Abstract

Increasing complexity of real-time systems and demands for enabling their configurability and tailorability are strong motivations for applying new software engineering principles such as aspect-oriented and component-based software development. The integration of these two techniques into real-time systems development would enable: (i) efficient system configuration from the components in the component library based on the system requirements, (ii) easy tailoring of components and/or a system for a specific application by changing the behavior (code) of the component by aspect weaving, and (iii) enhanced flexibility of the real-time and embedded software through the notion of system configurability and component tailorability.

In this thesis we focus on applying aspect-oriented and component-based software development to real-time system development. We propose a novel concept of aspectual component-based real-time system development (ACCORD). ACCORD introduces the following into real-time system development: (i) a design method that assumes the decomposition of the real-time system into a set of components and a set of aspects, (ii) a real-time component model denoted RTCOM that supports aspect weaving while enforcing information hiding, (iii) a method and a tool for performing worst-case execution time analysis of different configurations of aspects and components, and (iv) a new approach to modeling of real-time policies as aspects.

We present a case study of the development of a configurable real-time database system, called COMET, using ACCORD principles. In the

COMET example we show that applying ACCORD does have an impact on the real-time system development in providing efficient configuration of the real-time system. Thus, it could be a way for improved reusability and flexibility of real-time software, and modularization of crosscutting concerns.

In connection with development of ACCORD, we identify criteria that a design method for component-based real-time systems needs to address. The criteria include a well-defined component model for real-time systems, aspect separation, support for system configuration, and analysis of the composed real-time system. Using the identified set of criteria we provide an evaluation of ACCORD. In comparison with other approaches, ACCORD provides a distinct classification of crosscutting concerns in the real-time domain into different types of aspects, and provides a real-time component model that supports weaving of aspects into the code of a component, as well as a tool for temporal analysis of the weaved system.

Keywords: aspect-oriented software development, component-based software development, real-time systems, embedded systems, database systems, aspects, components, worst-case execution time

Acknowledgments

There are so many people that have helped in my professional and personal development that I find myself overwhelmed with the gratitude I feel and the ability to express it in words. Therefore, whatever I write cannot truly represent the depth of my gratitude, and the following lines are merely an attempt to capture at least a fragment of my thankfulness.

What I owe to my supervisor, Dr. Jörgen Hansson, is beyond what I can fit in this tiny space, and, in fact, to say my praise to him would require writing of a book on its own. He was there for me in each step of the way, providing guidance, support, and encouragement. Without him, this thesis would not be a reality.

The research presented in this thesis has been conducted under the COMET project umbrella, which encompasses two universities (Linköping and Mälardalen), and involves two influences that mold my research: Prof. Christer Norström, my secondary supervisor at Mälardalen University, and Dag Nyström, peer doctoral student. Christer helped immensely in providing constructive feedback on my work, and assisting in formulating and formalizing my fuzzy ideas. I am indebted to him for teaching me some of most intricate parts of the Swedish language (which are still a mystery to most Swedish people). Dag and I have had many inspiring and spirited discussions that generated volcanic irruptions of ideas, which have been a true corner stone and a building material for the research within the COMET project.

I am fortunate to have Dr. Simin Nadjm-Tehrani on my advisory team, who provided feedback on my work, and supported me gracefully

through these past few years of my PhD studies at RTSLAB.

Every administrative matter I touch turns into a nightmare, and without Anne Moe and Gunilla Mellheden I would simply be lost in the sea of messy administrative issues.

Lillemor Wallgren has been the very first person to help me in my voyage to the graduate degree; she was the first person I ever contacted at Linköping University, and has, since then, assisted me in all intricate administrative matters related to graduate studies. Lillemor and Bodil Carlsson were a great help when putting this thesis into a publication.

Dr. Nataša Gospić was my guide in the undergraduate education at Banjaluka University and she continues to be my inspiration. I am also grateful to Prof. Petar Hinić for his support and encouragement during my graduate education at Banjaluka University.

I wish to express my gratitude to all the members of RTSLAB who, with their versatile backgrounds and personalities, provided an enjoyable, lively, and interesting working environment.

I would not be able to accomplish this much without unconditional love and support from my family. I am grateful to my parents (for always believing in me), my sister (for being the best sister in the world), my wonderful grandparents, aunts, uncles, ... Most importantly, I am grateful to my husband Goran for being the sun in my life shining on me at all times, giving me energy to work, and keeping me warm and happy.

Hvala
Aleksandra

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 project has resulted in the following technical reports and published papers (the list also contains descriptions of the content of each paper, relating it to the other papers, and the role of the doctoral students in contributing to the papers).

Embedded Databases for Embedded Real-Time Systems: a Component-Based Approach
Aleksandra Tešanović, Dag Nyström, Jörgen Hansson, and Christer Norström
Technical report, Dept. of Computer Science, Linköping University, and Dept. of Computer Engineering, Mälardalen University, ISSN 1404-3041 ISRN MDH-MRTC-43/2002-1-SE, January 2002

This technical report is a survey of the state of the art in the areas of (i) real-time database systems, (ii) embedded database systems, (ii) component-based software engineering, (iii) component-based database systems, and (iv) component-based real-time and embedded systems. The report represents the background study for the overall

COMET project.
The survey of real-time and embedded databases was conducted by Dag Nyström, while the survey of component-based software engineering and its application on database, embedded, and real-time systems was conducted by Aleksandra Tešanović.

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 Euromicro International Conference on Real-Time Systems, pages 249-256, Vienna, Austria, 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 writing of this case study paper, was made possible by Nils-Erik Bånkestad. Tešanović and Nyström investigated two different real-time systems, one each. Additionally, Tešanović studied the impact of current data management in both systems, and Nyström proposed a way to integrate a real-time database in the existing 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 the aspect-oriented software systems.
Tešanović developed a way of representing temporal information of aspects and an algorithm that enable worst-case execution time analysis of aspect-oriented systems. Tešanović, together with Nyström, developed a way of specifying worst-case execution time of components.

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), Tainan City, Taiwan, Springer-Verlag, February 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, space and resource management constraints, and composability.
The main ideas and contributions of the paper are developed by Aleksandra Tešanović.

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), Tainan City, Taiwan, Springer-Verlag, February 2003

This paper introduces the concept of database pointers, which enable a fast and predictable way of accessing data in a database without the need of consulting the indexing system of a database. Database pointers allow fast and predictable accesses of data without violating temporal and logical consistency, and transaction serialization.
The main ideas and contributions of the paper are developed by Dag Nyström.

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, Poland, 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.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter is outlined as follows. In section 1.1 we motivate the need for applying new software engineering principles in real-time systems development. Our research goals are formulated in section 1.2, followed by the main contributions of this thesis in section 1.3, and our vision of the long term goals in section 1.4. Finally, section 1.5 gives a description of the thesis structure.

1.1 Motivation

Real-time and embedded systems are widely used in the modern society of today. However, successful deployment of embedded and real-time systems depends on low development costs, high degree of tailorability and quickness to market [94]. Thus, the introduction of the component-based software development (CBSD) into real-time and embedded systems development offers significant benefits, namely:

- configuration of embedded and real-time software for a specific application using components from the component library, thus reducing the system complexity as components can be chosen to provide the functionality needed by the system;
- rapid development and deployment of real-time software as many

software components, if properly designed and verified, can be reused in different embedded and real-time 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 aspects of real-time and embedded systems that cannot be encapsulated in a component with well-defined interfaces as they crosscut the structure of the overall system, e.g., synchronization, memory optimization, power consumption, and temporal attributes. Aspect-oriented software development (AOSD) [45] 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. Applying AOSD in real-time and embedded system development would reduce the complexity of the system design and development, and, thus, provide means for a structured and efficient way of handling crosscutting concerns in a real-time software system.

Hence, the integration of the two disciplines, CBSD and AOSD, into real-time systems development would enable: (i) efficient system configuration from the components and aspects from the library based on the system requirements, (ii) easy tailoring of components and/or a system for a specific application, i.e., reuse context by changing the behavior (code) of the component by applying aspects. This results in enhanced flexibility of the real-time and embedded software through the notion of system configurability and component tailorability. However, due to specific demands of real-time systems, applying AOSD and CBSD to real-time system development is not straightforward. For example, to be able to apply AOSD or CBSD in real-time system development, we need to provide methods for analyzing temporal behavior of individual aspects and components as the development process of real-time systems has to be based on a software technology that supports predictability in the time domain. Furthermore, if we want to use both AOSD and CBSD in real-time system development, we need to provide methods for efficient

temporal analysis of different configurations of components and aspects, i.e., components weaved with aspects. Additionally, CBSD assumes a component to be a black box, where internals of components are not visible, while AOSD promotes white box components, i.e., the entire code of the component is visible to 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.

1.2 Research Challenges

To successfully apply software engineering techniques such as AOSD and CBSD when developing real-time systems, a number of research challenges need to be addressed. In this thesis we focus on the following:

- issues that should be addressed and the criteria that should be enforced by a design method to allow integration of the two software engineering techniques into real-time systems;
- characteristics of a component model that can capture and adopt principles of the CBSD and AOSD in a real-time and embedded environment;
- relationship between components and tasks in real-time systems, with a focus on questioning the traditional view of real-time systems as systems composed out of tasks only; and
- methods and tools for temporal analysis of different configurations of aspects and components in a real-time system.

Investigating and resolving these issues would enable successful application of the ideas and notions from software engineering approaches, namely AOSD and CBSD, to the real-time system development.

1.3 Research Contributions

Our main contributions can be summarized as follows.

- **A novel concept of aspectual component-based real-time system development (ACCORD)**. Through the notion of aspects and components, ACCORD enforces the divide-and-conquer approach to complex real-time system development. ACCORD supports a decomposition process with the following two sequential phases: (i) decomposition of the real-time system into a set of components and a set of aspects, corresponding to the structural view of the components and the real-time system, and (ii) structuring of tasks, corresponding to the temporal view of the components and the real-time system.

- **A real-time component model** denoted RTCOM that describes what a real-time component, supporting different types of aspects and enforcing information hiding, should look like.

- **A method and a tool for worst-case execution time analysis** of different configurations of aspects and components.

- **A set of criteria for designing component-based real-time systems**, including: (i) a real-time component model that supports mapping of components to tasks, (ii) separation of concerns in real-time systems in different types of aspects, and (iii) composition support, namely support for configuration and analysis of the composed real-time software.

We developed ACCORD with hard real-time systems in mind; the approach is also general enough to be used for building both firm and soft real-time systems. In this thesis we focus only on hard real-time systems, and present a case study that shows how ACCORD can be applied to the design and development of a configurable embedded (hard) real-time database, called COMET. Using the COMET system as an example, we introduce an alternative way of handling concurrency in a real-time database system, where concurrency is modeled as an aspect crosscutting the overall system.

1.4 Goals

The work presented in this thesis is part of the long term goals we envision fulfilling in the future. Therefore, it is valuable to illustrate the long term goals clearly and to express the current contributions in terms of the extent to which we came in fulfilling these goals. Our long term goals are the following.

- The complete ACCORD method for aspectual real-time system development should:
  - follow the development cycle of real-time systems from the requirements specification to the implementation and verification of the system on the target platform,
  - provide a formalized design framework to exert sound system design cycle, and
  - provide automated tool support for each step of the design process.

  We want to verify the method by fully applying it to the development of a real-world real-time system.
- The component model should provide support for: (i) run-time environment of real-time systems, including support for specifying the resource, temporal, and memory requirements of the component for the target run-time environment, (ii) aspect weaving into the component code, (iii) formal specification and verification of component properties, e.g., worst-case execution time, (iv) composition process in terms of rules for component composition and interaction, and (v) generalizing to other application domains.

We have developed ACCORD and RTCOM and, in the current stage of their development, they have the following characteristics.

- ACCORD follows the development cycle from system design to the verification. The ACCORD design method, in its current form, is

not fully formal, and the ACCORD development process partially addresses the verification of the system by providing a support for temporal analysis of the system.

We have applied ACCORD to the COMET system development. While the current COMET implementation does not support all ACCORD notions, it provides a good experimental platform for judgment of ACCORD success so far.

- RTCOM supports: (i) a subset of temporal requirements, (ii) functional requirements in the form of different types of functional interfaces, (iii) aspect weaving into component code, and (iv) semi-formal specification. However, it does not have support for composition rules and generalization to other application domains.

1.5 Thesis Outline

The thesis has the following outline. Chapter 2 gives the background and defines the basic terminology used throughout the thesis. In chapter 3 we provide a set of criteria that a design method for the component-based real-time systems should fulfill. In chapter 4 we introduce the basic constituents of ACCORD, including a design method, a real-time component model, and a method for analyzing temporal behavior of the systems built on the ACCORD concept. We provide an evaluation of ACCORD based on the previously identified criteria. Chapter 5 presents the application of ACCORD to development of the COMET system. Chapter 6 contrasts ACCORD with current state of the art research in component-based and aspect-oriented real-time and database development. Finally, contributions and conclusions, together with directions for future research, are presented in chapter 7.

Chapter 2

Background

This chapter introduces the terminology used throughout the thesis. First, a background to real-time systems is presented in section 2.1. Basic notions in component-based and aspects-oriented development are introduced in sections 2.2, and 2.3, respectively. Main differences between components in component-based and aspect-oriented software development are discussed in section 2.4. In section 2.5 we give an overview of software engineering techniques that primarily focus on software composition, thus giving a preview of the future trends in software engineering. This work has been carried out as a part of the COMET project in which a configurable embedded real-time database has being built. Therefore, the chapter concludes with a discussion on embedded real-time database systems in section 2.6.

2.1 Real-Time and Embedded Systems

Digital systems can be classified in two categories: general-purpose systems and application-specific systems [41]. 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 can also be part of a larger host system and perform specific functions within the host system [22], and

such systems are usually referred to as *embedded systems*. An embedded system is implemented partly on software and partly on hardware. When standard microprocessors, micro-controllers or DSP processors are used, specialization of an embedded system for a particular application consists primarily on 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 the computer to interact with the external world (environment). They need to monitor sensors and control actuators for a wide variety of real-world devices. These devices interface to the computer via input and output registers and their operational requirements are device and computer dependent.

Most embedded systems are also real-time systems, i.e., the correctness of the system depends both on the logical result of the computation, and the time when the results are produced [92]. We refer to these systems as *embedded real-time systems*[1]. Embedded real-time systems are the focus of this thesis and, if not otherwise specified, when referring to real-time systems, we refer to embedded real-time systems.

In the last years the development and deployment of embedded and real-time systems has increased dramatically. Below follows a list of examples where embedded real-time systems can be found [21].

- Vehicle systems for automobiles, subways, aircrafts, railways, and ships.
- Traffic control for highways, airspace, railway tracks, and shipping lines.
- Process control for power plants and chemical plants.
- Medical systems for radiation therapy and patient monitoring.

[1]We distinguish between embedded and real-time systems, since there are some embedded systems that do not enforce real-time behavior, and there are real-time systems that are not embedded.

- Military uses such as advanced firing weapons, tracking, and command and control.
- Manufacturing systems with robots.
- Telephone, radio, and satellite communications.
- Multimedia systems that provide text, graphic, audio and video interfaces.
- Household systems for monitoring and controlling appliances.
- Building managers that control such entities as heat, lights, doors, and elevators.

Real-time systems are typically constructed out of concurrent programs, called tasks. The most common type of temporal constraint that a real-time system must satisfy is the completion of task deadlines. Depending on the consequence due to a missed 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 control, while in a *soft real-time system*, missing a deadline does not cause catastrophic damage to the system, but may affect performance negatively.

In real-time systems it is necessary to specify the order in which tasks should execute to ensure that tasks meet their respective deadlines. The process of determining the order in which tasks should execute is known as scheduling [21, 51]. Scheduling enables real-time system designers to predict behavior of a real-time system, i.e., make the system predictable, by ensuring that all tasks fulfill their execution requirements and meet their deadlines. Scheduling theory greatly depends on the ability to measure or estimate the amount of CPU time tasks require for execution [20]. Typically, scheduling tests for tasks in real-time systems require that the worst-case execution time (WCET) of a task is known. The execution needs of a task can be obtained either by [20, 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 WCET 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 relying on the model of the underlying hardware. The model used in the second method represents the 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 [82]. The estimated WCET should be as tight as possible in order to make a real-time system as predictable as possible.

2.1.1 Symbolic Worst-Case Execution Time Analysis

As mentioned, one of the most important elements in real-time system development is temporal analysis of the real-time software. Determining the WCET of the code provides guarantees that the execution time does not exceed the WCET bound. WCET analysis is usually done on two levels [82]: (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.

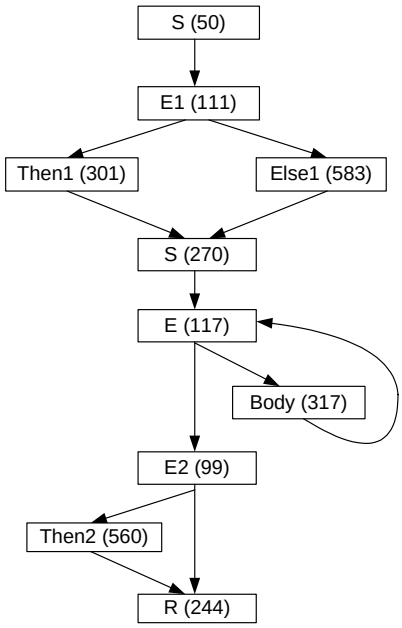
Symbolic WCET addresses the problem of obtaining the high-level tight estimate of the WCET by characterizing the context in which code is executed [13]. Hence, the symbolic WCET technique describes the WCET as a symbolic expression, rather than a fixed constant.

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 2.1(a) (the example is adopted from [13]). The function computes the `n`-th power of a float number `f`. If `n` is negative the function computes $\frac{1}{\texttt{f}^{\texttt{abs(n)}}}$.

