the following sections we refer to the application layer of the VECU and IECU as the VECU (software) system and the IECU (software) system.

7.1.1 Rubus

Rubus is a real-time operating system designed to be used in systems with limited resources [20]. The Rubus version of the operating system used in the studied

implementation of ECUs supports both off-line and on-line scheduling, and consists of two parts: red part, which deals with hard real-time, and blue part, which deals with soft real-time.

The red part of Rubus executes tasks scheduled off-line. The tasks in the red part, also referred to as red tasks, are periodic and have higher priority than the tasks in the blue part, invoked in an event-driven manner. The blue part of Rubus also supports functionality that can be found in many standard commercial real-time operating system, e.g., priority-based scheduling, message handling, and synchronization via semaphores. Each task has a set of input and output ports that are used for communication with other tasks.

7.1.2 VECU

The VECU system is used to control and observe the state of the vehicle. The system can identify anomalies, e.g., an abnormal temperature. Depending on the criticality of the anomaly, different actions can be taken, e.g., warning the driver and shutting down the system. Furthermore, some of the functionality of the vehicle is controlled by this system via sensors and actuators. Finally, logging and maintenance via the diagnostics link can also be performed using a service tool that can be connected to the vehicle.

All tasks in the system, except the communication task, are non-preemptive tasks scheduled off-line. The communication task uses its own data structures, e.g., message queues, and, thus, no resources are shared with other tasks. Since the tasks are non-preemptive and run until completion, mutual exclusion is not necessary. The reason for using non-preemptive off-line scheduled tasks is to minimize the run-time overhead and to simplify the verification of the system behavior.

The data in the system can be divided into five different categories: (1) sensor/actuator raw data, (2) sensor/actuator parameter data, (3) sensor/actuator base data, (4) logging data, and (5) parameter data.

The *sensor/actuator raw data* is a set of data elements that are either read from sensors or written to actuators. The data is stored in the same format as they are read/written. This data, together with the *sensor/actuator parameter data*, is used to derive the *sensor/actuator base data*, which can be used by the application. The sensor/actuator parameter data contains reference information about how to convert raw data received from the sensors into base data. For example, consider a temperature sensor, which outputs the measured temperature as a voltage T_{volt}. This voltage needs to be converted to a temperature T using a reference value T_{ref}, i.e., $T = T_{volt} \cdot T_{ref}$.

In the studied system, the sensor/actuator (raw and parameter) data is stored in a vector of data called a hardware database (HW Db), see figure 7.3. The HW Db is, despite its name, not a database but merely a memory structure. The base data is not stored in the system but is derived "on the fly" by data derivation tasks. Apart from data collected from local sensors and the application, sensor

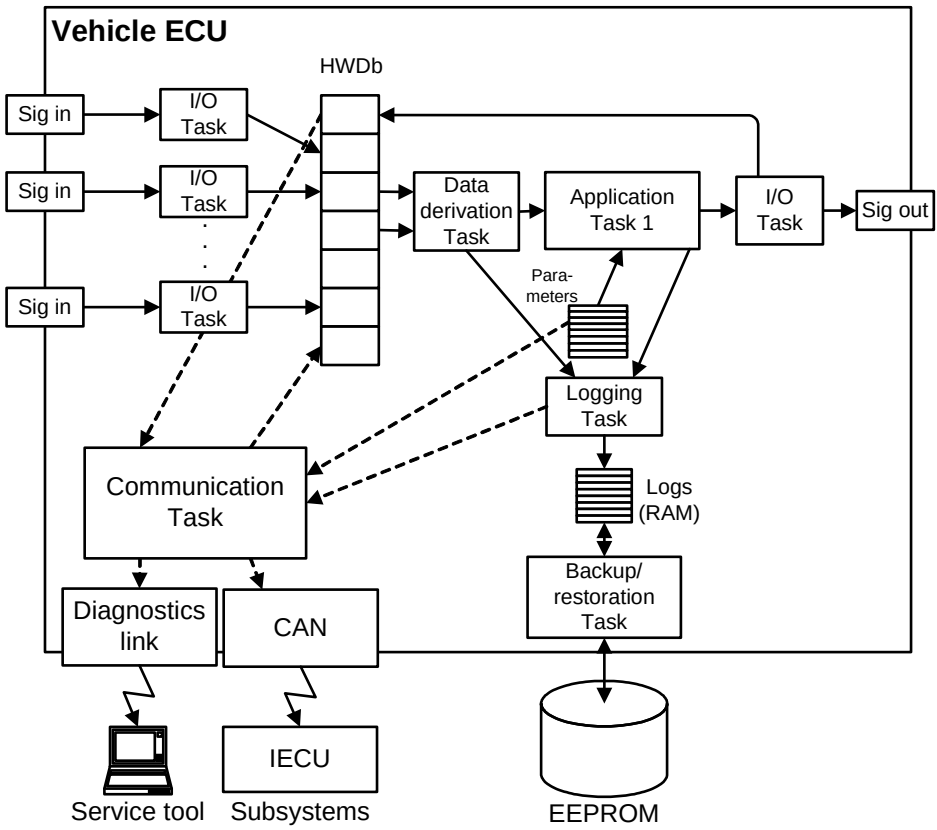


Figure 7.3: *The architecture of the VECU*

and actuator data derived in other ECUs is stored in the HW Db. The distributed data is sent periodically over the CAN bus. From the application's point of view, the locality of the data is transparent in the sense that it does not matter if the data is gathered locally or remotely.

Some of the data derived in the applications is of interest for statistical and maintenance purposes and therefore the data is logged (referred to as *logging data*) on permanent storage media, e.g., EEPROM. Most of the logging data is cumulative, e.g., the total running time of the vehicle. These logs are copied from EEPROM to RAM in the startup phase of the vehicle and are then kept in RAM during runtime, to finally be written back to EEPROM memory before shutdown. However, logs that are considered critical are copied to EEPROM memory immediately at an update, e.g., warnings. The *parameter data* is stored in a parameter area. There are two different types of parameters, permanent and changeable. The permanent parameters can never be changed and are set to fulfill

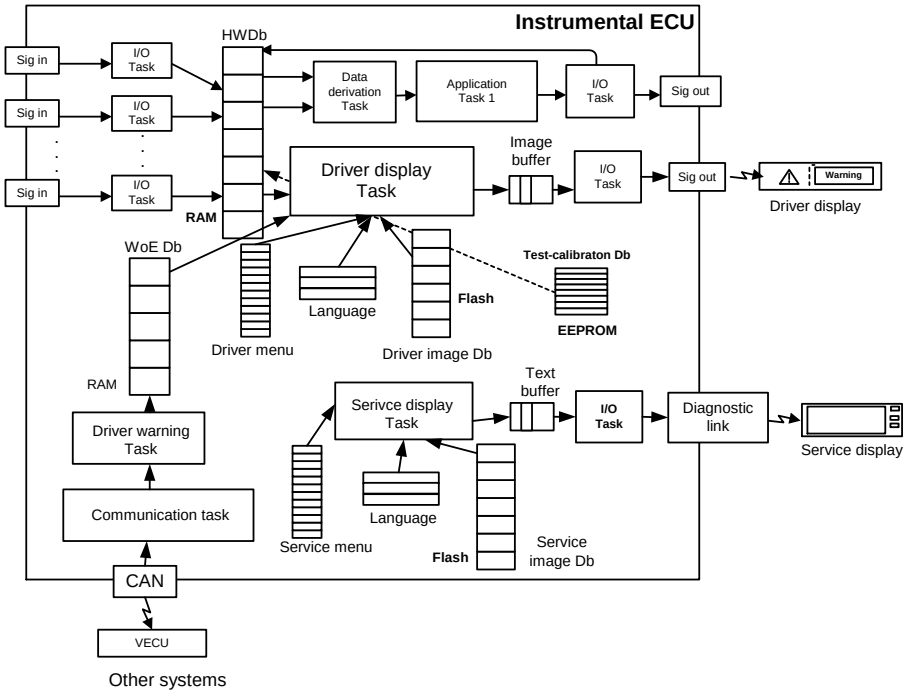


Figure 7.4: *The architecture of the IECU*

certain regulations, e.g., pollution and environment regulations. The changeable parameters can be changed using a service tool.

Most control applications in the VECU follow a common structure residing in one precedence graph. The sensors (Sig In) are periodically polled by I/O tasks (typically every 10 ms) and the values are stored in their respective slot in the HW Db. The data derivation task then reads the raw data from the HW Db, converts it, and sends it to the application task. The application task then derives a result that is passed to the I/O task, which both writes it back to the HW Db and to the actuator I/O port.

7.1.3 IECU

The IECU is a display electronic control unit that controls and monitors all instrumental functions, such as displaying warnings, errors, and driver information on the driver display. The IECU also controls display of the service information on the service display (a unit for servicing the vehicle), the I/O in the driver cabin, e.g., accelerator pedal, and the communication with other ECUs via CAN

and the diagnostic link.

The IECU differs from the VECU in several ways. Firstly, the data volume in the system is significantly higher since the IECU controls displays and, thus, works with a large amount of images and text information. Moreover, the data is scattered in the system and, depending on its nature, stored in a number of different data structures as shown in figure 7.4. Similarly to the HW Db, data structures in the IECU are referred to as databases, e.g., image databases, menu databases, and language databases. Since every text and image information in the system can be displayed in thirteen different languages, the interrelationships of data in different data storages are significant.

A dominating task in the system is the task updating the driver display. This is a red task, but, in contrast to other red tasks, it can be preempted but only by red tasks. However, scheduling of all tasks is performed such that all possible data conflicts are avoided.

Data from the HW Db in the IECU is periodically pushed on to the CAN link and copied to the HW Db of the VECU. Warnings or errors (WoE) are periodically sent through the CAN link from/to the VECU and are stored in the dedicated part of RAM, referred to as the WoE database (WoE Db). Hence, the WoE Db contains information of active warnings and errors in the overall wheel loader control system. While WoE Db and HW Db allow both read and write operations, the image and menu databases are read-only databases.

The driver display is updated as follows (see figure 7.4). The driver display task periodically scans the databases (HW, WoE, and menu Db) to determine the information that needs to be presented on the driver display. If any active WoE exist in the system, the driver display task reads the corresponding image, in the specified language, from the image database located in a persistent storage and then writes the retrieved image to the image buffer. The image is then read by the blue I/O task, which then updates the driver display with an image as many times as defined in the WoE Db. Similarly, the driver display task scans the HW Db and menu database. If the HW Db has been updated and this needs to be visualized on the driver display, or if data in the menu organization has been changed, the driver display task reads the corresponding image and writes it to the driver display as described previously. In the case a service tool is plugged into the system, the service display task updates the service display in the same way as described for the driver display, but then uses its own menu organization and image database, buffer, and the corresponding blue I/O task.

7.2 Data Management Requirements

Table 7.1 gives an overview of data management characteristics in the VECU and IECU systems. The following symbols are used in the table:

v — feature is true for the data type in the VECU,
i — feature is true for the data type in the IECU, and
x — feature is true for the data type in both VECU and IECU.

Data types		*Sensor*	*Actuator*	*Base*	*Parameters*	*WoE*	*Image&Text*	*Logs*
Management characteristics								
Data source	HW Db	x	x			i		
	Parameter Db				x			
	WoE Db					i		
	Image Db						i	
	Language Db						i	
	Menu Db						i	
	Log Db							v
Memory type	RAM	x	x	x	x	x		v
	Flash						i	
	EEPROM				x			v
Memory allocation	Static	x	x	x	x	x	i	v
	Dynamic							
Interrelated with other data		x	x	x	x	x	i	v
Temporal validity		x	x	x		x		v
Logging	Startup							v
	Shutdown							v
	Immediately			v[1]				
Persistence		x	x	v[1]	x	x		
Logically consistent		x	x	x	x			
Indexing							i	
Transaction type	Update	x	x	x	x	x		v
	Write-only	x		x				
	Read-only		x	x	x		i	
	Complex update	x	x	x				v
	Complex queries	x	x	x	x	x	i	v

Table 7.1: *Data management characteristics for the systems*

As can be seen from the table 7.1, all data elements in both systems are scattered in groups of different flat data structures referred to as databases. These databases are flat because the data is structured mostly in vectors, and the databases only contain data with no support for DBMS functionality.

The nature of the systems places special requirements on data management in terms of (see table 7.1):

- ❏ only static memory allocation is allowed, since dynamic memory allocation is not allowed due to the safety-criticality of the systems;
- ❏ small memory consumption, since production costs should be kept as low as possible; and

[1]The feature is true only for some base data in the VECU.

- ❑ diverse data accesses, since data can be stored in different storages, e.g., EEPROM, Flash, and RAM.

Most data, from different databases and even within the same database, is logically related. These relations are not intuitive, which makes the data hard to maintain for the designer and programmer as the software of the current system evolves. Raw values of sensor readings and actuator writings in the HW Db are transformed into base values by the data derivation task, as explained in section 7.1.2. The base values are not stored in any of the databases, rather they are placed in ports (shared memory) and given to application tasks when needed.

The period times of updating tasks ensure that data in both systems (VECU and IECU) is correct at all times with respect to absolute consistency. Furthermore, task scheduling, which is done off-line, enforces relative consistency of data by using an off-line scheduling tool. Thus, data in the system is temporally consistent (we denote this data property in the table as temporal validity). Exceptions are permanent data, e.g., images and text, which are not temporally constrained (see table 7.1).

One implication of the systems' demand on reliability, i.e., the requirement that a vehicle must be movable at all times, is that data must always be temporally consistent. A violation of temporal consistency is viewed as a system error, in which case three possible actions can be taken by the system: (1) use a predefined default data value (most often), (2) use an old data value, and (3) shutdown of involved functions (system exposes degraded functionality).

Some data is associated with a range of valid values, and is kept logically consistent by tasks in the application (see table 7.1). The negative effect of enforcing logical consistency by the tasks is that programmers must ensure consistency of the task set with respect to logical constraints.

Persistence in the ECUs is maintained by storing data on a stable storage. However, exceptions to this rule exist, e.g., RPM data is never copied to the stable storage. Also, some of the data is only stored on the stable storage, e.g., internal system parameters. In contrast, data imperative to systems' functioning is immediately copied to the stable storage, e.g., WoE logs are copied to/from the stable storage at startup/shutdown.

Several transactions exist in the VECU and IECU systems: (i) update transactions, which are application tasks reading data from the HW Db; (ii) write-only transactions, which are tasks updating sensor values; (iii) read-only transactions, which are tasks reading actuator values; and (iv) complex update transactions, which originate from other ECUs. In addition, complex queries are performed periodically to distribute data from the HW Db to other ECUs.

Data in the VECU is organized in two major data storages, RAM and Flash. Logs are stored in EEPROM and RAM (one vector of records), while 251 items structured in vectors are stored in the HW Db. Data in the IECU is scattered and interrelated throughout the system even more in comparison to the VECU (see table 7.1). For example, the menu database is related to the image database,

which in turn is related to the language Db and the HW Db. Additionally, data structures in the IECU are fairly large. HW Db and WoE Db reside in RAM. HW Db contains 64 data items in one vector, while WoE Db consists of 425 data items structured as 106 records with four items each. The image Db and the language Db reside in Flash. All images can be found in 13 different languages, each occupying 10Kb of memory. The large volume of data in the image and language databases requires indexing. Indexing is today implemented separately in every database, and even every language in the language Db has separate indexing on data.

The main problems we have identified in existing data management can be summarized as follows:

- ❏ data is scattered in the system in a variety of databases, each representing a specialized data store for a specific type of data;
- ❏ base data values are not stored in any of the data stores, but are placed in ports, which complicates maintenance and makes adding of functionality in the system a difficult task;
- ❏ application tasks must communicate with different data stores to get the data they require, i.e., the application does not have a uniform access or view of the data;
- ❏ temporal and logical consistency of data is maintained by the tasks, increasing the level of complexity for programmers when maintaining a task set; and
- ❏ data from different databases exposes different properties and constraints, which complicates maintenance and modification of the systems.

A possible solution to these problems is to integrate a database management system to ensure uniform access to data and ease overall data manipulation in the system. Moreover, we need to be able to configure a database system to meet specific requirements on data management in different systems, as we elaborate next.

7.3 Observations

As can be noted, a real-time system controlling a vehicle is operating in a closed environment as its workload characteristics are known beforehand and do not change during the operational lifetime of the system. Furthermore, vehicle control systems are real-time safety-critical systems consisting of several distributed nodes, each implementing a specific functionality. Although nodes depend on each other and collaborate to provide required behavior for the overall vehicle control system, each node can be viewed as a stand-alone real-time system. The size of the

nodes can vary significantly, from very small nodes to large nodes. For instance, a vehicle control system could consist of a small number of resource adequate ECUs responsible for the overall performance of the vehicle, e.g., 32-bit CPUs with a few Mb of RAM, and a large number of ECUs responsible for controlling specific subsystems in the vehicle, which are significantly resource-constrained, e.g., an 8-bit micro-controller and a few Kb of RAM [124].

Depending on the functionality of a node and the available memory, different database configurations are preferred. In safety-critical nodes, e.g., VECU, tasks are often non-preemptive and scheduled off-line, avoiding concurrency by allowing only one task to be active at any given time. This, in turn, influences functionality of a database in a given node with respect to concurrency control. Less critical nodes having preemptable tasks, e.g., IECU, would require concurrency control mechanisms. Furthermore, some nodes require critical data to be logged, e.g., warning and errors, and require backups on startup and shutdown, while other nodes only have RAM (main-memory), and do not require non-volatile backup facilities from the database. Hence, to provide support for data management in these types of systems we need to enable development of different database configurations to suit the needs of each node with respect to memory consumption, concurrency control, recovery, scheduling techniques, transaction models, and storage models. In the next chapter we show how we have reached this goal by applying ACCORD constituents to the design and development of the reconfigurable embedded real-time database COMET.

Chapter 8

COMET Database Platform

As mentioned before, real-time and embedded systems would benefit from having a real-time database that can be configured to meet the specific data management needs of the underlying system. In this chapter we present COMET, a real-time embedded database platform that can be reconfigured to meet the needs of various real-time and embedded applications. Reconfiguration in COMET is achieved by employing ACCORD. Following the ACCORD design process, we describe the decomposition of COMET into components and aspects, and then focus on the design and implementation of a number of COMET aspect packages.

8.1 COMET Decomposition

After the requirements on data management in the target application domain (vehicle systems) are gathered, the need for developing reconfigurable and reusable embedded real-time database has been identified in chapter 7.[1] To ensure development of such a database we employ ACCORD, starting with steps ❷ and ❸ of the ACCORD development process (see figure 4.1). Hence, we decompose data management of a real-time embedded system first into a set of components, followed by decomposition into a set of aspects.

8.1.1 Decomposition into Components

When decomposing into components, each component is identified such that it encapsulates a well-defined functionality of a database management system. Moreover, another criterium in decomposition is to ensure that application-dependent

[1] Nowadays, the COMET platform is extended and embraces a larger class of real-time systems, including systems operating in closed environments and those operating in open, unpredictable real-time environments.

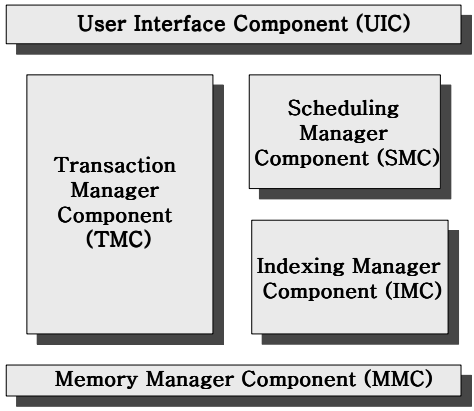


Figure 8.1: *Decomposition of COMET into components*

functionality is encapsulated into a component for easier exchange and reuse. Also, the identified components, when composed into a system, should at least provide basic functionality of the system that can later be reconfigured. Hence, for the case of an embedded real-time database system, we identified the following set of components (see figure 8.1):

- ❏ the user interface component (UIC),
- ❏ the scheduling manager component (SMC),
- ❏ the indexing manager component (IMC),
- ❏ the transaction manager component (TMC), and
- ❏ the memory manager component (MMC).

The UIC provides a database interface to the application, which enables an application to query and manipulate data elements. Depending on the target application, different user interfaces can be applicable implying that this part of the database functionality is a good candidate for encapsulation into a component in order to facilitate its reuse and reconfiguration.

Arriving transactions are scheduled in the system by the SMC. Hard real-time applications typically do not require sophisticated transaction scheduling, i.e., the system allows only one transaction to access the database at a time [124]. Therefore, for these types of applications the SMC is unnecessary while for others it might be a vital part of the system. Since in embedded environments it also important to minimize the footprint of the system, we need to ensure that the scheduling functionality can easily be omitted from the system, or added when necessary.

The TMC is responsible for executing incoming transactions, thereby performing the actual data manipulation. Transaction management can be done in a number of distinct ways, depending on the goals of the database system, e.g., relational vs. object-oriented transaction processing. Having transaction management encapsulated into a component provides a good basis for exchange of the overall transaction management functionality and, more importantly, ensures that other parts of the system functionality can straightforwardly be exchanged or modified without changing the way transaction is managed. In contrast, if transaction management would not be encapsulated into a component, but implemented partly in a number of other components, modifying the functionality of one component could also require extensive modifications of all components containing parts of transaction management.

The IMC is responsible for maintaining an index of all tuples in the database. Indexing strategies could vary depending on the real-time application with which the database should be integrated, e.g., T-trees [109] and multi-versioning are suitable for applications with a large number of read-only transactions [145]. To ensure that a database can be configured for a large number of applications requiring distinct indexing strategies and structures, the IMC should be made an exchangeable part of the database architecture.

The MMC manages access to data in the physical storage. Recall from chapter 7 that real-time systems can have different storage types, e.g., EEPROM, Flash, and RAM. Each of the storage types might require distinct memory manipulation methods for data writing and reading. Hence, to make sure that a database can be configured and used with various storage types, it is beneficial to have the functionality in charge of memory manipulation encapsulated into an exchangeable component.

The identified COMET components are necessary for creating a configuration that provides basic database functionality, i.e., a database that can take an arriving transaction via the UIC, register the transaction using the SMC, perform operations on data via the TMC by looking up the data (IMC) and reading/writing their values (MMC). A detailed description of the design and functionality of each COMET component is given later in section 8.2.

8.1.2 Decomposition into Aspects

Following ACCORD, after decomposing the system into a set of components we further decompose it into a set of aspects. The decomposition of COMET into aspects fully corresponds to the ACCORD decomposition (given in section 4.2) into three types of aspects: run-time, composition, and application aspects. Although the majority of application aspect might be developed as a part of an aspect package, it is nevertheless beneficial to identify these aspects after the decomposition into component has been made. This is to capture the crosscutting effects of aspects to previously identified components as these can influence the design and implementation of the components. Hence, in this section we illustrate various

COMET application aspects and discuss the possible impact of crosscutting.

Application aspects are determined based on the characteristics of the application domain, while keeping in mind previously made decomposition into components. For COMET, the initial application domain was vehicle control systems, and a database system in this domain needs to support both concurrent and non-concurrent execution of transactions. This has a profound impact on the way application aspects are defined.

For example, concurrent transaction execution can result in conflicts on data items, inducing a need for concurrency control algorithms. Recall from section 2.4.1 that a variety of concurrency control methods exist, each suitable for a particular group of applications. To ensure high reconfigurability of the database, concurrency control should be designed and implemented as an aspect in the database structure.

Additionally, it is possible to customize the indexing strategy depending on the number of transactions active in the system. For example, in vehicle control applications where only one transaction is active at a time there is no need to ensure mutually exclusive access to the indexing structure as it cannot become inconsistent. In more complex applications where multiple transactions can execute concurrently, there is a possibility that the index structure becomes inconsistent if accessed at the same time by a number of transactions. Such scenarios require explicit access strategies ensuring that only one transaction at the time is allowed to access the structure. This type of indexing where the indexing strategy provides explicit handling of inconsistencies, e.g., via mutual exclusion and semaphores, is referred to as thread-safe indexing. A way of ensuring thread-safe indexing is to define and weave a synchronization aspect into the database system. Note that the synchronization aspect ensuring mutual access to a resource is also necessary to be woven into the SMC, and TMC, i.e., all components that contain data structures affected by transaction execution.

