Aspects and Components in Real-Time System Development: Towards Reconfigurable and Reusable Software

Aleksandra Tešanović⋆, Dag Nyström†, Jörgen Hansson⋆, and Christer Norström†
⋆Linköping University, Department of Computer Science, Linköping, Sweden
†Mälardalen University, Department of Computer Engineering, Västerås, Sweden

***Abstract*— Increasing complexity of real-time systems, and demands for enabling their configurability and reusability are strong motivations for applying new software engineering principles, such as aspect-oriented and component-based development. In this paper we introduce 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. Initial results show that the successful application of the proposed concept has a positive impact on real-time system development in enabling efficient configuration of real-time systems, improved reusability and flexibility of real-time software, and modularization of crosscutting concerns. We provide arguments for this by presenting an application of the proposed concept on the design and development of a configurable embedded real-time database, called COMET. Furthermore, using the COMET system as an example, we introduce a novel way of handling concurrency in a real-time database system, where concurrency is modeled as an aspect crosscutting the system.**

***Index Terms*— Embedded systems, real-time systems, software components, aspects, database systems, temporal analysis.**

I. Introduction

REAL-TIME and embedded systems are being used widely in 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 [1]. Thus, the introduction of the component-based software development (CBSD) [2] paradigm 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) has emerged as a new principle for software development that provides an efficient way of modularizing crosscutting concerns in software systems [3]–[5]. 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 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 using 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 a 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, 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. 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 a component user. This implies that we need to provide support for aspect integration into component code, while preserving information hiding of a component to the largest degree possible. Hence, to be able to successfully

apply software engineering techniques such as AOSD and CBSD in real-time systems, the following questions need to be answered.

- What is the appropriate design method that will allow integration of the two software engineering techniques into real-time systems?
- What component model and aspects are appropriate for real-time and embedded environments?
- What component model can capture and adopt principles of CBSD and AOSD in real-time and embedded environments?

In this paper we investigate and address these research questions by proposing a novel concept of aspectual component-based real-time system development (ACCORD). The concept is founded on a design method that decomposes real-time systems into components and aspects, and provides a real-time component model (RTCOM) that supports the notion of time and temporal constraints, space and resource management constraints, and composability. RTCOM is the component model addressing real-time software reusability and composability by combining aspects and components. It is our experience so far that applying the proposed concept has a positive impact on the real-time system development in enabling efficient configuration of real-time systems, improved reusability and flexibility of real-time software, and a structured way of handling crosscutting concerns. We show that ACCORD can be successfully applied in practice by describing the way we have applied it in the design and development of a component-based embedded real-time database system (COMET). In the COMET example we present a novel approach to modeling and implementing real-time policies, e.g., concurrency control and scheduling, as aspects that crosscut the structure of a real-time system. Modularization of real-time policies into aspects allows customization of real-time systems without changing the code of the components.

The paper is organized as follows. In section II a background to component-based and aspects-oriented software development is presented. In section III we present an outline of ACCORD and its design method. We present RTCOM in section IV. In section V we show an application of ACCORD to the development of COMET. In the COMET example we describe a new way of modeling real-time concurrency control policy as an aspect in a real-time database system. Related work is presented in section VI. The paper finishes (section VII) with a summary containing the main conclusions and directions for our future research.

II. Background

In this section, background to component-based and aspects-oriented software development is presented (sections II-A, and II-B). Main differences between components in component-based and aspect-oriented software development are then discussed in section II-C.

A. Component-Based Software Development

The need for transition from monolithic to open and flexible systems has emerged due to shortcomings in traditional software development, such as high development costs, inadequate support for long-term maintenance and system evolution, and often unsatisfactory quality of software [6]. 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, such as decreased development costs, increased quality of software, shortened time-to-market, and reduced maintenance costs [6]–[9].

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 [6]. In systems where COM [10] is used as a component framework, a component is generally assumed to be a self-contained binary package with precisely defined standardized interfaces [11]. Similarly, in the CORBA component framework [12], 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 [13].