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 2.1(a)). Then, the WCET of the function is determined by adding the execution times of all sections of code, look-

```
1  double power(double fkn, int e) {
2      double res = 0;
3      int times = 0;
4      bool pos = true;
5
6      if e < 0 then
7          times = -e;
8      else
9          pos = false;
10         times = e;
11     end if
12     res = 1.0;
13
14     for(i in 1 ... times loop
15         res = res * fkn;
16     end loop
17
18     if(pos)
19         res = 1 / res;
20
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 2.1: The `power` function

ing for the longest path in conditional branches. Figure 2.1(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). For `n` in range [-10,10] and 10 as the maximum number of loop iterations, adding the execution times of code sections (given in figure 2.1(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 it holds dead paths, i.e., 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 algebraic expression. In this case, the WCET of the `power` function can be formulated as a function of the exponent, denoted e, as follows:

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

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

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

The WCET is maximal for $e = -10$, and 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. When the call is made, based on the passed value of the exponent e, the symbolic expression is evaluated and the tight bound on the execution time is obtained.

2.2 Component-Based Software Development

The need for transition from monolithic to open and flexible systems has emerged due to 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 [19]. Component-based software development (CBSD) is an emerging development paradigm that enables this transition by allowing systems to be assembled from a pre-defined set of components explicitly developed for multiple usages. Developing systems out of existing components offers many advantages to developers and users. In component-based systems [19, 27, 29, 36]:

- Development costs are significantly decreased because systems are built by simply plugging in existing components.
- System evolution is eased because system built on CBSD concepts is open to changes and extensions, i.e., components with new functionality can be plugged into an existing system.
- Quality of software is increased since it is assumed that components are previously tested in different contexts and have validated behavior at their interfaces. Hence, validation efforts in these systems have to primarily concentrate on validation of the architectural design.
- Time-to-market is shortened since systems do not have to be developed from scratch.
- Maintenance costs are reduced since components are designed to be carried through different applications and, thus, changes in a component are beneficial to multiple systems.

As a result, efficiency in the development for the software vendor is improved and flexibility of delivered product is enhanced for the user [52].

Component-based development also raises many challenging problems, such as [52]:

- Building good reusable components. This is not an easy task and a significant effort must be invested to produce a component that can be used in different software systems. In particular, components must be tested and verified to be eligible for reuse.
- Composing a reliable system out of components. A system built out of components is in risk of being unreliable if inadequate components are used for the system assembly. The same problem arises when a new component needs to be integrated into an existing system.
- Verification of reusable components. Components are developed to be reused in many different systems, which makes the component verification a significant challenge. For every component use, the developer of a new component-based system must be able to verify the component, i.e., determine if the particular component meets the needs of the system under construction.
- Dynamic and on-line configuration of components. Components can be upgraded and introduced at run-time; this affects the configuration of the complete system and it is important to keep track of changes introduced in the system.

2.2.1 Software Component

Software components are the core of CBSD. However, different 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, e.g., performance, real-time, and reliability [19]. In systems where COM [64] is used as a component framework, a component is generally assumed to

be a self-contained binary package with precisely defined standardized interfaces [67]. Similarly, in the CORBA component framework [71], 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 standardized) component interfaces [32].

Hence, there is no common definition of a component for every component-based system. The definition of a component clearly depends on the implementation, architectural assumptions, and the way the component is to be reused in the system. However, all component-based systems have one common fact: *components are for composition* [99].

While frameworks and standards for components today primarily focus on CORBA, COM, or JavaBeans, the need for component-based development has 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 [37, 63].

Common for all types of components, independent of their definition, is that they communicate with its environment through well-defined interfaces, e.g., in COM and CORBA interfaces are defined in an interface definition language (IDL), Microsoft IDL and CORBA IDL. Components can have more than one interface. For example, a component may have three types of interfaces: provided, required, and configura-

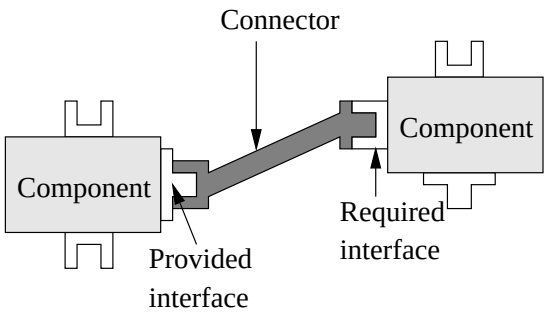


Figure 2.2: Components and interfaces

tion interface [19]. 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., software engineer (developer) who is constructing an application out of 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. When using components in an application there might be 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 components (see figure 2.2).

Independently of application area, a software component is normally considered to have *black box* properties [36, 32]: each component sees only interfaces to other components, thus, internal state and attributes of the component are strongly encapsulated.

Every component implements some field of functionality, i.e., a domain [19]. 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, analyze its suitability for a certain software product, etc. [19]. 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.

2.2.2 Software Architecture

Generally, every software system has an architecture, although it may not be explicitly modeled [62]. The software architecture represents a high level of abstraction where a system is described as a collection of interacting components [6]. A commonly used definition of software architecture is [12]:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, 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 [62]. Design of the software architecture is the first step in the design process of a software product, right after the specification of the system's requirements. Hence, it allows early system testing, that is, as pointed out by Bosch [19], assessment of design and quality attributes of a system. Quality attributes of a system are those that are relevant from the software engineering perspective, e.g., maintainability, reusability, and those that represent quality of the system in operation, e.g., performance, reliability, robustness, fault-tolerance. These quality attributes of a system could be further classified as functional (see table 2.1) and non-functional quality attributes (see table 2.2), and used for architectural analysis [105].

The software architectural design process results in component quality requirements, as well as several constraints and design rules components must obey, e.g., means of communication [19]. Hence, developing reusable components depends on the software architecture, since the software architecture to a large extent influences functionality and quality attributes of a component [19]. Thus, the software architecture represents the effective basis for reuse of components [78]. Moreover, there are indications that software evolution and reuse is more likely to receive higher payoff if architectures and designs can be reused and can guide low-level component reuse [60, 66].

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 does not endanger human life or the environment.
Temporal constraints	The real-time attributes such as deadlines, jitter, response time, worst-case execution time, etc.
Security	The ability of a software system to resist malicious intended actions.
Availability	The probability of a system functioning correctly at any given time.

Table 2.1: Functional quality attributes

Quality attribute	Description
Testability	The ability to easily prove correctness of a system by testing.
Reusability	The extent to which the architecture 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 2.2: Non-functional quality attributes

During the architectural design phase of the system development, the issue of handling conflicts in quality requirements should be explicitly addressed, since a solution for improving one quality attribute may affect other quality attributes negatively, e.g., reusability and performance are considered to be contradicting, as are fault-tolerance and real-time computing [19]. For example, consider a software system fine-tuned to meet extreme performance requirements. In such a system, a component is also optimized to meet specific performance requirements. Hence, reusing the same component in a different system, with different performance requirements, could result in degraded performance of the newly developed system. In such a scenario, additional efforts have to be made to ensure that the component and, thus, the system under construction meet the initial performance requirements.

2.3 Aspect-Oriented Software Development

Aspect-oriented software development (AOSD) has emerged as a new principle for software development, and is based on the notion of separation of concerns [45]. Typically, AOSD implementation of a software system has the following constituents:

- components, written in a component language, e.g., C, C++, and Java;
- aspects, written in a corresponding aspect language[2], e.g., AspectC [25], AspectC++ [91], and AspectJ [100] developed for Java; and
- an aspect weaver, which is a special compiler that combines components and aspects.

Components used for system composition in AOSD are not black box components (as they are in CBSD), rather they are *white box* components. A white box component is a piece of code, e.g., program, function, and method, completely accessible by the component user. White box

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

components do not enforce information hiding, and are fully open to changes and modifications of their internal structure. In AOSD one can modify the internal behavior of a component by weaving different aspects into the code of the component.

Aspects are commonly considered to be a property of a system that affect its performance or semantics, and that crosscuts the functionality of the system [45]. 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.

In existing aspect languages, each 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 component code where aspects should be weaved, e.g., a method, a type (struct or union). Figure 2.3 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[3].

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

Figure 2.3: An example of the pointcut definition

An *advice* is a declaration used to specify the code that should run when the join points, specified by a pointcut expression, are reached. Different kinds of advices can be declared, such as: (i) *before advice*, which is executed before the join point, (ii) *after advice*, which is executed immediately after the join point, and (iii) *around advice*, which is executed in place of the join point. Figure 2.4 shows an example of an

[3]The example presented is written in AspectC++.

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

Figure 2.4: An example of the advice definition

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.

2.4 Components vs. Aspects

The goal of this section is to clarify the notion of a component in CBSD and AOSD with a particular focus on abstraction metaphors: a white box and a black box component.

While CBSD uses black box as an abstraction metaphor for the components, the AOSD uses 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.

The main motivation and the main benefits of CBSD overlap and complement the ones of AOSD. Furthermore, making aspects and aspect weaving usable in CBSD would allow improved flexibility in tailoring of components and, thus, enhanced reuse of components in different systems. To allow aspects to invasively change the component code and still preserve information hiding to the largest extent possible requires

opening a black box component. This, in turn, implies using the gray box abstraction metaphor for the component. The gray box component preserves some of the main features of a black box component, such as well-defined interfaces as access points to the component, and it also allows aspect weaving to change the behavior and the internal state of the component.

2.5 From Components to Composition

Research in the component-based software engineering community increasingly emphasizes composition of the system as the way to enable development of reliable systems, and the way to improve reuse of components. In this section we give an overview of the software engineering techniques primarily focusing on system composition. Figure 2.5 provides hierarchical classification of composition-oriented approaches [10].

Component-based systems on the first level, e.g., CORBA, COM and JavaBeans, represent the first generation of component-based systems, and are referred to as "classical" component-based systems [10]. 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 component-based systems lack rules for the system composition, i.e., composition recipe.

The next level represents architecture systems, e.g., RAPIDE [59] and UNICON [106]. These systems provide an architectural description language (ADL), which is 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 [6], and a connector is a specific module that encapsulates the communication between application-specific components. This gives significant advancement in the composition compared to classical component-based systems, since communication and the architec-

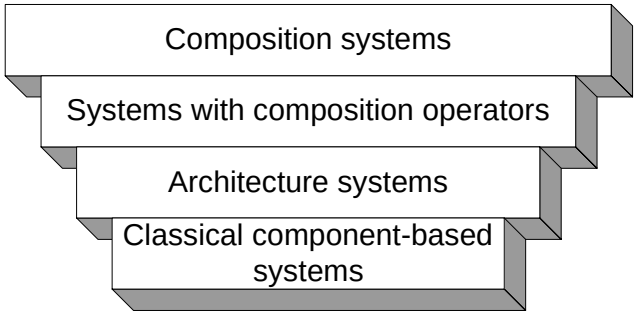


Figure 2.5: Classes of component-based systems

ture 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, adopted in the industry, can be viewed as a subset of architecture systems (which are not yet adopted by the industry), as they are in fact simple architecture systems with fixed communication.

The third level represents aspect systems that are developed using the AOSD principles [45]. Aspect 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 [15]. Only recently, several projects 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 [81]. The projects include ARIES [2], ISIS PCES [3], and FACET [4]. In aspect systems, aspects are separated from core components; they are recombined automatically through weaving. In AOSD, a core component is considered to be a unit of system functional decomposition, i.e., application-specific functionality [45]. Weavers are special compilers that combine aspects with core components at so-called joint points either statically (at compile time) or dynamically (at run-time). Weaving breaks the core component (at joint points) and cross-cuts aspects into the component and the weaving process results in an integrated component-based system. Hence, core components are no longer black boxes; rather they are white boxes as they are cross-cut with aspects. However, aspect weavers can be viewed as black boxes since they are written for a specific combination of aspects and core components, and for each new combination of aspects and core components 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 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 systems. Having different aspects improves reusability since various aspects can be combined (reused) with different core components. The main drawback of aspect systems is that they are built on special languages for aspects, requiring system developers to learn these languages.

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 composable out of components, and can be re-composed for every combination of aspects and components, further improving the reuse. Subject-oriented programming (SOP) [73], 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 weav-

ing 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.

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 [10] is one approach that aims to provide a language for the system composition, and here components may consist of a set of arbitrary program elements, and are called boxes [10]. 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 [104]. 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. Also, COMPOST is language-dependent as it only supports Java source-to-source transformations.

2.6 Embedded Real-Time Database Systems

The amount of data that needs to be managed by real-time and embedded systems is increasing, e.g., the amount of information maintained by real-time systems controlling a vehicle is increasing with the rate of 7-10% per year [24]. Hence, database functionality suitable for embedded and real-time systems is needed to provide efficient support for storage and manipulation of data.

Embedding databases into embedded systems have significant gains: (i) reduction of development costs due to the reuse of database systems; (ii) 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 (iii) increased maintainability as the software evolves. Consequently, this improves the overall reliability of the system. Furthermore, embedded databases provide mechanisms that support porting of data to other embedded systems or large central databases.

Naturally, embedded real-time systems put demands on such an embedded database that originate from requirements on embedded and real-time systems. For example, 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 [69]. Also, the resource load the database imposes on the embedded system should be carefully balanced, which includes memory footprint and power consumption. For example, in embedded systems used to control a vehicle minimization of the hardware cost is of utmost importance. This usually implies that memory capacity must be kept as low as possible, i.e., 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 put different set of demands on a database system. The data in the database used in real-time sys-

tems must be logically consistent, as well as temporally consistent [83]. 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 components:

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

We use the notation introduced by Ramamritham [83] to give a formal definition of temporal consistency.

A data element, denoted d, which is temporally constrained, is defined by three attributes:

- value d_{value}, i.e., the current state of data element d in the database,
- time-stamp d_{ts}, i.e., the time when the observation relating to d was made, and
- absolute validity interval d_{avi}, i.e., the length of the time interval following d_{ts} 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 [83]

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

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 [89]. 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) [83]. 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 [57].

There are many embedded databases on the market, but they vary widely from vendor to vendor. Existing commercial embedded database systems, e.g., Polyhedra [80], RDM and Velocis [61], Pervasive.SQL [79], Berkeley DB [90], and TimesTen [103], have different characteristics and are designed with specific applications in mind. They support different data models, e.g., relational vs. object-relational model, and different operating system platforms [101]. Moreover, they have different memory requirements and provide different 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 configurability of software architectures. Research projects that are building real-time database platforms, such as ART-RTDB [46], BeeHive [96], DeeDS [8] and RODAIN [54], have monolithic structure, and are built for a particular 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

Component-Based Real-Time Systems Design Criteria

This chapter presents a set of criteria for evaluating existing, and developing new, design methods for building component-based real-time systems. We identify the main issues addressed by design approaches developed by the real-time and software engineering research community, respectively. These help us form a common ground for the criteria that any design method for component-based real-time systems should fulfill.[1]

Since the component-based paradigm is a product of the advancements in the software engineering research community, to efficiently apply these ideas to real-time system development, it is essential to first identify the issues considered to be of prime importance when developing a general-purpose component-based software system (section 3.1), and contrast those with the most relevant issues addressed by the design approaches in the real-time community (section 3.2). We show (section 3.3) that there is a gap between the approaches from different communities as the real-time community has focused primarily on the real-time issues not exploiting modularity of software to the extent that the software en-

[1]We use italics to highlight the specific issues stressed by a particular design method.

gineering community has done. Hence, we present the first uniform set of criteria for the design of component-based real-time systems, and discuss the extent to which design approaches investigated fulfill the criteria. The criteria include component model (section 3.4), aspect separation (section 3.5), and system composability (section 3.6). The chapter finishes (section 3.7) with a summary and an observation that an existing design method specifically addressing all the identified criteria is missing, and a new design method fulfilling the criteria is needed.

3.1 Software Engineering Design Methods

The design methods for general-purpose software systems in software engineering community are mostly targeted towards a *component model*. The component model typically enables enforcement of the information hiding criterion [76] as components are considered to be black boxes that communicate with each other, or the environment, through well-defined *interfaces* [99]. This is a component view taken by the first generation of component-based systems, e.g., COM and CORBA. Both COM and CORBA provide a standardized component model with an interface definition languages, but lack support in composability of different system configurations [33, 74].

Some design approaches [100, 10, 91, 25, 7] have taken one step further in software configurability by providing support for *aspects* and *aspect weaving*, thus adopting the main ideas of aspect-oriented software development. A typical representative of programming languages that explicitly provide ways for specifying aspects is AspectJ [100]. It is accompanied by powerful configuration tools for development of software systems using aspects written in AspectJ and components written in Java. However, AspectJ is limited as it supports only the notion of white box components, i.e., components are not encapsulated and their behavior is visible to other parts of the system, thus not exploiting the powers of information hiding. Invasive software composition (ISC) [10] overcomes the drawbacks of pure aspect language approaches, and enforces information hiding by having a well-defined component model,

called the box. The box is a general component model supporting two types of interfaces: explicit interfaces and implicit interfaces. Explicit interfaces are used for inter-component communication, while implicit interfaces are used for aspect weaving into the code of components. Here, components can be viewed as gray boxes in the sense that they are encapsulated, but still their behavior can be changed by aspect weaving.

3.2 Real-Time Design Methods

There are several established design methods developed by the real-time community. We focus on a few representative approaches [39, 40, 50, 35, 20] to illustrate the types of requirements that a real-time domain places on a design method.

DARTS

DARTS [39], a design approach for real-time systems, and ADARTS [40], its extension for Ada-based systems, focus on the decomposition of real-time systems into tasks. DARTS emphasizes the need for having *task structuring criteria* that could help the designer to make the transition from tasks to modules. Modules in DARTS typically represent traditional functions. DARTS partially enforces information hiding through two different types of task interfaces: task communication and task synchronization interface. Decomposition of real-time systems using DARTS is done by decomposing a system into tasks that are then grouped into software modules. Hence, the method offers 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 (usually task-oriented) function. Configuration of DARTS-based systems is done using *configuration guidelines* that are very clear and have been refined over the years, especially through their use in industry. However, it has been recognized that DARTS does not provide mechanisms for checking and verifying temporal behavior of the system under development [50, 20].

TRSD

Kopetz et al. [50] introduced a method for real-time system design, which is a transaction-oriented approach to real-time system development. For the sake of presentation clarity, we refer to this method as TRSD, a transactional real-time system design . Building blocks of a real-time system are transactions consisting of one or several tasks. A transaction, in this context a group of tasks, 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 TRSD method is complete in the sense that it follows the design cycle of real-time systems from specification to the implementation and verification. Moreover, TRSD goes one step further than DARTS by providing support for temporal analysis of the real-time system under development. However, TRSD does not focus on providing the support for information hiding and configurability of the real-time software.

HRT-HOOD

HRT-HOOD [20], a hard real-time hierarchical object oriented design, introduces object-orientation into modeling of hard real-time systems. HRT-HOOD 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. This design step assumes that the designer knows the relationship of an object to a task, since an object can represent one or more tasks. 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 object reuse.

VEST

VEST [94, 95], 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 [95], the tool has been extended to support design-level cross-cutting 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 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 [94]. Currently VEST uses the well-defined CORBA component model [95].

RT-UML

RT-UML [35], 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 for the following reason. Although RT-UML provides powerful notation for modeling real-time systems it essentially provides only syntax, not semantics, for the real-time system design. Thus, 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 powerful expressiveness could be used by a design method as means of specifying real-time software components [26].

3.3 What Can We Learn?

From early '80s till now real-time design methods have mostly focused on task structuring and two different views on the system, temporal and structural, and only with moderate emphasis on the information hiding. The analysis of the real-time system under design, although missing from early design approaches, has been highlighted as important for the real-time system development. 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.

We can observe that there is a gap between the approaches from different communities, as the real-time community has focused primarily on real-time issues not exploiting modularity of software to the extent that the software engineering community has done. Bridging the gap between the approaches could be done by providing a uniform view of the criteria that a method for designing component-based real-time systems should fulfill. Thus, in the following sections we present a set of criteria that a design method for component-based real-time systems should fulfill.

3.4 Component Model

One of the most fundamental reasons behind the need to divide software into modules is to guarantee exchange of parts. 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 out of reusable components, we need a way of specifying what a component should look like, i.e., we need a component model. A component model describes what a software component should look like, and it supports modularity of the software in the sense that it defines what should be hidden in the component such that it can be exchanged and reused in several systems [10, 99, 28]. Further, a component model should enforce information hiding, implying that the deployers of a component cannot see when the manufacturer changes the component internals, and that newer versions of components can substitute older versions if the interfaces of components remain the same. Hence, the 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. Most software

Design approaches / Criteria		DARTS	TRSD	VEST	ISC	COM	AspectJ	HRT-HOOD
Component model								
CM1	Information hiding	◉	◉	◉	●	●	◉	●
CM2	Interfaces	◉	◉	◉	●	●	●	●
CM3	Component Views	◉	◉	●	◉	-	-	◉
CM4	Temporal attributes	-	◉	●	-	-	-	●
CM5	Task mapping	●	-	◉	-	-	-	-
Aspect separation								
AS1	Aspect support	-	-	◉	●	-	●	-
AS2	Aspect weaving	-	-	-	●	-	●	-
AS3	Multiple interfaces	-	-	-	●	●	◉	◉
AS4	Multiple aspect types	-	-	-	-	-	-	-
System composability								
SC1	Configuration support	◉	◉	●	●	◉	●	◉
SC2	Temporal analysis	-	-	◉	-	-	-	●

LEGEND: ● supported ◉ partially supported - not supported

DARTS: desing approach for real-time systems
ISC: invasive software composition
TRSD: transactional real-time system design
VEST: Virginia embedded systems toolkit
HRT-HOOD: a hard real-time hierarchical object-oriented desing

Table 3.1: The criteria for the evaluation of design approaches

engineering approaches enforce a good component model by providing information hiding and having well-defined interfaces for accessing components, e.g., COM, CORBA, and ISC (see table 3.1).

The most important criteria for a software component with respect to component model (CM) can be summarized as follows [10].

CM1 *Information hiding.* A component has to maintain one or several secrets, e.g., design, implementation, and programming language, which are encapsulated to enable component exchangeability.

CM2 *Interfaces.* A component should have one or more well-defined interfaces to the environment. The component can be accessed only through these interfaces (thus complementing the information hiding criterion).

Real-time systems sharing the component vision are faced with additional requirements on the component model, as components in a real-time system should provide mechanisms for handling temporal constraints. Moreover, since the traditional view of real-time systems implies tasks as building elements of the system, the relationship between tasks and components needs to be clearly identified. We argue that this relationship (between a real-time software component and a task) should not be fixed for several reasons. First, we would like to reuse not only pieces of code that correspond to tasks in the system, but also any software components applicable to the real-time system under construction, i.e., we do not want to limit reusability of the real-time software only to tasks. Second, perfect mapping of a component to a task for all applications is hard to determine. We illustrate this with a simple example as follows. Assume that we have a set of components, e.g., $c_1 \ldots, c_n$, in a component library or available on the market. When developing a real-time system, denoted RT_1, a designer determines that a subset of components from a library is needed for the real-time system under construction, namely $c_1 \ldots, c_k$ $(k < n)$. Furthermore, the target run-time environment of RT_1 is such that components $c_1 \ldots, c_3$ are allocated to one task $(c_1, \ldots, c_3 \Rightarrow t_1)$, while components $c_4, \ldots, c_k$ each are allocated to distinct tasks $(c_4 \Rightarrow t_2, \ldots, c_k \Rightarrow t_{k-3})$. On the other hand,