Various real-time policies, e.g., concurrency control, indexing, and scheduling, require additional attributes to be defined for a transaction to represent transaction models. Since changes to the transaction need to be propagated to all components in the system (see table 8.1), a transaction model can be viewed as a highly crosscutting concern and would therefore be beneficial to design and implement it as an aspect in the database system. Note that the application aspects vary depending on a specific real-time system and, thus, particular attention should be made to identify the application aspects for each real-time system.

In the sections that follow, we first present design and implementation of COMET components, constituting the basic database functionality. Since COMET initially was intended for vehicle systems, to ensure conformance with legacy software and current industrial practice, we chose the C programming language as the implementation language for COMET components. Aspects and components providing specific functionality are grouped and presented in the context of aspect packages. This is also to illustrate the way systems can be developed and evolved using aspect packages. Due to the current unavailability

COMET components / Application aspects	UIC	SMC	IMC	MMC	TMC
Transaction	X	X	X	X	X
Real-time scheduling		X			X
Concurrency control	X		X	X	X
QoS policy	X	X			X
Memory optimization	X	X	X		X
Synchronization		X	X		X
Security	X		X	X	X

Table 8.1: *Crosscutting effects of different application aspects on COMET components*

of AspectC [47], AspectC++ [158] has been used for implementation of aspects within COMET.

8.2 COMET Components

In this section we describe in detail the design and implementation of COMET components, and discuss the flow of transaction execution in COMET.

8.2.1 User Interface Component (UIC)

The UIC is the component that provides a user interface to the application. To facilitate aspect weaving and manipulation of transactions, transaction-specific details are stored in a structure called `DBTrans` (implemented using `struct`). Depending on the number of transactions in the system, this structure can be a part of an array of `DBTrans` structs, where each `DBTrans` contains the information for a specific transaction. The number of allowed transactions in the system is a changeable parameter and can be set by the designer via the UIC configuration interface.

Applications use operations provided by the UIC to create and execute new transactions (see figure 8.2(a)). A transaction is initialized by an application by invoking the operation `beginTrans()`[2] of the UIC. After a transaction has been

[2]The names of operations and mechanisms are simplified for presentation purposes. Actually, in the COMET implementation, the name of an operation or a mechanism is always preceded

initialized queries can be posed to the database using the operation `query()`. The query language in COMET has an SQL-like syntax. Currently, the most common database query commands are supported, namely `select`, `insert`, `update`, `delete`, `create table` and `drop table`. Queries are passed to the rest of the database as strings. When submitted by an application using the operation `query()`, a query is parsed by the UIC, which creates an execution plan for the query and stores it into a tree structure denoted execution tree. Each node in the execution tree represents an operation that needs to be performed on data items in the database in order to fulfill the query. The execution tree for a transaction is stored in the `DBTrans` struct of the transaction. The operation `endTrans()` is used by an application to signal the database that the actual execution of a transaction should be initiated. Once a transaction is submitted to the database, the UIC calls the SMC operations to ensure appropriate scheduling of the transaction. When the transaction is scheduled, the SMC notifies the UIC that the scheduling is done. Now, the UIC can initialize execution of operations on data items by calling appropriate operations of the TMC component. Hence, the UIC requires operations from both SMC and TMC, and these are recorded in the required interface of the UIC.

The UIC also provides an operation that enables the result of the transaction execution to be presented to the application. The mechanisms within the UIC are used for implementing UIC operations, as well as for aspect weaving, and are declared in the UIC composition interface.

8.2.2 Scheduler Manager Component (SMC)

The SMC provides mechanisms for scheduling transactions arriving into the system, based on the chosen scheduling policy. The SMC is also in charge of maintaining the list of all transactions that exist in the system, including scheduled transactions and unscheduled but active transactions, i.e., transactions submitted for execution. Note that the SMC in the non-concurrent database configuration still exists but it is only used for registering an active transaction in the system. The operations provided by the SMC to the system are depicted in figure 8.2(b).

The SMC manages a set of threads, called a thread pool. To each transaction submitted to the database system via the UIC, a thread is assigned. The number of threads available in the thread pool is a configurable parameter set by the system designer. Depending on the run-time environment constraints, this parameter can be changed (it is declared in the component configuration interface). The SMC maintains two queues of transactions, a ready queue and an active queue. Transactions currently executing are stored in the active queue, while transactions that arrived in the system but have not yet started executing, i.e.,

with the abbreviation of the component and an appropriate designation: `_op` for operations and `_mech` for mechanisms. Detailed explanation of naming and coding rules enforced in COMET can be found in the COMET User Manual [103].

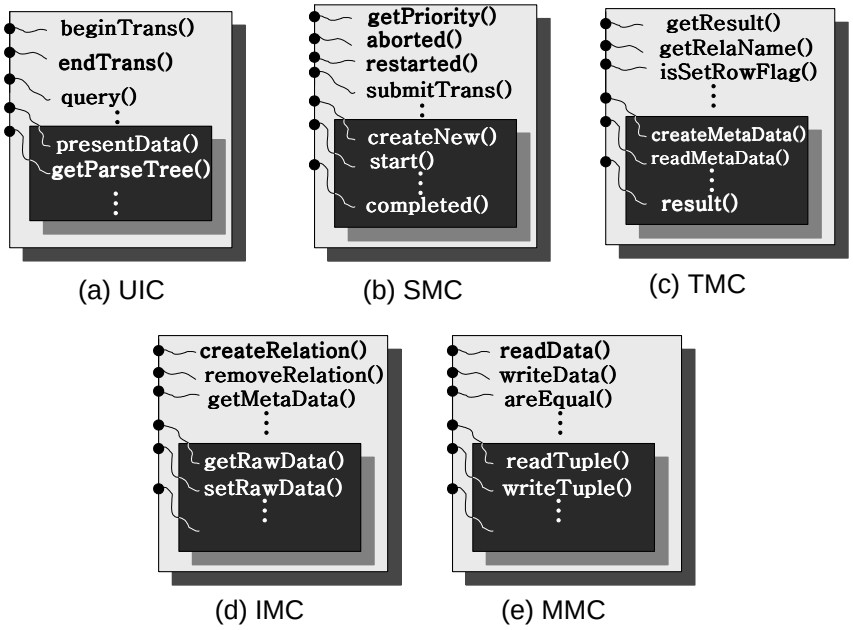


Figure 8.2: *The outlook of the functional part of the COMET components*

are pending for execution, are held in the ready queue. Since an arriving transaction is ready to be executed it is placed into the ready queue, which is sorted according to transaction priorities determined by the chosen scheduling policy. When the transaction starts executing, it is removed from the ready queue and placed into the active queue. Upon completion, the transaction is removed from the active queue and, thereby, leaves the system.

When a transaction, τ, is committed to the database in the manner described in section 8.2.1, the UIC uses the SMC operation `submitTrans()` to submit the request for scheduling this new transaction. Upon receiving the request for scheduling τ, the SMC places τ in the ready queue, and then the SMC checks wether it is possible to start the execution of τ. Three possible scenarios in transaction execution emerge.

1. **The thread pool contains at least one thread.** In this case transaction τ is immediately assigned to an available thread, removed from the ready queue, and placed in the active queue as it starts to execute.

2. **The thread pool is empty and transactions currently executing have at least the same priority as** τ. In this case τ remains in the ready queue waiting to be scheduled by the SMC. The next transaction to execute is chosen with the highest priority from the ready queue. Recall

that the ready queue is sorted according to transaction priorities which are determined by the scheduling algorithm, e.g., EDF.

3. **The thread pool is empty and at least one of the currently executing transactions has a lower priority than τ.** In this case the transaction with lower priority is rolled back before completion to return its thread to the thread pool. The released thread is then assigned to τ, which starts to execute. Thus, τ is removed from the ready queue and placed into the active queue.

The threads from the thread pool are scheduled by the underlying operating system. When a transaction is completed, it releases its thread and is removed from the active queue. If a transaction is rolled back before completion, it also releases its thread, and is removed from the active queue and then placed back in the ready queue.

The SMC component requires operations from the UIC, to inform the UIC that the transaction has successfully been scheduled. Furthermore, the SMC provides operations to the UIC and the LMC.

8.2.3 Transaction Manager Component (TMC)

The TMC coordinates the activities of all components in the system with respect to transaction execution. Hence, the TMC performs the actual manipulation of data in the database. The execution of a transaction is initiated by the UIC using operation `getResult()` provided by the TMC (see figure 8.2(c)). The TMC executes a transaction by traversing its execution tree. The traversal is done sequentially, using the recursive call to the mechanism `result()`, which is the core mechanism of the TMC. For each node in the execution tree, affected relations are loaded into buffers using operations provided by the IMC and the MMC; hence, these are part of the required interface of the TMC. The relations are examined tuple by tuple, and tuples not needed in the query are deleted. The operations of the query are performed on the tuples in the buffers. If a transaction contains an update query, all the affected tuples are written back to memory, again utilizing operations of the IMC and MMC. When finished with a previous node in the execution tree of a transaction, the TMC starts with the next one. When finalizing the last node, the TMC informs the SMC that the transaction is completed. The so-called buffer manager component (BMC), which is a part of TMC, is in charge of the buffer management.

8.2.4 Indexing Manager Component (IMC)

The IMC deals with indexing of data. Currently, the IMC operations, shown in figure 8.2(d), are used to find tuples in relations by storing their addresses in searchable trees. These addresses are used when reading or writing tuples with the MMC. The IMC provides functionality for managing relations and tuples, e.g.,

insertion of new tuples. The default IMC implements the T-tree index structure [109].

8.2.5 Memory Manager Component (MMC)

The MMC depicted in figure 8.2(e) manages access to data in the physical storage. For example, each time a tuple is added or deleted, the MMC is invoked to allocate and release memory. Generally, all reads or writes to/from the memory in COMET involve the MMC. The operations provided by the MMC, e.g., `readData`, `writeData`, `allocate`, and `deallocate`, are used by the TMC to manage relations, and by the IMC to manage the index.

8.2.6 Transaction Flow

Transaction execution in COMET is done in a number of steps depicted in figure 8.3. Each step is depicted with components involved in that step of execution. The following is an explantation of execution steps of update and read-only transactions, i.e., `update` and `select` queries.

1. An application sends an SQL query as a string to the UIC, initiating a new transaction for the user query. Initialization implies the UIC stores transaction-specific details in the `DbTrans` struct, parses the query string, and produces a corresponding execution tree, also stored in `DbTrans`. After that, the transaction is submitted to the the database for execution using the UIC.

2. The transaction is scheduled by the SMC as described in section 8.2.2.

3. The TMC loads the relations needed by the query into buffers. Each relation is loaded as follows. First, the IMC is used to get the address of the metadata for the relation. The metadata describes properties of the relation and its attributes, such as the name and types of the attributes. The MMC is then called to read the metadata from memory. The obtained metadata are stored by the TMC in a buffer. The TMC continues transaction execution by locating every tuple in the relation using the IMC, and reads the tuples into the buffer via the MMC.

4. The operations of the queries are performed on the tuples in the buffer, e.g., changing the value of an attribute. If the transaction is read-only, it leaves the TMC at this point, and moves to step 6.

5. If the transaction contains an update query, the tuples affected in the previous step are committed by writing the updated values to the memory using first the IMC to obtain the correct address of the tuple, and then using the MMC to write the tuple to memory.

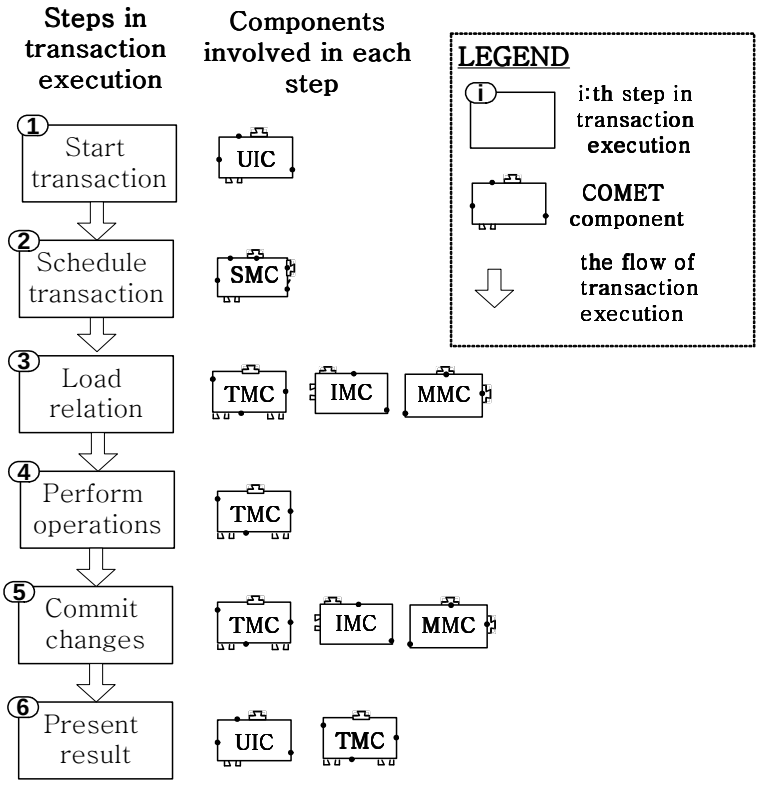


Figure 8.3: *The execution steps of a transaction in COMET*

6. The result is presented to the user via the UIC, which in turn utilizes the TMC.

The execution steps when performing `delete` or `insert` queries are analogous, with the exception that metadata is updated in step 5 as well.

8.3 Concurrency Control Aspect Package

The concurrency control (CC) aspect package was identified based on the guidelines presented earlier in section 4.4. Recall that an aspect package normally consists of a number of components and aspects providing a specific policy. The aspects in the aspect package can be categorized further into policy, task (transaction) model, and connector aspects. In the case of COMET, the CC aspect package consists of one component, locking manager component (LMC), and the following application aspects:

- CC policy aspects: high priority 2 phase locking with similarity aspect and optimistic divergence control aspect; and
- CC transaction model aspects: basic and epsilon-based transaction model aspects.

Since the LMC is the only component in the package, it was more convenient to integrate the functionality of the connector aspect with CC policy aspects (thereby leaving the class of connector aspects unpopulated).

The CC policy aspects define the policy used for resolving conflicts that occur when multiple transactions require the same data. The CC transaction model aspects modify the transaction model of the database to fit the model required by a particular CC policy.

Before going into details on the implementation of these aspects, we first briefly present the main ideas behind the chosen concurrency control algorithms for conflict resolution, and discuss the transaction models needed for realization of each of these algorithms.

8.3.1 CC Policies

For ensuring conflict resolution in the COMET database, we have chosen to implement two representative CC policies: (i) high-priority 2 phase locking with similarity as the representative of the class of pessimistic CC policies, and (ii) optimistic divergence control as a representative of the class of optimistic approaches.

HP-2PL with Similarity

High priority 2 phase locking (HP-2PL) with similarity [95] is a pessimistic concurrency control. It is founded on HP-2PL, which assumes that a transaction

Lock	Read	Write
Read	✓	✗
Write	✗	✗

Table 8.2: *The lock compatibility for HP-2PL concurrency control policy*

executes in two phases: (i) the growing phase, in which the transaction acquires all locks and is not allowed to release any of the locks, and (ii) the shrinking phase, in which all locks held by the transaction are released. HP-2PL uses two types of locks, read (shared) locks and write (exclusive) locks. Table 8.2 shows the compatibility between the locks, where ✓ indicates that locks are compatible and ✗ indicates that they are conflicting. If a transaction tries to lock an item, such that a lock conflict occurs, the conflict resolution policy is employed. In HP-2PL, the conflict is resolved depending on the priorities of the involved transactions as follows. If the requesting transaction has the highest priority of all transactions holding a lock, these are aborted and the requesting transaction acquires the lock. If the requesting transaction does not have the highest priority, it is blocked until it becomes the highest priority transaction.

When similarity is incorporated in HP-2PL, the conflicts resolution policy is based on the similarity of transactions in the value domain. To outline the way conflicts are handled we introduce the following notation:

- $\tau_1, \ldots, \tau_n$ is a collection of transactions in the system that manipulate data item x,
- $v(x)$ is the original value of data item x, before any transaction τ_i, $1 \leq i \leq n$, has accessed it,
- $v(\tau_i, x)$ is the new value of x that some transaction τ_i wants to write to x,
- τ_{new} is a new transaction coming into the system that intends to access data item x, and
- $v(\tau_{new}, x)$ is the value of data item x that τ_{new} intends to write to x.

If τ_{new} is an update transaction acquiring a lock on x in a conflicting mode with some of transactions τ_i already holding a lock, the conflicting transactions are considered similar and the lock is granted to τ_{new} if the following conditions hold:

1. $v(\tau_{new}, x)$ is similar to $v(x)$
2. $v(\tau_{new}, x)$ is similar to $v(\tau_i, x)$, $\forall i : 1 \leq i \leq n$

If τ_{new} is a read-only transaction, only one condition should hold in order for conflicting transactions to be similar:

1. $v(x)$ is similar to $v(\tau_i, x)$, $\forall i : 1 \leq i \leq n$

The conflict is handled using conventional HP-2PL if the conflicting transactions are not similar. The semantics of the "is similar" relation is application-specific and should be defined by the application designer. The similarity can be a time interval within which two values are read from a sensor, or a certain data value threshold that should not be exceeded.

Optimistic Divergence Control

Optimistic divergence control (ODC) [181] is based on an optimistic concurrency control method introduced by Wu and Dias [184] that uses weak and strong locks and assumes transaction execution in three phases: read, write, and validation phase. As a transaction executes it acquires weak read and write locks on the accessed data items, and updates data in the local space. If a strong lock is already held on any of the items, the requesting transaction is marked for abortion, and is aborted in the validation phase. If a transaction has not been marked for abortion during its read/write phase, it is committed in the validation phase. During commit, all weak write locks of a transaction are temporarily converted into strong locks. This implies that at this point all other transactions holding weak locks on data items that get locked by strong locks are marked for abortion.

In ODC transactions can be read-only, update, and complex queries. To distinguish between update and query transactions, we denote query transactions as τ_q, and update transactions as τ_u. Every query transaction τ_q has two attributes: $ImpValue(\tau_q)$, denoting the currently imported inconsistency, and $ImpLimit(\tau_q)$ setting the limit on the amount of inconsistency the query is allowed to import. Update transactions τ_u are associated with similar values. $ExpValue(\tau_u)$ denotes the currently exported inconsistency and $ExpLimit(\tau_u)$ sets the limit on the amount of inconsistency a transaction is allowed to export. Hence, when a transaction τ_q requests a weak read lock on a data item and a strong lock is already held by some other transaction τ_u, the import value of τ_q and the export value of τ_u is checked. If any of these values exceed their respective limits after an increment by the difference in absolute values on the data item caused by τ_u during its execution, τ_q is marked for abortion. If both values are still within their limits, τ_q is not marked for abortion, rather it is marked as non-serializable as the conflict is resolved based on epsilon serializability. In the validation phase transactions are allowed to commit as long as they are not marked for abortion.

8.3.2 Data and Transaction Model

The HP-2PL with similarity policy requires a simple transaction model, which we denote as the basic transaction model. Namely, in the basic transaction model data elements are associated only with their values and each transaction τ_i is characterized with a period p_i, and a relative deadline d_i. Transactions can be read-only transactions, update transactions, or complex query transactions. The

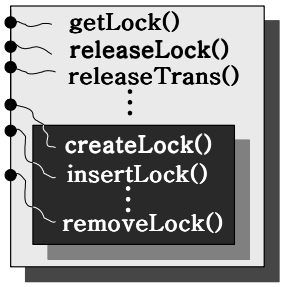


Figure 8.4: *The locking manager component in COMET*

basic transaction model suffices for HP-2PL with similarity as the only information required from a transaction during its execution is its priority. The transaction priority is set based on the scheduling policy.

The ODC method, however, requires broadening of the transaction model to embrace export and import limits on transactions. The epsilon-based transaction model reflects the requirements on the transactions posed by the ODC policy. In this model each transaction τ_i is characterized with:

- ❏ period p_i,
- ❏ relative deadline d_i,
- ❏ export limit exp_i, and
- ❏ import limit imp_i.

Export and import limits of transactions are set by the system designer depending on the need of an application with which the database is used.

8.3.3 COMET CC Aspect Package

Both HP-2PL with similarity and ODC can be implemented using locks and, thereby, belong to the class of lock-based concurrency control policies. Although these policies are different in the way they resolve conflicts, locking and unlocking of data is done in the same manner in both algorithms. This results in using the same LMC for both algorithms and an analogous implementation of HP-2PL and ODC when it comes to the reconfiguration locations (i.e., join points) in the code of COMET components.

The LMC deals with locking of data, providing mechanisms for lock manipulation, and maintaining lock records in the database. Operations provided by the LMC that enable lock management together with the underlying mechanisms are

```
aspect CCpolicy{

/* pointcuts */

/* relating to MMC */
  pointcut readTuple(DataElem *de)=
    execution("bool MMC_readTuple(...)")&&args(de);
  pointcut writeTuple(DataElem *de)=
    execution("bool MMC_writeTuple(...)")&&args(de);

/* relating to IMC */
  pointcut insertIndex(char *relation.char *key, DataElem *address)=
    execution("bool IMC_insert(...)")&&args(relation,key,address);


/* advices */
   advice readTuple(de): before(DataElem *de){
     CC_lockAttempt(de, 'W');
   }
   advice writeTuple(de):before(DataElem *de){
     CC_lockAttempt(de, 'R');
   }
...
  advice insertIndex(relation,key,address): before(char *relation, ...){
    CC_insertIndex(realtion,key,address,T)
  }
...
}
```

Figure 8.5: *The structure of a locking-based concurrency control aspect*

depicted in figure 8.4. The LMC provides an initial policy for the lock administration in which all locks are granted. The initial policy of the component can be changed into a specific policy where the LMC deals with lock conflicts by weaving a concurrency control aspect.

The concurrency control aspect is shown in figure 8.5. The functions implementing advices are implemented into a separate C file, and are in fact treated as operations provided by the aspect. Hence, these are for the purpose of the dynamic exchange also listed in the aspect provided interface. The reconfiguration locations of components used in pointcut expressions are, on the other hand, declared in the required interface of the aspect. In general, the aspect that implements HP-2PL with similarity has similar structure and pointcuts as the aspect that implements ODC. Table 8.3 shows what components are crosscut and used by an aspect that implements lock-based CC. ✓ in the crosscut column means that the component is crosscut by the aspect, while ✓ in the used column implies that component operations or mechanisms are used in the implementation of advices in the aspect. A short description of why a component is crosscut and/or

Comp.	Crosscut	Used	Description
UIC	✓		The UIC is crosscut by the aspect to enable initializations when new threads start.
TMC	✓		The TMC is crosscut to make it possible to release read locks when tuples are deleted from a buffer. Additionally, metadata locking on the insert and delete database operations is performed.
IMC	✓	✓	The IMC is crosscut to perform locking of the metadata when metadata is read. The IMC is also used for restoration of the index on rollbacks.
SMC		✓	The SMC operations are used to abort and restart transaction on conflicts.
LMC		✓	The LMC is used to manage all locking and unlocking of data.
MMC	✓	✓	The MMC is crosscut to ensure locking of tuples on tuple reads, as well as to restore tuples on rollbacks, and to perform deferred deallocation.

Table 8.3: *The components that are crosscut and used by lock-based concurrency control methods*

used is also given. Overall, HP-2PL consists of 19 pointcuts and 19 advices, while two more pointcuts and advices are defined for the ODC aspect, which has in total 21 pointcuts and 21 advices.