Hence, the universal definition of a component that would be suitable for every component-based system does not currently exist. 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 a common fact: *components are for composition* [2].

All types of components, independent of their definition, 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. Furthermore, independently of application area, a software component is normally considered to have *black box* properties [9], [13]: each component sees only interfaces of other components, thus, internal state and attributes of the component are strongly encapsulated.

While frameworks and standards for components today primarily focus on CORBA, COM, or JavaBeans, the increasing need for component-based development has also been identified in the area of real-time and embedded systems [1], [14], [15].

B. Aspect-Oriented Software Development

AOSD has emerged as a new principle for software development, and is based on the notion of separation of concerns [3]. 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, e.g., AspectC [16], AspectC++ [17], and AspectJ [18] developed for Java;[1] and

[1]These aspect languages share many similarities with AspectJ.

- an aspect weaver, which is a special compiler that combines components and aspects in a process called aspect weaving.

Components used for system composition in AOSD 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 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 [3]. 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 [19].

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 it is described by a pointcut expression. A *join point* refers to a point in the component code where aspects should be woven, e.g., a method, a type (struct or union). Figure 1 shows the definition of a named pointcut `getLockCall`, which refers to all calls to the function, i.e., join point, `getLock()` within the program with which the aspect is to be woven, and exposes a single integer argument to that call.[2]

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

Fig. 1. An example of the pointcut definition

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

Fig. 2. An example of the advice 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* code is executed before the join point, (ii) *after advice* code is executed immediately after the join point, and (iii) *around advice* code is executed in place of the join point. Figure 2 shows an example of an after advice. With this advice each call to `getLock()` is followed by the execution of the advice code, i.e., printing of the lock id.

C. Components vs. Aspects

Next, we discuss 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, AOSD uses the white box component metaphor to emphasize that all details of the component implementation should be revealed. Both black box and white box component abstractions have their advantages and disadvantages. For example, hiding all details of a component implementation in a black box manner has the advantage that a component user does not have to deal with the component internals. In contrast, having all details revealed in a white box manner allows a 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 changing 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.

III. ACCORD Design Method

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 section 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 with two sequential phases: (i) decomposition of the real-time system into a set of components, and (ii) decomposition of the real-time system into a set of aspects.
- Components, as software artifacts that implement a number of well-defined functions, and where they have well-defined interfaces.
- Aspects, as properties of a system affecting its performance or semantics, and crosscutting the functionality of the system [3].
- A real-time component model (RTCOM) that describes a real-time component, which supports aspects but also enforces information hiding. 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.

The design of a real-time system using ACCORD method is performed in two phases. In the first phase, a real-time system

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

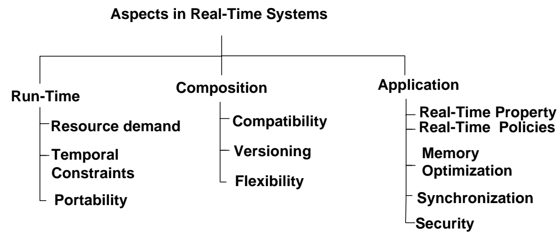

Fig. 3. Classification of aspects in real-time systems

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. Aspects crosscut components and the overall system. This phase typically deals with non-functional requirements[3] and crosscutting concerns of a real-time system, e.g., resource management and temporal attributes. After the design, components and aspects are implemented based on RTCOM.

Analogous to the classical object-oriented design method that initially identifies objects as building blocks of a system, ACCORD initially identifies components and aspects as building blocks of a real-time software system. Hence, ACCORD can be viewed as an extension to the classical object-oriented design method, which in turn implies that ACCORD is founded on a well-established design method.