when developing another real-time system (e.g., RT_2), a different set of components may be applicable, e.g., $c_1 \ldots, c_6$. Furthermore, the run-time environment of RT_2 can differ from the one described for RT_1, e.g., only two tasks in the system are allowed, so the resulting allocation of the components and tasks is such that components $c_1, \ldots, c_4$ are allocated to one task ($c_1, \ldots, c_4 \Rightarrow t'_1$), while components c_5 and c_6 to another task ($c_5, c_6 \Rightarrow t'_2$). We can observe that the same set of software components could be allocated differently to tasks in different real-time systems, depending on the actual needs of the real-time system and its underlying run-time environment.

The problem now is to distinguish between components and tasks in a component-based real-time system, and to determine their relationship. A similar problem is addressed in DARTS, since the DARTS method is concerned with the relationship between tasks and modules, and the way tasks can be efficiently structured into modules. To solve the problem (modules vs. tasks) DARTS proposes a set of task structuring criteria to guide the designer when grouping tasks into modules. Note that in DARTS the system is first decomposed into tasks, corresponding to the temporal view of the real-time system, and then tasks are grouped into modules, corresponding to the structural view of the system.

In a component-based real-time system each component should be specified in two dimensions (views):

- structural, and
- temporal.

We already argued for the necessity of having components and tasks distinguished from each other and their relationship clearly defined. One of the ways to achieve this is to distinguish between a temporal and a structural dimension of a component. The structural dimension (or structural view) of a component represents a traditional software component as perceived by software engineering community, i.e., software artifact with well-defined interfaces that enforces information hiding. The temporal dimension should reflect the 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 true for components built for use in hard real-time systems. In all approaches discussed, VEST is the only platform that can be considered to enforce the component views as it allows components (pieces of code) to be mapped to tasks for a particular platform (see table 3.1). However, VEST does not provide detailed information on how the actual mapping should be done. The graphical environment of VEST enables storing temporal attributes of components that are used for schedulability analysis. More traditional approaches to real-time system design, including DARTS and TRSD, do not provide temporal information for their components (see table 3.1). HRT-HOOD, in contrast, supports only a temporal dimension of objects as building parts of a real-time system.

We conclude that in a real-time environment, a software component model should support the following.

CM3 *Component views.* A design method should support decomposition of real-time systems into components as basic building blocks, and, further, components should support both a structural and a temporal view. In the structural view a real-time software is composed out of software components, and in the temporal view the real-time software is composed out of tasks.

CM4 *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 the real-time system, and enable static and dynamic temporal analysis of the overall real-time system.

CM5 *Task mapping.* A design method should provide clear guidelines (or possibly tools) to support mapping of components into tasks.

3.5 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 [10]. Separation

of concerns is the main idea of AOSD [45]. 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. AOSD offers several important advantages [10]:

- aspect specifications can be exchanged separately from the component[2] and independently from each other;
- aspect weaving creates system configurations that might have completely different functionality or non-functional features;
- separation of concerns into aspects eases configuration of the system as fewer parts of the overall system need to be modified for a particular system configuration; and
- a system becomes scalable as the same core components can be reused elsewhere.

These are the reasons why some design approaches in software engineering, e.g., ISC and AspectJ, provide support for aspects and aspect weaving (see table 3.1). A restricted form of aspect separation is also realized in COM and CORBA through defining multiple interfaces as access points for the component, as each of the interfaces can be viewed as one aspect.

It is clear that software engineering design methods increasingly emphasize support for aspects and aspect weaving, giving us the following criteria with respect to aspect separation (AS) [45, 10, 7].

AS1 *Aspect support.* A design method should support separation of concerns, namely it should provide support for identifying and specifying aspects in the software system.

AS2 *Aspect weaving* A design method should provide tools that weave aspect specifications to the final product.

[2] As already mentioned, a component in AOSD is typically a white box component, e.g., traditional program, function and method.

AS3 *Multiple interfaces.* A component should have multiple interfaces under which it can be accessed.

The frontier where aspects and aspect weaving meet real-time has not been explored yet, 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 crosscutting, in typical implementations of real-time systems, the overall system. Moreover, these concerns cannot be easily encapsulated in a component with well-defined interfaces. However, some real-time design approaches do not support aspects and aspect weaving at all, e.g., DARTS and TRSD (see table 3.1). HRT-HOOD supports two types of interfaces, hence, supporting only the AS3 criterion. Some work on aspect separation in the design of real-time systems has been addressed in VEST. However, VEST supports aspects like memory consumption describing the run-time behavior of the component with respect to memory consumption but not changing the code of the VEST components. From this, it can be observed that real-time systems have another different dimension of crosscutting compared to general-purpose software systems, e.g., the VEST memory consumption aspect describes the memory needs of components and crosscuts the entire real-time system composed out of components. Hence, 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. This implies that a design method of configurable real-time systems should support multiple aspect types. We express this in the following criterion.

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 method should support aspects that crosscut code of core components, as well as additional aspect types to specify the behavior of the real-time component in the run-time environment.

3.6 System Composability

Software for real-time systems should be produced quickly, reliably, and should be optimized for a particular application or a product. Successful deployment of software components for building real-time systems depends on the availability of tools that would improve development, implementation, and evaluation of component-based real-time systems. An example of an effort to create a tool helping the designer to configure and analyze a component-based real-time system is VEST. Configuration support, although not automated by tools, has also been addressed by both DARTS and TRSD as they provide detailed design guidelines for real-time system composition. Note that most approaches to design of general-purpose software systems, with the exception of COM and CORBA, have powerful tools for software configuration. However, analysis of the software behavior is of secondary (if any) interest as these approaches do not provide means for analyzing behavior of the composed software system (see table 3.1).

Temporal analysis of a real-time system is one of the key issues in real-time system development, and there are a number of approaches that provide support in performing temporal analysis of the real-time system in their design method, namely TRSD, HRT-HOOD, and VEST (see table 3.1).

We conclude that a design method for building reliable component-based real-time systems should provide support for exchanging components and configuring new systems out of existing, or newly developed, components. Hence, the following should be considered with respect to system composability (SC).

SC1 *Configuration support.* The design method should provide recipes for combining different components into system configurations.

SC2 *Temporal analysis.* The design method should provide support for temporal analysis of the composed real-time system. Tools to achieve predictable software behavior are preferable.

3.7 Observations

As can be observed, most of the design approaches discussed in this chapter both in the software engineering and real-time communities fulfill only a subset of the criteria identified. However, to fully exploit the benefits of component-based software development in real-time systems, new requirements are put on the design method. We have identified necessary criteria we have found so far that a design method needs to satisfy. 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, and this has not been fully addressed by existing real-time design approaches. Hence, a design method that would fully support aspects in the real-time system design 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 design method supporting aspects and components should provide methods and tools for the temporal analysis of the system composed out of components and aspects. This requirement is essential if a design method for building component-based real-time systems is targeted towards hard real-time environments.

Chapter 4

ACCORD

We have argued that the growing need for enabling development of configurable real-time and embedded systems that can be tailored for a specific application, and managing the complexity of the requirements in the real-time system design, calls for an introduction of new concepts and new software engineering paradigms into real-time system development. In this chapter we present ACCORD as a proposal to address these new needs. Through the notion of aspects and components, ACCORD enables efficient application of the divide-and-conquer approach to complex system development. To effectively apply ACCORD, we provide a design method with the following constituents.

- A decomposition process that supports the structural and temporal view of the components, and consists of the following sequential phases: (i) decomposition of the real-time system into a set of components and a set of aspects, corresponding to the structural view of the components and the real-time system; and (ii) task structuring, corresponding to the temporal view of the components and the real-time system.
- Aspects, as properties of a system affecting its performance or semantics, and crosscutting the functionality of the system [45].
- A real-time component model (RTCOM) that describes what a

real-time component, which supports aspects but also enforces the information hiding, should look like. RTCOM is specifically developed to: (i) enable an efficient decomposition process, (ii) support the notion of time and temporal constraints, and (iii) enable efficient analysis of components and the composed system.

- A method for performing the worst-case execution time analysis of different configurations of aspects and components, and a tool that allows analysis automation.

The design of a real-time system using ACCORD is performed in three phases. In the first phase, a real-time system is decomposed into a set of components. Decomposition is guided by the need to have functionally exchangeable units that are loosely coupled, but with strong cohesion. In the second phase, a real-time system is decomposed into a set of aspects (these aspects crosscut components and the overall system). This phase typically deals with non-functional requirements[1] and crosscutting concerns of a real-time system, e.g., resource management and temporal attributes. In the next phase, components and aspects are implemented based on RTCOM. In the last phase, the structural view of the real-time system, i.e., components and aspects, are mapped to the temporal view using the task structuring criteria.

ACCORD supports both static WCET and dynamic schedulability analysis. Task schedulability analysis is performed based on the WCET information of each component (possibly colored with aspects) and the mapping information obtained in the task structuring step. The WCET analysis of components colored with aspects is performed using our tool for automated aspect-level WCET analysis.

As non-functional requirements are essential in real-time system development, in this chapter we first focus on aspects and aspectual decomposition of real-systems (section 4.1), and then discuss RTCOM (section 4.2). Section 4.3 presents guidelines for mapping components to tasks. In section 4.4 we present the method and the tool for aspect-level WCET

[1]Non-functional requirements are sometimes referred to as extra-functional requirements [28].

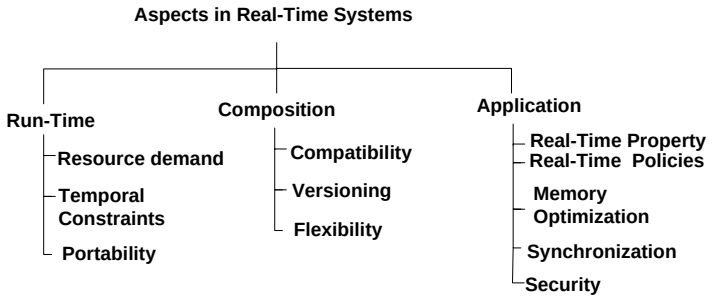


Figure 4.1: Classification of aspects in real-time systems

analysis of components, aspects, and the resulting system. Finally, in section 4.5 we present an evaluation of ACCORD.

4.1 Aspects in Real-Time Systems

We classify aspects in a real-time system as follows (see figure 4.1):

- application aspects (section 4.1.1),
- run-time aspects (section 4.1.2), and
- composition aspects (section 4.1.3).

The following is the description of each class of aspects in real-time systems.

4.1.1 Application Aspects

Application aspects can change the internal behavior of components as they crosscut the code of the components in the system. The application in this context refers to the application towards which a real-time and embedded system should be configured, e.g., memory optimization

aspect, synchronization aspect, security aspect, real-time property aspect, and real-time policy aspect. Since optimizing memory usage is one of the key issues in embedded systems and it crosscuts the structure of a real-time system, we view memory optimization as an application aspect. Security is another application aspect that influences the behavior and the structure of a system, e.g., the system should distinguish users with different security clearance. Synchronization, entangled over the entire system, is encapsulated and represented by a synchronization aspect. Memory optimization, synchronization, and security aspect are commonly mentioned aspects in AOSD [45]. Additionally, real-time properties and policies are viewed as application aspects as they influence the overall structure of the system. Depending on the system's requirements, real-time properties and policies could be further refined, which we show in detail in the example of the COMET system (see section 5.3). Application aspects enable tailoring of the components for a specific application, as they change the code of the components.

We informally define application aspects as programming (aspect) language-level constructs encapsulating crosscutting concerns that invasively change the code of the component. Formal definition of application aspects is given in section 4.2.2.

4.1.2 Run-Time Aspects

Run-time aspects are critical as they refer to aspects of the 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 system 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 behavior, contained in the temporal constraints aspect, e.g., WCET. The temporal 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 hard-

ware related information. This information is contained in the portability aspect. It is imperative that the information contained in the run-time aspect is provided to ensure predictability of the composed system, ease the integration into a run-time environment, and ensure portability to different hardware and/or software platforms.

We informally define run-time aspects as language-independent design-level constructs encapsulating crosscutting concerns that contain the information describing the component behavior with respect to the target run-time environment. This implies that the run-time aspects do not invasively change the code of the component.

4.1.3 Composition Aspects

Composition aspects describe compositional constraints when combining (compatibility aspect), the version of the component (version aspect), and possibilities of extending the component with additional aspects (flexibility aspect).

Composition aspects can be viewed as language-independent design-level constructs encapsulating crosscutting concerns that describe the component behavior with respect to the composition needs of each component. This implies that composition aspects do not invasively change the code of the component.

Having separation of aspects in different categories eases reasoning about different embedded and real-time related requirements, as well as the composition of the system and its integration into a run-time environment. For example, the run-time system could define what (run-time) aspects the real-time system should fulfill so that proper components and application aspects could be chosen from the library, when composing a monolithic system. This approach offers a significant flexibility since additional aspect types can be added to components, and therefore, to the monolithic real-time system, further improving the integration with the run-time environment.

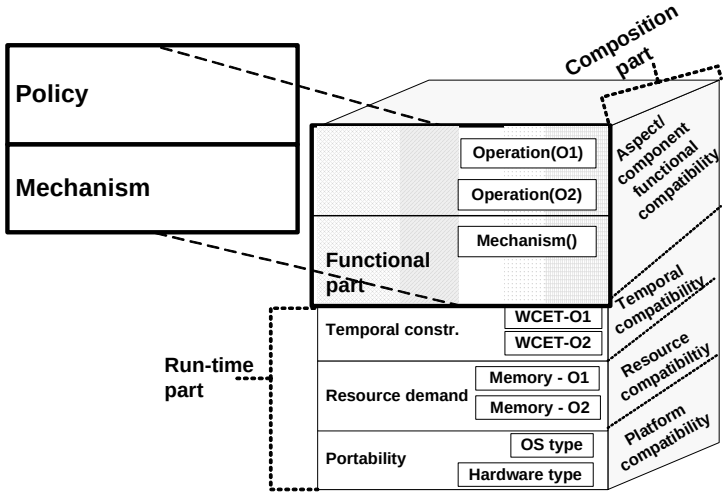


Figure 4.2: A real-time component model (RTCOM)

4.2 Real-Time Component Model

In this section we present RTCOM, a real-time component model, which allows easy and predictable weaving of aspects, i.e., integrating aspects into components, thus reflecting decomposition of the system into components and aspects. Furthermore, RTCOM supports information hiding and three types of interfaces.

RTCOM can be viewed as a component colored with aspects, both inside (application aspects), and outside (run-time and composition aspects). RTCOM is a language-independent component model, consisting of the following parts (see figure 4.2):

- functional part (section 4.2.2),
- run-time system dependent part (section 4.2.3), and
- composition part (section 4.2.4).

RTCOM represents a coarse-granule component model as it provides a

broad infrastructure within its functional part. This broad infrastructure enables tailoring of a component through weaving of application aspects, thereby changing the functionality and the behavior of the component to suit the needs of a specific application. In contrast, traditional component models are fine-grained and allow controlled configuration of a component to adopt it for use in different system. Although this type of fine-grained component is typically more optimal for a particular functionality provided by the component in terms of code size, it does not allow component tailoring, but merely fine-tuning restricted set of parameters in the component [28]. For each component implemented based on RTCOM, the functional part of the component is first implemented together with the application aspects, then the run-time system dependent part and the run-time aspects are implemented, followed by the composition part and rules for composing different components and application aspects. Interfaces supported by RTCOM are discussed in 4.2.5.

4.2.1 Notation

We use the following notation to provide a formalized framework for RTCOM:

- C denotes a set of components of a real-time system under development, i.e., the configuration, and $c \in C$ represents a component in the system;
- M denotes a set of mechanisms;
- O denotes a set of operations;
- A denotes a set of application aspects of a real-time system under development; and
- $I = I_f \cup I_c \cup I_g$ is the set of component interfaces, where
  - I_f is a set of functional interfaces of component c,

- I_c is a set of compositional interfaces of component c,
- I_g is a set of configuration interfaces of component c.

Within RTCOM we define a component as follows.

Definition 1 (Component) *A component c is a tuple $< M, O, I >$, where M is a set of mechanisms encapsulated by component c, O is a set of operations of component c, and I is a set of component interfaces.*

The following sections provide the follow-up definitions and extensive elaboration on each of the constituents of the definition 1 using the notation introduced in this section.

4.2.2 Functional Part of RTCOM

To define the functional part of the RTCOM, we first need to define the notion of mechanisms and operations of a component, as follows.

Definition 2 (Mechanisms) *A set of mechanisms M of component c is a non-empty set of functions encapsulated by component c.*

The implication of definition 2 is the establishment of mechanisms as fine-granule methods or functions of each component.

Definition 3 (Operations) *A set of operations O of component c is a set of functions implemented in c where for each operation $o \in O$ there exists a non-empty subset of mechanisms $K \subseteq M$, a subset of operations L from other components $(C \backslash \{c\})$, and a mapping such that $o = f(K, L)$.*

Definition 3 implies that each component provides a set of operations to other components and/or to the system. Operations can be viewed as coarse-granule methods or function calls as they are implemented using the underlying component mechanisms. Additionally, each operation within the component can call any number of operations provided by other components in the system. An example of how operations and

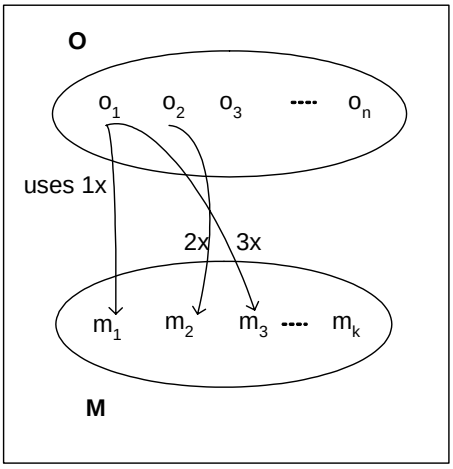


Figure 4.3: Operations and mechanisms in a component c

mechanisms could be related in the component is given in figure 4.3. For example, operation $o_1 \in O$ is implemented using the subset of component mechanisms $\{m_1, m_3\}$, while operation o_2 is implemented using the subset $\{m_2\}$ of component mechanisms. Furthermore, each operation in the component can use a mechanism in its implementation one or several times, e.g., o_1 uses m_1 once and m_3 three times.

Definition 4 (Recursively non-cyclic set) *Given the operation sequence* $< o_1, \ldots, o_n >$ *let* f_i *be the mapping that defines the operation* o_i*, i.e.,* $o_i = f_i(K_i, O_i)$*, where each* O_i *is defined by:*

- $O_i = \{o_{i1}, \ldots, o_{im}\}$ *for some* m*, and*
- $o_{ik} = f_{ik}(K_{ik}, O_{ik})$*,* $1 \leq k \leq m$*.*

Let $D_i = O_{i1} \cup \ldots \cup O_{ik}$ *be the operation domain of the functions at level* i*. The operation sequence* $< o_1, \ldots, o_n >$ *is recursively non-cyclic if and only if for all* D_i, D_j*,* $i < j$*,* $D_i \cap D_j \neq \emptyset$*.*

A set of operations $\{o_1, o_2, o_3\}$ is recursively cyclic if operation o_1 is implemented using operation o_2, which in turn is implemented using

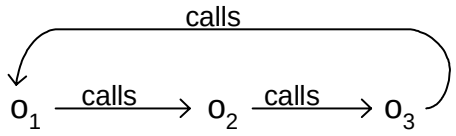


Figure 4.4: An example of the recursively cyclic set of operations $\{o_1, o_2, o_3\}$

operation o_3, and operation o_3 makes a recursive cycle by being implemented using the operation o_1 (see figure 4.4). Having recursively cyclic sets of operations in the component, and between different components, makes temporal analysis, e.g., WCET analysis, of the system composed out of components inherently difficult. Hence, RTCOM in its current form only supports recursively non-cyclic operation sets. The following definition characterizes this property.

Definition 5 *A component configuration C for which the operations O can be ordered into a sequence $< o_1, \ldots, o_n >$ with the recursively non-cyclic property, is considered to be well-formed for the purpose of the WCET analysis.*

The functional part of RTCOM represents the actual code implemented in the component, and is characterized by definition 6.

Definition 6 (Functional part) *Let c belong to a well-formed component set C. Then the functional part of component c is represented by the tuple $< M, O >$, where M is the set of mechanisms of the component and O is the set of operations implemented by the component.*

Definition 7 (Application aspects) *An application aspect $a \in A$ is a set of tuples $< a^t, P >$ where:*

- $t \in \{before, after, around\}$;
- a^t *is an advice of type* t *defined by mapping* $a^t = f(K)$, $K \subseteq M$; *and*

– *P is a set of pointcuts that describes the subset of operations in components that can be preceded, succeeded, or replaced by advice a^t depending on the type of the advice.*

Definition 7 extends the traditional definition of programming-language level aspects by specifying pointcuts and advices in terms of mechanisms and operations. This enables performing temporal analysis[2] of the weaved system, and thereby use of aspects in real-time environments. This also enables existing aspect languages to be used for implementing application aspects in real-time systems, and enables existing weavers to be used to integrate aspects into components while maintaining predictability of the real-time system. Hence, in RTCOM, pointcuts refer to operations. This implies that a pointcut in an application aspect points to one or several operation of a component where modifications of the component code are allowed. Also, having the mechanisms of the components as basic building blocks of the advices enables temporal analysis of the resulting weaved code. Furthermore, the implementation of a whole application 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 operations through out the system configuration. Advices, and, hence, application aspects can be implemented using the mechanisms from a number of components.

If x and y represent two code fragments then xy denotes sequential composition of the two code fragments (resulting from a textual concatenation of the two pieces of code).

Definition 8 *Let x and y be two pieces of code (two sequences of statements in some programming language). Let o and o' be the mathematical representation of x and y, respectively. Then we denote the mathematical representation of the code xy by $glue(o, o')$.*

Definition 9 (Weaving of application aspects) *Let $a =< a^t, P >$ be an application aspect where $a^t = f(K)$, $K \subseteq M$, and P is a set of*

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

pointcuts, $P \subseteq O$. Weaving of application aspect $a \in A$ in the component $c =< M, O, I >$, results in a component $c' =< M, O', I >$ where for all $o'_i \in O'$ the following holds:

- *if $o_i \in O \backslash P$ then $o'_i = o_i$*
- *if $o_i \in P$ then*

$$o'_i = \begin{cases} glue(a^t, o_i) & \text{if } t\text{=}before \\ glue(o_i, a^t) & \text{if } t\text{=}after \\ a^t & \text{if } t\text{=}around \end{cases}$$

For example, assume that we have component $c =< M, O, I >$ such that M is the set of mechanisms, $O = \{o_1, \ldots, o_6\}$ is the set of operations, and I the set of component interfaces. Then, weaving of application aspect a, consisting of advices a_1^{before}, a_2^{after}, and their respective pointcut sets, $P_1 = \{o_1, o_3\}$ and $P_2 = \{o_6\}$, would result in component $c' =< M, O', I >$ where

$$O' = \{glue(a_1^{before}, o_1), o_2, glue(a_1^{before}, o_3), o_4, o_5, glue(o_6, a_2^{after})\}.$$

Hence, in component c', the execution of operations o_1 and o_3 are preceded by the execution of the code of advice a_1^{before}, and the execution of operation o_6 is succeeded by the execution of advice a_2^{after}. Operations o_2, o_4 and o_5 remain unchanged.

Weaving application aspects into the code of a component does not change the implementation of mechanisms, only the implementation of operations within the component. Thus, operations are flexible parts of the component as their implementation can change by weaving application aspects, while mechanisms are fixed parts of the component infrastructure. Since advices are implemented using the mechanisms of the components, each advice can change the behavior of the component by changing one or more operations in the component.

To enable easy implementation of application aspects into a component, the design of the functional part of the component is performed

in the following manner. First, mechanisms, as basic blocks of the component, are implemented. Here, particular attention should be given to the identified application aspects, and the table that reflects the crosscutting effects of application aspects to different components should be made to help the designer in the remaining steps of the RTCOM design and implementation. Next, the operations of the component are implemented using component mechanisms (see definition 3). Note that the implemented operations provide an initial component policy, i.e., basic and somewhat generic component functionality. This initial policy we denote a *policy framework* of the component. The policy framework could be modified by weaving different application aspects to change the component policy. If all crosscutting application aspects are considered when implementing operations and mechanisms, then the framework is generic and highly flexible.

The development process of the functional part of a component results in a component colored with application aspects. Therefore, in the graphical view of RTCOM in figure 4.2, application aspects are represented as vertical layers in the functional part of the component as they influence component behavior, i.e., change component policy.

Consider a simple example of an ordinary linked list implemented based on RTCOM. The functional part of the component, i.e., the code, consists of mechanisms and the policy framework. The mechanisms needed are the ones for the manipulation of nodes in the list, i.e., `createNode, deleteNode, getNextNode, linkNode`, and `unlinkNode` (see figure 4.5). Operations implementing the policy framework, e.g., `listInsert, listRemove, listFindFirst`, define the behavior of the component, and are implemented using the underlying mechanisms. In this example, `listInsert` 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 policy framework is FIFO.

Assume that we want to change the policy of the component from FIFO to priority-based ordering of the nodes. Then, this can be achieved by weaving an appropriate application aspect. Figure 4.6 shows the `listPriority` application aspect, which consists of one pointcut

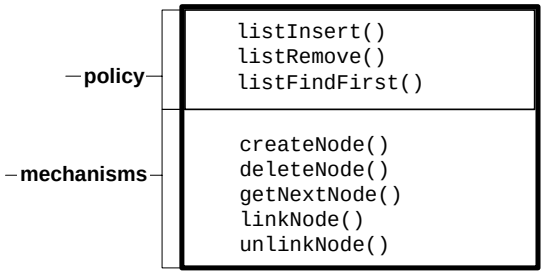


Figure 4.5: The functional part of the linked list component

`listInsertCall`, identifying `listInsert` as a join point in the component code (lines 2-3). When this join point is reached, the code in the `before` advice `listInsertCall` is executed. Hence, the application aspect `listPriority` intercepts the operation (a method or a function call to) `listInsert`, and before the code in `listInsert` is executed, the advice, using the component mechanisms (`getNextNode`), determines the position of the node based on its priority (lines 5-14).

4.2.3 Run-Time System Dependent Part of RTCOM

The run-time system dependent part of RTCOM accounts for temporal behavior of the functional part of the component code, not only without aspects but also when aspects are weaved into the component. Hence, run-time aspects are part of the run-time dependent part of RTCOM; they are represented as horizontal parallel layers to the functional part of the component as they describe component behavior (see figure 4.2). In the run-time part of the component, run-time aspects are expressed as attributes of operations, mechanisms, and application aspects, since those are the elements of the functional part of the component, and thereby influence the temporal behavior of the component.

We now illustrate how run-time aspects are represented and handled in RTCOM using one of the most important run-time aspects as an example, i.e., WCET. One way of enabling predictable aspect weaving is