The epsilon-based transaction model aspect augments the basic COMET transaction model so that it suits the epsilon transaction model, and has export and import limits. The augmentation of the transaction model is achieved with an inter-type advice that adds new parameters to the to the basic transaction model by adding members, the export limit exp_i and the import limit imp_i, to `DBTrans` struct.

8.4 Index Aspect Package

The index aspect package consists of one component, an alternative IMC that implements B-tree structure (IMC_B-tree), and the GUARD policy aspect. As the GUARD indexing policy requires only the basic data and transaction model, additional transaction model aspects are not needed to be defined. Moreover, as the IMC_B-tree replaces the default IMC component, connector aspects that would ensure integration of the component into the system are unnecessary.

8.4.1 Indexing Policy

Haritsa and Seshadri [71] proposed the GUARD-link indexing protocol to further strengthen the index management of a soft real-time system by taking into account transaction deadlines. We adopt this policy in COMET as a part of the index aspect package.

GUARD-link Policy

The GUARD-link indexing protocol augments the classical B-tree algorithm with the GUARD admission control. In the GUARD policy, all transactions entering the system are divided into two groups, `admit` and `deny`. The goal is to collect the largest set of transactions that can be completed before their deadlines into the `admit` group. Assigning transactions to groups is done using an admission controller that determines whether the incoming transaction is suitable for the `admit` or for the `deny` group. The variable *AdmitCapacity*, based on which the assignment of transactions into groups is performed, represents the size of the `admit` group. Each transaction τ is assigned a random integer and inserted in a list, which is sorted based on this integer. If the position pos_τ of transaction τ in the list is less than or equal to *AdmitCapacity* the transaction is assigned to the `admit` group, otherwise it is assigned to the `deny` group. EDF scheduling policy is then employed by the scheduler to schedule transactions within the `admit` group. While the transactions in the `admit` group are allowed to execute, the

```
1: aspect Guard{
2:
3:  pointcut transCreated(SMC_ScheduleRecord sr, void
4:   *args) =  execution("bool SMC_CreateNew()")
5:                          &&args(sr, arg)
6:  //GUARD admission control
7:  advice transCreated(sr, args) : around(...) {
8:      ...
9:    pos=getPosition(sr.transId);
10:  if (pos<=admitCapacity){
11:     assignToAdmitGroup(sr.transId):
12:     tjp->proceed();
13:  } else {
14:      assignToDenyGroup(sr.transId);
15:  }
16:  ...
17: };
```

Figure 8.6: *The GUARD policy aspect*

transactions in the `deny` group are denied entry to the system and are discarded when their deadlines expire.

8.4.2 COMET Index Aspect Package

As mentioned earlier, the index package consists of the IMC_B-tree component and the GUARD aspect, which are described next.

The IMC_B-tree is a component that implements indexing based on B-trees. Namely, the index structure that COMET uses by default in an IMC component described in section 8.1 is a T-tree structure [98]. As GUARD-link indexing concurrency control requires a B-link tree, it was necessary to implement this type of structure as a replacement for the T-tree. An index structure does not have characteristics of a crosscutting concern as it is a functionally coherent unit, clearly separated from the rest of the system. Thus, we decided to implement the B-link tree structure within a new version of an IMC component as a part of the aspect package. The new IMC_B-tree component has the same interfaces as the default IMC that uses T-trees. Thus, switching between the two IMCs (and, thus, indexing policies) is transparent to the rest of the COMET database. The version of the IMC needed for the database configuration can be chosen at compile time during static system reconfiguration. Furthermore, the version of the IMC can be exchanged at run-time by means of dynamic system reconfiguration.

The GUARD aspect implements the GUARD admission control. It only crosscuts one component, the SMC. The aspect also uses mechanisms of the SMC

in its advices to perform an admission check to establish whether the transaction can violate its deadlines and, thereby, needs to be placed in the `deny` group. Figure 8.6 illustrates how the GUARD aspect is implemented. As prescribed by the GUARD policy, every transaction is assigned to either the `admit` or the `deny` group as follows. If the incoming transaction passes the admission test it is assigned to the `admit` group and allowed to execute (lines 10-12). This implies that a schedule record for that transaction is created in the SMC and transaction continues execution in the normal flow of execution. If the transaction does not pass the admission test, its execution is not initiated and it is assigned to the `deny` group (lines 13-15). The transactions in the `deny` group for which deadlines have elapsed are removed from the system.

8.5 QoS Aspect Package

Applying the notion of an aspect package on COMET also resulted in the development of the COMET QoS aspect package that enables the database to be used in applications that have uncertain workloads and where requirements for data freshness are essential. The COMET QoS aspect package consists of two components, the QAC and the FCC, and the following aspects:

- QoS management policy aspects: QAC utilization policy, missed deadline monitor, missed deadline controller, scheduling strategy, data access monitor, QoS through update scheduling aspect, self-tuning regulator aspect, and adaptive regression model aspect;
- QoS transaction model aspects: utilization transaction model aspect and data differentiation aspect; and
- QoS connector aspects: QAC connector and FCC connector aspect.

8.5.1 QoS Policies

Given that we want to use the COMET database with applications that require performance guarantees, we need to employ existing QoS policies and integrate them into the database. Hence, in this section we give an overview over three instances of feedback-based QoS management we found in our case study to be especially suitable for ensuring performance guarantees in real-time database systems. First we briefly review the feedback miss ratio control (FC-M) [108], where deadline miss ratio is controlled by modifying the number of admitted transactions. This is followed by a description of the QoS sensitive approach for miss ratio and freshness guarantees (QMF) [80], used for managing QoS in real-time databases. Finally, we give a description of two adaptive QoS algorithms.

FC-M Policy

Recall that FC-M uses a control loop to control the deadline miss ratio by adjusting the utilization in the system. The deadline miss ratio,

$$m(k) = \frac{missedTransactions(k)}{admittedTransactions(k)} \tag{8.1}$$

denotes the ratio of transactions that have missed their deadlines. The performance error, $e_m(k) = m_r(k) - m(k)$, is computed to quantize the difference between the desired deadline miss ratio $m_r(k)$ and the measured deadline miss ratio $m(k)$. The change to the utilization $\delta u(k)$ is the manipulated variable. Admission control is used to carry out the change in utilization.

QMF Policy

Another way to change the requested utilization is to apply the policy used in QMF [80], where a feedback controller, similar to that of FC-M, is used to control the deadline miss ratio. The actuator in QMF manipulates the quality of data in real-time databases in combination with admission control to carry out changes in the controlled systems. If the database contains rarely requested data items, then updating them continuously is unnecessary, i.e., they can be updated on-demand. On the other hand, frequently requested data items should be updated continuously, because updating them on-demand would cause serious delays and possibly deadline overruns. When a lower utilization is requested via the deadline miss ratio controller, some of the least accessed data objects are classified as on-demand, thus, reducing the utilization. In contrast, if a greater utilization is requested then the data items that were previously updated on-demand, and have a relatively higher number of accesses, are moved from on-demand to immediate update, meaning that they are updated continuously. This way the utilization is changed according to the system performance.

Adaptive QoS Policies

The QoS management approaches presented so far in this section are using linear feedback-control assuming that real-time system is time-invariant and implying that a controller is tuned and fixed for that particular environment setting. For time-varying real-time systems it is beneficial to use adaptive QoS management that enables the controller in the feedback loop to dynamically adjust its control algorithm parameters such that the overall performance of the system is improved. Two representative adaptive QoS approaches are the self-tuning regulator and the least squares regression model [146]. In these, the control algorithm parameters are adjusted either by self-tuning or by employing least squares techniques.

Attribute	Periodic transactions	Aperiodic transactions
d_i	$d_i = p_i$	$d_i = r_{A,i}$
$u_{E,i}$	$u_{E,i} = x_{E,i}/p_i$	$u_{E,i} = x_{E,i}/r_{E,i}$
$u_{A,i}$	$u_{A,i} = x_{A,i}/p_i$	$u_{A,i} = x_{A,i}/r_{A,i}$

Table 8.4: *The utilization transaction model*

8.5.2 Data and Transaction Model

QoS algorithms like FC-M, QMF, and adaptive algorithms require distinct and more complex data and transaction models than the ones used in previously described aspect packages.

In the differentiated data model, data objects are classified into two classes, temporal and non-temporal. For temporal data we only consider base data, i.e., sensor engineering data objects that hold the view of the real-world and are updated by sensors. A base data object b_i is considered temporally inconsistent or stale if the current time is later than the timestamp of b_i followed by the absolute validity interval avi_i of b_i, i.e., $currenttime > timestamp_i + avi_i$. Both FC-M and QMF policies require a transaction model where transaction τ_i is classified as either an update or a user (query) transaction. Update transactions arrive periodically and may only write to base data objects. User transactions arrive aperiodically and may read temporal and read/write non-temporal data. In this model, denoted the utilization transaction model, each transaction has the following characteristics:

- ❏ period p_i (update transactions),
- ❏ estimated average inter-arrival time $r_{E,i}$ (user transactions),
- ❏ actual average inter-arrival time $r_{A,i}$ (user transactions),
- ❏ estimated execution time $x_{E,i}$,
- ❏ actual execution time $x_{A,i}$,
- ❏ relative deadline d_i,
- ❏ estimated utilization[3], $u_{E,i}$, and
- ❏ actual utilization, $u_{A,i}$.

Table 8.4 presents the complete utilization transaction model. Upon arrival, a transaction presents the estimated average utilization $u_{E,i}$ and the relative deadline d_i to the system. The actual utilization of the transaction $u_{A,i}$ is not known in advance due to variations in the execution time.

[3]Utilization is also referred to as load.

```
1: aspect QAC_composition{
2:   // Insert QAC between UIC and SMC.
3:   advice call("bool SMC_CreateNew(...)") : around() {
4:     if (QAC_admit(*(ScheduleRecord *)tjp->arg(0)))
5:       tjp->proceed();
6:     else
7:       *(bool *)tjp->result() = false;
8:   }
9: };
```

Figure 8.7: *The QAC connector aspect*

8.5.3 COMET QoS Aspect Package

The current COMET QoS aspect package provides components and aspects that implement the FC-M, QMF, self-tuning, and least squares regression QoS policies.

The QAC decides, based on an admission policy, whether to allow new transactions into the system. Operations provided by the QAC are `QAC_Admit()`, which performs the admission test, and `QAC_Adjust()`, which adjusts the number of transactions that can be admitted. The default admission policy is allowing all transactions to be admitted to the system. This admission policy of the QAC can be changed by weaving specific QoS actuator policy aspects.

The FCC computes the input to the admission policy of the QAC at regular intervals. By default, an input of zero is generated, but by using QoS controlling policy aspects different feedback policies can be used. The FCC provides the operation `FCC_Init()` that initializes the FCC component. FCC calls `QAC_Adjust()` after computing the manipulated variable.

The utilization transaction model aspect augments the basic COMET transaction model so that it suits the utilization transaction model described in section 8.5.2. This is done using inter-type declaration that adds new parameters to the basic model, e.g., estimated utilization $u_{E,i}$ and estimated execution time $x_{E,i}$.

The QAC connector aspect enables QAC to intercept requests to create new transactions that are posed by the UIC to the SMC. This is done via an advice of type around which is executed when the SMC operation `SMC_CreateNew()` is called (lines 3-8 in figure 8.7). Since this operation of the SMC is in charge of registering a new transaction to the system, the advice ensures that, before the transaction is actually registered, an admission test is made by the QAC (line 4). If the transaction can be admitted the transaction registration is resumed; the `proceed()` in line 5 enables the normal continuation of the join point `SMC_CreateNew()`. If the transaction is to be aborted, then the around advice

```
1: aspect QAC_utilization_policy{
2:   // Add a utilization reference to the system
3:   advice "UIC_SystemParameters" : float utilizationRef;
4:   // Changes the policy of the QAC to the utilization
5:  advice execution("% QAC_admit(...)") : around() {
6:    // Get the current estimated total utilization
7:    totalUtilization = GetTotalEstimatedUtilization();
8:   // Check if the current transaction can be admitted
9:   if (utilizationTarget > totalUtilization + sr->utilization)
10:    { (*(bool *)tjp->result()) = true; }
11:      else
12:    { (*(bool *)tjp->result()) = false; }
13:  }
```

Figure 8.8: *The QAC utilization policy aspect*

replaces the execution of the transaction registration in full and, thus, ensures that the transaction is rejected from the system (line 7).

The QAC utilization policy aspect shown in figure 8.8 replaces, via the around advice (lines 5-13), the default admission policy of QAC with an admission policy based on utilization (lines 9-12). The current transaction examined for admission in the system is denoted `ct` in figure 8.8.

The FCC connector aspect facilitates the composition of FCC with all other components in the system by ensuring that the FCC is properly initialized during the system initialization.

The missed deadline monitor aspect modifies the SMC to keep track of transactions that have missed their deadlines, $missedTransactions$, and transactions that have been admitted to the system, $admittedTransactions$. This is done by having a number of advices of different types intercepting SMC operations that handle completion and abortion of transactions (see figure 8.9). For example, the advice of type after, which intercepts the call to `SMC_CreateNew()`, increments the number of admitted transactions once transactions have been admitted to the system (lines 2-4). Similarly, the advice in lines 5-12 checks if the number of transactions with missed deadlines should be incremented before the transaction has completed, i.e., before invoking the SMC operation `SMC_Completed()`.

The missed deadline controller aspect is illustrated in figure 8.10. This aspect is an instance of the feedback control policy aspect and it modifies the SMC to keep track of the deadline miss ratio, using equation 8.1. The aspect does so with two advices of type after. One is executed after the initialization of the UIC (lines 3-11), thus, ensuring that the appropriate variables needed for FCC policy are initialized. The other advice modifies the output of the FCC to

```
1: aspect missed_deadline_monitor {
2:   advice call("% SMC_CreateNew(...)") : after(){
3:       if (*(bool *)tjp->result()) { admittedTransactions++; }
4:   }
5:   advice call("% SMC_Completed(...)") : before(){
6:      ScheduleRecord *sr = (ScheduleRecord *)tjp->arg(0);
7:       _getTime(&currentTime);
8:       node = findNode(ActiveQueue_root, sr->id);
9:       if ((node != NULL) && (_compareTimes(&currentTime,
10:                                &(node->data->deadline))))
11:         { missedTransactions++; }
12:   }
13:   advice call("% SMC_Aborted(...)") : before(){…
14:      admittedTransactions--;}
15:   advice call("% SMC_RejectLeastValuableTransaction(...)") :  after(){
16:      if (*(bool *)tjp->result()) { admittedTransactions--;}
17:   }
18:   advice call("% getTimeToDeadline(...)") && within("%
19:      getNextToExecute(...)") : after() {… missedTransactions++;}
20: }
```

Figure 8.9: *The missed deadline monitor aspect*

suit the chosen feedback control policy, which is deadline miss ratio in this case (lines 13-17).

The data differentiation aspect enriches the data model of the basic COMET configuration to differentiate between base data and derived data. Differentiation is done by assigning avi_i and $timestamp_i$ attributes to data items manipulated by the transaction. Inserted data items containing fields for avi_i and $timestamp_i$ are assumed to be base data. Whenever these data values are inserted or modified, $timestamp_i$ is set to the current time.

The scheduling strategy aspect modifies the scheduling strategy and the data model of COMET to support two distinct update strategies for base data: immediate and on-demand [139]. To accommodate these strategies the aspect adds an update wait queue in the SMC (advice of type after in lines 2-6 in figure 8.11). The name of the update strategy is stored in a field in the relation (see line 15 for an example). Hence, an inserted data that contains a field for update strategy as well as fields for avi_i and $timestamp_i$ is handled by this aspect. Note that, when a transaction reads a base data item, the freshness of the item is examined in the advice that is executed after TMC mechanism `readData()` is called, i.e., after the data is read from the memory (lines 18-22 in figure 8.11). If the base data item is stale and the updating strategy is set to on-demand, the transaction is rolled back and moved to the update wait queue. If the updating strategy is set to immediate, the transaction is rolled back and restarted. Updates of base data items set to immediate are always allowed, while updates of base data items

```
1: aspect missed_deadline_control{
2:  // Initialize the new variales need for control
3:  advice call("% UIC_init(...)") : after() {
4:   UIC_SystemParameters *sp =
5:       (UIC_SystemParameters *)tjp->arg(0);
6:   if (*(bool *)tjp->result()) {
7:    missRatioReference = sp->missRatioReference;
8:    missRatioControlVariableP =
9:              sp->missRatioControlVariableP;
10:    ...
11:  }
12:  // Modify the calculation of the control output
13:  advice call("% calculateOutput(...)") : after(){
14:   missRatioOutputHm =
15:      calculateMissRatioOutput(RSMC_GetDeadlineMissRatio());
16:   *((float *)tjp->result()) = missRatioOutputHm;
17:  }
18: }
```

Figure 8.10: *The missed deadline control aspect*

```
1: aspect scheduling_policy{
2:   advice call("% SMC_constructor(...)") : after(){
3:     // Initialize the update-wait queue
4:    UpdateWaitQueue_root = SMC_createNode(...);
5:   ...
6:   }
7:   advice call("% insert(...)") : before(){
8:      // Set update type to immediate upon
9:       //if the data is base data.
10:     if (isUpdateTypeData(buffer)){
11:     while (updateTypeNr > counter){
12:          counter++;
13:          treePtr = treePtr->right;
14:     }
15:    strcpy(treePtr->left->Data.operandID, "IMMEDIATE");
16:   }
17:  }
18:  advice call("% ReadData(...)") : after(){
19:    // If it is a base data relation...
20:    // If it is not an update or insert transaction...
21:    // If it is invalid...
22:  }
23: …..
24: }
```

Figure 8.11: *The scheduling strategy aspect*

set to on-demand are rejected unless these data items have been requested by a transaction in the update wait queue. If so, the requesting transaction is moved to the ready queue and the update executes normally.

The data access monitor aspect modifies the TMC to keep track of how often base data items are accessed. Remember that in QMF data base items are updated on-demand or immediate based on how often they are accessed.

The QoS through update scheduling aspect uses the data differentiation aspect, scheduling strategy aspect, and the data access monitor aspect to modify the QAC such that the actuator policy in QMF is used. Hence, when applying the QoS through update scheduling aspect, changes to quality of data in combination with admission policy is used to enforce utilization changes based on the control signal from FCC.

The self-tuning regulator aspect and adaptive regression model aspect implement the adaptive policies in controllers so that the control algorithm parameters can be adjusted dynamically and, thereby, enable COMET to be perceived as time-varying system. Both aspects modify the policies of FCC and QAC, and enhance the scheduling policy of the system in the process, hence, crosscutting the SMC.

Chapter 9

COMET Configuration and Analysis

Here we discuss the possible COMET configurations that can be obtained using aspects and components described in the previous chapter. Furthermore, we discuss how the ACCORD development tool environment can be used to support configuration and analysis of COMET. Finally, we present performance evaluations showing that both static and dynamic system reconfiguration yields a new and functionally correct COMET configuration that exhibits the desired behavior.

9.1 COMET Configurations

A significant number of COMET configurations fulfilling distinct requirements can be made from existing COMET components and aspects. In this section we describe a subset of possible COMET configurations, their relationship, and constituents.

The basic configuration consists of five COMET components: the UIC, TMC, SMC, IMC, and MMC (see table 9.1). The configuration uses the basic transaction model, and it is especially suitable for small resource-constrained embedded vehicular systems [123]. All remaining database configurations are built upon the basic COMET configuration by adding appropriate aspects and components from aspect packages.

The high priority two-phase locking concurrency control configuration (COMET HP-2PL) includes all basic components, as well as the LMC and the HP-2PL with similarity aspect from the concurrency control aspect package. This

configuration is suitable for more complex real-time systems where transactions can execute concurrently.

The optimistic divergence control configuration (COMET ODC) is built on top of the basic COMET configuration by adding the LMC, the ODC aspect, and the epsilon-based transaction model aspect, as indicated in table 9.1. This configuration, therefore, enables database system to resolve conflicts based on the ODC policy. As such, COMET ODC is suitable for soft real-time applications having concurrent transactions.

The GUARD-link configuration (COMET GUARD) requires all components from the basic COMET configuration, except the IMC. It also requires the constituents of the CC aspect package. Furthermore, from the index aspect package the IMC_B-tree and the GUARD link policy aspect are required. Two distinct COMET GUARD configurations can be made, depending on the choice of the CC policy from the CC aspect package. Namely, if the HP-2PL with similarity aspect is used, COMET GUARD with HP-2PL is created. If ODC is used, the configuration of COMET GUARD with ODC is made (see table 9.1). COMET GUARD configurations are desirable in soft real-time applications where indexing structure should further be optimized in terms of meeting transaction deadlines.

A number of COMET QoS configurations can be made using the COMET QoS aspect package together with different combinations of aspects and components from the CC aspect package and the index aspect package. For simplicity here we discuss only COMET QoS configurations that are distinct with respect to QoS-related aspect and components. Any of the concurrent or GUARD COMET configurations from table 9.1 can be used as a foundation for adding aspects and components from the COMET QoS aspect package, and creating a COMET QoS configuration. Hence, we discuss in detail five distinct COMET QoS configurations that provide admission control, FC-M, QMF, self-tuning, and least squares regression QoS. However, note that depending on the chosen aspects and components from the COMET QoS aspect package, the number of possible configurations in COMET QoS family is higher (see figure 9.1). Table 9.2 illustrates the elements of the QoS aspect package used in the five representative COMET QoS configurations.

The admission control QoS configuration includes one component from the QoS aspect package, the QAC. The configuration also requires two aspects, the QAC connector aspect and the utilization transaction model aspect. The QAC connector aspect adds the QAC to the existing controlled system, while the utilization transaction model aspect extends the transaction model (see table 9.2). The admission control configuration is simple as it only provides facilities for admission control.