A. Aspects in Real-Time Systems

We classify aspects in a real-time system as follows (see figure 3): (i) application aspects, (ii) run-time aspects, and (iii) composition 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 of the system. Security is another application aspect that influences behavior and structure of a system, e.g., the system must be able to distinguish users with different security clearance. Synchronization, entangled in the entire system, is encapsulated and represented by a synchronization aspect. Memory optimization, synchronization, and security are commonly mentioned aspects in AOSD [3]. Additionally, real-time properties and policies are viewed as application aspects as they influence the overall structural behavior of the system. Depending on the requirements of a system, real-time properties and policies could be further refined, which we show in the example of the COMET system (see section V-C). Application aspects enable tailoring of the components for a specific application, as they change 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.

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. Run-time aspects give information needed by the run-time system to ensure that integrating a real-time system would not compromise timeliness, nor 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., worst-case execution time (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 hardware 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.

Composition aspects describe with which components a component can be combined (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 as 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.

After aspects are identified, we recommend that a table is made with all the components and all identified application aspects, in which the crosscutting effects to different components are recorded (an example of one such table is given in section V-C). As we show in the next section, this step is especially useful for the next phase of the design, where

[3]Non-functional requirements are sometimes referred to as extra-functional requirements [20].

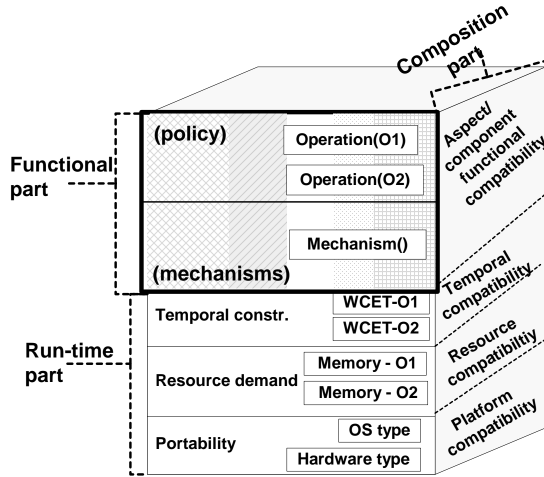

Fig. 4. A real-time component model (RTCOM)

each component is modeled and designed to take into account identified application aspects.

IV. Real-Time Component Model (RTCOM)

In this section we present RTCOM, which 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): (i) the functional part, (ii) the run-time system dependent part, and (iii) the composition part.

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 of a restricted set of parameters in the component [20]. 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.

A. 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.

B. Functional Part of RTCOM

To define the functional part of 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.

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 $o = f(K, L)$.

The implication of definition 2 is the establishment of mechanisms as fine-granule methods or functions of each component. 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 mechanisms could be related in a component is given in figure 5. 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.

The relationship between the operations in a component and within a system configuration should be precisely defined to enable temporal analysis[4] of real-time software composed of RTCOM components. Given that RTCOM is primarily targeted towards hard real-time systems, efficient WCET analysis is an essential requirement. To provide means for satisfying this requirement, we introduce the notion of non-recursively cyclic set of operations as follows.

Definition 4 (Non-recursively cyclic set): Given the operation sequence $< o_1, \ldots, o_n >$ let f_i be the mapping that

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

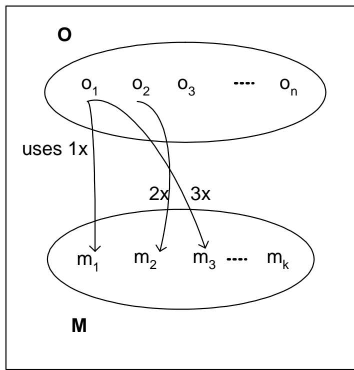

Fig. 5. Operations and mechanisms in a component c

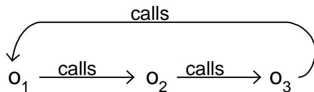

Fig. 6. An example of the recursively cyclic set of operations $\{o_1, o_2, o_3\}$

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 non-recursively 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 operation o_3, and operation o_3 makes a recursive cycle by being implemented using the operation o_1 (see figure 6). 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 non-recursively 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 non-recursively cyclic property, is considered to be well-formed for the purpose of 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.