```
aspect listPriority{
1:
2: pointcut listInsertCall(List_Operands * op)=
3:   call("void listInsert(List_Operands*)")&&args(op);
4:
5: advice listInsertCall(op):
6:   void before(List_Operands * op){
7:     while
8:        the node position is not determined
9:      do
10:       node = getNextNode(node);
11:       /* determine position of op->node based
12:       on its priority and the priority of the
13:       node in the list*/
14:    }
15: }
```

Figure 4.6: The `listPriority` application aspect

to ensure an efficient way of determining the WCET of the operations and/or real-time system that have been modified by weaving of aspects. WCET specifications in RTCOM are based on the following two observations:

- aspect weaving does not change WCET of mechanisms since mechanisms are fixed parts of the RTCOM; and
- aspect weaving changes operations by changing the number of mechanisms that an operation uses, thus, changing their temporal behavior.

Therefore, if the WCETs of mechanisms are known and fixed, and the WCET of the policy framework and aspects are given as a function of mechanism used, then the WCET of a component weaved with aspect(s) can be computed by calculating the impact of aspect weaving to WCETs of operations within the component (in terms of mechanism usage). To facilitate efficient WCET analysis of different configurations of aspects and components, WCET specifications within run-time part of RTCOM should satisfy the following:

- the WCET for each mechanism is known and declared in the WCET specification;
- the WCET of every operation is determined based on the WCETs of the mechanisms, used for implementing the operation, and the internal WCET of the body of the function or the method that implements the operation, i.e., manages the mechanisms; and
- the WCET of every advice that changes the implementation of the operation is based on the WCETs of the mechanisms used for implementing the advice and the internal WCET of the body of the advice, i.e., code that manages the mechanisms.