		COMET configurations				
			Concurrent COMET		COMET GUARD	
		Basic COMET	COMET HP-2PL	COMET ODC	with HP-2PL	with ODC
Basic COMET components	UIC	X	X	X	X	X
	TMC	X	X	X	X	X
	SMC	X	X	X	X	X
	IMC	X	X	X		
	MMC	X	X	X	X	X
Concurrency control aspect package						
policy aspects	HP-2PL		X		X	
	ODC			X		X
transaction model aspects	Epsilon-based			X		X
connector aspects						
components	LMC		X	X	X	X
Index aspect package						
policy aspects	GUARD link				X	X
transaction model aspects						
connector aspects						
components	IMC_B-tree				X	X

LEGEND

X in the table means that an aspect/component is part of a configuration

UIC - user interface component
TMC - transaction manager component
SMC - scheduling manager component
IMC - index manager component
MMC - memory manager component
LMC - locking manager component

COMET HP-2PL - The high-priority 2 phase locking configuration
COMET ODC- The optimistic divergence control configuration

Table 9.1: *Relationship between different parts of the concurrency control and the index aspect package and various COMET configurations*

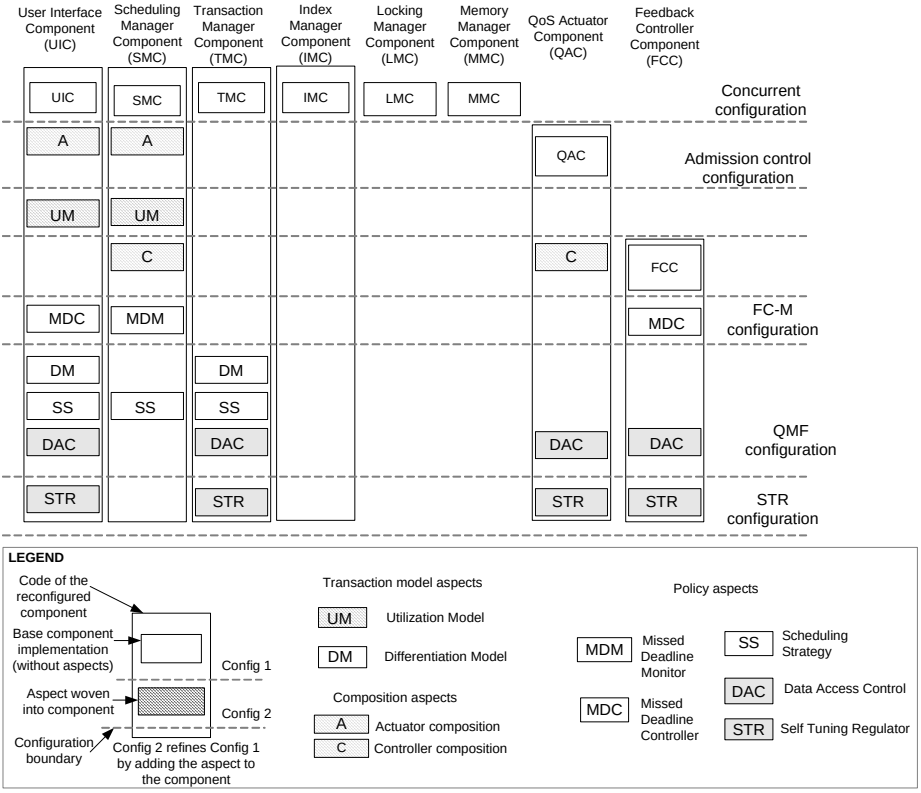


Figure 9.1: *Creating a family of real-time systems from the COMET QoS aspect package*

QoS aspect package		COMET configurations				
		Admission control	COMET FC-M	COMET QMF	COMET STR	COMET RM
policy aspects	Actuator utilization policy	X	X	X	X	X
	Missed deadline monitor		X	X	X	X
	Missed deadline controller		X	X	X	X
	Scheduling strategy			X		
	Data access monitor			X		
	QoS through update scheduling			X		
	Self-tuning regulator				X	
	Adaptive regression model					X
transaction model aspects	Utilization transaction model	X	X	X	X	X
	Data differentiation			X		
connector aspects	Actuator connector	X	X	X	X	X
	Controller connector		X	X	X	X
components	QAC	X	X	X	X	X
	FCC		X	X	X	X

LEGEND

X in the table means that an aspect (or a component) is part of a configuration

QAC - QoS Actuator Component

FCC - Feedback Controller Component

COMET FC-M - The miss ratio feedback configuration

COMET STR - The self-tuning regulator configuration

COMET QMF - The update scheduling configuration

COMET RM - The regression model configuration

Table 9.2: *Relationship between different parts of the QoS package and various COMET QoS configurations*

The miss ratio feedback configuration (COMET FC-M) provides QoS guarantees based on the FC-M policy. The configuration includes the QAC and FCC components and their corresponding connector aspects, the utilization transaction model aspect, the missed deadline monitor aspect, and the missed deadline controller aspect (see table 9.2). These aspects modify the policy of the SMC and FCC to ensure that QoS with respect to controlling the number of deadline misses is satisfied in soft real-time systems where workload characteristics are generally unpredictable.

The update scheduling configuration (COMET QMF) provides the QoS guarantees based on QMF policy. Here the data differentiation aspect and scheduling strategy aspect are used to enrich the transaction model. Moreover, the data access monitor aspect is required to ensure the metric used in QMF. Also, the QoS through update scheduling aspect is used to further adjust the policy of QAC to suit the QMF algorithm. COMET QMF is especially suitable for soft real-time applications where maintaining a certain level of data quality is of interest.

The self-tuning regulator configuration (COMET STR) provides adaptive QoS control where the control algorithm parameters are adjusted by using the self-tuning regulator. This aspect is added to the aspects and component constituting the COMET FC-M configuration to ensure the adaptability of the control already provided by the COMET FC-M configuration. The COMET STR configuration can be used for real-time systems that cannot be modeled as time-invariant.

The regression model configuration (COMET RM) provides adaptive QoS control where the control algorithm parameters are adjusted by using the least square technique and the regression model. This aspect also requires all the aspects needed for the FC-M configuration to ensure adaptability of QoS management.

9.2 Static and Dynamic COMET Configuration

Each of the COMET configurations discussed in the previous section can be configured statically and dynamically. Regardless of the way COMET is configured, tools in the ACCORD development environment can be used for aiding the designer in configuring. In this section, therefore, we first illustrate the way COMET is configured using ACCORD tools. Then we discuss how dynamic reconfiguration of the system by on-line component exchange is performed.

9.2.1 Configuring COMET using ACCORD Tools

The following example illustrates how COMET can be tailored for a particular real-time application using existing aspects and components. We focus here on using ACCORD-ME for configuring COMET for a specific ECU in vehicles, which is a new generation of ECUs used for controlling the engine in a vehicle, and it has the following set of data management requirements.

R1: The application performs computations using data obtained from sensors and forwards the computation results directly to actuators; recall that sensor data items are typically referred to as base data items.

R2: Sensor data should reflect the state of the controlled environment implying that transactions used for updating data items should be associated with real-time properties, such as periods and deadlines.

R3: Values of data should be updated only if they are stale[1] to save computation time of the CPU.

R4: Multiple tasks can execute concurrently in an ECU, and operate on the same data. Hence, consistency needs to be enforced.

R5: The tasks in the ECU should be scheduled according to priorities.

R6: The memory footprint of the system should be within the *memBound*, which can be obtained from the ECU specifications.

When configuring COMET the ECU we start by specifying the requirements using the configurator tool in ACCORD-ME. Given that we know the system requirements, then the requirement-based option in the configurator can be used to guide the system composition. Now, based on the requirements R1-R5 we can choose options in the requirement-based form (shown in figure 9.2) as follows. The configuration of the COMET database suitable for the ECU contains base data items. Since tasks using the base data are not storing intermediate results, there is no need for storing derived data (R1). Furthermore, the database should provide mechanisms for dealing with concurrency such that conflicts on data items are resolved (R4) and data items that are stale are updated (R3). This can be achieved using HP-2PL with similarity [95]. The transaction model should be chosen such that transactions are associated with periods and deadlines (R2). We choose the RMS scheduling policy to enforce priority-based scheduling of tasks (R5). Performance guarantees in terms of levels of quality of service are not required.

When these decisions are submitted, the configurator loads candidate components and aspects into ACCORD-ME. For more efficient composition process one can use help in terms of composition rules provided in the description tab of components and aspects in the ACCORD-ME editing window.

[1] A data item is stale if its value does not reflect the current state of the environment.

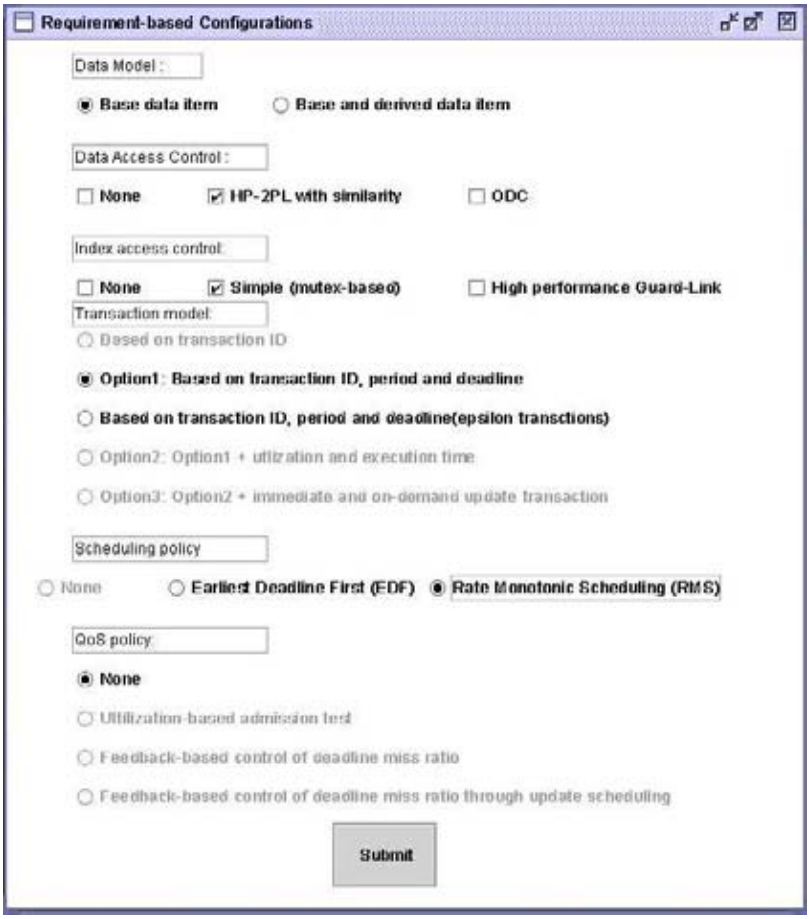


Figure 9.2: *Requirement-based configuration of the real-time database system*

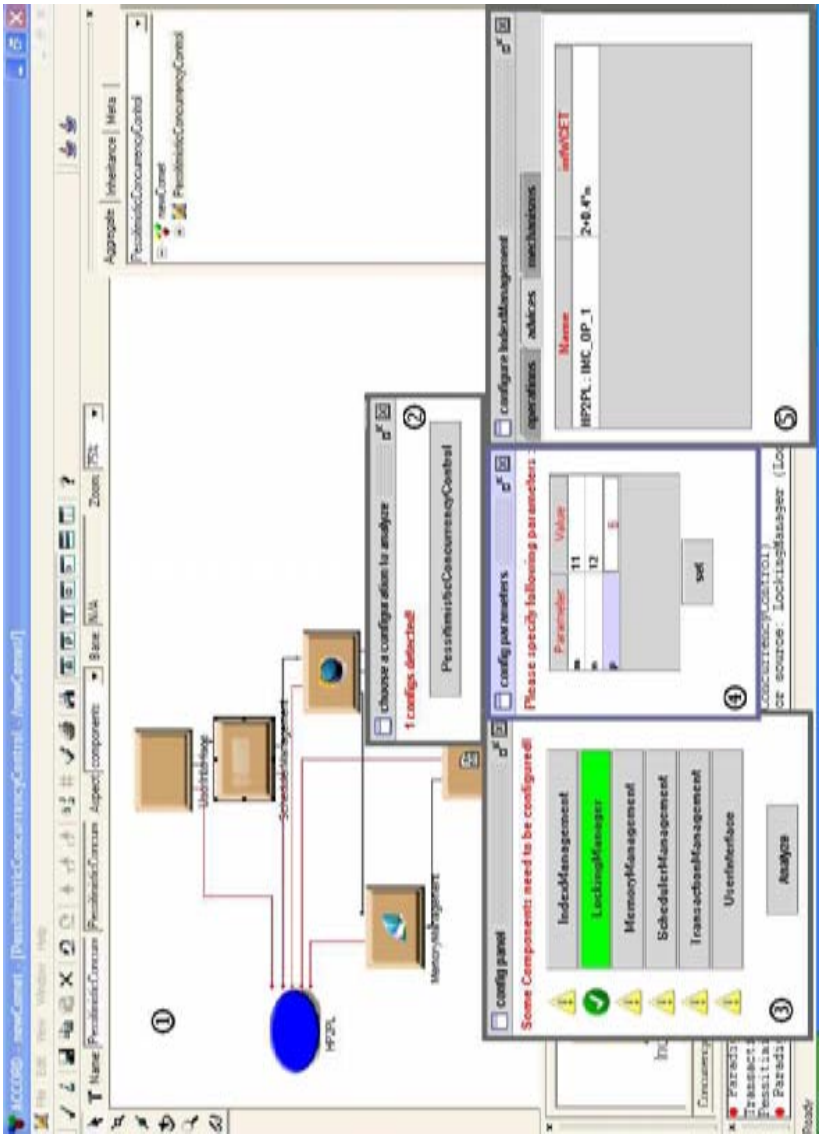


Figure 9.3: *The snapshot of ACCORD-ME when doing analysis on a real-time system configuration*

A system configuration satisfying functional requirements R1-R5 is shown in the upper part of figure 9.3 (denoted as step 1 in the figure). Recall that in ACCORD-ME ovals are used as the graphical representation of aspects, while squares represent components. When the composition of the system is made, it should be analyzed to determine the memory needs of the configuration and contrast these to the available memory in the ECU. Hence, when the configuration part of the system development is finished then the obtained configuration can be analyzed using the M&W analyzer tool. Figure 9.3 is the snapshot of the analysis process. When the M&W analyzer is invoked, it detects the configuration(s) one might have made in ACCORD-ME and prompts for the choice of a configuration. In our example, we created only one configuration and denoted it PessimisticConcurrecyControl. This configuration is detected by the M&W analyzer, as illustrated by step 2 in figure 9.3. After the configuration is chosen, the appropriate files describing run-time aspects of components and aspects are loaded for analysis. Since run-time properties of aspects and components are described in terms of symbolic expressions with parameters, the values of these parameters are instantiated during analysis, and the list of components that require instantiation of parameters is displayed during analysis (step 3 in the figure). One can also make an inspection of the symbolic expressions and input the values of parameters in the expressions, as depicted by step 4. Note that advices that modify components are included in the component run-time description as shown in step 5. Once the values of parameters are set for this particular ECU, the tool outputs the resulting WCET and/or memory consumption values which can be compared with the values given in the memory requirement (R6).

If the obtained configuration satisfies the requirements of the target ECU, the next step is to compile the system and deploy it into the run-time environment, which is done using the configuration compiler. As mentioned previously, the configuration compiler also provides documentation of the composed configuration of COMET.

9.2.2 Dynamic Reconfiguration

The same procedure as described in the previous section can be employed for obtaining a dynamically reconfigurable COMET configuration. The difference is in compiling and deploying the configuration as the middleware layer needs to be included into the configuration to ensure that component exchange can occur at run-time. Moreover, to ensure that performance of the database is maintained even when the exchange occurs, the system should be made QoS adaptive.

Figure 9.4 depicts an instance of dynamic COMET configuration that is QoS adaptive. The depicted configuration is COMET HP-2PL described earlier in section 9.1, augmented with the middleware layer, an instantiation of the middleware layer described in section 4.6. Hence, the configuration in the figure consists of all basic COMET components, as well as the HP-2PL with similarity concurrency control aspect, and utilization-based transaction model. The utilization-based

Attribute	Description
AE_i	Average execution time of transaction τ_i
AE	Average execution time of all transactions
AI_i	Average inter-arrival time of transaction τ_i
AI	Average inter-arrival time of all transactions

Table 9.3: *Measured attributes of transactions*

transaction model is further enriched with the parameters presented in table 9.3 that are derived by aspects in charge of the QoS adaptability, i.e., they are measured at run-time. The application running the database is also considered to be a component plugged into the middleware layer.

As previously discussed, components constituting a configuration of a real-time system are typically mapped to a number of tasks in the run-time environment [53, 150]. Hence, in COMET a subset of COMET components is mapped to one or several transactions. Since a transaction is executed by invoking operations provided in COMET components, the variations of the execution time of components[2] directly influence the execution time of a transaction.

Reconfiguration of the COMET is initiated from the application via the `exchange()` operation provided by the middleware layer. The application using the database regularly performs self-inspection where it checks if any updates of components and aspects are available. If a new version of a component is available, the reconfiguration takes place. Correct transaction execution during reconfiguration is ensured by emptying the active queue maintained by the SMC. Completion of transactions that are executing in the system at the time reconfiguration is requested is a precondition for successful preservation of correct states of the components after reconfiguration. After the queue has been emptied the actual reconfiguration of a component takes place, i.e., re-pointing in the jump table. In section 9.3.2 we experimentally show that the overhead in terms of the time it takes to perform the reconfiguration is negligible.

QoS guarantees are satisfied in COMET by having the QAC and FCC components and the aspects constituting the FC-M policy (see figure 9.4). Recall that the FC-M aspects are applied to the QAC and FCC to control deadline miss ratio. Hence, the FC-M aspects constitute the deadline miss ratio feedback loop, consisting of measuring the deadline miss ratio, forming the performance error $e_m(k) = m_r(k) - m(k)$, and computing the manipulated variable $\delta_l(k)$.

[2]Here we refer to the execution time of an operation provided by a component, and used by a transaction, as the execution time of the component.

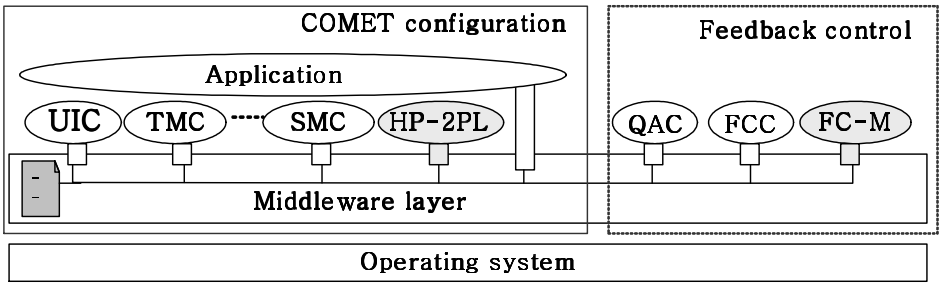


Figure 9.4: *Dynamic COMET configuration*

9.3 Performance Evaluation

In this section we present experimental evaluations of the COMET platform. The goal of the evaluations is twofold. In the first set of experiments, we show that reconfiguring COMET statically by adding aspects from an aspect package to an existing COMET configuration yields a new and functionally correct COMET configuration. In the second set of experiments, we show that the dynamically reconfigurable COMET is suitable for embedded environments as the middleware layer does not introduce significant performance overhead. Moreover, we show that QoS guarantees are provided when a system undergoes dynamic reconfiguration.

9.3.1 Static System Reconfiguration

The goal of these experiments is to show that adding aspects and components from an aspect package into an existing configuration results in a new system configuration exhibiting the expected behavior. To that end, we chose the COMET QoS aspect package as a representative of aspect packages, and concurrent COMET HP-2PL configuration as a configuration on top of which aspects and components from the QoS aspect package are added. In this context, the goal of the experiments is to show that the QoS management in COMET performs as expected and, thereby, show that, when adding the QoS aspect package, we achieve required performance guarantees. It should be noted that we have performed several other experiments to show that we achieve the desired behavior under different COMET QoS configurations (see [32]). Extensive performance evaluations have also shown that the concurrent COMET configurations, created by adding aspects and components from the CC aspect package on top of the basic COMET, exhibit expected functional behavior with respect to how data access conflicts are detected and handled [58].

For doing the experiment we have chosen the following experiment setup. The database consists of eight relations, each containing ten tuples. Note that this

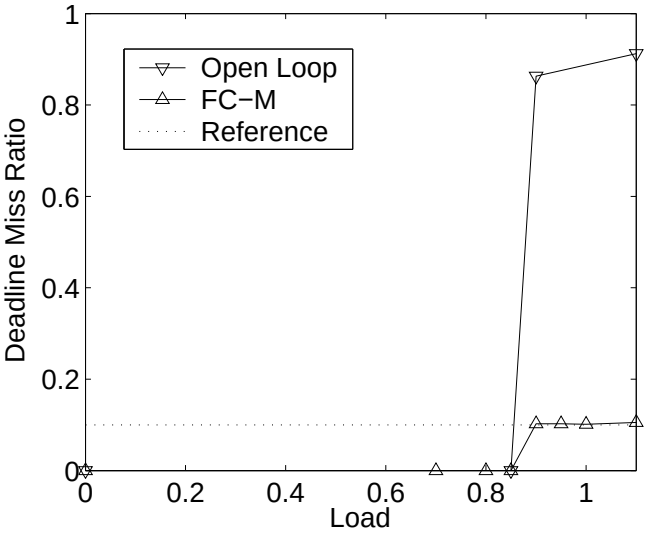


Figure 9.5: *Deadline miss ratio as a function of load*

relatively low data volume is acceptable for the experiments as the experimental results do not depend on the data volume but the load, i.e., number of transactions, that is imposed on the database. To that end, we ensure that constant streams of transaction requests are used in the experiments. Update transactions arrive periodically, whereas user transactions arrive aperiodically. To vary the load on the system, the inter-arrival times between transactions are altered. The deadline miss ratio reference, i.e., the desired deadline miss ratio, is set to 0.1.

The experiment is applied to the COMET FC-M configuration, where the load applied on the database varies. This way we can determine the behavior of the system under an increasing load. We use the behavior of the concurrent COMET configuration without the QoS aspect package as a baseline. For all the experiment data, we have taken the average of 10 runs, each one consisting of 1500 transactions. We have derived 95% confidence intervals based on the samples obtained from each run and used the t-distribution [54]. It has been shown that the 10 runs are sufficient as we have obtained tight confidence intervals (shown later in this section). Figure 9.5 shows the deadline miss ratio of concurrent COMET and COMET with the FC-M configuration. The dotted line indicates the reference deadline miss ratio, i.e., the desired QoS. We vary the applied load from 0% to 130%. This interval of load captures the transition from an underloaded system to an overloaded system. Hence, at 130% applied load we capture the case when the system is overloaded.

Starting with concurrent COMET, the deadline miss ratio starts increasing at

approximately 0.85 load. However, the deadline miss ratio increases more than the desired deadline miss ratio and, hence, concurrent COMET does not provide any QoS guarantees. Studying the FC-M configuration we see that the deadline miss ratio for loads $0.90, \ldots, 1.30$ are $0.1025 \pm 0.0070, 0.1023 \pm 0.0027, 0.1012 \pm 0.0025, 0.1052 \pm 0.0030$, and 0.1082 ± 0.0011. In contrast with concurrent COMET, the added FC-M configuration manages to keep the deadline miss ratio at the reference, even during high loads. This is in line with earlier observations where feedback control has shown to be very effective in guaranteeing QoS [15, 12, 13]. Hence, the results of our experiment show that the COMET FM-C configuration is able to provide QoS guarantees under varying load.

9.3.2 Dynamic System Reconfiguration

We perform experiments on the dynamically reconfigurable COMET HP-2PL database configuration shown in figure 9.4. Before each experiment run, ten relations are created in the database, each relation consisting of ten tuples. To create a load during the experiment runs, transactions are created and submitted to the database with an average inter-arrival time AI that is uniformly distributed. The transaction deadlines are uniformly distributed in the interval $[3 \cdot AE\ ms, 17 \cdot AE\ ms]$, where AE denotes the average execution time of the transactions. The experiments are run on a Sun Blade 1500 and Solaris 9 [157].

Reconfiguration Overhead

In the following experiments we want to quantify the effect of enabling dynamic system reconfiguration on the system performance. Our focus is primarily on quantifying the overhead in transaction execution times. Note however that there also exists a performance overhead with respect to memory, since, e.g., exchanging a component requires loading of a new component into the memory while the old component still exists in the system. Hence, for reconfiguration to take place, the system has to have an amount of free memory corresponding to the memory of the largest component in the system.

Table 9.4 gives the impact of dynamic reconfiguration on the transaction execution times. We measure the average execution time of a transaction in both statically and dynamically reconfigurable COMET HP-2PL configurations. Lower bound (LB) and upper bound (UB) of a 95% confidence interval are also presented in the table. As can be seen, the differences in execution times are statistically negligible.

Tables 9.5 and 9.6 give the results of performance measurements where we evaluate the mechanisms for dynamic reconfiguration by measuring the time it takes to carry out the reconfiguration in the system. The magnitude of the component internal states increases as the number of transactions increases, resulting in an increase in the execution time of the `import` and `export` operations. Results in table 9.5 show the time it takes to do the reconfiguration in the simple case

	original ACCORD	**dynamic ACCORD**
AE_i $[\mu s]$	59117	59054
LB $[\mu s]$	58899	58847
UB $[\mu s]$	59336	59262

Table 9.4: *Execution times for transactions*

Comp.	**Task**	**Time** $[\mu$**s**$]$
MMC	Component exchange	7959
TMC	Component exchange	7803
SMC	Component exchange	8034
MMC and TMC	Component exchange	10875

Table 9.5: *Reconfiguration times with one transactions running*

when elaborate component states do not have to be exported and imported due to only one transaction running. Comparing the times needed for reconfiguration of every component (table 9.5) with the execution time of transactions (table 9.4), it is clear that that the time it takes to reconfigure a component is substantially lower than the execution time of a transaction, re-confirming that reconfiguration does not induce significant overhead in system performance.