The functional part of RTCOM, its constituents, their relationship and properties introduced so far, we illustrate through a small example of an ordinary linked list implemented based on RTCOM. The functional part of the linked list component, i.e., the code, consists of component mechanisms and operations (as prescribed by definition 6). The mechanisms needed are the ones for the manipulation of nodes in the

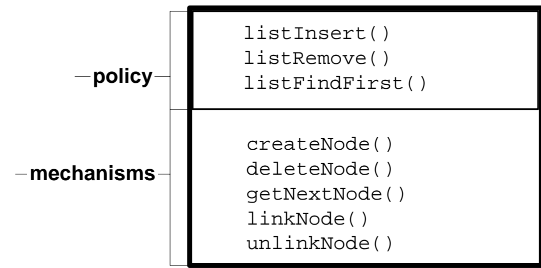

Fig. 7. The functional part of the linked list component

list, i.e., `createNode`, `deleteNode`, `getNextNode`, `linkNode`, and `unlinkNode` (see figure 7). Operations in the linked list component, namely `listInsert`, `listRemove`, and `listFindFirst`, are implemented using the underlying component 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. Thus, the implementation of the operations in a component defines the behavior of the component. Here, the component provides a FIFO ordering of nodes in the list, and, hence, exhibits the FIFO component policy.

The policy of a component can be changed or modified by weaving of application aspects into the component code, i.e., functional part of the component. Therefore, application aspects are directly dependent on the functional part of RTCOM. The definition of application aspects is influenced by two requirements: (i) preserving information hiding of the component to the largest extent possible, and (ii) enabling temporal analysis of the resulting woven components. To satisfy the two requirements we utilize the notion of mechanisms as building blocks of application aspects, and provide application aspects as follows.

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 of the woven 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. In RTCOM, pointcuts refer to operations, implying that a pointcut in an application aspect points to one or several operations of a component where modifications of the component code are allowed. Having mechanisms of the components as basic building blocks of the advices is a

```
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: }
```

Fig. 8. The `listPriority` application aspect

decisive factor in enabling temporal analysis of the resulting woven code (see section IV-C for details on performing WCET analysis). 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.

Assume that we want to change the policy of the linked list component given in figure 7 from FIFO to priority-based ordering of the nodes. This can be achieved by weaving an appropriate application aspect. Figure 8 shows the `listPriority` application aspect, which consists of one named pointcut `listInsertCall`, identifying `listInsert` operation of the component as a join point in the component code (lines 2-3). The `listInsertCall` before advice is implemented using the component mechanism `getNextNode` to determine the position of the node based on its priority (lines 5-14). Weaving of the `listPriority` application aspect into the code of the linked list component would result in a component where each execution of the operation `listInsert` is preceded by the execution of the advice `listInsertCall`, i.e., before placing the node into the list its position is determined based on its priority.

Expressing semantics of aspect weaving formally (definition 9), requires an introduction of the mathematical interpretation of the sequential execution of two code fragments in the following manner. Given that x and y represent two code fragments, and xy denotes their sequential compositions (resulting from a textual concatenation of the two pieces of code), then we can define the mathematical interpretation of the code xy as follows.

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 $glue(o, o')$.

Using the formal notation introduced so far, we can formally express the semantics of application aspect weaving as follows.

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 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.

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, application aspects are represented as vertical layers in the functional part of the component as they influence component behavior, i.e., change component policy.

C. 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 woven 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). 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 WCET as an example of a run-time aspect. One way of enabling predictable aspect weaving is 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 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 woven 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.

Figure 9 shows the WCET specification for mechanisms in the component, where for each mechanism the WCET is declared and assumed to be known. Similarly, figure 10 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 11 shows the WCET specification of an application aspect. For each advice type (before, around, after) that modifies an operation, the operation it modifies is declared together with the mechanisms used for 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.