```
mechanisms(listOfParameters){
  mechanism{
    name    [nameOfMechanism];
    wcet      [value of wcet];
    }
  mechanism{
    name    [nameOfMechanism];
    wcet      [value of wcet];
    }
              .........
}
```

Figure 4.7: Specification of the WCET of component mechanisms

Figure 4.7 shows the WCET specification for mechanisms in the component, where for each mechanism the WCET is declared and assumed to be known. Similarly, figure 4.8 shows the WCET specification of the component policy framework. Each operation defining the policy of the component declares what mechanisms it uses, and how many times it uses a specific mechanism. This declaration is used for computing WCETs of the operations or the component (without aspects). Figure 4.9 shows the WCET specification of an application aspect. For each

```
policy(listOfParameters){
  operation{
    name    [nameOfOperation];
    uses{
         [Name of mechanism] [Number of times used];
    }
    intWcet [Value of internal operation wcet
                    (called mechanisms excluded)]
  }
  operation{
    ...
  }
  ...
}
```

Figure 4.8: Specification of the WCET of a component policy framework

```
aspect(listOfParameters){
  advice{
    name [nameOfAdvice];
    type [typeOfAdvice:before, after, around];
    changes{
      name  [nameOfOperation];
      uses{
        [nameOfMechanism]  [Number of times used];
      }
     intWcet[Value of internal advice wcet
              (called mechanisms excluded)]
    }
  }

 ......
}
```

Figure 4.9: Specification of the WCET of an application aspect

advice type (before, around, after) that modifies an operation, the operation it modifies is declared together with the mechanisms used for the implementation of the advice, and the number of times the advice uses these mechanisms. WCET specifications of aspects and components can also have a list of parameters used for expressing the value of WCETs. The resulting WCET of the component (or one operation within the component), colored with application aspects, is computed using the algorithm presented in [102] and described in detail in section 4.4.2. The algorithm utilizes the knowledge obtained from WCET specifications of the mechanisms and operations involved, as well as WCET specifications of aspects that change a specific operation.

As a consequence of weaving an application aspect into the code of the component, the temporal behavior of the resulting component, colored with aspects, changes. Hence, run-time aspects need to be defined for the policy framework (the component without application aspects) as well as the application aspects, so we can determine the run-time aspects of a component colored with different application aspects. Figure 4.10 presents an instantiation of a WCET specification for the policy framework of the linked list component. Each operation in the framework is named and its internal WCET (`intWcet`) with the number of times it uses a particular mechanism are declared (see figure 4.10). The WCET specification for the application aspect `listPriority` that changes the policy framework is shown in figure 4.11. Since the maximum number of elements in the linked list can vary, the WCET specifications are parameterized with the `noOfElements` parameter. We continue this example of WCET specifications of the linked list component in section 4.4 and use it to illustrate the way WCET analysis is performed within the ACCORD.

4.2.4 Composition Part of RTCOM

The composition part refers both to the functional part and the run-time part of a component, and is represented as the third dimension of the component model (see figure 4.2). Given that there are different application aspects that can be weaved into the component, composition

```
policy(noOfElements){
  operation{
    name listInsert;
    uses{
      createNode 1;
      linkNode   1;
    }
    intWcet 1ms;
  }
  operation{
    name listRemove;
    uses{
      getNextNode noOfElements;
      unlinkNode  1;
      deleteNode   1;
    }
    intWcet 4ms;
  }
....
}
```

```
mechanisms{
  mechanism{
    name createNode;
    wcet  5ms;
  }
  mechanism{
    name deleteNode;
    wcet  4ms;
  }
  mechanism{
    name getNextNode;
    wcet  2ms;
  }