Table 9.6 shows the time it takes to perform the reconfiguration when the load in the system is increased 300%, i.e., when we have several transactions running. In the heavily loaded system, reconfiguration of a component must also include the time it takes to complete transactions by emptying the transaction queue, which increases the total reconfiguration time of one component. If several components are exchanged, then emptying the queue is done only once before the components are exchanged. Hence, the action constituting the great portion of the component exchange time, i.e., emptying the queue, is executed only once followed by the time it takes to actually exchange the components (given in table 9.5). Consequently, even with many transactions in the system, the reconfiguration mechanism takes a bounded amount of time, equivalent to the execution time of one transaction.

From the performed experiments, we can conclude the following. Developing and deploying a system configuration to be dynamically reconfigurable as prescribed by ACCORD does not introduce a significant overhead in the system performance. Observe that reconfiguration in the presented experiments is done at arbitrary points in time, exchanging arbitrary number of components, indicating that reconfiguration can be done at any point in time and that the system does not posses a priori knowledge of components that are going to be exchanged.

Effects of Reconfiguration on System Performance

In the following experiments we examine how the deadline miss ratio is affected when replacing components. Specifically, the experiments show how QoS is varied

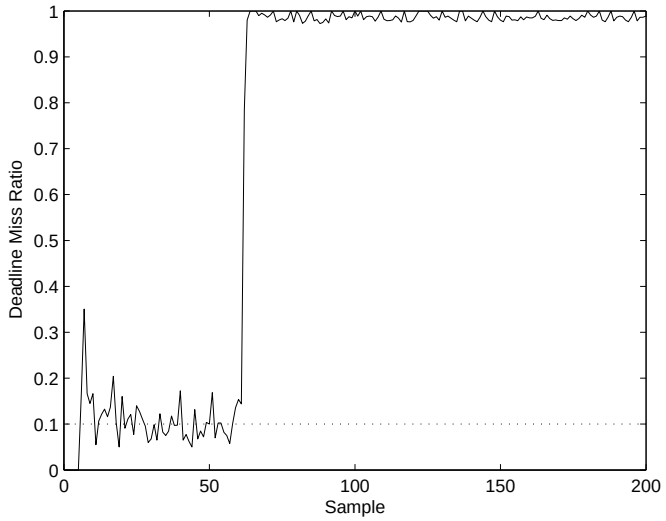
Comp.	Task	Time [μs]
MMC	Empty queue	57663
	Component exchange	8037
	Total exchange time	**65700**
SMC	Empty queue	64438
	Component exchange	9754
	Total exchange time	**74192**

Table 9.6: *Reconfiguration times in COMET with a 300% load*

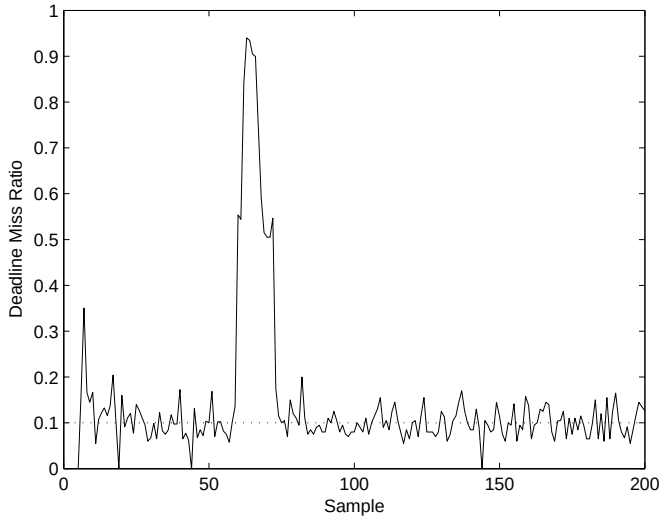
during transient state. We consider two versions of the TMC, namely, TMC^1 and TMC^2, where TMC^2 is a later version of a TMC component that has a longer execution time than TMC^1. The execution time of TMC^2 is twice the execution time of TMC^1. We chose the TMC for reconfiguration in experiments since it is the largest component in COMET and has the greatest impact on the transaction execution time in the system. Additionally, we chose to double the execution time of one component to simulate an extreme case condition where several components with great variations in the execution time are replaced. If our approach performs well under extreme cases then it should also perform well under normal circumstances where the increase in the execution time is less. A transaction running in the COMET configuration with TMC^1 has the execution time AE of $400ms$[3]. When the TMC^1 is replaced with TMC^2, transactions in the resulting COMET configuration with TMC^2 have the average execution time of $800ms$. Hence, the component exchange resulted in significant increase of the execution time of transactions in the system.

At the 60th sample TMC^1 is replaced with TMC^2, hence, introducing a longer execution time. The results of the transient behavior of a single experiment run can be seen in figures 9.6(a) and 9.6(b). The deadline miss ratio increases when the component is replaced and the system becomes over-utilized. Without QoS adaptation the system cannot adjust to changes in the execution time and continues to be over-utilized, resulting in that the deadline miss ratio remains at approximately 98%. Although the execution time of the transactions is doubled, representing an extreme case condition, our approach adjusts to the reconfiguration, since the deadline miss ratio drops to the miss ratio reference value within a bounded settling time. This is done by utilizing the feedback information and reducing the admitted load, i.e., rejecting more transactions. Also, as we can see the worst-case QoS during reconfiguration (the overshoot in figure 9.6(b)) is bounded. Since we are simulating an extreme case of reconfiguration overload, the allowed worst-case QoS under reconfiguration is, consequently, relatively high. In normal reconfiguration scenarios, where one or several components with relatively

[3] AE of $400ms$ for a transaction is obtained in a separate set of experiments where artificial delays are introduced in transaction execution and it, therefore, differs from AE given in table 9.4.



(a) No QoS Adaptation



(b) QoS Adaptation

Figure 9.6: *Deadline miss ratio when components are replaced*

small variations in execution times are reconfigured, the allowed worst-case QoS will be significantly smaller.

We have shown that in the face of a reconfiguration we are able to adjust the deadline miss ratio, representing QoS, such that the worst-case system performance is bounded and that deadline miss ratio converges toward the desired reference in a timely manner. Hence, having dynamic reconfiguration as prescribed by ACCORD has shown to be very effective in satisfying performance objectives in a reconfigurable real-time system.

9.4 Experience Report

This section contains observations we made with respect to usage of a combination of aspects and components for ensuring configurability and reusability of real-time software.

Lesson 1: *Both aspects and components are needed to ensure configurability.* We already discussed in section 4.4 that many real-time policies have similar infrastructure but they provide distinct policies, concluding that this is reason why an aspect package has both components (providing infrastructure) and aspects (providing, e.g., QoS policies). Here, again on the example of QoS management, we would like to reaffirm that without both aspects and components high reconfigurability and reusability would be difficult to achieve. Namely, if all QoS management approaches where implemented only using aspects, each aspect would have to contain the necessary functionality of the infrastructure for QoS management. This is not conducive to the development of families of real-time systems with distinct needs on the QoS configuration, since we are not able to add aspects to the system incrementally because each aspect would contain redundant parts of the infrastructure. Furthermore, the footprint of the system would be increased with additional code, which is not preferable for resource-constrained real-time systems. Another solution would be to implement one basic aspect that would contain the functionality of an infrastructure and possibly a QoS policy. However, this option is not favorable as it implies dependencies among aspects that could lead to dependency problems in the implementation phase. This in turn could induce decreased configurability (as fewer aspects could be combined). Having components that provide functionality used by aspects decreases dependency issues and the overall memory footprint for the system, and increases reusability of aspects and configurability of the overall system.

Lesson 2: *Explicitly declared reconfiguration locations in component interfaces lead to an efficient and analyzable product-line architecture.* We observed that for development of a variety of system configurations using the same set of components in combination with various aspects, it is beneficial to have explicitly defined

places in the architecture where extensions can be made, i.e., aspects woven. Although we restrain the join point model of the aspect language, we obtain clear points in the components and the system architecture where variations can be made. Thus, the system developer who is extending and reconfiguring an existing system does not have to have a complete insight into the system or component internals to perform successful reconfiguration. Therefore, in our component model we enforce that the places where possible extensions can be done (a component can be reconfigured) are explicitly declared in the component interfaces as reconfiguration locations. It is our experience (confirmed by the experiences of third-party COMET users) that these points are not difficult to identify in the design and implementation phase of the component development. For example, reconfiguration locations for the QAC and the FCC from the COMET QoS aspect package were straightforwardly determined in the design phase by taking into consideration a few possible policies that could be applied to these components.

Hence, relevant reconfiguration locations in the components should be favorably identified in the design phase of the system development with regard to possible policy variations. In development of the COMET database we have experienced that these locations are bounded and relatively low in number and once identified they are even suitable for aspects that are developed later on, i.e., aspects that were not anticipated when the were initially identified. Moreover, the explicitly declared reconfiguration locations are desirable in the real-time domain as they provide pre-defined places where code modifications can be done and, therefore, the system can be analyzed during the design phase to establish if it satisfies temporal constraints and has desired functional characteristics [170, 174]. We conclude that by using components with explicitly declared reconfiguration locations we enable efficient development of an analyzable product line architecture of a real-time system that has the ability to satisfy specified performance guarantees.

Lesson 3: *There is a tradeoff between configurability, reusability, and maintenance.* Having a large number of aspects leads to high demands on maintainability of the aspects and the system, while fewer aspects lead to better maintainability at the expense of limiting configurability and reusability of aspects in the system. This further confirms the conclusions made in [175] where a tradeoff between requirements for configurability and maintenance when using aspects in embedded software systems was identified. In the case of development of a QoS aspect package for COMET our primary goal was to ensure reuse of QoS-related aspects and to increase system configurability. Therefore, we have chosen to separate concerns such that we have a great number of aspects that each can be used in multiple COMET configurations. For example, the missed deadline monitor aspect is decoupled from the missed deadline controller aspect (both are part of the QoS policy aspects and implement FC-M policy) to ensure that reuse of aspects is increased since the missed deadline monitor aspect can generally be

used in combination with another controller policy. In the case when there is a focus on maintainability, the missed deadline monitor aspect and the missed deadline control aspect could be combined into one aspect that both monitors and controls the deadline misses. The same is true for the scheduling strategy aspect and the QoS through the update scheduling aspect that both implement parts of the QMF algorithm.

Hence, if reusability and configurability is of foremost concern, as it is typically the case in the context of creating families of real-time systems, real-time policies should be decomposed into greater number of aspects. Thus, trading maintainability for reusability and configurability. To deal with maintainability issues, an efficient way of organizing aspects and components for easier access and modification within an aspect package is needed.

Lesson 4: *Aspects can be reused in various phases of the system development.* We found that aspects can efficiently be reused in different phases of the system development. This is true for reusing aspects both in the system design and implementation phase, and the evaluation phase. For example, due to the nature of QoS policies, one or several aspects constituting a policy normally control the load of the system and in some way monitor the system performance. Hence, in addition to reusing these aspects in a number of QoS configurations, they can be reused in the testing and evaluation phase for evaluating performance and gathering statistics. As a simple illustration, the missed deadline monitor aspect within the COMET QoS aspect package is used in the design and implementation phase of the system as a part of a QoS management to implement a specific QoS policy, and is later reused in the the evaluation phase of the system development for performance evaluations (presented in section 9.3).

Lesson 5: *Aspect languages are a means of dealing with legacy software.* Since we initially developed the COMET database system to be primarily suited for hard real-time systems in the vehicular industry [124], the programming language used for the development of the basic database functionality needed to be suited for software that already existed in a vehicular control system. Moreover, analysis techniques that have been used in the existing vehicle control system should be applicable to our basic database components. This leads to the development of the COMET basic configuration using the C programming language. Aspects provide efficient means for introducing extensions to the system; we used the AspectC++ weaver since a weaver for the C language [46] is not yet publicly available. Overall, we believe that if extending existing real-time systems, which are typically developed in a non-object-oriented language such as C, aspects are of greater value than rebuilding the system using an object-oriented language and then making extensions to it using an object-oriented language such as C++.

Lesson 6: *Less is more when it comes to aspect languages for embedded and real-time systems.* When developing aspect languages for real-time systems operating in resource-constrained environments, the focus should be on providing basic aspect language features that facilitate encapsulating and weaving aspects into the code of components in the simplest and most memory-efficient way possible. We believe that minimizing memory usage should be of primary importance for aspect languages suitable for these types of systems. Given that most real-time computing systems are developed using non-object-oriented languages, the inter-type declaration could be kept as simple as possible, e.g., allowing weaving of single members in structs. We also observe that due to the nature of many real-time operating systems, e.g., Rubus [20] and MicroC [93], the advice and pointcut model could be simplified. Namely, pointcut syntax in most cases does not need to be as elaborate as it is in current aspect languages (e.g., AspectJ and AspectC++). We have developed most of the COMET aspects using `call` and `execution` pointcuts, and occasionally `within`.

Part IV

Epilogue

Chapter 10

Related Work

Given that representative design approaches are already contrasted against ACCORD in previous chapters, we here primarily focus on existing component-based real-time systems and database platforms. As the research in applying aspect-orientation to real-time system development is in its early stages and, thus, considerably sparse, we direct our attention to existing aspect-oriented database platforms.

Since a constituent of ACCORD is an approach to formal verification of reconfigurable components, we conclude this chapter with an overview of approaches to formal verification of reconfigurable systems, which we relate to our verification method.

10.1 Component-Based Real-Time Systems

In this section we review three distinct types of component-based embedded real-time systems.

- Systems where extensions are possible by plugging components that provide non-standard features or functionality. An example of this type of systems is SPIN [31], an extensible microkernel.
- Systems that provide efficient management of resources in dynamic heterogeneous environments, e.g., 2K [83] is a CORBA-based distributed operating system specifically developed for management of resources in a distributed environment.
- Systems that have fine-grained components and an architecture that facilitates system configurability, e.g., VEST [160], Ensemble [106], and the port-based object (PBO) approach [167].

SPIN

SPIN [31, 132] is an extensible operating system that allows applications to define customized application-specific operating system services. An application-specific service is one that precisely satisfies the functional and performance requirements of an application, e.g., multimedia applications impose special demands on the scheduling, communication and memory allocation policies of an operating system. SPIN provides a set of core services that manage memory and processor resources, such as device access, dynamic linking, and events. All other services, such as user-space threads and virtual memory, are provided as extensions. A reusable component, called an extension, is a code sequence that can be installed dynamically into the operating system kernel by the application or on behalf of it. The mechanism that integrates extensions (components) with the core system are events, i.e., communication in SPIN is event-based. Event-based communication allows considerable flexibility of the system composition as all relationships between the core system and components are subject to changes by changing the set of event handlers associated with any given event.

The correctness of the composed system depends only on the language safety and encapsulation mechanisms, specifically interfaces, type safety, and automatic storage management. Analysis of the composed system is not performed since it is assumed that the configuration support provided within the Modula-3 language is enough to guarantee the system to be correct and safe. Provided the right extension for real-time scheduling policy, this operating system can be used for soft real-time applications such as multimedia applications [31, 132].

2K

2K [85, 83] is an operating system specifically developed for manipulation of resources in a distributed heterogeneous environment (different software systems on different hardware platforms). As shown in figure 10.1, the 2K middleware architecture is realized using standard CORBA services such as naming, trading, security, and persistence, as well as extending the CORBA service model with additional services, such as QoS-aware management, automatic configuration, and code distribution.

Integration of components into the middleware is done through a component called dynamic TAO, the adaptive communication environment ORB. The dynamic TAO is a CORBA compliant reflective ORB as it allows inspection and reconfiguration of its internal engine [84]. The dynamic TAO component has a memory footprint greater than a few megabytes, which makes it inappropriate for use in environments with limited resources. A variant to the dynamic TAO, a LegORB component, is developed by the 2K group and it has a small footprint and is appropriate for embedded environments, e.g., 6 Kbytes on the PalmPilot running on PalmOS.

2K provides automated installation and configuration of new components and

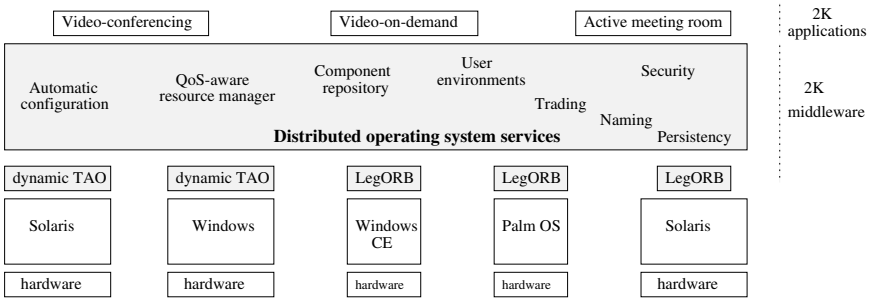


Figure 10.1: *The 2K middleware architecture*

the development of new components is done using CORBA component specifications [127]. However, it is assumed that inter-component dependencies provide good basis for the system integration and guarantee correct system behavior (other guarantees of the system behavior, obtained by appropriate analysis, do not exist).

Ensemble

Ensemble is a high performance network protocol architecture designed to support group membership and communication protocols [106]. Ensemble does not enforce real-time behavior, but is nevertheless interesting because of the configurable architecture and the way it addresses the problem of configuration and analysis of the system. Ensemble includes a library of over sixty micro-protocol components that can be stacked, i.e., formed into a protocol in a variety of ways to meet communication demands of an application. Each component has a common event-driven Ensemble micro-protocol interface, and uses message-passing as communication. Ensemble's micro-protocols implement basic sliding window protocols and functionality such as fragmentation and re-assembly, flow control, signing and encryption, group membership, and message ordering. The Ensemble system provides an algorithm for calculating the stack, i.e., composing a protocol out of micro-protocols, given the set of properties that an application requires. This algorithm encodes knowledge of protocol designers and appears to work quite well, but it does not assure generation of a correct stack (the methodology for checking correctness is not automated yet). Thus, Ensemble can be efficiently customized for different protocols, i.e., it has a high level of tailorability. In addition, Ensemble gives the possibility of formal optimization of the composed protocol. This is done in Nuprl [106] and appears to give good results in optimizing a protocol for a particular application.

VEST

VEST aims to enable the construction of an embedded real-time system with strengthened resource needs [160, 161, 164, 162]. The VEST development process is fairly well-defined with an emphasis on configuration and analysis tools. System development starts with the design of the infrastructure, which can be saved in a library for further reuse (see figure 10.2). The infrastructure consists of micro-components: interrupt handlers, indirection tables, dispatchers, plug and unplug primitives, and proxies for state mapping.

After a system is composed, dependency checks are invoked to establish certain properties of the composed system. If the properties are satisfied and the system does not need to be refined, the user can invoke analysis tools to perform real-time and reliability analysis. As can be seen, VEST offers a high degree of tailorability for the designer, i.e., a specific system can be composed out of appropriate components as well as infrastructure from the component library.

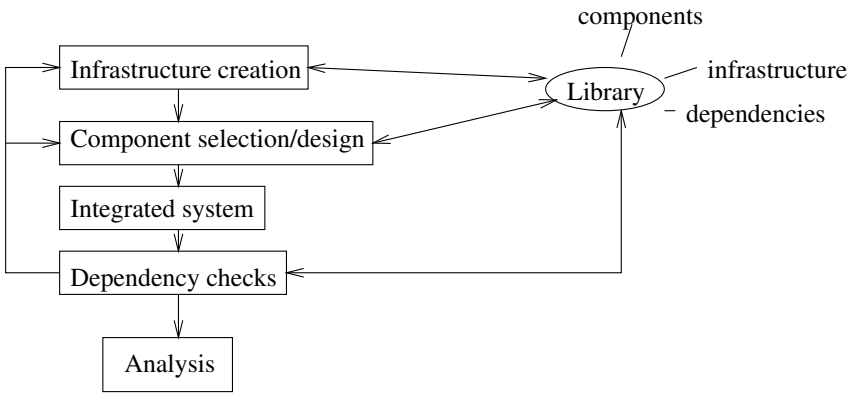


Figure 10.2: *Embedded system development in VEST*

It should be noted that in the first version of VEST, components were fine-granule, but VEST did not have an explicit component model, implying that components could be pieces of code, classes, and objects [160, 161]. however, currently VEST uses the well-defined CORBA component model [164]. Moreover, each component is associated with real-time properties such as WCET, deadline, and precedence and exclusion constraints, which enable real-time analysis of the composed system. In addition to temporal properties, components have explicit memory needs and power consumption requirements, needed for efficient use in an embedded system.

Designing and selecting the appropriate component(s) is a fairly complex process, since both real-time and non-real-time aspects of a component must be considered and appropriate configuration support has to be available. Dependency

checks proposed in VEST are one good way of providing configuration support. Due to its complexity dependency checks are broken into four types:

- ❏ factual: component-by-component dependency checks (WCET, memory, importance, deadline, etc.);
- ❏ inter-component: pairwise component checks (interface requirements, version compatibility, is a component included in another, etc.);
- ❏ aspects: checks that include issues that affect the performance or semantics of components (real-time, concurrency synchronization and reliability issues); and
- ❏ general: checks of global properties of the system (e.g., the system should not experience deadlocks and hierarchical locking rules must be followed).

Having well-defined dependency checks is vital since they minimize possible errors in the system composition. In its recent edition [164, 162], VEST has been extended with prescriptive aspects, which are defined as design-level aspects describing the relationship and the interaction between the components. Prescriptive aspects help in doing the factual and general dependency checks of the system.

The VEST configuration tool allows tool plug-ins, thus enabling temporal analysis of the composed system by enabling plugging off-the-shelf analysis tools into the VEST environment.

PBO Model

A component-based system based on the PBO model is suitable for development of embedded real-time control software system [167]. Components from the component library, in addition to newly created ones, can be used for the system assembly. A component is the PBO that is implemented as an independent concurrent process. Components are interconnected through ports, and communicate through shared memory.

The PBO defines module specific code, including input and output ports, configuration constants (for adopting components for different applications), the type of the process (periodic and aperiodic), and temporal parameters such as deadline, frequency, and priority. Support for composing a system out of components is limited to general guidelines given to the designer and the design process is not automated. This approach to componentization is somewhat unique since it gives methods for creating a framework that handles the communication, synchronization and scheduling of each component. Any C programming environment can be used to create components with minimal increase in performance or memory usage. Creating code using PBO methodology is an "inside out" programming paradigm as compared to a traditional coding of real-time processes. With this, the reconfiguration of the system is traded for performance and memory optimization.

The PBO method provides consistent structure for every process and OS system services, such as communication, synchronization, scheduling. Only when necessary, OS calls methods of PBO to execute application code. Analysis of the composed system is not considered.

10.2 Component-Based Database Systems

Component-based database management systems can be classified as follows [55]:

- ❏ Databases that can be extended with non-standard functionality, e.g., Oracle8*i* [128], Informix Universal Server with its DataBlade technology [77], Sybase Adaptive Server [126], and DB2 Universal Database [42].
- ❏ Databases that integrate existing data stores into a database system and provide users and applications with a uniform view of the entire system, e.g., OLE DB [116].
- ❏ Platforms that provide database functionality in a standardized form unbundled into services, e.g., (real-time) CORBAService [131].
- ❏ Databases that enable composition of non-standard DBMSs out of reusable components, e.g., KIDS [64].

Next, we review representatives of each class.

Oracle8i

Oracle8*i* is an extensible database system. It allows developers to create their own application-domain-specific data types [128]. Capabilities of the Oracle data server can be extended by means of data cartridges, which represent components in the Oracle8*i* architecture. A data cartridge consists of one or more domain-specific types and can be integrated with the server. Data cartridges can be integrated into a system through extensibility interfaces. There are three types of these interfaces: DBMS and data cartridge interfaces, used for communication between components and the DBMS, and service interfaces used by the developers of a component.