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

Fig. 9. Specification of the WCET of component mechanisms

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

Fig. 10. 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)]
    }
  }

 ......
}
```

Fig. 11. Specification of the WCET of an application aspect

Figure 12 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 12). The WCET specification for the application aspect `listPriority` that changes the policy framework is shown in figure 13. Since the maximum number of elements in the linked list can vary, the WCET specifications are parameterized with the `noOfElements` parameter.

The resulting WCET of the component, woven with application aspects, is computed using a tool we developed called aspect WCET analyzer [21]. The aspect WCET analyzer

```
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 linkNode;
    wcet  4ms;
  }
  mechanism{
    name getNextNode;
    wcet  2ms;
  }

....
}
```

Fig. 12. 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;
  }
....
}
```

Fig. 13. The WCET specification of the `listPriority` application aspect

(aspectualized)listInsertWcet
= listInsertWcet(without aspects) + (before)listInsertCallWcet
= 14 + 2.4*noOfElements

where

listInsert(without aspects)
= intWcet + $\sum$ mechanism*usage
= 1 + createNodeWcet*1 + linkNodeWcet*1
= 1 + 5*1 + 4*1 = 10

(before)listInsertWcet
= intWcet+ $\sum$ mechanism*usage
= 4 + 0.4*noOfElements+ getNextNodeWcet*noOfElements
= 4+2.4*noOfElements

Fig. 14. An example of WCET calculations for an operation modified with an advice

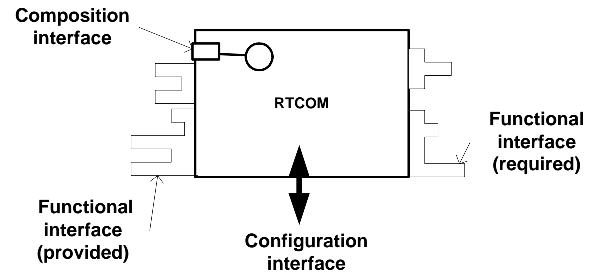

Fig. 15. Interfaces supported by RTCOM

performs automated aspect-level WCET analysis [22], [23], which is an approach for determining the WCET of a real-time system composed using aspects and components. 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 environment without paying the price of aspect weaving for each individual candidate configuration. The aspect WCET analyzer performs the computations using a set of rules that define how to compute a new WCET of an operation woven with aspects, depending on the type of an advice in the aspect. For example, for the advice of the type before modifying an operation, the new WCET of the operation would be computed using the value of an old WCET (i.e., WCET of an operation without aspects), and augmenting that value with the WCET of the before advice. This rule reflects the fact that the code of the before advice would, after aspect weaving, be inserted before the code of the operation. Similar rules exist for the advices of types after and around. Following the example of the linked list component, we can compute the WCET of the operation `listInsert` modified with the advice `listInsertCall` of the type before as illustrated in figure 14.

D. Composition Part of RTCOM

The composition part of RTCOM 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). Given that there are different application aspects that can be woven into the component, composition aspects represented in the composition part of RTCOM should contain information about component compatibility with respect to different application aspects, as well as with respect to different components.

E. RTCOM Interfaces

RTCOM supports three different types of interfaces (see figure 15): (i) functional interface, (ii) configuration interface, and (iii) composition interface.

Functional interfaces of components are classified in two categories, namely provided functional interfaces, and required functional interfaces. Provided interfaces reflect a set of operations that a component provides to other components or to the system. Required interfaces reflect a set of operations that a component requires from other components. Having separation to provided and required interfaces eases component exchange and addition of new components into the system.

The *configuration interface* is intended for the integration of a real-time system with the run-time environment. This interface provides information of temporal behavior of each component, and reflects the run-time aspect of the component. Combining multiple components results in a system that also has a configuration interface, and enables the designer to inspect the behavior of the system towards the run-time environment (see figure 16).