....
}
```

Figure 4.10: The WCET specification of the policy framework

```
aspect listPriority(noOfElements){
  advice{
    name listInsertCall;
    type before;
    changes{
      name listInsert;
        uses{
          getNextNode noOfElements;
        }
    }
    intWcet 4ms+0.4*noOfElements;
  }
....
}
```

Figure 4.11: The WCET specification of the `listPriority` application aspect

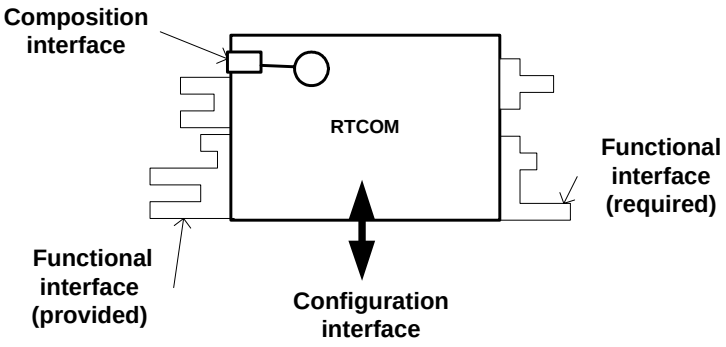


Figure 4.12: Interfaces supported by the RTCOM

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 it is currently under development.

4.2.5 RTCOM Interfaces

RTCOM supports three different types of interfaces (see figure 4.12): (i) functional interface, (ii) configuration interface, and (iii) composition interface.

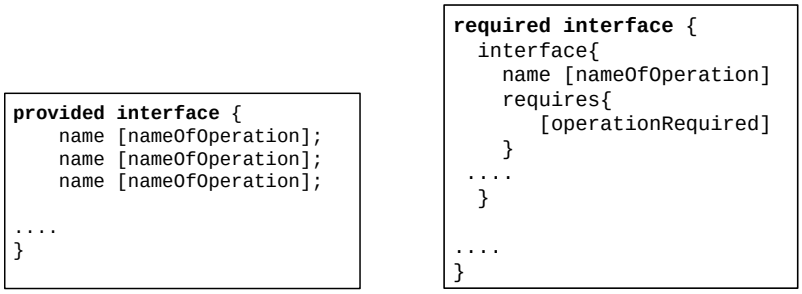
Typically, in a component-based software system, a component functional interface specification reflects the operations of the component. Namely, the interface specification provides the operation name, type, and parameters of the operation, e.g., input, output, or input/output parameters [26]. We denote such operation specification as operation signature and use this notation in the remainder of this section.

Functional Interface

A functional interface I_f of component c is a tuple $< I_f^p, I_f^r >$, where I_f^p is an interface provided by the component consisting of a set of signatures of

operations O provided by the component, and I_f^r is an interface required by the component c consisting of a set of signatures of operations L from other components ($C \backslash \{c\}$) called from the component c.

Thus, functional interfaces of components are classified in two categories, namely provided functional interfaces and required functional interfaces. The way required and provided interfaces are implemented in the first application of RTCOM is illustrated in the figure 4.13. Provided interfaces reflect a set of operations that a component provides to other components or to the system (see figure 4.13(a)). Required interfaces reflect a set of operations that a component, i.e., each operation within the component, requires from other components (see figure 4.13(b)). In



(a) The provided functional interface

(b) The required functional interface

Figure 4.13: An example of the specification of the functional interface

general, it is considered that having separation to provided and required interfaces eases component exchange and addition of new components into the system [19].

Configuration Interface

A configuration interface I_g of a component c is a tuple $< O, X >$, where O is a set of signatures of operations implemented in the component,

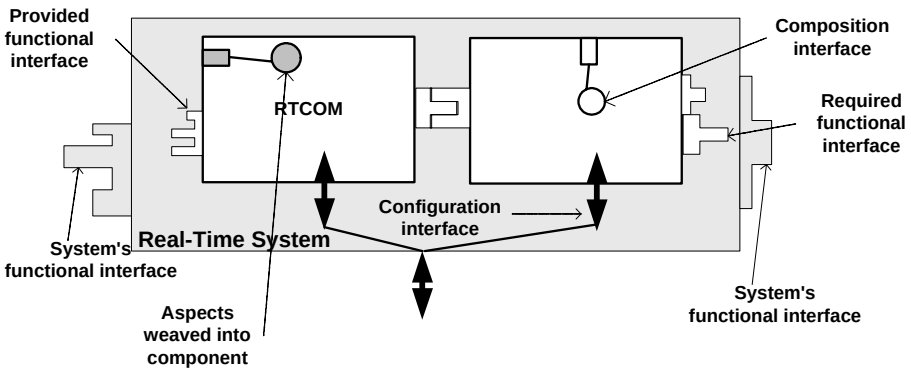


Figure 4.14: Interfaces and their role in the composition process

and X is a set of changeable parameters specifying the behavior of the operations.

The configuration interface is intended for supporting the integration of a real-time system with the run-time environment. This interface provides a set of parameters X that can be changed within the component so that the mapping of the component on the target run-time environment is eased. Combining multiple components results in a system that also has a configuration interface, which enables the designer to inspect the behavior of the system towards the run-time environment (see figure 4.14). The precise role and the full functionality of this interface are still under development.

Composition Interface

A composition interface I_c of component c is a set of signatures of mechanisms M of the component.

Composition interfaces, which correspond to join points, are embedded into the functional part of a component in form of the mechanism declarations. The weaver identifies composition interfaces and uses them for aspect weaving. Composition interfaces are ignored at component/system compile-time if they are not needed; they are "acti-

vated" only when certain application aspects are weaved into the system. Thus, the composition interface allows integration of the component and aspectual part of the system. Aspect weaving can be performed either on the component level, weaving application aspects into component functionality, or on the system level, weaving application aspects into the monolithic system.

4.3 Task Structuring

When aspects and components are identified in the real-time system under development, the final step is to map those to tasks in the system. We argued (see section 3.4) that the need for having reusable components in a component library results in a real-time system decomposition into components and aspects as building blocks of the system. The mapping of these into tasks is dependent on the current system in which the component is to be used and, hence, should be done after the components and aspects required in the system have been identified. To help designers in the mapping process we have adopted the task structuring guidelines from DARTS [39] to perform mapping from components to tasks. The guidelines should be applied after the software components in the system have been identified. Hence, components can be mapped to tasks based on the following.

- Event dependency, which includes the following:

  - Asynchronous I/O device dependency. A component, providing certain operations dependent on the device input and output, is often constrained to execute at the speed of the I/O device with which it interacts. In particular, if the device is asynchronous then such a component could be structured into a separate I/O dependent task.

  - Periodic event. If operations provided by one or more components need to be executed at regular intervals of time, the

component providing these operations could be structured as a periodically activated task.

- Task cohesion, which includes the following:
  - Sequential cohesion. Certain components have operations that need to be performed sequentially with respect to operations of some other components. The first operation of the first component in the sequence is triggered by an asynchronous or periodic event. These sequential components can be combined into one task.
  - Temporal cohesion. Components that perform operations activated on the same event may be grouped into a task so that they are executed each time the task receives a stimulus.
- 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 should be structured as a separate high priority task.
  - Computational intensity. Non-time-critical computationally intensive operations of the component may run as a low priority task consuming spare CPU cycles.

The given guidelines present a way of assigning components to tasks. The assignments result in the "ownership" relation between tasks and components in the system, where each task can "own" one or more components. The mapping information describing the ownership of each task obtained in this step should be the end result of this process. Note, however, that the guidelines presented in this section represent the initial effort in defining the way components can be mapped to tasks, and issues not discussed here, such as inter-process synchronization, are subject to further research.

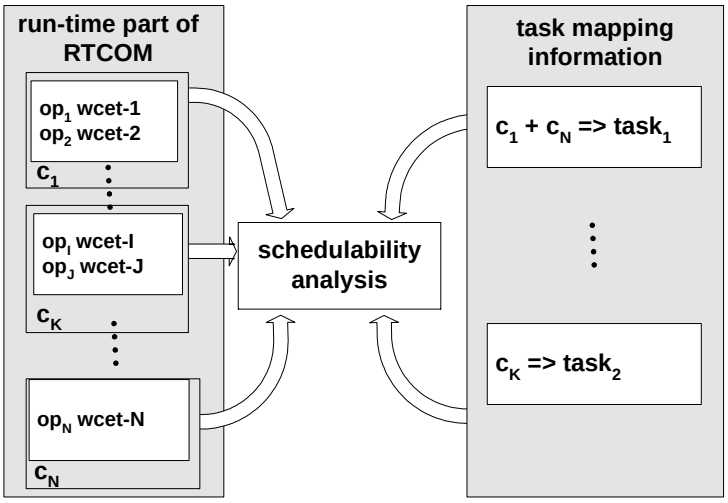


Figure 4.15: Temporal analysis in ACCORD

4.3.1 Task Analysis

Temporal analysis is rather straightforward in ACCORD. The analysis is based on the following elements:

- the WCET information contained in the run-time part of the component (components can possibly be modified with aspects), and
- the mapping information describing the task ownership of components obtained in the task structuring step.

Given the configuration of the system consisting of N components ($c_1, \ldots, c_N$), the information about the WCET of each operation provided by a component is contained in the run-time part of the components (see figure 4.15). For example, in the run-time part of the component c_1 the value of the WCET of the operations (op_i, i=1,2), is provided. The task mapping information specifies on which task's thread of execution a component runs, e.g., component c_1 runs on $task_1$ thread of execution together with component c_N. By combining the WCET

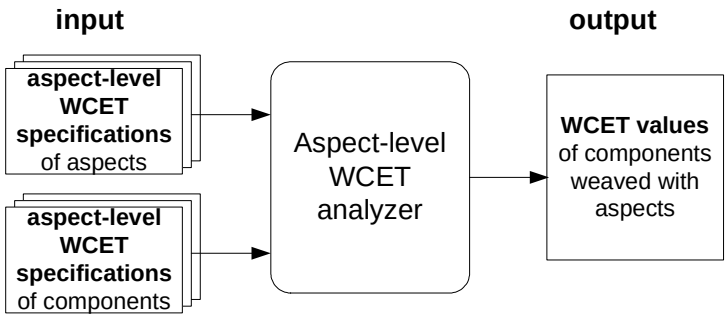


Figure 4.16: An overview of the automated aspect-level WCET analysis process

knowledge of the operations in the component with the task mapping information we obtain precise WCETs of each task in the system, which then makes it easy to analyze a given task set with respect to finding a feasible schedule on a specific platform, e.g., by using some off-the-shelf tool for schedulability analysis.

4.4 Aspect-Level Worst-Case Execution Time Analysis

In this section we present an approach for determining the WCET of a real-time system composed using aspects[3] and components, called aspect-level WCET analysis. The aspect-level WCET analysis is based on the concept of symbolic WCET analysis [13]. The main goal of aspect-level WCET analysis is determining the WCET of different real-time system configurations consisting of aspects and components before any actual aspect weaving (system configuration) is performed, and, hence, help the designer of a configurable real-time system to choose the system configuration fitting the WCET needs of the underlying real-time envi-

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

ronment without paying the price of aspect weaving for each individual candidate configuration.

Figure 4.16 presents an overview of the automated aspect-level WCET analysis with its main constituents, namely:

- input files, which are aspect-level WCET specifications of aspects and components,
- the aspect-level WCET analyzer, which performs the actual computation of the WCET of components weaved with aspect, and
- output files, which are the result of the aspect-level WCET analysis.

Our tool, the aspect-level WCET analyzer, produces WCET estimates of components weaved with aspects to determine if the configuration of aspects and components under consideration can be integrated into the target run-time environment. If necessary, i.e., if very precise WCET estimates are needed, the tool for aspect-level WCET analysis can be followed by further analysis of the resulting weaved code using a more specialized 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.

4.4.1 Aspect-Level WCET Specification

Aspect-level specifications of components and aspects correspond to the run-time part of the RTCOM that implements the WCET aspect. The reason we introduce a notion of aspect-level specifications to represent WCET aspect is to emphasize that the approach to aspect-level WCET analysis could be generalized beyond the RTCOM model and ACCORD, if aspects and components are implemented in conformance with the guidelines presented in section 4.2.2.

The aspect-level WCET specification of an aspect and a component consists of internal and external WCET specifications. *The internal*

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 4.1: Aspect-level WCET specifications of aspects and components

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 not 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 4.1 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 specifications and variable external WCET specifications (see table 4.1). External WCETs of an operation is specified through usage of mechanisms in the operation as aspect weaving changes the operation implementation by changing the number of the 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 4.17 illustrates the aspect-level WCET specifications for the

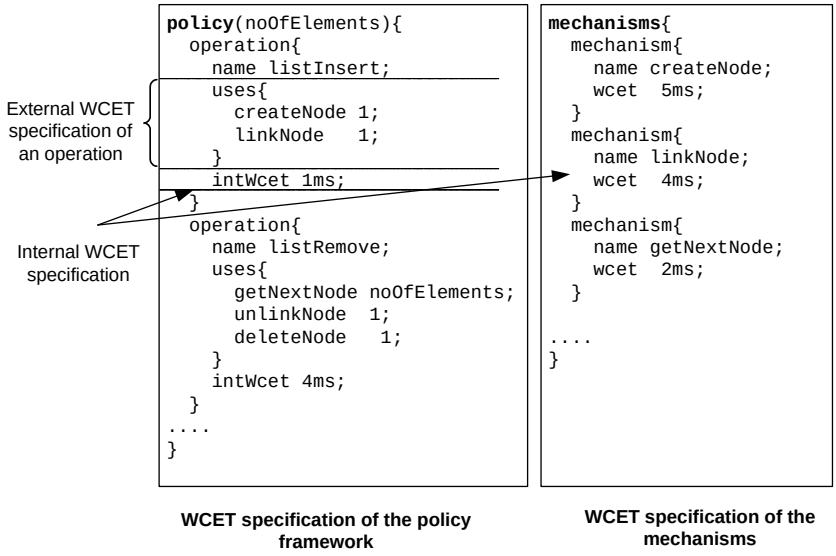


Figure 4.17: Aspect-level WCET specifications of the operations and mechanisms of the locking component

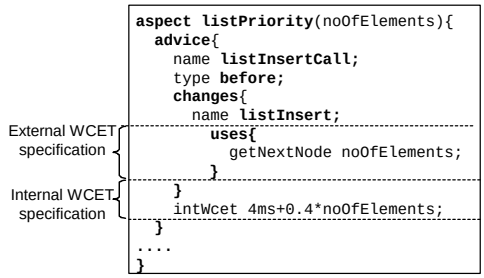
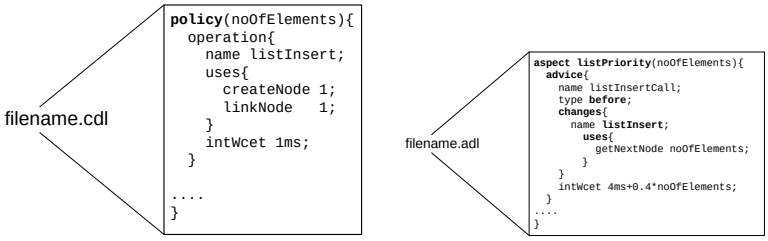


Figure 4.18: The aspect-level WCET specification of the `CCpolicy` aspect



(a) Component description language

(b) Application aspect description language

Figure 4.19: An example of the input and output files

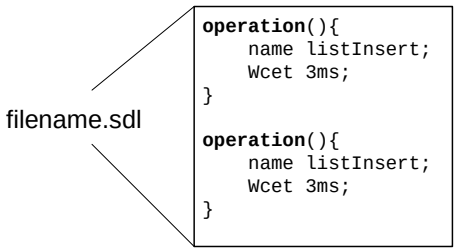


Figure 4.20: An example of the output file of the aspect-level WCET analyzer

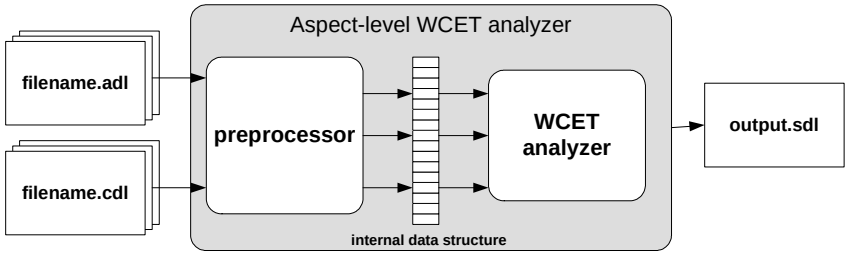


Figure 4.21: Main constituents of the aspect-level WCET analyzer

linked list component. The aspect-level specification for the aspect `listPriority` changing the code of the linked list component is shown in figure 4.18.

Aspect-level WCET specifications of aspects and components are inputs to the aspect-level WCET analyzer. The specifications are currently implemented such that an aspect-level WCET specification of a component is contained in a file that has the extension *cdl* (component description language) (see figure 4.19(a)), while aspect-level WCET specifications of aspects have the extension *adl* (aspect description language) (see figure 4.19(b)). The tool outputs the file with an extension *sdl* (system description language) that contains all the operations of the components in the configuration of the real-time system under analysis, and their respective resulting WCETs (see figure 4.20).

4.4.2 Aspect-Level WCET Analyzer

The aspect-level WCET analyzer consists of two main parts, the preprocessor and the WCET analyzer, as shown in figure 4.21. In the remainder of this section we give detailed description of these parts of the tool and discuss their interaction.

Preprocessor

The task of the preprocessor of the aspect-level WCET analyzer is to transform the information contained in aspect-level WCET specifica-

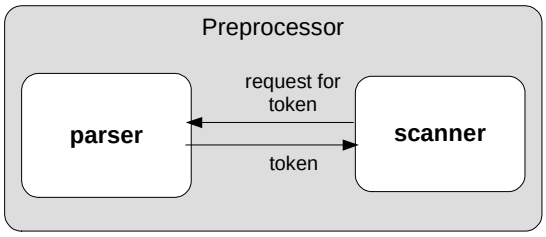


Figure 4.22: The structure of the preprocessor

tions, which are given as an input in the form of the files with extensions cdl and adl, to a form useful for the WCET analyzer. The preprocessor analyzes the WCET specifications given, and produces data structures containing the WCET values and interdependency information for all components and aspects. These data structures are internal to the aspect-level WCET analyzer and they are coupling the preprocessing and the analyzing part of the tool (see figure 4.21).

The preprocessor has a typical structure consisting of a parser and a scanner as shown in figure 4.22. It was implemented using Flex [77] and Bison [34], which are code-generating tools designed to assist in compiler development. Their detailed description is out of the scope of this thesis, and here we only give a short overview of these tools focusing primarily on the issues important to understand preprocessing part of the aspect-level WCET analyzer. Flex, a fast lexical analyzer, is a tool for generating scanners, which are programs that recognize lexical patterns in the text. Bison is a tool used for generating parsers, which convert the grammar descriptions into a C or C++ program to facilitate easy parsing.

The parser and scanner are interrelated as parser cannot complete its actions without the interaction with the scanner. As shown in the figure 4.22, the interaction is realized in the form of token request-reply. The parser breaks down all the expressions and sentences in the description files into words, i.e., tokens, and in communication with scanner checks if the token is a valid part of the description language. It then groups the tokens into sentences based on the grammar rules. When a

grammar rule is recognized, a semantic action is executed. The semantic action for the preprocessor is to generate a data structure to store the information contained in the aspect-level WCET specification files. The structures created by semantic actions are realized in the form of maps and vectors to facilitate fast access (for the WCET analyzer) to the data. The data structures store: (i) values of internal and external WCETs for operations and advices, (ii) values of external WCETs of and mechanisms, (iii) parameters existing in the symbolic expressions of operations, mechanisms, and advices, and (iv) dependency information, e.g., the mechanisms used by an operation, and advices modifying an operation.

WCET Analyzer

The WCET analyzer implements the aspect-level WCET algorithm that computes the WCETs of components weaved with aspects. After the preprocessing step, the internal data structures created by the preprocessor contain the parameters and the WCET values needed by the algorithm. Since internal WCETs in the aspect-level specifications are symbolic expressions, the values of these need to be determined, and the fist step is to obtain the values of parameters in the expressions.

This is done by the aspect-level WCET analyzer in the step before actually applying the aspect-level WCET algorithm. The global function `checkParameters()` of the aspect-level WCET analyzer checks the data structures created in the preprocessing step detecting the parameters of operations and mechanisms (used in symbolic expressions), and prompts the human user for their values. The values of the parameters of an operation are then stored within the same data structure (e.g., an object) in a vector that contains all the information and a WCET values for a particular operation. The resulting parameterized data structures are then used by the WCET analyzer as input to calculate the WCETs of all the operations within the real-time system configuration under development.

Algorithm aspect-level WCET

```
WCETanalyzer ()
 1: for every operation_i do
 2:    newWCET=operationWCET(operation_i)
 3: end for

operationWCET (operation)
 1: operationWCET= 0
 2: if an advice is modifying the operation then
 3:    for every advice_i in the aspect_k modifying the operation do
 4:       if around advice then
 5:          operationWCET=operationWCET+codeBlockWCET(advice_i)
 6:       else return before or after advice
 7:          operationWCET=operationWCET+codeBlockWCET(advice_i)+
             codeBlockWCET(operation)
 8:       end if
 9:    end for
10: else
11:    operationWCET=codeBlockWCET(operation)
12: end if
13:
14: if operation requires other operations then
15:    for every operation_k required by the operation do
16:       operationWCET=operationWCET+operationWCET(operation_k)
17:    end for
18: end if
19: return  operationWCET