The architecture of the Oracle8*i* is fixed and defines the places where extensions can be made (components added), i.e., the system has low degree of tailorability. Provided configuration support by the Oracle Designer family of products is adequate, since the system already has a fixed architecture and predefined extensions, and that extensions are allowed only in well-defined places of the architecture. This type of system emphasizes on satisfying only one requirement - handling non-standard data types. Also, these systems cannot easily be integrated in all application domains, e.g., real-time system, since there is no analysis support for checking temporal behavior.

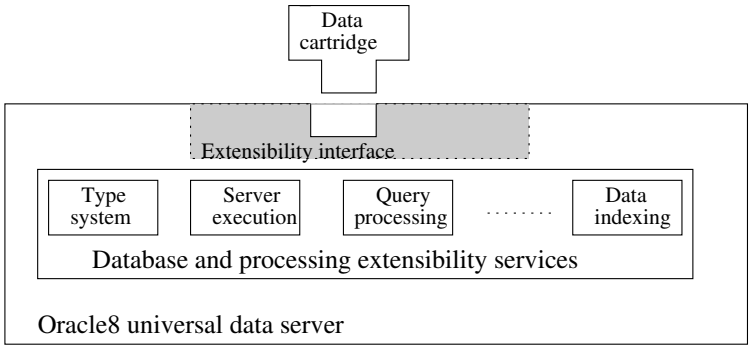


Figure 10.3: *The Oracle extensibility architecture*

Informix DataBlade Technology

DataBlade modules are standard software modules that can be plugged into the Informix Universal Server database to extend its capability [77]. DataBlade modules are components in the Informix Universal Server. These components are designed specifically to enable users to store, retrieve, update, and manipulate any domain-specific type of data. Similar to Oracle, Informix has provided low degree of tailoring, since the database can only be extended with standardized components that enable manipulation of non-standard data types. Configuration support is provided for development and installation of DataBlade modules, e.g., BladeSmith, BladePack, and BladeManager.

DB2 Universal Database

DB2 Universal Database [42, 52] also allows extensions in the architecture to provide support for comprehensive management of application-specific data types. Application-specific data types and new index structures for that data types are provided by DB2 Relational Extenders, reusable components in the DB2 Universal Database architecture. There are DB2 Relation Extenders for text (text extender), image (image extender), audio and video (extender). Each extender provides the appropriate functions for creating, updating, deleting, and searching through data stored in its data type. An extender developer's kit with wizards for generating and registering extenders provides support for the development and integration of new extenders in the DB2 Universal Database.

Sybase Adaptive Server

Similar to other extensible database systems, the Sybase Adaptive Server [126] enables extensions in its architecture, called Sybase's adaptive component

architecture (ACA), to enable manipulation of application-specific data types. Components that enable manipulation of these data types are called Speciality Data Stores, e.g., speciality data stores for text, time series, and geospatial data. The Sybase Adaptive Server differs from other database systems in the extensible DBMS category in that it provides support for standard components in distributed computing environments. Through open (Java) interfaces, Sybase's ACA provides mechanisms for communication with other database servers. Also, Sybase enables interoperability with other standardized components in the network, such as JavaBeans.

OLE DB

OLE DB [34, 35] is a specification for a set of data access interfaces designed to enable a variety of data stores to work together. OLE DB provides a way for any type of data store to expose its data in a standard and tabular form, thus unifying data access and manipulation. In Microsoft's OLE-DB infrastructure, a component is thought of as [116]:

> "... the combination of both process and data into a secure, reusable object..."

and as a result, both consumers and providers of data are treated as components. A data consumer can be any piece of the system or the application code that needs access to a broad range of data. In contrast, data providers are reusable components that represent data sources, such as Microsoft ODBC, Microsoft SQL server, Oracle, Microsoft Access, which are all standard OLE DB providers. Thus, OLE DB enables building component-based solutions by linking data providers and data consumers through providing services that add functionality to existing OLE DB data and where the services are treated as components in the system (see figure 10.4). The architecture in figure 10.4 is called the universal data access (UDA) architecture. It is possible to develop new, customized, data providers that reuse existing data providers as the underlying component or a component building block of more complex (data provider) components.

Although OLE DB provides unified access to data and enables developers to build their own data providers, there is no common implementation on either the provider or consumer side of the interface [36]. Compatibility is provided only through the specification and developers must follow the specification exactly to make interoperable components, i.e., adequate configuration support for this is not yet provided. To make up for inadequate configuration support, Microsoft has made available, in Microsoft's software developer's kit (SDK), tests that validate conformance of the specification. However, analysis of the composed system is missing.

OLE DB is not applicable for the real-time domain since it does not provide support for specifying and enforcing temporal constraints on the components and

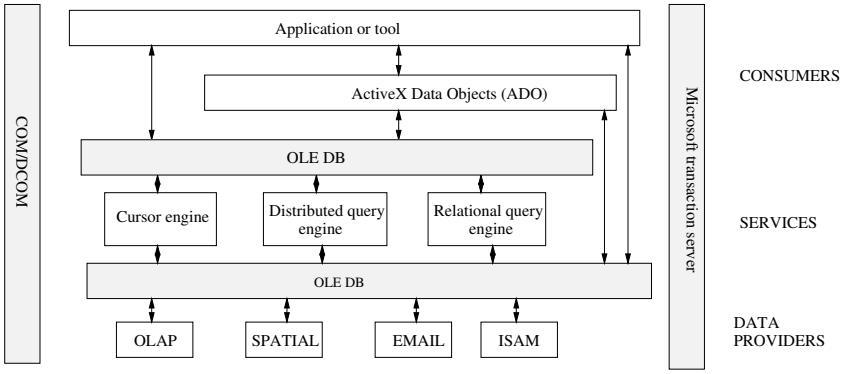


Figure 10.4: *The Universal Data Access (UDA) architecture*

the system. Additionally, OLE DB is limited with respect to software platforms, since it can only be used in Microsoft software environments.

(Real-time) CORBAservices

One single DBMS could be obtained by gluing together CORBAservices that are relevant for databases, such as transaction service, backup and recovery service, and concurrency service. Adding the real-time services of CORBA, such as scheduling service, could result in real-time CORBAservices DBMS. CORBAservices are implemented on the top of the object request broker (ORB). Service interfaces are defined using the interface definition language [56]. In this scenario a component would be one of the database (or real-time) relevant CORBAservices. This would mean that applications could choose, from a set of stand-alone services, those services (components) that they need. However, this approach is (still) not viable because it requires writing significant amount of glue code. In addition, performance overhead could be a problem due to the inability of an ORB to efficiently deal with fine-granularity objects [131]. Also, an adequate value-added framework that allows development of components and use of these components in other applications is still missing.

KIDS

The KIDS [64], kernel-based implementation of database management systems, approach to constructing configurable component-based databases is an interesting research project at the University of Zürich, since it offers a high level of reusability, where virtually any results obtained in a previous system construction is reused (designs, architectures, specifications, etc.). Components in KIDS are DBMS subsystems that are collections of brokers. Brokers are responsible

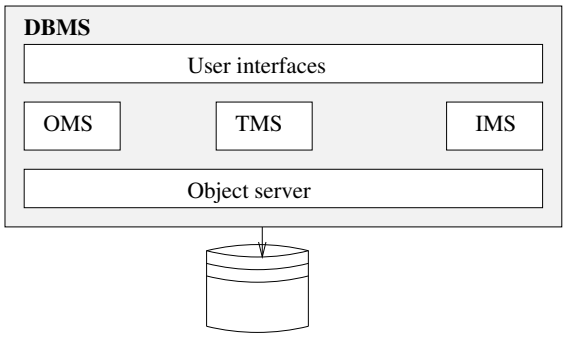


Figure 10.5: *The KIDS subsystem architecture*

for a related set of tasks, e.g., object management, transaction management, and integrity management. A structural view of the KIDS architecture is shown in figure 10.5. The DBMS architecture consists of two layers. The first layer is the object server component, which supports the storage and retrieval of storage objects. The object server component is reused in its entirety, and it belongs to the fixed part of the DBMS architecture (this is because the object server implements functionality needed by any DBMS). The second layer is variable to a large extent, and can be decomposed into various subsystems. In the initial decomposition of KIDS, three major subsystems exist in the second layer:

- ❏ the object management subsystem (OMS), which implements the mapping from data model objects into storage objects, retrieval of data model objects, and meta data management;
- ❏ the transaction management subsystem (TSM), which implements the concept of a transaction, including concurrency control, recovery, and logging; and
- ❏ the integrity management subsystem (IMS), which implements the (DBMS-specific) notion of semantic integrity, and is responsible for checking whether database state transitions result in consistent states.

These three subsystems (OMS, TMS, and IMS) implement basic database functionality. Additional functionality can be provided by adding new subsystems in the second layer of the KIDS architecture, i.e., expanding decomposition of this layer to more than three subsystems.

By expanding the initial set of components in the KIDS architecture with the functionality (components) needed by a particular application, one could be able to design "plain" object-oriented DBMS, a DBMS video-server, or a real-time plant control DBMS. Of course, in the proposed initial design of KIDS, real-time properties of the system or components are not considered.

A defined process of a DBMS construction, reusability of components and architectures, and high degree of componentization (tailorability) of a system differentiates this CDBMS from all others.

10.3 Aspect-Oriented Database Systems

In the area of database systems the AOD [18], aspect-oriented databases, initiative aims to incorporate the notion of separation of concerns into databases. The focus of this initiative is on providing a non-real-time database that can be effectively customized using aspects [143].

The AOD initiative separates aspects in database systems in two levels [144]:

- DBMS level, which are aspects that provide features affecting the software architecture of the database system, and
- database level, which are aspects that relate to the data maintained by the database and their relationship, i.e., database schema.

Aspects on the DBMS level correspond to application aspects defined within ACCORD. Within the AOD initiative, the aspect-oriented approach has been employed to achieve customization in SADES [142], a semi-autonomous database evolution system.

Following is a description of main features of SADES with the focus on aspect support. As mentioned, SADES is a database system that incorporates the notions from AOSD to provide support for effective customization. SADES has been implemented on top of the commercially available Jasmine object DBMS [144]. The SADES architecture is divided into a set of spaces as follows:

- object space, which holds all objects, i.e., data, residing in the database,
- meta-object space, which holds meta-data, e.g., classes, class member definitions, and definition scopes,
- meta-class space, which holds entities that are used to instantiate meta-objects in the meta-object space, and
- aspect space, which holds all the aspects residing in the database.

Meta-class "aspect" residing in the meta-class space is used to instantiate aspects. SADES uses aspects to provide customization of the following features on the database level [144]:

- changes to links among entities, such as predecessor/successor links between object versions or class versions, inheritance links between classes, etc.,
- changes to version strategy for object and class versioning,

- ❏ changes to structural consistency approach, and
- ❏ extending the system with new meta-classes.

Although COMET goals overlap partly with the goals for SADES in the effort to enable customization of the database system by aspect weaving, aspects supported by SADES differ from aspects supported by COMET. Namely, COMET supports aspects on the DBMS level, while the main focus of SADES is aspect support on the database level. SADES has been developed for non-real-time environments and, thus, does not address real-time issues. Although it is claimed that the SADES approach to aspect support could be applied to existing component-based database systems [144], it is not clear how this can be achieved since the components in SADES are typical AOSD-type components, i.e., white-box components.

10.4 Formal Approaches for Aspects and Components

Li et al. and Sipma have provided basic formalization methods for verification of reconfigurable component-based designs [100, 101, 156]. Both of these research efforts primarily focus on proving the correctness of the functional behavior of components and aspects, and do not consider timing behavior of the system. Sipma [156] provides a formal method for crosscutting, where system and aspects are modeled as modular transition system, and verification of crosscuttings is done applying deductive reasoning. Her work is more oriented toward existing aspect languages, providing support for explicit specification of aspects and advices as well as the pointcuts.

Li et al. [100, 101] model the system (components) and extensions (features) as state machines. They provide a compositional verification based on model checking, and quasi-sequential composition of features with the base system. Moreover, their work concentrates on the features, and feature-oriented systems, and therefore considers crosscutting on a more abstract level than crosscutting defined by aspect weaving and existing aspect languages; their approach, in the named papers, does not support specification of aspects by means of advices and pointcuts. Krishnamurthi et al. [90] extended this feature-oriented formalization to support verification of aspect-oriented programs. Here aspects are explicitly modeled as state machines. They give an informal description of a method as follows. The verification is done by first extracting an appropriate interface from the crossproduct of the state machines that model the program and pointcut designators. These interfaces are then used in the property preservation step, in which the advice is checked to find out whether the weaving preserves proven program property.

In the real-time domain, a lot of work has been done in the area of verification of real-time systems, see [8] for a survey. Approaches to real-time system

verification typically use timed automata as the underlying formalization and primarily focus on parallel composition of the components and the system, and the space-explosion reduction [8, 9, 96]. We build upon this work, focusing on the representation of the timed automata in terms of zones, and provide a method for verification of the real-time systems obtained by quasi-sequential composition of aspects with components.

Chapter 11

Conclusions

This final chapter presents a summary of our work and restates the research contributions. The issues for the future work are also identified.

11.1 Summary

The cost-effective development of real-time software through reuse and reconfiguration is one of the key issues that needs to be investigated. Using software engineering techniques specifically developed to facilitate reuse and reconfiguration could be beneficial for engineering real-time systems. Especially applying the main principles of the component-based and aspect-oriented software development to real-time systems development would enable:

- efficient and fast system configuration from the components in the component library based on the system requirements;
- (re)configuring, i.e., tailoring, components and/or a system for a specific application by changing the behavior (code) of the component via aspect weaving; and
- enhanced reusability of software as both components and aspects can be reused across different applications.

However, applying aspect-oriented and component-based principles to real-time system development is challenging for a number of reasons. Namely, if both aspects and components are to be used for system development, a real-time component model should be defined such that it enables aspect weaving into component code, while preserving information hiding. Moreover, the model should also provide adequate means for specifying temporal and resource constraints and,

thereby, ensure that analysis of temporal correctness of the resulting system can be performed.

To fully capitalize on the benefits that component-based development offers, it is desirable to enable dynamic reconfiguration of a real-time system. Dynamic reconfiguration is often preferable for embedded real-time systems that require continuous hardware and software upgrades in response to technological advancements, environmental change, or alteration of system goals during system operation. However, dynamic reconfiguration of a real-time system also changes the temporal properties of the tasks in a system, which in turn affects the performance of the system negatively, e.g., increasing the deadline miss ratio. Hence, dynamic reconfiguration of real-time systems should be enabled, but only under the condition that a desirable performance level can be guaranteed in the reconfigured system.

Also, when composing systems using components and aspect it is expedient to be able to formally prove temporal and functional properties of components, aspects, and the resulting system. The verification challenge for reconfigurable systems is great as the verification methodology needs to ensure that components are verified only once and the verification of reconfigured designs is done on aspects. This is to overcome the possible state explosion that might happen in cases where verification is done on the resulting, woven systems.

Since there could be many aspects and components in the library, the designer might need assistance in choosing the relevant subset for configuring and analyzing the system for a specific real-time application. Therefore, appropriate tools for configuration and analysis also need to be provided.

Resolving the identified issues would enable successful integration of the ideas and notions from component-based and aspect-oriented software development into real-time system development. Thereby, cost-effective development of reconfigurable and reusable real-time software would be feasible. In this thesis we have proposed the following solutions.

1. The RTCOM component model, which describes how a real-time component, supporting different aspects and enforcing information hiding, could be efficiently designed and implemented.

2. Support for static and dynamic reconfiguration of a real-time system in terms of:

   (a) Design guidelines for development of real-time systems out of components and aspects, which prescribe that a real-time system design should be carried out in the following sequential phases: (i) decomposition of the real-time system into a set of components, followed by (ii) decomposition into a set of aspects, and (iii) implementation of components and aspects based on RTCOM.

   (b) A method for dynamic system reconfiguration suited for resource-constrained real-time applications ensuring that components and

aspects can be added, removed, or exchanged in the system at runtime. Thus, in addition to traditional static reconfiguration, we support dynamic reconfiguration of a system.

3. Methods for ensuring satisfaction of real-time constraints, namely:

   (a) A method for aspect-level worst-case execution time analysis of real-time systems assembled out of aspects and components, which is performed at system composition time.

   (b) A method for formal verification of temporal properties of reconfigurable real-time components that enables (i) proving temporal properties of individual components and aspects, and (ii) proving that reconfiguration of a component via aspect weaving preserves expected temporal behavior in the reconfigured component.

   (c) A method for reconfigurable quality of service that enables configuring quality of service in real-time systems in terms of desired performance metric and performance level based on the underlying application requirements. The method ensures that the specified level of performance is maintained during system operation and after reconfiguration.

We have implemented a tool set with which the designer can efficiently configure a real-time system to meet functional requirements and analyze it to ensure that non-functional requirements in terms of temporal constraints and available memory are satisfied. The analysis tools represent an automation of the analysis methods from (3).

We refer to these methods and tools collectively as the ACCORD framework to indicate that, in addition of being used in isolation, the solutions can be used together to further alleviate efficient development of reconfigurable and reusable real-time software.

We have shown how ACCORD can be used in practice by describing the way we have used it in the design and development of COMET, a configurable real-time database. From this case study we conclude that ACCORD could have a positive impact on real-time system development in general by enabling efficient configuration of real-time systems, and improving reusability and flexibility of real-time software.

11.2 Future Work

ACCORD with its accompanied tools and methods could still be enhanced to ensure more successful application to real-time systems development. The following is a number of issues that, if resolved, could further increase applicability and readiness of ACCORD to be used in all phases of the real-time system development.

Currently, with our method for formal verification it is possible to verify a limited set of aspects and components. Extending the existing method to embrace the verification of a number of aspects and components is needed to ensure scalability of the verification method to complex real-time systems. Moreover, the formal verification is currently not supported by tools within the ACCORD development environment. A tool that would automatically translate models of ACCORD-based systems into formal models (timed automata) based on our formalizations would provide a necessary foundation for tool support.

The method for QoS-aware dynamic system reconfiguration could further be enhanced so that the system developer can determine if dynamic reconfiguration of the system is feasible or not before the actual reconfiguration takes place. This way we can ensure that the reconfiguration is carried out only if it is safe to do so, i.e., when we made sure that reconfiguring a system will not violate the quality of service specification and that the performance of the system will be at the desired level under and after reconfiguration.

We would also like to refine the composition part of RTCOM, i.e., a language for describing the composition needs of components/aspects. Currently the component model supports only simple composition rules specifying for an aspect or a component with which components and/or aspects it can be combined. We would like to develop composition rules that account for both functional and run-time needs of components and aspects.

Appendix A

Abbreviations

ACCORD	AspeCtual COmponent-based Real-time system Development
ACCORD-ME	ACCORD Modeling Environment
ACID	Atomicity, Consistency, Isolation, Durability
ADARTS	ADA-based Design Approach for Real-Time Systems
ADL	Architectural Description Language
AOD	Aspect-Oriented Databases
AOSD	Aspect-Oriented Software Development
AS	Aspect Separation
CBSD	Component-Based Software Development
CC	Concurrency Control
CDBMS	Component-based DataBase Management Services
CM	Component Model
COMET	COMponent-based Embedded real-Time database
DARTS	Design Approach for Real-Time Systems
DBMS	DataBase Management System
ECU	Electronic Control Unit
EDF	Earliest Deadline First
FCC	Feedback Controller Component
FC-M	Feedback Control based on Miss ratio

GME	Generic Modeling Environment
GUARD	Gatekeeping Using Adaptive eaRliest Deadline
HP-2PL	Hight Priority 2 Phase Locking
HRT-HOOD	Hard Real-Time Hierarchical Object Oriented Design
Hw Db	Hardware Database
IMC	Indexing Manager Component
IDL	Interface Definition Language
IECU	Instrumental Electronic Control Unit
ISC	Invasive Software Composition
KIDS	Kernel-based Implementation of Database management
LMC	Locking Manager Component
MMC	Memory Manager Component
OCC	Optimistic Concurrency Control
ODC	Optimistic Divergence Control
ORB	Object Request Broker
OS	Operating System
PBO	Port-Based Object
QAC	QoS Actuator Component
QoS	Quality of Service
RMS	Rate Monotonic Scheduling
RT-UML	Real-Time Unified Modeling Language
RTCOM	Real-Time COmponent Model
SADES	Semi-Autonomous Database Evolution System
SC	System Composability
SOP	Subject-Oriented Programming
TAO	The Adaptive communication environment ORB
TCTL	Timed Computational Tree Logic
TMC	Transaction Manager Component
TRSD	Transactional Real-Time System Design
SADES	Semi-Autonomous Database Evolution System
SMC	Scheduling Manager Component
UIC	User Interface Component
UML	Unified Modeling Language
VECU	Vehicle Electronic Control Unit
VEST	Virginia Embedded Systems Toolkit
WCET	Worst-Case Execution Time
WoE	Warnings or Errors
WoE Db	Warnings or Errors Database

Bibliography

[1] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions: a performance evaluation. *ACM Transactions on Database Systems*, 17(3):513–560, September 1992.

[2] T. Abdelzaher, Y. Lu, R. Zhang, and D. Henriksson. Practical application of control theory to web services. In *Proceedings of American Control Conference (ACC)*, 2004.

[3] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):80–96, 2002.

[4] T. F. Abdelzaher, J. A. Stankovic, C. Lu, R. Zhang, and Y. Lu. Feedback performance control in software services. *IEEE Control Systems Magazine*, 23(3):74–90, June 2003.

[5] G. D. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architectures. *ACM Software Engineering Notes*, 18(5):9–20, 1993.

[6] AIRES: Aspects in real-time embedded systems. Project website at `http://www.dist-systems.bbn.com/projects/AIRES/`, February 2003.

[7] M. Aksit, J. Bosch, W. van der Sterren, and L. Bergmans. Real-time specification inheritance anomalies and real-time filters. In *Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP'94)*, volume 821 of *Lecture Notes in Computer Science*, pages 386–407. Springer-Verlag, 1994.

[8] R. Alur. Timed automata. In *Proceedings of 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 8–22. Springer Verlag, 1999.

[9] R. Alur, C. Courcoubetis, and D. Dill. Model checking for real-time systems. In *Proceedings of the 5th IEEE International Symposium on Logic in Computer Science*, Philadelphia, 1990. IEEE Computer Scoiety Press.

[10] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[11] M. Amirijoo, J. Hansson, S. Gunnarsson, and S. H. Son. Enhancing feedback control scheduling performance by on-line quantification and suppression of measurement disturbance. In *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'05)*, pages 2–11. IEEE Computer Society Press, 2005.

[12] M. Amirijoo, J. Hansson, and S. H. Son. Error-driven QoS management in imprecise real-time databases. In *Proceedings of the 15th IEEE Euromicro Conference on Real-Time Systems (ECRTS'03)*, pages 63–72. IEEE Computer Society Press, 2003.

[13] M. Amirijoo, J. Hansson, and S. H. Son. Specification and management of QoS in imprecise real-time databases. In *Proceedings of the 7th IEEE International Database Engineering and Applications Symposium (IDEAS'03)*, pages 192–201. IEEE Computer society Press, 2003.

[14] M. Amirijoo, J. Hansson, and S. H. Son. Algorithms for managing QoS for real-time data services using imprecise computation. In *Proceedings of the Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA'03)*, volume 2968 of *Lecture Notes in Computer Science*, pages 136–157. Springer-Verlag, 2004.

[15] M. Amirijoo, J. Hansson, S. H. Son, and S. Gunnarsson. Robust quality management for differentiated imprecise data services. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS'04)*, pages 265–275. IEEE Computer Society Press, 2004.

[16] M. Amirijoo, J. Hansson, S. H. Son, and S. Gunnarsson. Generalized performance management of multi class real-time imprecise data services. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 38–49, 2005.

[17] S. F. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and N. Elftring. DeeDS towards a distributed and active real-time database system. *ACM SIGMOD Record*, 25(1), 1996.

[18] AOD: Aspect-oriented databases. Project website at `http://www.comp.lancs.ac.uk/computing/aod/`, May 2003.