Composition interfaces, which correspond to join points, are embedded into the functional component part. The weaver identifies composition interfaces and uses them for

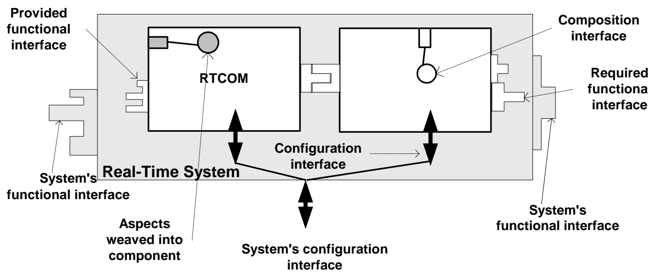

Fig. 16. Interfaces and their role in the composition process

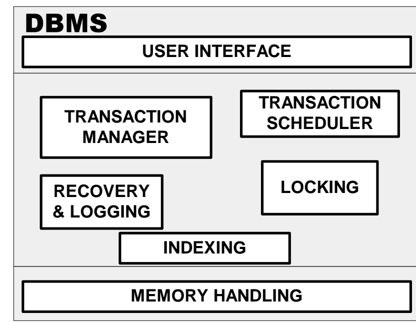

Fig. 17. COMET decomposition into a set of components

aspect weaving. Composition interfaces are ignored at component/system compile-time if they are not needed, and are "activated" only when certain application aspects are woven 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.

Explicit separation of software component interfaces into composition interfaces and functional interfaces was introduced in [19].

V. COMET: A COMPONENT-BASED EMBEDDED REAL-TIME DATABASE

This section shows how to apply the introduced concept of aspectual component-based development on a design and development of a concrete real-time system by presenting the application of the design method to development of a configurable real-time embedded database system, called COMET.

A. Background

The goal of the COMET project 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 on the example of one of the COMET targeting application areas: control systems in the automotive industry. These 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, e.g., nodes can implement transmission, engine, or instrumental functions. 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, and transaction 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.

B. COMET Components

Following the ACCORD design method presented in section III 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 17): 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 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 [24]. 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 [25]. Therefore, the TSC should be a flexible and exchangeable part of the database architecture.

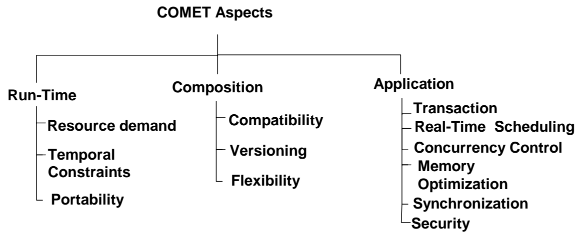

Fig. 18. Classification of aspects in an embedded real-time database system

The *locking component* (LC) deals with locking of data, and it provides mechanisms for lock manipulation and 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 V-D).

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 [26] and multi-versioning suitable for applications with a large number of read-only transactions [27]. Additionally, it is possible to customize an indexing strategy depending on the number of transactions active in the system and the indexing algorithm needed.

The *recovery and logging component* (RLC) is in charge of recovery and logging of data in the database. As COMET stores data in main-memory, there is a need for different recovery and logging techniques, depending on the type of the storage, e.g., non-volatile EEPROM or Flash.

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.

C. 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 18, and it fully corresponds to the ACCORD decomposition (given in section III-A) in three types of aspects: run-time, composition, and application aspects. However, as COMET is the real-time database system, refinement to the application aspects is made to reflect both real-time and database issues. Hence, in the COMET decomposition of application aspects, the real-time policy aspect is refined to include real-time scheduling and concurrency control policy aspects, while the real-time property aspect

TABLE I
CROSSCUTTING EFFECTS OF DIFFERENT APPLICATION ASPECTS ON THE COMET COMPONENTS

Application aspects \ Components	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				X
Memory optimization	X	X	X	X	X		X
Synchronization		X	X	X	X		X
Security	X		X	X		X	X

(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 the table I. Note that application aspects can also crosscut or depend on other application aspects. In this paper, however, we primarily focus on crosscutting effects of application aspects to different components. For more details on dependencies and inter-relationships of aspects we refer interested readers to [28], [29].

As can be seen from table I, 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 application of the real-time system, thus, particular attention should be made to identify the application aspects for each real-time system.

D. COMET RTCOM

Components and aspects in COMET are implemented based on RTCOM (discussed in section IV). 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 the LC) and one application aspect (namely the CC aspect).