codeBlockWCET (codeBlock)
 1: codeBlockWCET=intcodeBlockWCET
 2: for every mechanism_i used by the codeBlock do
 3:    codeBlockWCET=codeBlockWCET+WCET_i * N_i
 4: end for
 5: return  codeBlockWCET
```

The aspect-level WCET algorithm, used for calculating the total WCET of components, possibly colored with aspects, consists of three interdependent parts (top-down description):

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

`operationWCET()` computes the WCET of an operation taking into account that the operation might be modified by aspect weaving. If the operation is modified by aspect weaving, the following is applicable (lines 2-10). 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 a `codeBlockWCET()`, where around advice now is a code block (lines 4-5). 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 (lines 6-10). If the operation is not modified by aspect weaving, then the above described actions are ignored (lines 2-10) and the value of the WCET of the operation is obtained simply by calling `codeBlockWCET()` (lines 10-12). 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 (lines 14-18).

`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 (line 1). 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` (lines 2-4) such that the value of WCET of a mechanism (a symbolic expression) is multiplied with the number of times the `codeBlock` uses the mechanism (line 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. Here, to simplify the explanations, we consider a simple configuration consisting of one component and one aspect, namely linked list component and the `listPriority` aspect. The aspect-level WCET specification of these we already discussed in 4.4.1. Hence, if the specifications of liked list component in figure 4.17, and the aspect `listPriority` in figure 4.18, are given as an input to the aspect-level WCET analyzer, then the preprocessor extracts the information needed by the WCET analyzer in a suitable form. In this case, the information would be in the form illustrated in figure 4.23. The policy object is created for the linked list component containing the pointers to the list of parameters found in symbolic expressions, and the list of operations. In this case, the list of parameters consists of only one parameter `noOfElements`, while the list of operations consists of all the operations within the linked list component, e.g., `listInsert` operation. The aspect object is also created and contains the pointer to all the advices within the aspect. In our example, only one advice `listInsertCall` exists, thus, there is only one object created for that advice. The advice object consists of the name of the advice, type of the advice, the list of all mechanisms used by the advice, the pointer to the list of parameters, and the node that

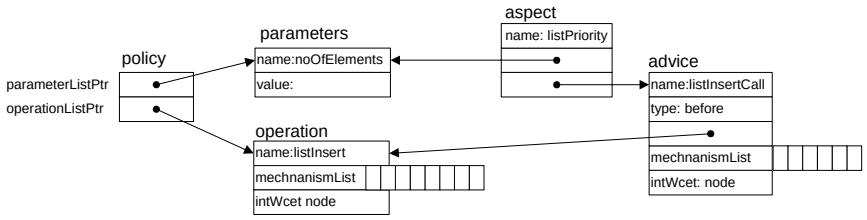


Figure 4.23: An example of internal data structures in the aspect-level WCET analyzer

points to the symbolic expression representing the internal WCET of the advice. Similarly, for the operation `listInsert` an object is created that contains the name of the operation, the list of mechanisms used by the operation, and the node of the symbolic expression. The list of mechanisms contains names of each of the mechanisms used by the operation `listInsert` together with the number of times `listInsert` uses a particular mechanism. When these data structures are created by the preprocessor, the function `checkParameters()` detects that there exists a parameter `noOfElements`, and prompts the human user for its value. Let us assume that, when prompted by the aspect-level WCET analyzer, we set this value to 5. Now the WCET analyzer can apply the algorithm to compute the WCET values of the operations in the linked list component as follows. Since the operation `listInsert` is modified by the advice `listInsertCall` of the `listPriority` aspect, the WCET analyzer applies lines 2-10 of the `operationWCET()` part of the aspect-level WCET algorithm to compute a new value of the WCET of the operation `listInsert` weaved with the before advice `listInsertCall`, as follows:

$$operationWCET = operationWCET + codeBlockWCET(listInsertCall) + codeBlockWCET(listInsert).$$

This results, after applying the `codeBlockWCET()` part of the algorithm, in:

$$operationListInsertWCET = 0 +$$

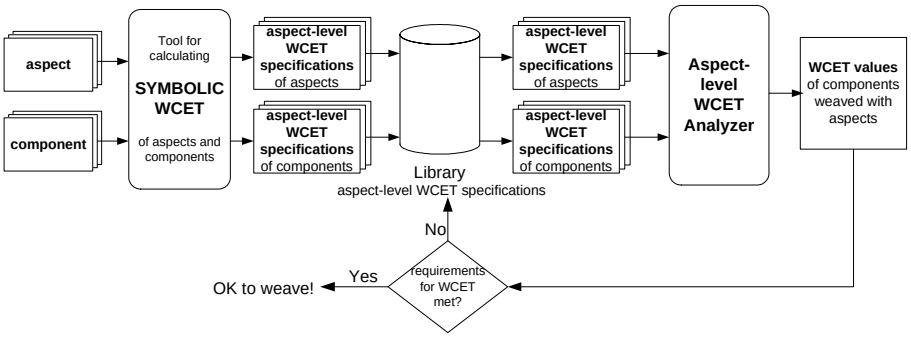


Figure 4.24: An overview of the aspect-level WCET analysis lifecycle

$$(1 + createNodeWCET * 1 + linkNodeWCET * 1) +$$
$$(4 + 0.4 * noOfElements + getNextNode * noOfElements) =$$
$$= 0 + (1 + 5 * 1 + 2 * 4) + (4 + 0.4 * 5 + 2 * 5) = 30.$$

4.4.3 Limitations and Benefits

The aspect-level WCET analyzer assumes that the aspect-level WCET specifications of the aspects and components are provided by symbolic WCET analysis. Hence, the aspect-level WCET analyzer depends on the ability to extract this WCET information (in the form presented in this section) from aspects and components.

Ideally, the complete process of the aspect-level WCET analysis should have a lifecycle as presented in figure 4.24. 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, and extracts these into aspect-level WCET specifications. These specifications are stored in a library and are used by the aspect-level WCET analyzer, which computes the WCET the different configurations of components and aspects to determine the 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 4.24 also illustrates limitations of current automated aspect-level WCET analysis. The tool that computes WCETs in the form of symbolic expressions and extracts these to aspect-level WCET specifications should be an adaptation of the tool for symbolic WCET analysis to the aspect-level WCET analysis. The current implementation of the aspect-level WCET analyzer works only with aspect-level WCET specifications. The current implementation, although given the limitations, provides benefits over traditional WCET analysis performed on weaved code since 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.

4.5 ACCORD Evaluation

This section provides an evaluation of ACCORD and it main constituents. The evaluation is performed by relating the design criteria for component-based real-time systems we identified in chapter 3 to ACCORD. The recap of the main criteria identified previously is given in table 4.2. The table also gives the comparison of ACCORD and the design approaches discussed previously (see chapter 3), as well as its evaluation with respect to the criteria.

Component model - relation to ACCORD. ACCORD provides a component model for real-time systems, which enforces information hiding and supports three different types of interfaces, hence, fulfilling the criteria CM1 and CM2, respectively (see section 4.2). The method assumes that a real-time system should first be decomposed into a set of components, which are later mapped to tasks. Hence, the relationship between tasks and components is not fixed. ACCORD has adopted and refined task structuring criteria from DARTS to provide guidelines

Design approaches / Criteria		DARTS	TRSD	VEST	ISC	COM	AspectJ	HRT-HOOD	ACCORD
Component model									
CM1	Information hiding	◐	◐	◐	●	●	◐	●	●
CM2	Interfaces	◐	◐	◐	●	●	●	●	●
CM3	Component Views	◐	◐	●	◐	-	-	◐	●
CM4	Temporal attributes	-	◐	●	-	-	-	●	●
CM5	Task mapping	●	-	◐	-	-	-	-	◐
Aspect separation									
AS1	Aspect support	-	-	◐	●	-	●	-	●
AS2	Aspect weaving	-	-	-	●	-	●	-	●
AS3	Multiple interfaces	-	-	-	●	●	◐	◐	●
AS4	Multiple aspect types	-	-	-	-	-	-	-	●
System composability									
SC1	Configuration support	◐	◐	●	●	◐	●	◐	◐
SC2	Temporal analysis	-	-	◐	-	-	-	●	◐

LEGEND: ● supported ◐ partially supported - not supported

DARTS: desing approach for real-time systems
TRSD: transactional real-time system design
ISC: invasive software composition
VEST: Virginia embedded systems toolkit
HRT-HOOD: a hard real-time hierarchical object-oriented desing
ACCORD: Aspectual component-based real-time system development

Table 4.2: Evaluation criteria for ACCORD

for mapping of components into tasks (see section 4.3). However, the guidelines need to be refined to include issues not addressed, such as inter-process communication. Hence, ACCORD partially fulfills CM5 (task mapping).

ACCORD provides a description language for defining temporal attributes of components (see section 4.2.3), corresponding to the CM4 criterion, and aims at providing tools that can automatically extract temporal information from any real-time software component build on RTCOM. The description language for temporal attributes of components and guidelines for mapping of components to tasks, enable ACCORD to enforce the CM3 criterion by supporting both temporal and structural views of components and the overall real-time system.

Aspect separation - relation to ACCORD. ACCORD enforces the criterion AS1 and AS4 by supporting three different types of aspects: application, run-time and composition aspects (see section 4.1). It also fulfills the criterion 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 criterion AS3 is supported as RTCOM provides three types of interfaces: functional, configuration and composition interfaces (see section 4.2.5).

System composability - relation to ACCORD. ACCORD provides configuration support for the component-based real-time system design by providing design guidelines, but not the tools, for configuration of the real-time system using components and aspects, hence partially meeting the criterion SC1. ACCORD also partially enforces the SC2 criterion as it supports static WCET analysis of composed system (see section 4.4); the dynamic schedulability analysis is not automated and relies on the guidelines for the task mapping, and, hence, requires additional refinements (see section 4.3.1).

Chapter 5

Applying ACCORD to COMET: a Case Study

In this chapter we present how we applied ACCORD when developing COMET - a component-based embedded real-time database. The goal with the COMET platform is to enable development of a configurable real-time database for embedded systems, i.e., enable development of different database configurations for different embedded and real-time applications. The types of requirements we are dealing with can best be illustrated by example of one of the COMET targeting application areas: vehicle control systems. Hence, first we present a study of two different hard real-time systems developed at Volvo Construction Equipment Components AB, Sweden, with respect to data management (section 5.1). We then show how we have reached our goal by applying ACCORD when developing COMET. We present decomposition of COMET into components (section 5.2), the decomposition of COMET into aspects (section 5.3), and a detailed description of the implementation of COMET components based on RTCOM (section 5.4). The chapter finishes with a discussion on experienced benefits and drawbacks of ACCORD (section 5.5).

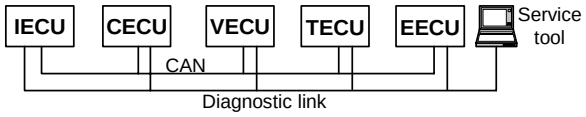


Figure 5.1: The overall architecture of the vehicle controlling system

5.1 Data Management in 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 [68]. These systems are embedded into two different vehicles, an articulated hauler and a wheel loader. These are typical representative real-time systems for this class of vehicular systems. Both of these vehicle control systems consist of several subsystems called electronic control units (ECU), connected through two serial communication links: the fast CAN link and the slow diagnostic link, as shown in the figure 5.1. 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 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 cost of

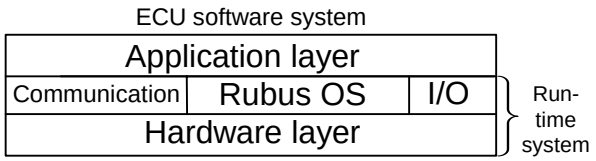


Figure 5.2: The structure of an ECU.

the ECU 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 5.2). 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 real-time operating system Rubus. 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 (in following sections we refer to the application layer of the VECU and IECU as the VECU (software) system and the IECU (software) system).

5.1.1 Rubus

Rubus is a real-time operating system designed to be used in systems with limited resources [9]. Rubus supports both off-line and on-line scheduling, and consists of two parts: (i) red part, which deals with hard real-time; and (ii) 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 (referred to as blue tasks). The blue part supports tasks that can be invoked in an event-driven manner. The blue part of Rubus 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 red tasks. Rubus is used in all the ECUs.

5.1.2 VECU

The vehicle system is used to control and observe the state of the vehicle. The system can identify anomalies, e.g., an unnormal temperature. Depending on the criticality of the anomaly, different actions, such as warning the driver, system shutdown etc., can be taken. 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 being 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 non-preemptive tasks run until completion and cannot be preempted, mutual exclusion is not necessary. The reason for using non-preemptive off-line scheduled tasks is to minimize the runtime overhead and to simplify the verification of the system.

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 engineering 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 *sen-*

sor/actuator parameter data, is used to derive the *sensor/actuator engineering 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 engineering 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}, e.g., $T = T_{volt} \cdot T_{ref}$.

In the current system, the sensor/actuator (raw and parameter) data are stored in a vector of data called a hardware database (HW Db), see figure 5.3. The HW Db is, despite its name, not a database but merely a memory structure. The engineering 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 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 certain regulations, e.g., pollution and environment regulations. The changeable parameters can be changed using a service tool.

Most controlling 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

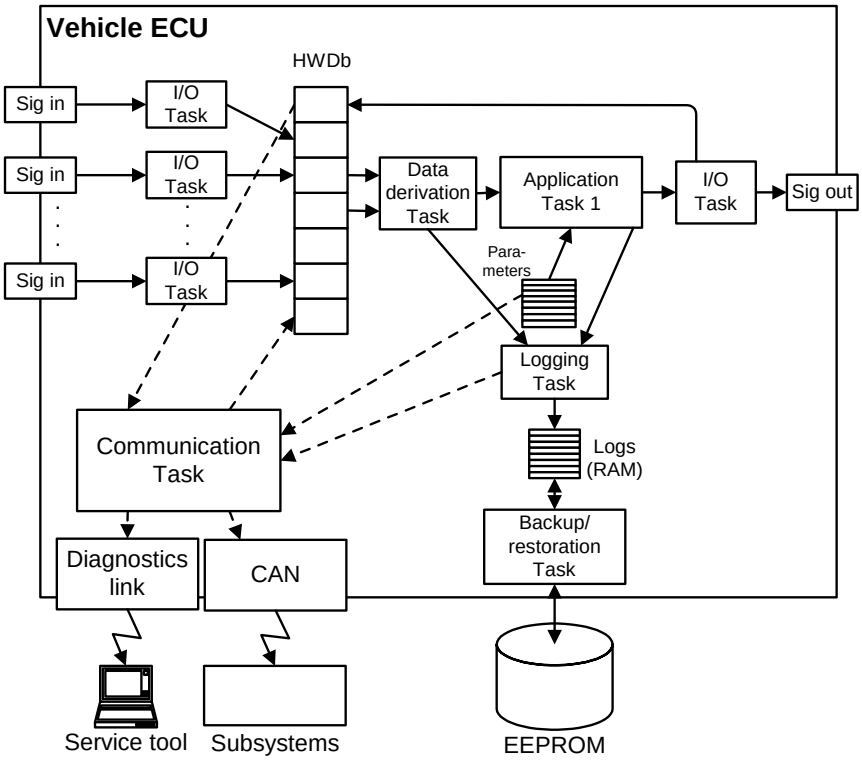


Figure 5.3: The architecture of the VECU

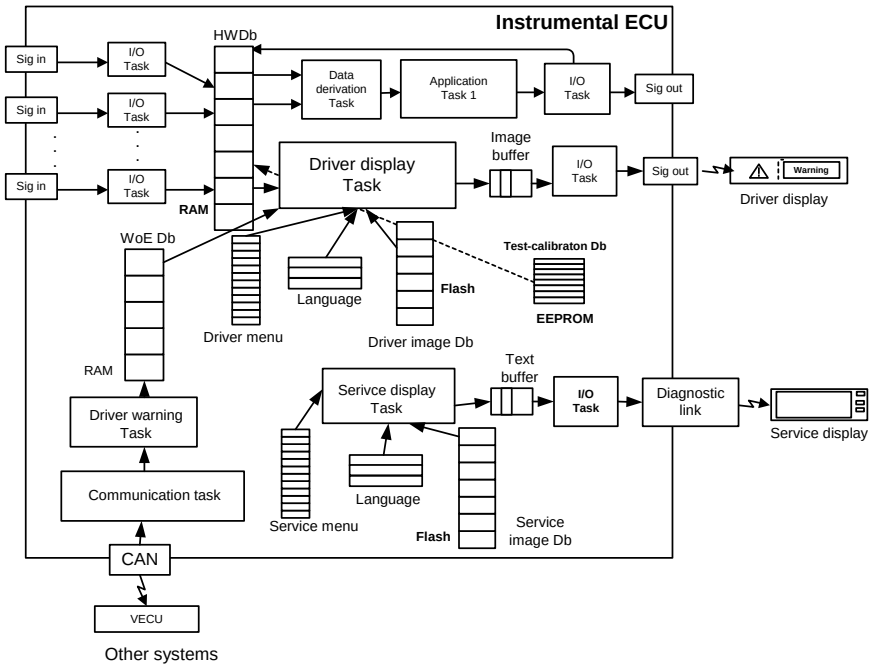


Figure 5.4: The architecture of the IECU

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.

5.1.3 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 displaying service information on the service display (a unit for servicing the vehicle). It furthermore controls the I/O in the driver cabin, e.g.,

accelerator pedal, and communicates 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 5.4. 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 it differs from other red tasks in the system since it can be preempted by other red tasks in the IECU. 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 5.4). The driver display task periodically scans the databases (HW Db, WoE Db, and menu Db) to determine the information that needs to be displayed on the driver display. If any active WoE exists 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 hardware database 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 case the 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.

5.1.4 Data Management Requirements

Table 5.1 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.

As can be seen from the table 5.1, all data elements in both systems are scattered in groups of different flat data structures referred to as databases, e.g., HW Db, image Db, WoE Db and language Db. 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 put special requirements on data management (see table 5.1): (i) static memory allocation only, since dynamic memory allocation is not allowed due to the safety-critical aspect of the systems; (ii) small memory consumption, since production costs should be kept as low as possible; and (iii) 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 engineering values

by the data derivation task, as explained in section 5.1.2. The engineering values are not stored in any of the databases, rather they are placed in ports (shared memory) and given to application tasks when needed.

Data types		Sensor	Actuator	Engineering	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 5.1: Data management characteristics for the systems

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

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

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 is not temporally constrained (see table 5.1).

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. Violation of temporal consistency is viewed as a system error, in which case three possible actions can be taken by the system: use a predefined default data value (most often), use an old data value or shutdown of the functions involved (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 5.1). 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 systems is maintained by storing data on stable storage, but there are some exceptions to the rule, e.g., RPM data is never copied to stable storage. Also, some of the data is only stored in stable storage, e.g., internal system parameters. In contrast, data imperative to systems' functioning is immediately copied to stable storage, e.g., WoE logs are copied to/from 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 sensor value update tasks; (iii) read-only transactions, which are actuator reading tasks; 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 5.1). 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 resides 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;
- engineering 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.

5.1.5 Observations

The vehicle control systems are typically hard real-time safety-critical systems consisting of several distributed nodes implementing 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. Depending on the functionality of a node and the available memory, different database configurations are preferred. In safety-critical nodes 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, 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, in the narrow sense of this application area, the goal was to enable development of different COMET configurations to suit the needs of each node with respect to memory consumption, concurrency control, recovery, different scheduling techniques, transaction models and storage models. In the following sections we show how we have reached our goal by applying ACCORD to the design and development of the COMET system.

5.2 COMET Components

Following the ACCORD design method presented in chapter 4 we have first performed the decomposition of COMET into a set of components with well-defined functions and interfaces. COMET has seven basic components (see figure 5.5): user interface component, transaction scheduler component, locking component, indexing component, recovery and logging component, memory handling component, and transaction manager component.

The *user interface component* (UIC) enables users to access data in the database, and different applications often require different ways of accessing data in the system. All the operations on data in the database are received via the UIC. The main activities of the UIC consist of receiving and parsing the incoming requests from the application and the

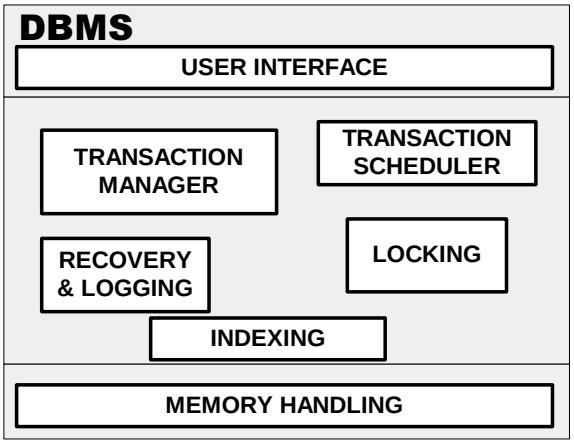


Figure 5.5: COMET functional decomposition

user. UIC takes the incoming requests and devises the execution plans.

The *transaction scheduler component* (TSC) provides mechanisms for performing scheduling of transactions coming into the system, based on the scheduling policy chosen. COMET is designed to support a variety of scheduling policies, e.g., EDF and RM [55]. The TSC is also in charge of maintaining the list of all transactions in the system, including scheduled transactions as well as unscheduled but active transactions, i.e., transactions submitted for execution. Hard real-time applications, such as real-time embedded systems controlling a vehicle, typically do not require sophisticated transaction scheduling and concurrency control, i.e., the system allows only one transaction to access the database at a time [68]. Therefore, the TSC should be a flexible and exchangeable part of the database architecture.

The *locking component* (LC) deals with locking of data, and it provides mechanisms for lock manipulation and it maintains lock records in the database. The LC provides the policy framework for the lock administration in which all locks are granted. This policy framework can

be changed into a specific policy according to which the LC deals with lock conflicts by weaving concurrency-control aspect (see section 5.4).

The *indexing component* (IC) deals with indexing of data. Indexing strategies could vary depending on the real-time application with which the database should be integrated, e.g., t-trees [58] and multi-versioning suitable for applications with a large number of read-only transactions [87]. 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, non-thread safe indexing can be used, while in more complex applications appropriate aspects could be weaved into the component to allow thread-safe processing of indexing strategy (this can be achieved by weaving the synchronization aspect).

The *memory handling component* (MHC) manages access to data in the physical storage. For example, each time a tuple is added or deleted, the MHC is invoked to allocate and release memory. Generally, all reads or writes to/from the memory in COMET involve the MHC.

The *transaction manager component* (TMC) coordinates the activities of all components in the system with respect to transaction execution. For example, the TMC manages the execution of a transaction by requesting lock and unlock operations provided by the LC, followed by requests to the operations, which are provided by the IC, for inserting or updating data items.

5.3 COMET Aspects

Following ACCORD, after decomposing the system into a set of components with well-defined interfaces, we decompose the system into a set of aspects. The decomposition of COMET into aspects is presented in figure 5.6, and it fully corresponds to the ACCORD decomposition (given in section 4.1) in three types of aspects: run-time, composition, and application aspects. As COMET is a real-time database system, the application aspects are made to reflect both real-time and database issues. Hence, in the COMET decomposition of application aspects,

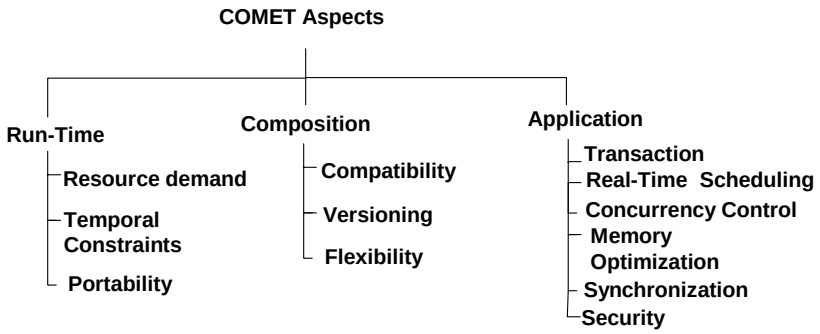


Figure 5.6: Classification of aspects in an embedded real-time database system

the real-time policy aspect is refined to include real-time scheduling and concurrency control policy aspects, while the real-time property aspect (in ACCORD) is replaced with the transaction model aspect, which is database-specific. The crosscutting effects of the application aspects to COMET components are shown in table 5.2. As can be seen from the table, all identified application aspects crosscut more than one component. For example, the concurrency control (CC) aspect crosscuts several components, namely TSC, LC, and TMC in the following manner. The TMC is responsible for invoking the LC to obtain and release locks. The way the LC is invoked by the TMC depends on the CC policy enforced in the database and, hence, needs to be adjusted separately for each type of CC policy, i.e., each type of the CC aspect. Furthermore, the way to deal with lock conflicts is enforced by the LC. Hence, the LC should be modified with CC aspect to facilitate lock resolution policy prescribed by the CC policy of the CC aspect. Since scheduling and CC are tightly coupled in the sense that CC polices typically require information about the transactions in the system maintained by the TSC, this means that the TSC should be modified by CC aspect to provide adequate support for the chosen CC policy.

The application aspects could vary depending on the particular ap-

Components / Application aspects	UIC	TSC	LC	IC	RLC	MHC	TMC
Transaction	X	X	X	X	X	X	X
Real-time scheduling		X					X
Concurrency control		X	X				X
Memory optimization	X	X	X	X	X		X
Synchronization		X	X	X	X		X
Security	X		X	X		X	X

Table 5.2: Crosscutting effects of different application aspects on the COMET components

plication of the real-time system, thus particular attention should be made to identify the application aspects for each real-time system.

5.4 COMET RTCOM

Components and aspects in COMET are implemented based on RTCOM (discussed in section 4.2). Hence, the functional part of components is implemented first, together with application aspects. We illustrate this process, its benefits and drawbacks, by the example of one component (namely LC) and one application aspect (namely concurrency control).

The LC performs the following functionality: assigning locks to requesting transactions and maintaining a lock table, thus, it records all locks obtained by transactions in the system. As can be seen from the table 5.2, the LC is crosscut with several application aspects. The application aspect that influences the policy, i.e., changes the behavior of the LC, is concurrency control (CC) aspect, which defines the way lock conflicts should be handled in the system. To enable tailorability of the LC,

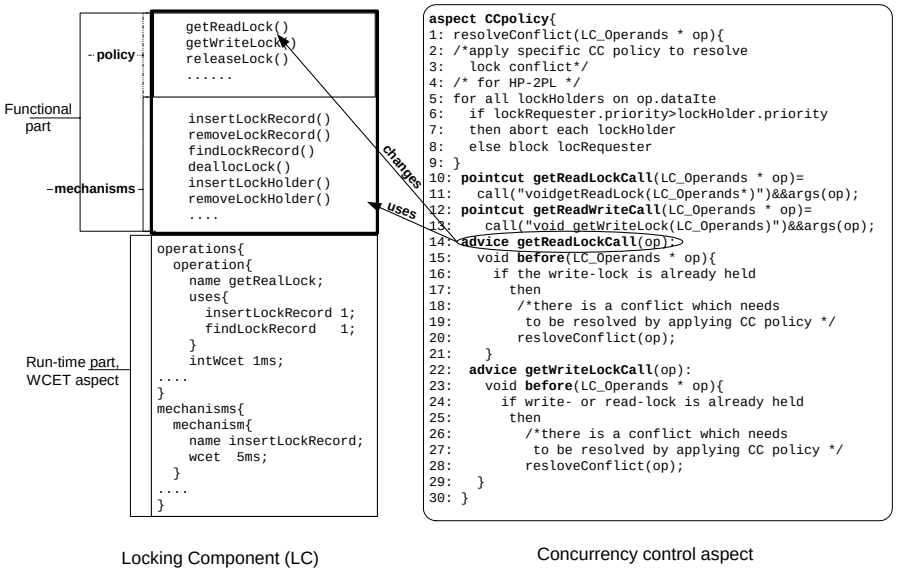


Figure 5.7: The locking component and the concurrency control aspect

and reuse of code to the largest possible extent, the LC is implemented with a policy framework where lock conflicts are ignored and locks are granted to all transactions. The policy framework can be modified by weaving CC aspects that define other ways of handling lock conflicts. As different CC policies in real-time database systems exist, the mechanisms in the LC should be compatible with most of the existing CC algorithms.

The LC contains mechanisms such as (see left part of the figure 5.7): `insertLockRecord()`, `removeLockRecord()`, etc., for maintaining the table of all locks held by transactions in the system. The policy part consists of the operations performed on lock records and transactions holding and/or requesting locks, e.g., `getReadLock()`, `getWriteLock()`, `releaseLock()`. The operations in the LC are implemented using underlying LC mechanisms. The mechanisms provided by the LC are used by the CC aspects implementing the class of pessimistic (locking) protocols, e.g., HP-2PL [5] and RWPCP [88]. However, as a large class of optimistic protocols is implemented using locking mechanisms, the mechanisms provided by the LC can also be used by CC aspects implementing optimistic protocols, e.g., OCC-TI [53] and OCC-APR [30].

The right part of figure 5.7 represents the specification for the real-time CC aspect (lines 1-30) that can be applied to a class of pessimistic locking CC protocols. We choose to give more specific details for the HP-2PL protocol as it is both commonly used in main-memory database systems and a well-known pessimistic CC protocol.

The CC aspect has several pointcuts and advices that execute when the pointcut is reached. As defined by the RTCOM pointcut model, the pointcuts refer to the operations `getReadLockCall()` and `getWriteLockCall()` (lines 10 and 12). The first pointcut intercepts the call to the function `getReadLock()`, which grants a read lock to the transaction and records it in the lock table. Similarly, the second pointcut intercepts the call to the function that gives a write lock to the transaction and records it in the lock table. Before granting a read or write lock, the advices in lines 14-21 and 22-29 check if there is a lock conflict. If a conflict exists, the advices deal with it by calling the local aspect function `resolveConflict()` (lines 1-9), where the resolution of

the conflict should be done by implementing a specific CC policy. The advices that check for conflicts are implemented using the LC mechanisms to traverse the lock table and the list of transactions holding locks.

So far we have shown that the CC aspect affects the policy of the LC, but the CC aspect also crosscuts other components (see table 5.2). In the example of the CC aspect implementing pessimistic HP-2PL protocol (see figure 5.7), the aspect uses the information about transaction priority (lines 5-8), which is maintained by the TSC, thus crosscutting the TSC. Optimistic protocols, e.g., OCC-TI, require additional pointcuts to be defined in the TMC, as the protocols (as compared to pessimistic protocols) assume execution of transactions in three phases: read, validate, and write.

Additionally, depending on the CC policy implemented, the number of pointcuts and advices varies. For example, some CC policies (like RWPCP, or optimistic policies) require additional data structures to be initialized. In such cases, an additional pointcut named `initPolicy()` could be added to the aspect that would intercept the call to initialize the LC. The before advice `initPolicy` would then initialize all necessary data structures in the CC aspect after data structures in the LC have been initialized.

5.5 Wrap-up

Here, we give the benefits and drawbacks of applying ACCORD to the development of COMET platform. We use the given example of the LC and CC aspect (see section 5.4) to draw our conclusions. The benefits of applying ACCORD to the development of COMET platform are the following (in the context of the given example of the LC and CC aspect).

- Clean separation of concurrency control as an aspect that crosscuts the LC code is enabled, thus, allowing high code reusability as the same component mechanisms are used in almost all CC aspects.

- Efficient tailoring of the component and the system to fit a specific

requirement (in this case specific CC policy), as weaving of a CC aspect into the LC changes the policy of the component by changing the component code, and leaving the configuration of COMET unchanged.

- Having the LC functionality encapsulated into a component, and the CC encapsulated into an application aspect enables reconfiguring COMET to support non-locking transaction execution (excluding the LC), if other completely non-locking CC protocol is needed.

The drawbacks experienced in applying ACCORD to real-time system development are the following.

- A great number of components and aspects available for system composition can result in an explosion of possible combinations of components and aspects. This is a common problem for all software systems using aspect and components, and extensive research has being done in identifying and defining good composition rules for the components and aspects [19, 11, 10].

- The coarse-granularity of RTCOM may result in non-negligible component code overhead, e.g., due to a large number of mechanisms implemented in the component in order to support tailorability through weaving of application aspects.

Hence, there is a trade-off between achieving good tailorability and flexibility of components, tractable combinations of aspects and components, and the optimization of the component infrastructure, i.e., number of mechanisms, for a particular application.

Chapter 6

Related Work

This chapter focuses on alternative approaches and related research relevant to our work, namely component-based and aspect-oriented real-time and database systems. As the research in applying AOSD to real-time system development is in its early stages and, thus, considerably sparse, we focus primarily on component-based real-time systems and discuss how they relate to aspects. Hence, in section 6.1 we survey existing component-based real-time systems. In section 6.2 we discuss related work on aspects-oriented and component-based database systems. This chapter finishes with a tabular overview comparing COMET, an example of an ACCORD-based system, to other approaches.

6.1 Component-Based Real-Time Systems

We have identified three distinct types of component-based embedded real-time systems:

- Extensible systems. An example of this type of systems is SPIN [14], an extensible microkernel. Extensions in the system are possible by plugging components, which provide non-standard features or functionality.
- Middleware systems. These are characterized by providing efficient

management of resources in dynamic heterogeneous environments, e.g., 2K [47] is a distributed operating system specifically developed for management of resources in a distributed environment.

- Configurable systems. An architecture of a configurable system allows new components to be developed and integrated into the system. Components in such systems are true building parts of the system. A variety of configurable systems exists, e.g., VEST [93], Ensemble [56], and the port-based object (PBO) approach [98].

6.1.1 Extensible Systems

SPIN

SPIN [14, 75] 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.

6.1.2 Middleware Systems

2K

2K [49, 47] 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 6.1, 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.

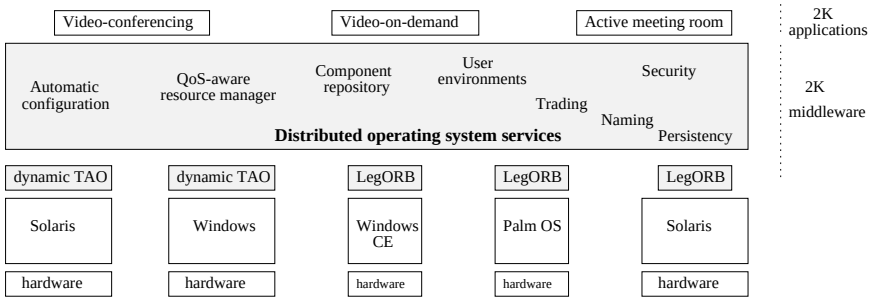


Figure 6.1: The 2K middleware architecture

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 [48]. 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 the development of new components is done using CORBA component specifications [71]. 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).

6.1.3 Configurable Systems

Ensemble

Ensemble is a high performance network protocol architecture designed to support group membership and communication protocols [56]. 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, message ordering etc. 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 [56] 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 [93]. 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 6.2). 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.

It should be noted that components in VEST are passive (collection of code fragments, functions and objects) and are mapped into run-time structures (tasks). Each component can be composed out of subcomponents. For example, the task management component can be made of components such as create task, delete task, and set task priority. Components have 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, each component has 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

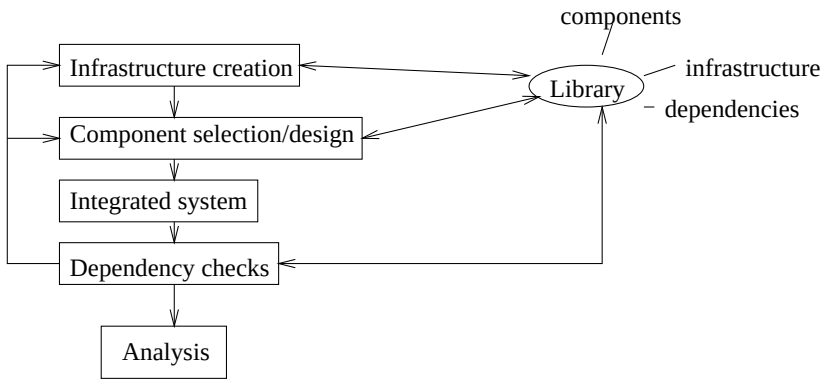


Figure 6.2: Embedded system development in VEST

be available. Dependency checks proposed in VEST are one good way of providing configuration support and the strength of the VEST approach. 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. Interface problems in VEST are only identified but are not further addressed; thus it is not obvious how

components can be interconnected. Also, analysis of the system is possible plugging off-the shelf analysis tools into the VEST platform. Finally, VEST is an ongoing project developing the platform for configuration and analysis of embedded real-time systems.

PBO Model

A component-based system based on the PBO model can be classified as configurable, and is suitable for development of embedded real-time control software system [98]. 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.

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.

6.2 Aspects and Components in Database Systems

6.2.1 Aspects in Database Systems

In the area of database systems the AOD [1], 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 [85].

The AOD initiative separates aspects in database systems in two levels [86]:

- 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 [84], a semi-autonomous database evolution system.

Following is a description of main features of SADES with the focus on aspect support.

SADES

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 [86]. 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, i.e., the classes, their member definitions, definition scopes, etc.,

- 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 [86]:

- 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 the real-time issues at all. Although it is claimed that the SADES approach to aspect support could be applied to existing component-based database systems [86], it is not clear how this can be achieved since the components in SADES are typical AOSD-type components, i.e., white-box components.

6.2.2 Components in Database Systems

Four different categories of component-based database management systems (CDBMSs) have been identified in [32]:

- *Extensible DBMS* extends existing DBMS with non-standard functionality, e.g., Oracle8*i* [72], Informix Universal Server with its

DataBlade technology [43], Sybase Adaptive Server [70], and DB2 Universal Database [23].

- *Database middleware* integrates existing data stores into a database system and provide users and applications with a uniform view of the entire system, e.g., OLE DB [65].
- *DBMS service* provides database functionality in a standardized form unbundled into services, e.g., CORBAService [74].