[19] AOSD: Aspect-Oriented Software Development. Official AOSD website: `http://www.aosd.net/`, January 2005.

[20] Articus Systems. *Rubus OS - Reference Manual*, 1996.

[21] U. Aßmann. *Invasive Software Composition*. Springer-Verlag, December 2002.

[22] L. Bass, P. Clements, and R. Kazman. *Software Architecture In Practice*. SEI Series in Software Engineering. Addison Wesley, 1998.

[23] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(4):355–398, 1992.

[24] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9:1–21, 1977.

[25] J. Bengtsson and W. Yi. *Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, chapter Timed Automata: Semantics, Algorithms and Tools. Springer-Verlag, 2004.

[26] Berkeley DB. Sleepycat Software Inc., `http://www.sleepycat.com`, March 2003.

[27] G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *Proceedings of the 25th IFAC Workshop on Real-Time Programming*. Elsevier, 2000.

[28] G. Bernat, A. Coling, and S. Petters. pWCET: A tool for probabilistic worst-case execution time analysis. In *Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis (WCET'03)*, Porto, Portugal, July 2003.

[29] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[30] P.A. Bernstein and N. Goodman. Multiversion concurrency control - theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, 1983.

[31] B. N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. G. Sirer. SPIN - an extensible microkernel for application-specific operating system services. Technical Report 94-03-03, Department of Computer Science and Engineering, University of Washington, Seattle, WA, USA, February 1994.

[32] M. Björk. QoS management in configurable real-time databases. Master's thesis, Department of Computer Science, Linköping University, Sweden, 2004.

[33] L. Blair and G. Blair. A tool suite to support aspect-oriented specification. In *Proceedings of the Aspect-Oriented Programming Workshop at 13th European Conference on Object-Oriented Programming (ECOOP'99)*, pages 7–10, Lisbon, Portugal, June 1999.

[34] J. A. Blakeley. OLE DB: a component DBMS architecture. In *Proceedings of the 12th IEEE International Conference on Data Engineering (ICDE'96)*, pages 203–204. IEEE Computer Society Press, March 1996.

[35] J. A. Blakeley. Universal data access with OLE DB. In *Proceedings of the 42nd IEEE International Computer Conference (COMPCON'97)*, pages 2–7, San Jose California, February 1997. IEEE Computer Society Press.

[36] J. A. Blakeley and M. J. Pizzo. *Component Database Systems*, chapter Enabling Component Databases with OLE DB. Morgan Kaufmann Publishers, 2000.

[37] J. Bosch. *Design and Use of Software Architectures*. ACM Press in collaboration with Addison-Wesley, 2000.

[38] A. Burns and A. Wellings. *HRT-HOOD: a Structured Design Method for Hard Real-Time Ada Systems*, volume 3 of *Real-Time Safety Critical Systems*. Elsevier, 1995.

[39] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. International Computer Science Series. Addison-Wesley, 1997.

[40] G. C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 1997.

[41] R. Camposano and J. Wilberg. Embedded system design. *Design Automation for Embedded Systems*, 1(1):5–50, 1996.

[42] M. J. Carey, L. M. Haas, J. Kleewein, and B. Reinwald. Data access interoperability in the IBM database family. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Interoperability*, 21(3):4–11, 1998.

[43] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano - a revolution in on-board communications. Technical report, VolvoTechnologyReport, 1998.

[44] A. Cerpa and D. Estrin. ASCENT: Adaptive Self-Configuring sEnsor Networks Topologies. *IEEE Transactions on Mobile Computing*, 3(3):272–285, 2004.

[45] A. Cervin, J. Eker, B. Bernhardsson, and K. Årzén. Feedback-feedforward scheduling of control tasks. *Real-time Systems Journal*, 23(1/2), July/September 2002. Special Issue on Control-Theoretical Approaches to Real-Time Computing.

[46] Y. Coady and G. Kiczales. Back to the future: A retroactive study of aspect evolution in operating system code. In *Proceedings of the 2nd ACM International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 50–59, Boston, USA, 2003. ACM Press.

[47] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, 2002.

[48] Com: The component object model specification, microsoft. Available at: `http://www.microsoft.com/com/resources/comdocs.asp`, February 2001.

[49] TimeSys Corp. Timewiz. `http://www.timesys.com`, October 2005.

[50] I. Crnkovic, B. Hnich, T. Jonsson, and Z. Kiziltan. Specification, implementation, and deployment of components. *Communications of the ACM*, 45(10):35–40, October 2002.

[51] I. Crnkovic and M. Larsson, editors. *Building Reliable Component-Based Real-Time Systems*. Artech House Publishers, July 2002.

[52] J. R. Davis. Creating an extensible, object-relational data management environment: IBM's DB2 Universal Database. Database Associated International, InfoIT Services, November 1996. Available at `http://www.dbaint.com/pdf/db2obj.pdf`.

[53] M. Daz, D. Garrido, L. M. Llopis, F. Rus, and J. M. Troya. Integrating real-time analysis in a component model for embedded systems. In *Proceedings of the 30th IEEE Euromicro conference*, pages 14–21. IEEE Computer Society Press, 2004.

[54] M. H. DeGroot and M. J. Schervish. *Probability and Statistics*. Addison-Wesley, 3rd edition, 2002.

[55] K. R. Dittrich and A. Geppert. *Component Database Systems*, chapter Component Database Systems: Introduction, Foundations, and Overview. Morgan Kaufmann Publishers, 2000.

[56] A. Dogac, C. Dengi, and M. T. Öszu. Distributed object computing platform. *Communications of the ACM*, 41(9):95–103, 1998.

[57] B. P. Douglass. *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Addison-Wesley, 2000.

[58] K. Erlandsson. Concurrency control in a configurable component-based real-time database. Master's thesis, Department of Computer Science, Linkköping University, Linköping, Sweden, August 2004.

[59] FACET: Framework for aspect composition for an event channel. Project website at `http://www.cs.wustl.edu/~doc/RandD/PCES/`, February 2003.

[60] Kronos tool. `http://www-verimag.imag.fr/TEMPORISE/kronos/`, January 2005.

[61] W. Fleisch. Applying use cases for the requirements validation of component-based real-time software. In *Proceedings of 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99)*, pages 75–84. IEEE Computer Society Press, May 1999.

[62] G. F. Franklin, J. D. Powell, and M. Workman. *Digital Control of Dynamic Systems*. Addison-Wesley, third edition, 1998.

[63] L. Freidrich, J. Stankovic, M. Humphrey, M. Marley, and J. Haskins. A survey of configurable, component-based operating systems for embedded applications. *IEEE Micro*, 21(3):54–68, May/Jun 2001.

[64] A. Geppert, S. Scherrer, and K. R. Dittrich. KIDS: Construction of database management systems based on reuse. Technical Report ifi-97.01, Department of Computer Science, University of Zurich, September 1997.

[65] GME: The genric modeling environement. Institute for Software Integrated Systems, Vanderbilt University, `http://www.isis.vanderbilt.edu/Projects/gme/`, December 2004.

[66] H. Gomaa. A software design method for real-time systems. *Communications of the ACM*, 27(9):938–949, September 1984.

[67] H. Gomaa. A software design method for Ada based real time systems. In *Proceedings of the 6th Washington Ada Symposium*, pages 273–284. ACM Press, 1989.

[68] H. Gomaa. Designing real-time and embedded systems with the COMET/UML method. *Dedicated Systems Magazine*, pages 44–49, 2001.

[69] R. K. Gupta. *Co-Synthesis of Hardware and Software for Digital Embedded Systems*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1995.

[70] J. Håkansson, L. Mokrushin, P. Pettersson, and W. Yi. An analysis tool for UML models with SPT annotations. In *Proceedings of the International Workshop on Specification and Validation of UML models for Real Time and Embedded Systems (SVERTS'04)*, 2004.

[71] J. R. Haritsa and S. Seshadri. Real-time index concurrency control. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):429–447, 2000.

[72] J. R. Haritsa and S. Seshadri. Real-time index concurrency control. In *Real-Time Database Systems: Architecture and Techniques*, pages 59–74. Kluwer Academic Publishers, 2001.

[73] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. Wiley-IEEE Press, 2004.

[74] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *Proceedings of the 7th IEEE Symposium of Logics in Computer Science*, pages 394–406. IEEE Computer Society Press, 1992.

[75] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *ACM SIGOPS Operating Systems Review*, 34(5):93–104, 2000.

[76] S. Hissam, G. Moreno, J. Stafford, and K. Wallnau. Enabling predictable assembly. *The Journal of Systems and Software*, 65:185–198, 2003.

[77] Developing DataBlade modules for Informix-Universal Server. Informix DataBlade Technology. Informix Corporation, 22 March 2001. Available at `http://www.informix.com/datablades/`.

[78] ISIS-PCES project: Constraint-based embedded program composition. Project website at `http://www.isis.vanderbilt.edu/Projects/PCES/`, February 2003.

[79] D. Isovic and C. Norström. Components in real-time systems. In *Proceedings of the 8th IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA'02)*, pages 135–139, Tokyo, Japan, March 2002.

[80] K.-D. Kang, S. H. Son, J. A. Stankovic, and T. F. Abdelzaher. A QoS-sensitive approach for timeliness and freshness guarantees in real-time databases. In *Proceedings of the 14th IEEE Euromicro Conference on Real-time Systems (ECRTS'02)*, pages 203–212. IEEE Computer Society Press, 2002.

[81] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.

[82] Y. Kim, M. Lehr, D. George, and S. Son. A database server for distributed real-time systems: Issues and experiences. In *Proceedings of the 2nd Workshop on Parallel and Distributed Real-Time Systems*, Cancun, Mexico, April 1994.

[83] F. Kon, R. H. Campbell, F. J. Ballesteros, M. D. Mickunas, and K. Nahrsted. 2K: A distributed operating system for dynamic heterogeneous environments. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, pages 201–208, Pittsburgh, August 2000.

[84] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhales, and R. H. Campbell. Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, volume 1795 of *Lecture Notes in Computer Science*, pages 121–143. Springer-Verlag, 2000.

[85] F. Kon, A. Singhai, R. H. Campbell, and D. Carvalho. 2K: A reflective, component-based operating system for rapidly changing environments. In *Proceedings of the Workshop on Reflective Object-Oriented Programming and Systems at the 12th European Conference on Object-Oriented Programming (ECOOP'98)*, volume 1543 of *Lecture Notes in Computer Science*, Brussels, Belgium, July 1998. Springer-Verlag.

[86] H. Kopetz and H. Obermaisser. Temporal composability. *Computing and Control Engineering Journal*, pages 156–162, August 2002.

[87] H. Kopetz, R. Zainlinger, G. Fohler, H. Kantz, P. Puschner, and W. Schütz. The design of real-time systems: from specification to implementation and verification. *Software Engineering Journal*, 6(3):72–82, 1991.

[88] X. Koutsoukos, R. Tekumalla, B. Natarajan, and C. Lu. Hybrid supervisory utilization control of real-time systems. In *Proceedings of 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'05)*, pages 12–21. IEEE Computer Society Press, 2005.

[89] C. M. Krishna and K. G. Shin. *Real-time Systems*. McGraw-Hill Series in Computer Science. The McGraw-Hill Companies, Inc., 1997.

[90] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularity. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. ACM Press, November 2004.

[91] T.-W. Kuo. *Real-Time Database Systems: Architecture and Techniques*, chapter Conservative and Optimistic Protocols, pages 29–44. Kluwer Academic Publishers, 2001.

[92] T.-W. Kuo and A. K. Mok. Application semantics and concurrency control of real-time data-intensive applications. In *Proceedings of 13th IEEE Real-Time Systems Symposium (RTSS'92)*, pages 35–45. IEEE Computer Society Press, 1992.

[93] J. J. Labrosse. *MicroC/OS-II the Real-Time Kernel*. CMPBooks, 2002.

[94] K. Y. Lam, T.-W. Kuo, B. Kao, T. S. H. Lee, and R. Cheng. Evaluation of concurrency control strategies for mixed soft real-time database systems. *Information Systems*, 27(2):123–149, 2002.

[95] K. Y. Lam and W. C. Yau. On using similarity for concurrency control in real-time database systems. *The Journal of Systems and Software*, 43(3):223–232, 1998.

[96] K. Larsen, P. Pettersson, and W. Yi. Compositional and symbolic model-checking of real-time systems. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 76–87. IEEE Computer Society Press, 1995.

[97] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on b-trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.

[98] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *Proceedings of the 12th International Conference on Very Large Databases (VLDB'86)*, pages 294–303.

[99] B. Li and K. Nahrstedt. A control theoretical model for quality of service adaptations. In *Proceedings of the 6th IEEE International Workshop on Quality of Service*, pages 145–153, 1998.

[100] H. Li, S. Krishnamurthi, and K. Fisler. Interfaces for modular feature verification. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE'02)*, pages 195–204. IEEE Computer Society Press, September 2002.

[101] H. Li, S. Krishnamurthi, and K. Fisler. Verifying cross-cutting features as open systems. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. ACM Press, November 2002.

[102] J. Lindström, T. Niklander, P. Porkka, and K. Raatikainen. A distributed real-time main-memory database for telecommunication. In *Proceedings of the International Workshop on Databases in Telecommunications*, volume 1819 of *Lecture Notes in Computer Science*, pages 158–173, 1999.

[103] Linköping University, Sweden. *COMET User Manual*.

[104] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[105] J. W. S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise computations. *IEEE Computer*, 82(1), 1994.

[106] X. Liu, C. Kreitz, R. Renesse, J.Hickey, M. Hayden, K. Birman, and R. Constable. Building reliable, high-performance communication systems from components. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, volume 34, pages 80–92, December 1999. Published as Operating Systems Review.

[107] D. Locke. *Real-Time Database Systems: Issues and Applications*, chapter Real-Time Databases: Real-World Requirements. Kluwer Academic Publishers, 1997.

[108] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Feedback control real-time scheduling: Framework, modeling and algorithms. *Journal of Real-time Systems*, 23(1/2), July/September 2002.

[109] H. Lu, Y. Ng, and Z. Tian. T-tree or B-tree: Main memory database index structure revisited. In *Proceedings of the 11th IEEE Australasian Database Conference*, pages 65–73. IEEE Computer Society, 2000.

[110] Y. Lu, A. Saxena, and T. F. Abdelzaher. Differentiated caching services; a control-theoretical approach. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS'01)*, pages 615–622. IEEE Computer Society Press, 2001.

[111] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995. Special Issue on Software Architecture.

[112] D. Mahrenholz, O. Spinczyk, A. Gal, and W. Schröder-Preikschat. An aspect-orientied implementation of interrupt synchronization in the PURE operating system family. In *Proceedings of the 5th Workshop on Object Orientation and Operating Systems at 16th European Conference on Object-Oriented Programming (ECOOP'02)*, Malaga, Spain, June 2002.

[113] MBrane Ltd. RDM Database Manager. http://www.mbrane.com.

[114] N. Medvedovic and R. N. Taylor. Separating fact from fiction in software architecture. In *Proceedings of the 3rd International Workshop on Software Architecture*, pages 10–108. ACM Press, 1999.

[115] B. Meyer and C. Mingins. Component-based development: From buzz to spark. *IEEE Computer*, 32(7):35–37, July 1999. Guest Editors' Introduction.

[116] Universal data access through OLE DB. OLE DB Technical Materials. OLE DB White Papers, 12 April 2001. Available at `http://www.microsoft.com/data/techmat.htm`.

[117] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes. In *Proceedings of the 16th International Conference on Very Large Databases (VLDB'90)*, pages 392–405. Morgan Kaufmann Publishers, 1990.

[118] A. Möller, I. Peake, M. Nolin, J. Fredriksson, and H. Schmidt. Component-based context-dependent hybrid property prediction. In *Proceedings of the ERCIM Workshop on Dependable Software Intensive Embedded systems*, 2005.

[119] Y. Mond and Y. Raz. Concurrency control in b^+-trees using preparatory operations. In *Proceedings of the 11th International Conference on Very Large Databases ()VLDB'85*, pages 331–334. Morgan Kaufmann Publishers, 1985.

[120] J. Montgomery. A model for updating real-time applications. *Real-Time Systems*, 7:169–189, 2004.

[121] A. Münnich, M. Birkhold, G. Färber, and P. Woitschach. Towards an architecture for reactive systems using an active real-time database and standardized components. In *Proceedings of 3rd IEEE International Database Engineering and Application Symposium (IDEAS'99)*, pages 351–359. IEEE Computer Society Press, 1999.

[122] D. Nyström, M. Nolin, A. Tešanović, C. Norström, and J. Hansson. Pessimistic concurrency-control and versioning to support database pointers in real-time databases. In *Proceedings of the 16th IEEE Euromicro Conference on Real-Time Systems (ECRTS'04)*, pages 261–270. IEEE Computer Society Press, 2004.

[123] D. Nyström, A. Tešanović, M. Nolin, C. Norström, and J. Hansson. COMET: A component-based real-time database for automotive systems. In *Proceedings of the IEEE Workshop on Software Engineering for Automotive Systems*, pages 1–8, May 2004.

[124] D. Nyström, A. Tešanović, C. Norström, J. Hansson, and N-E. Bånkestad. Data management issues in vehicle control systems: a case study. In *Proceedings of the 14th IEEE Euromicro International Conference on Real-Time Systems (ECRTS'02)*, pages 249–256. IEEE Computer Society Press, 2002.

[125] M. A. Olson. Selecting and implementing an embedded database system. *IEEE Computers*, 33(9):27–34, Sept. 2000.

[126] S. Olson, R. Pledereder, P. Shaw, and D. Yach. The Sybase architecture for extensible data management. *Data Engineering Bulletin*, 21(3):12–24, 1998.

[127] OMG. The common object request broker: Architecture and specification. OMG Formal Documatation (formal/01-02-10), February 2001. Available at: `ftp://ftp.omg.org/pub/docs/formal/01-02-01.pdf`.

[128] All your data: The Oracle extensibility architecture. Oracle Technical White Paper. Oracle Corporation. Redwood Shores, CA, February 1999.

[129] H. Ossher and P. Tarr. Subject-oriented programming: a critique of pure objects. In *Proceedings of the eighth annual conference on object-oriented programming systems, languages, and applications*, pages 411–428, Washington, USA, September 26 - October 1 1993. ACM Press.

[130] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[131] M. T. Özsu and B. Yao. *Component Database Systems*, chapter Building Component Database Systems Using CORBA. Data Management Systems. Morgan Kaufmann Publishers, 2000.

[132] P. Pardyak and B. N. Bershad. Dynamic binding for an extensible system. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*, Operating Systems Review, Special Issue, pages 201–212, Seattle WA, USA, October 1996. ACM and USENIX Association.

[133] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance managment. *Journal of Real-time Systems*, 23(1/2), July/September 2002.

[134] Pervasive Software Inc. Pervasive.SQL Database Manager. `http://www.pervasive.com`.

[135] Polyhedra Plc. Polyhedra database manager. `http://www.polyhedra.com`.

[136] DARPA ITO projects. Program composition for embedded systems. `http://www.darpa.mil/ito/research/pces/index.html`, 7 August 2001.

[137] P. Puschner and A. Burns. A review of worst-case execution-time analysis (editorial). *Journal of Real-Time Systems*, 18(2/3):115–128, May 2000.

[138] P. Puschner and R. Kirner. Avoiding timing problems in real-time software. In *Proceedings of the IEEE Workshop on Software Technologies for Future Embedded Systems*, pages 75–78, May 2003.

[139] K. Ramamritham. Real-time databases. *Distributed and Parallel Databases*, 1(2):199–226, 1993.

[140] K. Ramamritham, S. H. Son, and L. C. DiPippo. Real-time databases and data services. *Journal of Real-Time Systems*, 28(2-3):179–215, 2004.

[141] RapidRMA. Tri-Pacific Software, Inc., `http://www.tripac.com/html/prod-fact-rrm.html`, October 2005.

[142] A. Rashid. A hybrid approach to separation of concerns: the story of SADES. In *Proceedings of the 3rd International REFLECTION Conference*, volume 2192 of *Lecture Notes in Computer Science*, pages 231–249, Kyoto, Japan, September 2001. Springer-Verlag.

[143] A. Rashid and E. Pulvermueller. From object-oriented to aspect-oriented databases. In *Proceedings of the 11th International Conference on Database and Expert Systems Applications (DEXA'00)*, volume 1873 of *Lecture Notes in Computer Science*, pages 125–134. Springer-Verlag, 2000.

[144] A. Rashid and P. Sawyer. Aspect-orientation and database systems: an effective customization approach. *IEE Software*, 148(5):156–164, October 2001.

[145] R. Rastogi, S. Seshadri, P. Bohannon, D. W. Leinbaugh, A. Silberschatz, and S. Sudarshan. Improving predictability of transaction execution times in real-time databases. *Journal of Real-Time Systems*, 19(3):283–302, November 2000.

[146] K. J. Åström and B. Wittenmark. *Adaptive Control*. Addison-Wesley, second edition, 1995.

[147] R. Reussner, I. Poernomo, and H. Schmidt. Using the TrustME tool suite for automatic component protocol adaptation. In *Proceedings of the International Conference on Computational Science (ICCS'02)*, volume 2330 of *Lecture Notes in Computer Science*, pages 854–862. Springer-Verlag, 2002.

[148] A. Robertson, B. Wittenmark, and M. Kihl. Analysis and design of admission control in web-server systems. In *Proceedings of American Control Conference (ACC)*, 2003.

[149] ROBOCOP: Robust open component based software architecture for configurable devices project. Project website at `http://www.hitech-projects.com/euprojects/robocop/`, January 2005. ROBOCOP ITEA project.

[150] K. Sandström, J. Fredriksson, and M. Åkerholm. Introducing a component technology for safety critical embedded realtime systems. In *Proceedings of the International Symposium on Component-based Software Engineering (CBSE7)*, pages 194–208, Scotland, May 2004. Springer-Verlag.

[151] H. Schmidt. Trustworthy components-compositionality and predicition. *The Journal of Systems and Software*, 65:215–225, 2003.

[152] H. Schmidt, I. Peake, J. Xie, I. Thomas, B. Kramer, A. Fay, and P. Bort. Modelling predictable component-based distributed control architectures. In *Proceedings of the 9th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'03)*, pages 339–346. IEEE Computer Society Press, 2003.

[153] L. Sha, X. Liu, Y. Lu, and T. Abdelzaher. Queuing model based network server performance control. In *Proceedings of 23rd IEEE Real-Time Systems Symposium (RTSS'02)*. IEEE Computer Society Press, 2002.

[154] L. Sha, R. Rajkumar, S. H. Son, and C.-H. Chang. A real-time locking protocol. *IEEE Transactions on Computers*, 40(7):793–800, September 1991.

[155] A. Silberschatz, H. F. Korthe, and S. Sudarshan. *Database System Concepts*. The McGraw-Hill Companies, Inc., 1997.

[156] H. Sipma. A formal model for cross-cutting modular transition systems. In *In Proceedings of the Workshop on Foundations of Aspect-Oriented Languages (FOAL'03)*, Boston, USA, March 2003.

[157] Solaris 9 operating system. `http://www.sun.com/solaris/`, January 2005.

[158] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: an aspect-oriented extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'02)*, Sydney, Australia, February 2002. Australian Computer Society.

[159] J. Stankovic. Misconceptions about real-time computing: a serious problem for next-generation systems. *IEEE Computer*, 21(10):10–19, October 1988.

[160] J. Stankovic. VEST: a toolset for constructing and analyzing component-based operating systems for embedded and real-time systems. Technical Report CS-2000-19, Department of Computer Science, University of Virginia, May 2000.

[161] J. Stankovic. VEST: a toolset for constructing and analyzing component based operating systems for embedded and real-time systems. In *Proceedings of the 1st International Conference on Embedded Software, (EMSOFT'01)*, volume 2211 of *Lecture Notes in Computer Science*, pages 390–402. Springer-Verlag, 2001.