The LC performs the following functionality: assigning locks to requesting transactions and maintaining a lock table,

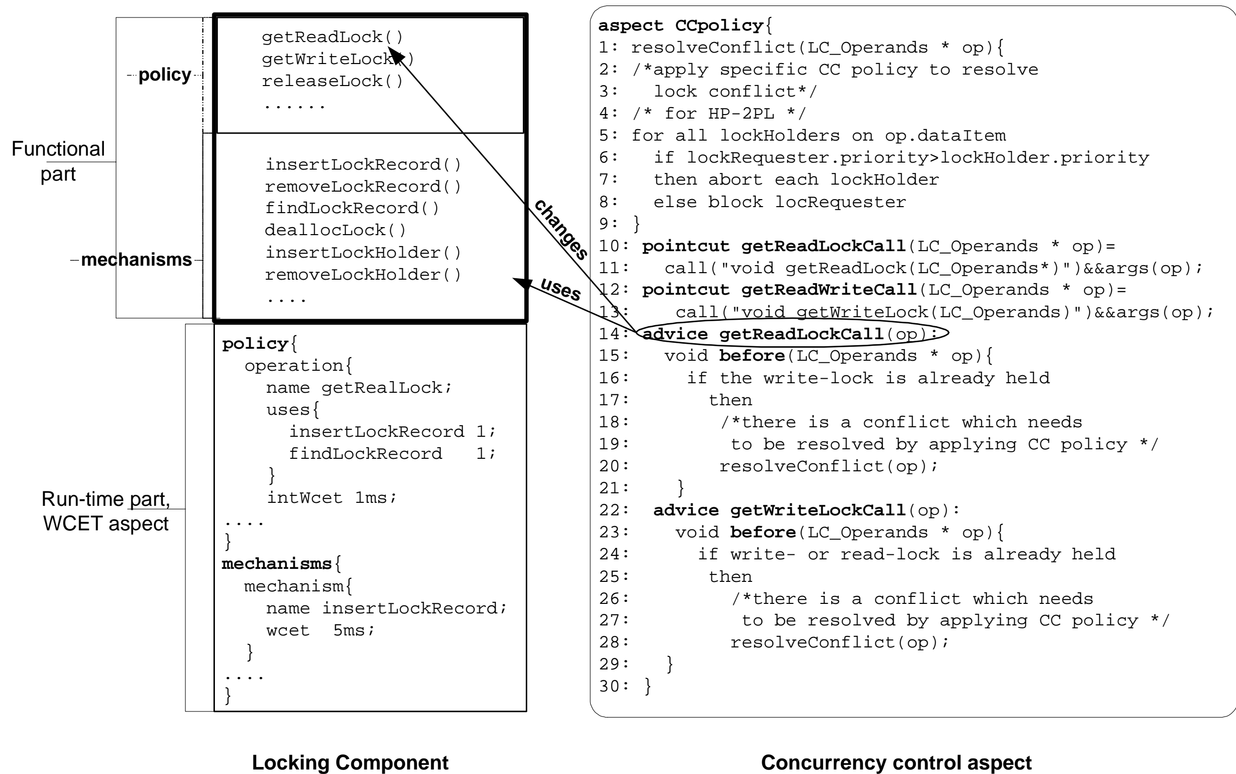

Fig. 19. The locking component and the concurrency control aspect

thus, it records all locks obtained by transactions in the system. As can be seen from the table I, the LC is crosscut with several application aspects. The application aspect that influences the policy, i.e., changes the behavior of the LC, is a CC aspect, which defines the way lock conflicts should be handled in the system. To enable tailorability of the LC, and reuse of code in the largest possible extent, the LC is implemented with the policy framework in which 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 19): `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 [30] and RWPCP [31]. 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 [32] and OCC-APR [33].