- *Configurable DBMS* enables composition of a non-standard DBMS out of reusable components, e.g., KIDS [38].

Sections that follow focus on characteristics of systems in each of these four different categories.

6.2.3 Extensible DBMS

Oracle8*i*

Oracle8*i* allows developers to create their own application-domain-specific data types [72]. 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 pre-defined 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

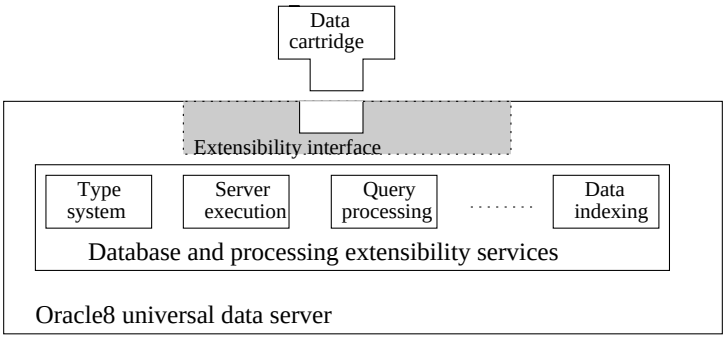


Figure 6.3: The Oracle extensibility architecture

all application domains, e.g., real-time system, since there is no analysis support for checking temporal behavior.

Informix DataBlade Technology

DataBlade modules are standard software modules that can be plugged into the Informix Universal Server database to extend its capability [43]. 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 [23, 31] 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 database systems in this category, the Sybase Adaptive Server [70] 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.

6.2.4 Database Middleware

OLE DB

OLE DB [16, 17] 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 [65]:

> "... the combination of both process and data into a secure, reusable object..."

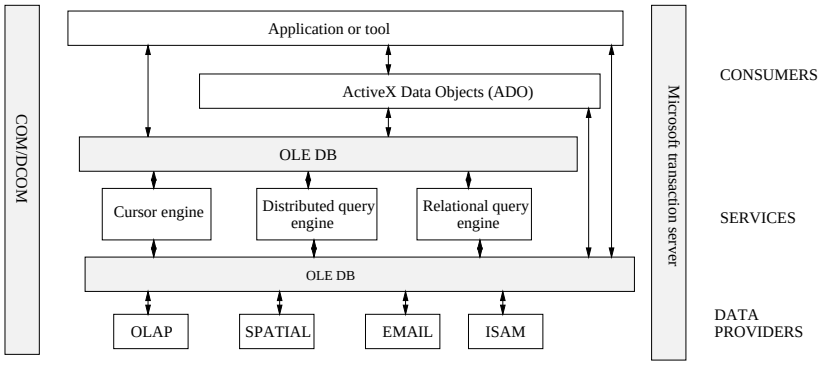


Figure 6.4: The Universal Data Access (UDA) architecture

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 6.4). The architecture in figure 6.4 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 [18]. 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 the system. Additionally, OLE DB is limited with respect to software platforms, since it can only be used in Microsoft software environments.

6.2.5 DBMS Service

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. CORBAservices are implemented on the top of the object request broker (ORB). Service interfaces are defined using the interface definition language [33]. In this scenario a component would be one of the database 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 [74]. Also, an adequate value-added framework that allows development of components and use of these components in other applications is still missing. Also, there is no support for performing configuration of the system nor analyzing it.

6.2.6 Configurable DBMS

KIDS

The KIDS [38], kernel-based implementation of database management systems, approach to constructing CDBMSs 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 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 6.5. 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

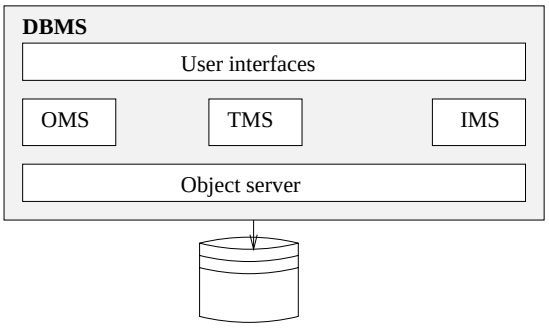


Figure 6.5: The KIDS subsystem architecture

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.

6.3 Tabular Overview

This chapter concludes with a tabular summary of investigated systems and their characteristics. The tables 6.1 and 6.2 provide an additional instrument for comparing and analyzing discussed approached, and contrasting them to COMET, an example of the real-time system built on ACCORD.
The following symbols are used in the table:

x — feature is supported in/true for the system, and

x/p — feature is partially supported in/true for the system, i.e, the system fulfills the feature to a moderate extent.

Below follows a description of the criteria.

Platforms / Characteristics of platforms		DBMS platforms: Oracle8i/ UDB2/ Sybase	DBMS platforms: OLE-DB	DBMS platforms: CORBA Service	DBMS platforms: KIDS	ACCORD/ COMET	Embedded and real-time platforms: SPIN	Embedded and real-time platforms: 2K	Embedded and real-time platforms: Ensemble	Embedded and real-time platforms: VEST	Embedded and real-time platforms: PBO
A. Type of the system	1) database	x	x	x	x	x					
	2) embedded					x	x	x	x	x	x
	3) real-time					x				x	x
B. Component granularity	1) system					x		x			
	2) part of the system	x	x	x	x	x	x	x	x	x	x
C. Category of the system	1) extensible	x					x				
	2) middleware		x					x			
	3) service			x							
	4) configurable				x	x			x	x	x
D. Component type	1) domain data type	x									
	2) service			x							
	3) CORBA object							x			
	4) microprotocol								x		
	5) passive									x	
	6)PBO										x
	7) general-purpose component model					x					

Table 6.1: Evaluation criteria for component-based real-time, embedded and database systems

Platforms / Characteristics of platforms		DBMS platforms					Embedded and real-time platforms				
		Oracle8i/ UDB2/ Sybase	OLE-DB	CORBA Service	KIDS	ACCORD COMET	SPIN	2K	Ensemble	VEST	PBO
E. Real-time properties	1) not preserved 2) preserved	x	x	x	x	x	x/p	x	x	x	x/p
F. Interface communication	1) standardized 2) system specific	x	x	x	x	x	x	x	x		x
H. Configuration tools	1) not supported 2) supported	x	x/p	x	x/p	x/p	x/p	x	x	x	x
I. Analysis tools	1) not supported 2) supported	x	x	x	x/p	x	x	x	x	x/p	x
E. Tailoring ability	1) low 2) moderate 3) high	x	x	x	x	x	x	x	x	x	x
F. Aspects	1) not supported 2) supported	x	x	x	x	x	x	x	x	x/p	x

Table 6.2: Evaluation criteria for component-based real-time, embedded and database systems

A. Type of the system We investigated integration of the component-based and aspect-oriented software development for the system development in the following areas:

1. database,
2. embedded, and
3. real-time.

As can be seen from table 6.1, there are few component-based systems that can be classified as embedded real-time; all component-based systems are either embedded and real-time systems, or database systems. COMET is the only component-based system that can be classified both as an embedded real-time, and a database system.

B. Component granularity There are two major granularity levels of components:

1. system as a component, and
2. part of the system as a component.

As can be seen from table 6.1, most embedded real-time systems have lightweight components, i.e., parts of the system. An exception is COMET where a component can even be the entire database system that needs to be integrated with a particular real-time and embedded system.

C. Category of the system The investigated component-based systems can be classified as follows (see sections 6.2.2 and 6.1):

1. extensible,
2. middleware,
3. service, and
4. configurable.

D. Component type A component in a component-based embedded real-time system and a component-based database system is one of the following:

1. domain-specific data type or a new index,
2. service,
3. CORBA object,
4. microprotocol,
5. passive component,
6. PBO, and
7. general-purpose component model.

In some systems components are not explicitly defined, and can only be classified as passive components, e.g., in VEST. Note that almost every system, with the exception of COMET that is built using a general RTCOM component model, has its own notion and a definition of a component that suites the purpose and requirements of the system.

E. Real-time properties Component-based database and embedded real-time systems may:

1. not preserve, or
2. preserve

real time properties. All component-based database systems, except COMET, do not enforce real-time behavior (see table 6.2). In addition, issues related to embedded systems such as low-resource consumption are not addressed at all. Accent in a database component is on providing a certain database functionality. In contrast, a component in existing component-based embedded real-time systems is usually assumed to be mapped to a task, i.e., passive components [93], PBO components [98], and RTCOM components are all mapped to a task.

F. Interfaces / communication The component communicates with its environment (other components) through well-defined interfaces. Generally, existing solutions use:

1. standard interfaces, or
2. system specific.

For example, standardized interfaces defined in the IDL are used in CORBAservices and in 2K. Also, OLE DB interface is used in the Microsoft's Universal Data Access architecture. Interfaces developed within the system are used in other systems, e.g., Oracle8*i* has extensibility interfaces and KIDS has component-specific interfaces. Inter-component communication in database systems and embedded real-time systems has different goals. Interfaces in embedded real-time systems must be such that inter-component communication can be performed in a timely predictable manner. There are two possible ways of a real-time component communication:

- Buffered communication. The communication is done through message passing, e.g., ENSEMBLE [56].
- Unbuffered communication. Unbuffered data is accessed through shared memory, e.g., PBO [97, 44].

Note that most component-based database systems use buffered communication since predictability of communication is not of importance in such systems. Traditionally, systems enforcing real-time behavior use unbuffered communication (an exception is VEST where interfaces of components are not defined), due to several disadvantages of buffered communication [42, 44]:

- Sending and receiving messages incur significant overhead.
- Tasks waiting for data might block for an undetermined amount of time.
- Crucial messages can get lost as a result of the buffer overflow if tasks do not execute at the same frequency.

- Sending messages in control systems, which have many feedback loops, creates a risk for deadlock.
- The upper bound on the number of produced/consumed messages must be determined to enable guarantee of temporal properties.

Generally, a real-time system using buffered communication is difficult to analyze due to dependencies among tasks. Unbuffered communication eliminates direct dependencies between tasks, since they only need to bind to a single element in the shared memory. Communication through shared memory incurs less overhead as compared to a message-passing system. Also, it is easier to check system temporal behavior if unbuffered communication is used [44]. Hence, unbuffered style of communication is the preferred style of communication in embedded real-time systems. It is suggested that interfaces in (hard) real-time systems should be unbuffered [44]. RTCOM (and thus COMET) has general enough interfaces to enable support for both buffered and unbuffered communication.

G. Configuration tools The development process should be well defined to enable efficient system assembly out of existing components from the component library or newly created ones. Adequate and automated configuration support must exist to help system designer with this process, e.g., rules for composing a system out of components, support for selection of an appropriate component from the component library, and support for the development of new components. However, in some systems configuration tools are not supported. Hence, we identify that configuration tools in investigated systems can be either:

1. not supported, or
2. supported.

In most configurable embedded real-time systems, some configuration support is provided. For example, PBO model gives good guidelines to help the designer when composing a system out of components. In VEST, the necessity of having good configuration tools is recognized.

Composition rules are defined through four types of dependency checks. COMET has been built using the ACCORD design guidelines.

H. Analysis tools Since the reliability of the composed system depends on the level of correctness of the component, analysis tools are needed to verify the behavior of the component and the composed system. In particular, real-time systems must meet their temporal constraints, and adequate analysis to ensure that the system has meet the temporal constraints is required. Thus, analysis tools can be either:

1. not supported, or
2. supported.

The problem of analysis of the composed component-based database system is rather straightforward. In most cases, analysis of the composed system is unavailable (see table 6.2). Importance of having good analysis of the composed system is recognized in KIDS, but is not pursued beyond that, i.e., analysis tools are not provided. Some component-based embedded and real-time systems also do not support analysis of the composed system. This is true for SPIN, 2K, and systems based on the PBO model. VEST introduces notion of reliability and real-time analysis of the system, but does not give more detailed description of such analysis. ACCORD provides automated analysis of WCET of different configurations of aspects and components.

I. Tailoring ability The benefit of using component-based development in database systems is customization of the database for different applications. There are four degrees of tailorability in component-based database and embedded real-time systems:

1. none,
2. low,
3. moderate, and

4. high.

It can be observed that extensible systems have low tailorability, middleware moderate, while configurable systems have high tailorability as those are built out of components only (see table 6.2). Since the goal is to provide an optimized real-time system for a specific application with low development costs and short time-to-market, it is safe to say that configurable systems are the most suitable in this respect.

J. Aspects In the investigated systems aspects are either:

1. not supported, or
2. supported.

Typically, software systems are build either using components or using aspects (see table 6.2). Besides COMET, there are only two more approaches that provide support for aspects. In the real-time and embedded domain, VEST provides support for aspects that do not crosscut the functional code of components, rather, they describe the component behavior. These aspect can be viewed as one out of three types of aspects, namely run-time aspects, that are supported by COMET. In the database area, SADES is developed with the explicit support for aspects.

Chapter 7

Conclusions

This final chapter presents a summary of our work and the research contributions in section 7.1. The issues for our future work are identified in section 7.2.

7.1 Summary

The integration of component-based and aspect-oriented software development into real-time systems development would enable (i) efficient system configuration from the components in the component library based on the system requirements, (ii) easy tailoring of components and/or a system for a specific application by changing the behavior (code) of the component by applying aspects, and (iii) enhanced flexibility of the real-time and embedded software through the notion of system configurability and component tailorability. However, due to specific demands of real-time systems, applying aspect-oriented and component-based notions to real-time system development is not straightforward for several reasons. First, the real-time system design should support decomposition of the system into a set of components and a set of aspects. Second, the component model should provide mechanisms for handling temporal constraints if used for building real-time systems. Furthermore, to support tailorability and separation of concerns in the form of aspects,

the real-time component model should also provide explicit support for aspect weaving, while preserving the information hiding. Since the traditional view of real-time systems implies tasks as building elements of the system, the relationship between tasks and components needs to be clearly identified. Additionally, temporal analysis, both static worst-case and dynamic schedulability analysis, of different configurations of aspects and components in a real-time system should be provided. Resolving these issues would enable successful integration of the ideas and notions from component-based and aspect-oriented software development into the real-time system development.

In this thesis we have presented the following.

- A novel concept of aspectual component-based real-time system development (ACCORD). Through the notion of aspects and components, ACCORD enforces the divide-and-conquer approach to complex real-time system development. ACCORD supports a decomposition process with the following two sequential phases: (i) decomposition of the real-time system into a set of components and a set of aspects, corresponding to the structural view of the components and the real-time system, and (ii) structuring of tasks, corresponding to the temporal view of the components and the real-time system.

- A real-time component model denoted RTCOM that describes what a real-time component, supporting different types of aspects and enforcing information hiding, should look like.

- A method and a tool for worst-case execution time analysis of different configurations of aspects and components.

- A set of criteria for designing component-based real-time systems, including: (i) a real-time component model that supports mapping of components to tasks, (ii) separation of concerns in real-time systems through the notion of different types of aspects, and (iii) composition support, namely support for configuration and analysis of the composed real-time software.

We have shown how ACCORD can be applied in practice by describing the way we have applied it in the design and development of COMET, a configurable real-time database. Analyzing the impact of applying ACCORD, 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.

7.2 Future Work

There is a number of research challenges left to be resolved. We consider the following issues crucial to successful application of ACCORD, and, thus, the focus of our future work.

To successfully apply ACCORD to real-time system development we should provide a fully formalized framework for ACCORD and develop a tool environment that would support the ACCORD development process, including: (i) identification of components and aspects based on system requirements, (ii) automated extraction of information that reflects run-time behavior of components and aspects built on RTCOM, (iii) automated extraction of the compositional needs of components, and (iv) automated configuration of a real-time systems out of chosen set of components and aspects. An essential issue is mapping components to tasks to embrace all the elements found to be important in real-time systems, and especially hard real-time systems. We intend to address this issue by extending the guidelines for mapping of components to tasks based on case studies where ACCORD is applied. This also includes providing a framework, possibly using synchronization aspects, for efficiently dealing with inter-process communication of mapped tasks. Currently, there is a limited understanding of effects on the performance and memory consumption when building systems with components and aspects. Further investigation is essential for this class of performance-constrained systems.

RTCOM needs to be expanded to fully embrace resource and temporal constraints in a real-time system in its run-time part (now we

provide support for WCET). We also need to refine the composition part of RTCOM, i.e., a language for describing the composition needs of each component is valuable, as well as develop composition rules that account both for functional and run-time needs of components and aspects. Currently the component model is constrained with respect to the relationship between operations and mechanisms as mechanisms cannot use operations for their implementation. We intend to relax the component model such that an application aspect can be derived both from operations and mechanisms of a component, and mechanisms can be derived from operations in the component and operations required from other components. Relaxing RTCOM could result in several challenges, as follows. First, the number of possible conflicting application aspect weavings in one or several components might increase, and, thus, increasing the need for an efficient way of resolving these. Second, the weaving of application aspects in the components could result in the need for weaving of required interfaces as weaved application aspects could inject new required operations in the component. Finally, the run-time aspects need to be refined to reflect the new relationship between the component constituents, namely operations and mechanisms. Additionally, temporal analysis of aspects and components becomes more complex as mechanisms can be modified by aspect weaving. This implies modifications of the preprocessing part of the automated aspect-level WCET analysis to include the variability of mechanisms WCETs, as well as modification of the WCET analyzer part, i.e., the algorithm, of the aspect-level WCET analyzer.

The ideas and notions introduced by RTCOM could be applicable to a wider spectrum of application domains, and not necessarily limited to real-time systems. Thus, on a larger scale, formalizing the model would help generalizing it to different application domains. On a smaller scale, we need to identify tradeoffs in the model with respect to mechanisms in the component that enable tailorability by aspect weaving.

Appendix A

Abbreviations

ACCORD	AspeCtual COmponent-based Real-time system Development
ACID	Atomicity, Consistency, Isolation, Durability
ADL	Architectural Description Language
ADARTS	ADA-based Design Approach for Real-Time Systems
AOSD	Aspect-Oriented Software Development
AOD	Aspect-Oriented Databases
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 Service

HP-2PL	Hight Priority - Two Phase Locking
HRT-HOOD	Hard Real-Time Hierarchical Object Oriented Design
Hw Db	Hardware Database
WoE	Warnings or Errors
WoE Db	Warnings or Errors Database
IC	Indexing Component
IDL	Interface Definition Language
IECU	Instrumental Electronic Control Unit
ISC	Invasive Software Composition
ECU	Electronic Control Unit
EDF	Earliest Deadline First
KIDS	Kernel-based Implementation of Database management Systems
LC	Locking Component
MHC	Memory Handler Component
PBO	Port-Based Object
RLC	Recovery and Logging Component
RM	Rate Monotonic
RTCOM	Real-Time Component Model
RT-UML	Real-Time Unified Modeling Language
OCC	Optimistic Concurrency Control
OS	Operating System
ORB	Object Request Broker
SADES	Semi-Autonomous Database Evolution System
SC	System Composability
SOP	Subject-Oriented Programming
TMC	Transaction Manager Component
TRSD	Transactional Real-Time System Design
TSC	Transaction Scheduler Component
UIC	User Interface Component
UML	Unified Modeling Language
VECU	Vehicle Electronic Control Unit
VEST	Virginia Embedded Systems Toolkit
QoS	Quality of Service
WCET	Worst-Case Execution Time

Bibliography

[1] Aspect-oriented databases (AOD). Project web-site at `http://www.comp.lancs.ac.uk/computing/aod/`, May 2003.

[2] Aspects in real-time embedded systems (AIRES). Project web-site at `http://www.dist-systems.bbn.com/projects/AIRES/`, February 2003.

[3] Constraint-based embedded program composition. Project web-site at `http://www.isis.vanderbilt.edu/Projects/PCES/default.html`, February 2003.

[4] Framework for aspect composition for an event channel (FACET). Project web-site at `http://www.cs.wustl.edu/~doc/RandD/PCES/`, February 2003.

[5] 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.

[6] 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.

[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 ECOOP '94*, volume 821 of *Lecture notes in computer science*, pages 386–407. Springer-Verlag, 1994.

[8] S. F. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring. DeeDS: Towards a distributed and active real-time database system. *ACM Sigmod Record*, Volume 25, 1996.

[9] Articus Systems. *Rubus OS - Reference Manual*, 1996.

[10] U. Aßmann. *Invasive Software Composition*. Springer-Verlag, December 2002.

[11] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau. Technical concepts of component-based software engineering. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, 2000.

[12] L. Bass, P. Clements, and R. Kazman. *Software Architecture In Practice*. SEI Series in Software Engineering. Addison Wesley, 1998.

[13] 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*, Palma, Spain, May 2000.

[14] 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.

[15] L. Blair and G. Blair. A tool suite to support aspect-oriented specification. In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP '99*, pages 7–10, Lisbon, Portugal, June 1999.

[16] J. A. Blakeley. OLE DB: a component DBMS architecture. In *Proceedings of the 12th International Conference on Data Engineering (ICDE)*, pages 203–204, New Orleans, Louisiana, USA, March 1996. IEEE Computer Society Press.

[17] J. A. Blakeley. Universal data access with OLE DB. In *Proceedings of the 42nd IEEE International Computer Conference (COMPCON)*, pages 2–7, San Jose California, February 1997. IEEE Computer Society Press.

[18] J. A. Blakeley and M. J. Pizzo. *Component Database Systems*, chapter Enabling Component Databases with OLE DB. Morgan Kaufmann Publishers, 2000.

[19] J. Bosch. *Design and Use of Software Architectures.* ACM Press in collaboration with Addison-Wesley, 2000.

[20] 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.

[21] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages.* International Computer Science Series. Addison-Wesley, 1997.

[22] R. Camposano and J. Wilberg. Embedded system design. *Design Automation for Embedded Systems*, 1(1):5–50, 1996.

[23] 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.

[24] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano - a revolution in on-board communications. Technical report, Volvo Technology Report, 1998.

[25] 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.

[26] 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.

[27] I. Crnkovic and M. Larsson. A case study: Demands on component-based development. In *Proceedings of 22th International Conference of Software Engineering*, pages 23–31, Limerick, Ireland, June 2000. ACM.

[28] I. Crnkovic and M. Larsson, editors. *Building Reliable Component-Based Real-Time Systems*. Artech House Publishers, July 2002.

[29] I. Crnkovic, M. Larsson, and F. Lüders. State of the practice: Component-based software engineering course. In *Proceedings of 3rd International Workshop of Component-Based Software Engineering*. IEEE Computer Society, January 2000.

[30] A. Datta and S. H. Son. Is a bird in the hand worth more than two birds in the bush? Limitations of priority cognizance in conflict resolution for firm real-time database systems. *IEEE Transactions on Computers*, 49(5):482–502, May 2000.

[31] 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`.

[32] K. R. Dittrich and A. Geppert. *Component Database Systems*, chapter Component Database Systems: Introduction, Foundations, and Overview. Morgan Kaufmann Publishers, 2000.

[33] A. Dogac, C. Dengi, and M. T. Öszu. Distributed object computing platform. *Communications of the ACM*, 41(9):95–103, 1998.

[34] C. Donnely and R. Stallman. *Bison: The YACC-Compatible Parser Generator*, 2002. Available at: `http://www.gnu.org/manual/bison-1.25/bison.html`.

[35] B. P. Douglass. *Real-Time UML: Developing Efficient Objects for Embedded Systems.* Addison-Wesley, 2000.

[36] 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)*, pages 75–84, Saint-Malo, France, May 1999. IEEE Computer Society Press.

[37] 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.

[38] 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.

[39] H. Gomaa. A software design method for real-time systems. *Communications of the ACM*, 27(9):938–949, September 1984.

[40] H. Gomaa. A software design method for Ada based real time systems. In *Proceedings of the 6th Washington Ada symposium on Ada*, pages 273–284, McLean, Virginia, United States, 1989. ACM Press.

[41] 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.

[42] M. Hassani and D. B. Stewart. A mechanism for communication in dynamically reconfigurable embedded systems. In *Proceedings of High Assurance Software Engineering (HASE) Workshop*, pages 215–220, Washington DC, August 1997.

[43] Developing DataBlade modules for Informix-Universal Server. Informix DataBlade Technology. Informix Corporation, 22 March 2001. Available at `http://www.informix.com/datablades/`.

[44] D. Isovic, M. Lindgren, and I. Crnkovic. System development with real-time components. In *Proceedings of ECOOP Workshop - Pervasive Component-Based Systems*, France, June 2000.

[45] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.

[46] Y-K. Kim, M. R. Lehr, D. W. George, and S. H. Son. A database server for distributed real-time systems: Issues and experiences. In *Proceedings of the Second IEEE Workshop on Parallel and Distributed Real Time Systems*, 1994.

[47] 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'9)*, pages 201–208, Pittsburgh, August 2000.

[48] 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 (Middleware'2000)*, volume 1795 of *Lecture Notes in Computer Science*, pages 121–143, New York, USA, April 2000. Springer-Verlag.

[49] 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 ECOOP Workshop on Reflective Object-Oriented Programming and Systems*, volume 1543 of

Lecture Notes in Computer Science, Brussels, Belgium, July 1998. Springer-Verlag.

[50] 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.

[51] C. M. Krishna and K. G. Shin. *Real-time Systems*. McGraw-Hill Series in Computer Science. The McGraw-Hill Companies, Inc., 1997.

[52] M. Larsson and I. Crnkovic. New challenges for configuration management. In *Proceedings of System Configuration Management, 9th International Symposium (SCM-9)*, volume 1675 of *Lecture Notes in Computer Science*, pages 232–243, Toulouse, France, August 1999. Springer-Verlag.

[53] J. Lee and S. H. Son. Using dynamic adjustment of serialization order for real-time database systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, December 1993.

[54] J. Lindstrom, T. Niklander, P. Porkka, and K. Raatikainen. A distributed real-time main-memory database for telecommunication. In *Proceedings of the Workshop on Databases in Telecommunications*, pages 158–173, 1999.

[55] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in hard real-time traffic environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[56] 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.

[57] D. Locke. *Real-Time Database Systems: Issues and Applications*, chapter Real-Time Databases: Real-World Requirements. Kluwer Academic Publishers, 1997.

[58] H. Lu, Y. Yeung Ng, and Z. Tian. T-tree or B-tree: Main memory database index structure revisited. *11th Australasian Database Conference*, 2000.

[59] 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.

[60] C. H. Lung, S. Bot, K. Kalaichelvan, and R. Kazman. An approach to software architecture analysis for evolution and reusability. In *Proceedings of CASCON*, Toronto, ON, November 1997. IBM Center for Advanced Studies.

[61] MBrane Ltd. `http://www.mbrane.com`, February 2001.

[62] N. Medvedovic and R. N. Taylor. Separating fact from fiction in software architecture. In *Proceedings of the Third International Workshop on Software Architecture*, pages 10–108. ACM Press, 1999.

[63] B. Meyer and C. Mingins. Component-based development: From buzz to spark. *IEEE Computer*, 32(7):35–37, July 1999. Guest Editors' Introduction.

[64] Microsoft. The component object model specification. Available at: `http://www.microsoft.com/com/resources/comdocs.asp`, February 2001.

[65] 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`.

[66] R. T. Monroe and D. Garlan. Style-based reuse for software architectures. In *Proceedings of the Fourth International Conference on Software Reuse*, pages 84–93. IEEE Computer Society Press, April 1996.

[67] 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 International Database Engineering and Application Symposium (IDEAS)*, pages 351–359, Montreal, Canada, August 1999. IEEE Computer Society Press.

[68] 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 Euromicro International Conference on Real-Time Systems*, Vienna, Austria, June 2002.

[69] M. A. Olson. Selecting and implementing an embedded database system. *IEEE Computers*, 33(9):27–34, Sept. 2000.

[70] S. Olson, R. Pledereder, P. Shaw, and D. Yach. The Sybase architecture for extensible data management. *Data Engineering Bulletin*, 21(3):12–24, 1998.

[71] 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.

[72] All your data: The Oracle extensibility architecture. Oracle Technical White Paper. Oracle Corporation. Redwood Shores, CA, February 1999.

[73] 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.

[74] M. T. Özsu and B. Yao. *Component Database Systems*, chapter Building Component Database Systems Using CORBA. Data Management Systems. Morgan Kaufmann Publishers, 2000.

[75] P. Pardyak and B. N. Bershad. Dynamic binding for an extensible system. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Operating Systems Review, Special Issue, pages 201–212, Seattle WA, USA, October 1996. ACM and USENIX Association.

[76] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[77] V. Paxson. *Flex: A fast scanner generator*, 2002. Available at: `http://www.gnu.org/manual/flex-2.5.4/flex.html`.

[78] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[79] Pervasive Software Inc. `http://www.pervasive.com`, February 2001.

[80] Polyhedra Plc. `http://www.polyhedra.com`, February 2001.

[81] DARPA ITO projects. Program composition for embedded systems. `http://www.darpa.mil/ito/research/pces/index.html`, 7 August 2001.

[82] P. Puschner and A. Burns. A review of worst-case execution-time analysis (editorial). *Real-Time Systems*, 18(2/3):115–128, May 2000.

[83] K. Ramamritham. Real-time databases. *International Journal of distributed and Parallel Databases*, pages 199–226, 1993.

[84] A. Rashid. A hybrid approach to separation of concerns: the story of SADES. In *Proceedings of the third International REFLECTION Conference*, volume 2192 of *Lecture Notes in Computer Science*, pages 231–249, Kyoto, Japan, September 2001. Springer-Verlag.

[85] A. Rashid and E. Pulvermueller. From object-oriented to aspect-oriented databases. In *Proceedings of DEXA 2000*, volume 1873 of *Lecture Notes in Computer Science*, pages 125–134. Springer-Verlag, 2000.

[86] A. Rashid and P. Sawyer. Aspect-orientation and database systems: an effective customization approach. *IEE Software*, 148(5):156–164, October 2001.

[87] R. Rastogi, S. Seshadri, P. Bohannon, D. W. Leinbaugh, A. Silberschatz, and S. Sudarshan. Improving predictability of transaction execution times in real-time databases. *Real-Time Systems*, 19(3):283–302, November 2000. Kluwer Academic Publishers.

[88] 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.

[89] A. Silberschatz, H. F. Korthe, and S. Sudarshan. *Database System Concepts*. The McGraw-Hill Companies, Inc., 1997.

[90] Sleepycat Software Inc. `http://www.sleepycat.com`, February 2001.

[91] 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 Pacific 2002)*, Sydney, Australia, February 2002. Australian Computer Society. AspectC++ can be downloaded from: `http://www.aspectc.org`.

[92] J. Stankovic. Misconceptions about real-time computing: a serious problem for next-generation systems. *IEEE Computer*, 21, Oct 1988.

[93] 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.

[94] J. Stankovic. VEST: a toolset for constructing and analyzing component based operating systems for embedded and real-time systems. In *Proceedings of the Embedded Software, First International Workshop (EMSOFT 2001)*, volume 2211 of *Lecture Notes in Computer Science*, pages 390–402, Tahoe City, CA, USA, October 2001. Springer-Verlag.

[95] 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 Real-Time Applications Symposium 2003*, Toronto, Canada, May. IEEE Computer Society Press.

[96] J. A. Stankovic, S. H. 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.

[97] D. B. Stewart, M. W. Gertz, and P. K. Khosla. Software assembly for real-time applications based on a distributed shared memory model. In *Proceedings of the 1994 Complex Systems Engineering Synthesis and Assessment Technology Workshop*, pages 214–224, Silver Spring, MD, July 1994.

[98] D. S. Stewart. Designing software components for real-time applications. In *Proceedings of Embedded System Conference*, San Jose, CA, September 2000. Class 408, 428.

[99] C. Szyperski. *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley, 1999.

[100] The AspectJ Team. *The AspectJ Programming Guide.* Xerox Corporation, September 2002. Available at: `http://aspectj.org/doc/dist/progguide/index.html`.

[101] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Embedded databases for embedded real-time systems: a component-based approach. Technical report, Dept. of Computer Science, Linköping University, and Dept. of Computer Engineering, Mälardalen University, 2002.

[102] 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.

[103] TimesTen Performance Software. `http://www.timesten.com`, February 2001.

[104] Germany University of Karlsruhe. Compost. Documentation available at: `http://i44w3.info.uni-karlsruhe.de/~compost/`, June 2001.

[105] A. Wall. Software architecture for real-time systems. Technical Report 00/20, Mälarden Real-Time Research Center, Mälardalen University, Västerås, Sweden, May 2000.

[106] 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/`.