[162] J. Stankovic, P. Nagaraddi, Z. Yu, and Z. He. Exploiting perscriptive aspects: A design time capability. In *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT'04)*, Pisa, Italy, September 2004. ACM Pess.

[163] J. Stankovic, S. Son, and J. Liebeherr. *Real-Time Databases and Information Systems*, chapter BeeHive: Global Multimedia Database Support for Dependable, Real-Time Applications, pages 409–422. Kluwer Academic Publishers, 1997.

[164] J. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis. VEST: an aspect-based composition tool for real-time systems. In *Proceedings of the 9th IEEE Real-Time Applications Symposium (RTAS'03)*, pages 58–69. IEEE Computer Society Press, 2003.

[165] J. A. Stankovic, K. Ramamritham, and D. Towsley. Scheduling in real-time transaction systems. Technical report, Department of Computer and Information Science, University of Massachusetts, 1991.

[166] D. B. Stewart, R. Volpe, and P. K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Transactions on Software Engineering*, 23(12), 1997.

[167] D. S. Stewart. Designing software components for real-time applications. In *Proceedings of Embedded System Conference*, San Jose, CA, September 2000. Class 408, 428.

[168] D. S. Stewart and G. Arora. Dynamically reconfigurable embedded software - does it make sense? In *Proceedings of IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 217–220. IEEE Computer Society Press, October 1996.

[169] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

[170] A. Tešanović, S. Nadjm-Tehrani, and J. Hansson. *Component-Based Software Development for Embedded Systems - An Overview on Current Research Trends*, volume 3778 of *Lecture Notes in Computer Science*, chapter Modular Verification of Reconfigurable Components, pages 59–81. Springer-Verlag, 2005.

[171] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Embedded databases for embedded real-time systems: a component-based approach. Technical report, Department of Computer Science, Linköping University, and Department of Computer Engineering, Mälardalen University, 2002.

[172] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Integrating symbolic worst-case execution time analysis into aspect-oriented software development. OOPSLA 2002 Workshop on Tools for Aspect-Oriented Software Development, November 2002.

[173] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Aspect-level worst-case execution time analysis of real-time systems compositioned using aspects and components. In *Proceedings of the 27th IFAC/IFIP/IEEE Workshop on Real-Time Programming (WRTP'03)*, Poland, May 2003. Elsevier.

[174] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Aspects and components in real-time system development: Towards reconfigurable and reusable software. *Journal of Embedded Computing*, 1(1), October 2004.

[175] A. Tešanović, K. Sheng, and J. Hansson. Application-tailored database systems: a case of aspects in an embedded database. In *Proceedings of the 8th IEEE International Database Engineering and Applications Symposium (IDEAS'04)*, pages 291–301. IEEE Computer Society, 2004.

[176] TimesTen Performance Software. TimesTen DB. `http://www.timesten.com`.

[177] Germany University of Karlsruhe. Compost. Documentation available at: `http://i44w3.info.uni-karlsruhe.de/~compost/`, June 2001.

[178] UPPAAL tool. `http://www.uppaal.com`, January 2005.

[179] R. van Ommering. Building product populations with software components. In *Proceedings of the 24th International Conference on Software Engineering*, pages 255–265. ACM Press, 2002.

[180] Anders Wall. *Architectural Modeling and Analysis of Complex Real-Time Systems*. PhD thesis, Mlardalen University, 2003.

[181] K. L. Wu, P. S. Yu, and C. Pu. Divergence control algorithms for epsilon serializability. *IEEE Transactions on Knowledge and Data Engineering*, 9(2):262–274, March-April 1997.

[182] Xerox Corporation. *The AspectJ Programming Guide*, September 2002. Available at: http://aspectj.org/doc/dist/progguide/index.html.

[183] M. Yannakakis. Perspectives on database theory. *SIGACT News*, 27(3):25–49, 1996.

[184] P. S. Yu and D. M. Dias. Impact of large memory on the performance of optimistic concurrency control schemes. In *Proceedings of the International Conference on Databases, Parallel Architectures, and Their Applications (PARBASE'90)*, pages 86–90. IEEE Computer Society Press, 1990.

[185] G. Zelesnik. *The UniCon Language User Manual*. Carnegie Mellon University, Pittsburgh, USA, May 1996. Available at `http://www.cs.cmu.edu/afs/cs/project/vit/www/unicon/reference-manual/`.

[186] Y. Zhu and F. Mueller. Feedback EDF scheduling exploiting dynamic voltage scaling. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, pages 84–93. IEEE Computer Society Press, 2004.

Index

absolute consistency, 35
ACCORD, 9, 61, 213
    development environment, 125
ACCORD-ME, 125, 179
ACID, 35
advice, 16
    after, 17, 170
    around, 17, 168
    before, 16, 103
AOSD, 5, 15, 42, 45, 53, 66
application aspects, 65, 68
architecture systems, 19
arrival time, 23
aspect, 15, 46, 68
    application, 65
    composition, 67
    connector, 86, 89
    join points, 15
    language, 15, 193
    pointcuts, 15
    policy, 86
    run-time, 67, 76, 129, 182
    weaver, 18, 46, 192
    weaving, 73, 76
aspect package, 84, 86
    COMET, 147
    concurrency control, 157, 174
    index, 163, 174
    QoS, 86, 165, 168, 174
aspect-level WCET analysis, 97
aspect-oriented software development, *see* AOSD, 45
aspect-oriented systems, 19
aspects
    FC-M, 166
    GUARD, 163
    HP-2PL, 157
    ODC, 159
    QMF, 166
assembly, 15

CBSD, 3, 13, 42, 51
COMET, 9, 146, 173, 182, 191
    admission control, 174
    aspect packages, 157
    aspects, 149, 157
    components, 147, 151
    configuration, 173, 182
    FC-M, 178, 185
    GUARD, 174
    HP-2PL, 173, 182, 184
    middleware layer, 91
    ODC, 174
    QMF, 178
    QOS, 174
    QoS, 184
    RM, 178
    STR, 178
component, 13, 42
    black box, 14, 42
    connector, 14, 43, 68
    domain, 14
    grey box, 42, 69
    inteface
        provided, 14
    intefaces
        analytical, 42
        constructive, 42

functional, 42
non-functional, 42
provided, 42
required, 42
interfaces, 13, 42
provided, 54
required, 54
middleware layer, 91, 94
white box, 15, 42
component views, 44
component-based software development, *see* CBSD
component-based systems, 18
composition aspects, 67
composition language, 20
composition operators, 20
concurrency control, 36
2V-DBP, 38
ARIES, 38
B-tree, 38
epsilon-based, 38
GUARD, 39, 163
HP-2PL, 37, 157
IM, 38
indexing CC, 38
OCC, 37
ODC, 159
priority inheritance, 37
RWPCP, 37
similarity-based, 38
timestamp-based, 37
versioning, 37
configuration, 15
dynamic, 49, 178
static, 47, 178
connector aspects, 89
QoS, 89
connectors, 43

DAG, 70
DBMS, 34, 202, 205
deadline, 22
deadline miss ratio, 107
direct acyclic graph, *see* DAG

ECU, 137, 146, 179
IECU, 138
VECU, 138
electronic control unit, *see* ECU
embedded
real-time, 22
embedded database, 34, 39, 145
commercial, 39
research, 40
embedded systems, 21, 76
execution time, 22
execution tree, 152

FC-M, 107, 166, 168, 178, 183
FCC, 86, 107, 165, 168
feedback control, 24
feedback loop, 107, 166
feedback-based QoS management, 27
FIFO, 75
formal methods, 25, 208

GME, 127
GUARD, 163, 174

HP-2PL, 157, 173

IECU, 138, 141
information hiding, 15, 42, 68
interfaces, 43
composition, 83
configuration, 82
provided, 82
required, 82

join points, 15, 160
jump table, 183

least square regression, 178

mechanisms, 69
memory, 34, 76
footprint, 34, 76
static, 76
memory footprint, 79
middleware layer, 91

ODC, 159, 174

path, 32
period, 23
pointcuts, 15
policy, 69
power consumption, 34
predictability, 23, 73

QAC, 86, 107, 165, 168
QMF, 166, 168, 178
QoS, 24, 49, 50, 66, 106, 165, 182, 184, 187, 192
  infrastructure, 84
QoS management
  feedback control-based, 27
  feedback-based, 27
QoS policy, 86, 87
  FC-M, 166, 168, 183
  least squares regression, 168
  QMF, 166, 168
  self-tuning, 168
quality attributes, 12
quality of service, *see* QoS

reachability analysis, 32, 114
real-time software, 21, 25, 61, 213
real-time systems, 21
  closed, 23, 145
  hard, 23
  open, 23
  soft, 23
reconfiguration
  dynamic, 49, 89
  static, 47, 89
reconfiguration locations, 72, 91, 109, 114, 160, 161, 191
relative consistency, 35
release time, 22
reuse context, 6, 83, 114
RM, *see* least square regression
RTCOM, 8, 68, 69, 91
  interfaces, 81
    composition, 83
    configuration, 82
    provided, 82
    required, 82
  mechanisms, 69
  policy, 69
Rubus, 138
run-time aspects, 67, 76, 129
run-time environment, 22, 25, 64, 67, 152, 183

scheduling, 24
  dynamic, 24
  earliest-deadline first, 24
  feedback control scheduling, 24
  rate-monotonic, 24, 37
  static, 24
self-tuning, 178
sensor component, 86
serializability, 36
signature, 16
  function, 16
  method, 16
software architecture, 11
static memory, 76
steady state, 24, 49, 106
STR, *see* self-tuning
symbolic transition, 32

task, 22, 43
  aperiodic, 23
  model, 22, 86, 87
    arrival time, 23
    deadlines, 22
    execution time, 22
    inter-arrival time, 23, 186
    period, 23
    release time, 22
  periodic, 23
  priority, 24
  sporadic, 23
task mapping, 44, 45, 105
TCTL, 33, 115
temporal consistency, 34, 35
temporal operators, 33

thread pool, 152
timed automata, *see* timed automaton
timed automaton, 30, 31
    clocks, 30, 31
    guards, 30
    locations, 30
    semantics
        operational, 31
        symbolics, 32
    transition, 30
transaction, 35, 110, 144
    model, 35, 157, 160, 167
transaction flow, 155
transient state, 24, 188

VECU, 138, 139

WCET, 64, 67, 76, 78, 95, 97, 182
    analysis, 97
        aspect-level, 97
        symbolic, 97
worst-case execution time, *see* WCET
    analysis, 25, 95

XML, 129

zone graph, 32, 116
    enriched, 119

Department of Computer and Information Science
Linköpings universitet

Dissertations

Linköping Studies in Science and Technology

No 14 **Anders Haraldsson:** A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.

No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.

No 18 **Mats Cedwall:** Semantisk analys av processbeskrivningar i naturligt språk, 1977, ISBN 91-7372-168-9.

No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.

No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System 1978, ISBN 91-7372-232-4.

No 51 **Erland Jungert:** Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.

No 54 **Sture Hägglund:** Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.

No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.

No 58 **Bengt Johnsson, Bertil Andersson:** The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.

No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.

No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.

No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91-7372-527-7.

No 94 **Hans Lunell:** Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.

No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.

No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation,1984, ISBN 91-7372-801-2.

No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372-805-5.

No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.

No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.

No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.

No 174 **Johan Fagerström:** A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.

No 192 **Dimiter Driankov:** Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.

No 213 **Lin Padgham:** Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.

No 214 **Tony Larsson:** A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.

No 221 **Michael Reinfrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.

No 239 **Jonas Löwgren:** Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.

No 244 **Henrik Eriksson:** Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.

No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies,1991, ISBN 91-7870-784-6.

No 258 **Patrick Doherty:** NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.

No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.

No 264 **Nils Dahlbäck:** Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.

No 265 **Ulf Nilsson:** Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.

No 270 **Ralph Rönnquist:** Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.

No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.

No 276 **Staffan Bonnier:** A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.

No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.

No 281 **Christer Bäckström:** Computational Complexity of Reasoning about Plans, 1992, ISBN 91-7870-979-2.

No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.

No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.

No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2.

No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.

No 338 **Simin Nadjm-Tehrani**: Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.

No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea/ABB 1968-1993, 1995, ISBN 91-7871-494-X.

No 375 **Ulf Söderman:** Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.

No 383 **Andreas Kågedal:** Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.

No 396 **George Fodor:** Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.

No 413 **Mikael Pettersson:** Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.

No 414 **Xinli Gu:** RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.

No 416 **Hua Shu:** Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.

No 429 **Jaime Villegas:** Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.

No 431 **Peter Jonsson:** Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.

No 437 **Johan Boye:** Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.

No 439 **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.

No 448 **Patrick Lambrix:** Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.

No 452 **Kjell Orsborn:** On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.

No 459 **Olof Johansson:** Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.

No 461 **Lena Strömbäck:** User-Defined Constructions in Unification-Based Formalisms,1997, ISBN 91-7871-857-0.

No 462 **Lars Degerstedt:** Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.

No 475 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.

No 480 **Mikael Lindvall:** An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.

No 485 **Göran Forslund**: Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.

No 494 **Martin Sköld**: Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.

No 495 **Hans Olsén**: Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.

No 498 **Thomas Drakengren:** Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.

No 502 **Jakob Axelsson:** Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.

No 503 **Johan Ringström:** Compiler Generation for Data-Parallel Programming Langugaes from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.

No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.

No 520 **Mikael Ronström:** Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.

No 522 **Niclas Ohlsson:** Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.

No 526 **Joachim Karlsson:** A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.

No 530 **Henrik Nilsson:** Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-x.

No 555 **Jonas Hallberg:** Timing Issues in High-Level Synthesis,1998, ISBN 91-7219-369-7.

No 561 **Ling Lin:** Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.

No 563 **Eva L Ragnemalm:** Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.

No 567 **Jörgen Lindström:** Does Distance matter? On geographical dispersion in organisations, 1999, ISBN 91-7219-439-1.

No 582 **Vanja Josifovski:** Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.

No 589 **Rita Kovordányi**: Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.

No 592 **Mikael Ericsson:** Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.

No 593 **Lars Karlsson:** Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.

No 594 **C. G. Mikael Johansson:** Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.

No 595 **Jörgen Hansson:** Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.

No 596 **Niklas Hallberg:** Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.

No 597 **Vivian Vimarlund:** An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.

No 598 **Johan Jenvald:** Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.

No 607 **Magnus Merkel:** Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.

No 611 **Silvia Coradeschi:** Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.

No 613 **Man Lin:** Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.

No 618 **Jimmy Tjäder:** Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.

No 627 **Vadim Engelson:** Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.

No 637 **Esa Falkenroth:** Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.

No 639 **Per-Arne Persson:** Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.

No 660 **Erik Larsson:** An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.

No 688 **Marcus Bjäreland:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.

No 689 **Joakim Gustafsson:** Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.

No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN-91-7373-126-9.

No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91 7373 207 9.

No 725 **Tim Heyer**: Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91 7373 208 7.

No 726 **Pär Carlshamre:** A Usability Perspective on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91 7373 212 5.

No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91 7373 258 3.

No 745 **Johan Åberg:** Live Help Systems: An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.

No 746 **Rego Granlund:** Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.

No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.

No 747 **Anneli Hagdahl:** Development of IT-supported Inter-organisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.

No 749 **Sofie Pilemalm:** Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-318-0.

No 765 **Stefan Holmlid:** Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.

No 771 **Magnus Morin:** Multimedia Representations of Distributed Tactical Operations, 2002, ISBN 91-7373-421-7.

No 772 **Pawel Pietrzak:** A Type-Based Framework for Locating Errors in Constraint Logic Programs, 2002, ISBN 91-7373-422-5.

No 758 **Erik Berglund:** Library Communication Among Programmers Worldwide, 2002, ISBN 91-7373-349-0.

No 774 **Choong-ho Yi:** Modelling Object-Oriented Dynamic Systems Using a Logic-Based Framework, 2002, ISBN 91-7373-424-1.

No 779 **Mathias Broxvall:** A Study in the Computational Complexity of Temporal Reasoning, 2002, ISBN 91-7373-440-3.

No 793 **Asmus Pandikow:** A Generic Principle for Enabling Interoperability of Structured and Object-Oriented Analysis and Design Tools, 2002, ISBN 91-7373-479-9.

No 785 **Lars Hult:** Publika Informationstjänster. En studie av den Internetbaserade encyklopedins bruksegenskaper, 2003, ISBN 91-7373-461-6.

No 800 **Lars Taxén:** A Framework for the Coordination of Complex Systems´ Development, 2003, ISBN 91-7373-604-X

No 808 **Klas Gäre:** Tre perspektiv på förväntningar och förändringar i samband med införande av informationsystem, 2003, ISBN 91-7373-618-X.

No 821 **Mikael Kindborg:** Concurrent Comics - programming of social agents by children, 2003, ISBN 91-7373-651-1.

No 823 **Christina Ölvingson:** On Development of Information Systems with GIS Functionality in Public Health Informatics: A Requirements Engineering Approach, 2003, ISBN 91-7373-656-2.

No 828 **Tobias Ritzau:** Memory Efficient Hard Real-Time Garbage Collection, 2003, ISBN 91-7373-666-X.

No 833 **Paul Pop:** Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems, 2003, ISBN 91-7373-683-X.

No 852 **Johan Moe:** Observing the Dynamic Behaviour of Large Distributed Systems to Improve Development and Testing - An Emperical Study in Software Engineering, 2003, ISBN 91-7373-779-8.

No 867 **Erik Herzog:** An Approach to Systems Engineering Tool Data Representation and Exchange, 2004, ISBN 91-7373-929-4.

No 872 **Aseel Berglund:** Augmenting the Remote Control: Studies in Complex Information Navigation for Digital TV, 2004, ISBN 91-7373-940-5.

No 869 **Jo Skåmedal:** Telecommuting's Implications on Travel and Travel Patterns, 2004, ISBN 91-7373-935-9.

No 870 **Linda Askenäs:** The Roles of IT - Studies of Organising when Implementing and Using Enterprise Systems, 2004, ISBN 91-7373-936-7.

No 874 **Annika Flycht-Eriksson:** Design and Use of Ontologies in Information-Providing Dialogue Systems, 2004, ISBN 91-7373-947-2.

No 873 **Peter Bunus:** Debugging Techniques for Equation-Based Languages, 2004, ISBN 91-7373-941-3.

No 876 **Jonas Mellin:** Resource-Predictable and Efficient Monitoring of Events, 2004, ISBN 91-7373-956-1.

No 883 **Magnus Bång:** Computing at the Speed of Paper: Ubiquitous Computing Environments for Healthcare Professionals, 2004, ISBN 91-7373-971-5

No 882 **Robert Eklund:** Disfluency in Swedish human-human and human-machine travel booking dialogues, 2004. ISBN 91-7373-966-9.

No 887 **Anders Lindström:** English and other Foreign Linquistic Elements in Spoken Swedish. Studies of Productive Processes and their Modelling using Finite-State Tools, 2004, ISBN 91-7373-981-2.

No 889 **Zhiping Wang:** Capacity-Constrained Production-inventory systems - Modellling and Analysis in both a traditional and an e-business context, 2004, ISBN 91-85295-08-6.

No 893 **Pernilla Qvarfordt:** Eyes on Multimodal Interaction, 2004, ISBN 91-85295-30-2.

No 910 **Magnus Kald:** In the Borderland between Strategy and Management Control - Theoretical Framework and Empirical Evidence, 2004, ISBN 91-85295-82-5.

No 918 **Jonas Lundberg:** Shaping Electronic News: Genre Perspectives on Interaction Design, 2004, ISBN 91-85297-14-3.

No 900 **Mattias Arvola:** Shades of use: The dynamics of interaction design for sociable use, 2004, ISBN 91-85295-42-6.

No 920 **Luis Alejandro Cortés:** Verification and Scheduling Techniques for Real-Time Embedded Systems, 2004, ISBN 91-85297-21-6.

No 929 **Diana Szentivanyi:** Performance Studies of Fault-Tolerant Middleware, 2005, ISBN 91-85297-58-5.

No 933 **Mikael Cäker:** Management Accounting as Constructing and Opposing Customer Focus: Three Case Studies on Management Accounting and Customer Relations, 2005, ISBN 91-85297-64-X.

No 937 **Jonas Kvarnström:** TALplanner and Other Extensions to Temporal Action Logic, 2005, ISBN 91-85297-75-5.

No 938 **Bourhane Kadmiry:** Fuzzy Gain-Scheduled Visual Servoing for Unmanned Helicopter, 2005, ISBN 91-85297-76-3.

No 945 **Gert Jervan:** Hybrid Built-In Self-Test and Test Generation Techniques for Digital Systems, 2005, ISBN: 91-85297-97-6.

No 946 **Anders Arpteg:** Intelligent Semi-Structured Information Extraction, 2005, ISBN 91-85297-98-4.

No 947 **Ola Angelsmark:** Constructing Algorithms for Constraint Satisfaction and Related Problems - Methods and Applications, 2005, ISBN 91-85297-99-2.

No 963 **Calin Curescu:** Utility-based Optimisation of Resource Allocation for Wireless Networks, 2005. ISBN 91-85457-07-8.

No 972 **Björn Johansson:** Joint Control in Dynamic Situations, 2005, ISBN 91-85457-31-0.

No 974 **Dan Lawesson:** An Approach to Diagnosability Analysis for Interacting Finite State Systems, 2005, ISBN 91-85457-39-6.

No 979 **Claudiu Duma:** Security and Trust Mechanisms for Groups in Distributed Services, 2005, ISBN 91-85457-54-X.

No 983 **Sorin Manolache:** Analysis and Optimisation of Real-Time Systems with Stochastic Behaviour, 2005, ISBN 91-85457-60-4.

No 986 **Yuxiao Zhao:** Standards-Based Application Integration for Business-to-Business Communications, 2005, ISBN 91-85457-66-3.

No 1004 **Patrik Haslum:** Admissible Heuristics for Automated Planning, 2006, ISBN 91-85497-28-2.

No 1005 **Aleksandra Tesanovic:** Developing Reusable and Reconfigurable Real-Time Software using Aspects and Components, 2006, ISBN 91-85497-29-0.

Linköping Studies in Information Science

No 1 **Karin Axelsson:** Metodisk systemstrukturering- att skapa samstämmighet mellan informa-tionssystemarkitektur och verksamhet, 1998. ISBN-9172-19-296-8.

No 2 **Stefan Cronholm:** Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998. ISBN-9172-19-299-2.

No 3 **Anders Avdic:** Användare och utvecklare - om anveckling med kalkylprogram, 1999. ISBN-91-7219-606-8.

No 4 **Owen Eriksson:** Kommunikationskvalitet hos informationssystem och affärsprocesser, 2000. ISBN 91-7219-811-7.

No 5 **Mikael Lind:** Från system till process - kriterier för processbestämning vid verksamhetsanalys, 2001, ISBN 91-7373-067-X

No 6 **Ulf Melin:** Koordination och informationssystem i företag och nätverk, 2002, ISBN 91-7373-278-8.

No 7 **Pär J. Ågerfalk:** Information Systems Actability - Understanding Information Technology as a Tool for Business Action and Communication, 2003, ISBN 91-7373-628-7.

No 8 **Ulf Seigerroth:** Att förstå och förändra systemutvecklingsverksamheter - en taxonomi för metautveckling, 2003, ISBN91-7373-736-4.

No 9 **Karin Hedström:** Spår av datoriseringens värden - Effekter av IT i äldreomsorg, 2004, ISBN 91-7373-963-4.

No 10 **Ewa Braf:** Knowledge Demanded for Action - Studies on Knowledge Mediation in Organisations, 2004, ISBN 91-85295-47-7.

No 11 **Fredrik Karlsson:** Method Configuration - method and computerized tool support, 2005, ISBN 91-85297-48-8.

No 12 **Malin Nordström:** Styrbar systemförvaltning - Att organisera systemförvaltningsverksamhet med hjälp av effektiva förvaltningsobjekt, 2005, ISBN 91-85297-60-7.

No 13 **Stefan Holgersson:** Yrke: POLIS - Yrkeskunskap, motivation, IT-system och andra förutsättningar för polisarbete, 2005, ISBN 91-85299-43-X.