The right part of the figure 19 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 chose 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 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. As this function is called from the advices it can be considered a part the body of each advice (equivalent would be to place the code of the function in each advice separately). Furthermore, `resolveConflict()` traverses the list of transactions holding a lock using underlying mechanisms of the LC. Hence, the overall advices are implemented using mechanisms of the LC to traverse the lock table (lines 16-19 and 24-27) and the list of transactions holding a lock (in the function `resolveConflict()`).

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 I). In the example of the CC aspect implementing pessimistic HP-2PL protocol (see figure 19), 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, would require additional pointcuts to be defined in the TMC, as the protocol (as compared to pessimistic protocols) assumes 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. A before advice `initPolicy` would then initialize all necessary data structures in the CC aspect after data structures in the LC have been initialized.

E. 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 V-D) 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 components, and extensive research has being done in identifying and defining good composition rules for the components [6], [19], [34].
- 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. Restricting the number of mechanisms in the component policy framework initially, and adding the mechanisms in the component "on-demand", i.e., when required by the application or an application aspect, could be one way of dealing with the code overhead.[5]

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.

VI. Related Work

In this section we address the research in the area of component-based real-time and database systems, and the real-time and database research projects that are using aspects to separate concerns.

The focus in existing component-based real-time systems is enforcement of real-time behavior. In these systems a component is usually mapped to a task, e.g., passive component [1], binary component [35], and port-based object component [36]. Therefore, analysis of real-time components in these solutions addresses the problem of temporal scopes at a component level as task attributes [1], [35], [36]: WCET, release time, deadline. ACCORD with its RTCOM model supports mapping of a component to a task, and takes a broader view of the composition process by allowing real-time systems to be composed out of tasks and components that are not necessarily mapped to a task. ACCORD, in contrast to other approaches building real-time component-based systems [1], [35], [36], enables support for multidimensional separation of concerns and allows integration of aspects into the component code. VEST [1], [37] indeed uses aspect-oriented paradigm but does not provide a component model that enables weaving of application aspects into the component code, rather it focuses on composition aspects.

In the area of database systems, the aspect-oriented databases (AOD) initiative aims at bringing the notion of separation of concerns to databases. The focus of this initiative is on providing a non-real-time database with limited configurability using only aspects (i.e., no components) [38]. To the best of our knowledge, KIDS [39] is the only research project focusing on construction of a configurable database composed out of components (database subsystems), e.g., object management and transaction management. Commercial component-based databases introduce limited customization of the database servers [40], [41], by allowing components for managing non-standard data types, data cartridges and DataBlade modules, to be plugged into a fully functional database system. A somewhat different approach to componentization is Microsoft's Universal Data Access Architecture [42], where the components are data providers and they wrap data sources enabling the translation of all local data formats from different data stores to a common format. However, from a real-time point of view none of the component-based database approaches discussed enforce real-time behavior and use aspects to separate concerns in the system.

Existing real-time design methods [1], [37], [43]–[46] focus on task structuring and two different views on the system,

[5]Note that the ACCORD framework is not restrictive and allows flexible augmentation of mechanisms within the component.

temporal and structural, with moderate emphasis on the information hiding. The analysis of the real-time system under design, although missing from early design approaches [43], [44], has been highlighted as important for the real-time system development [1], [37], [45], [46]. 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 existing real-time design methods. RT-UML [47] is an example an infrastructure that provides configuration tools in a form of a visual language. Note, however, that RT-UML cannot be considered a design method as it essentially provides only syntax, not semantics, for the real-time system design, e.g., its powerful expressiveness could be used by a design method as means of specifying real-time software components [48].

In contrast to real-time design methods, modern software engineering design methods [2], [19], [49], [50] 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 [16]–[19], [51].

It can be observed that there is a gap between the design 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. ACCORD helps in bridging this gap as it provides support for aspects and aspect weaving into the code of the components, efficient component and system tailoring, and better reusability and flexibility of real-time software - the issues that have not been fully addressed by existing real-time design approaches.

VII. Summary

In recent years, one of the key research challenges in software engineering research community has been enabling configuration of systems and reuse of software by composing systems using components from a component library. Our research focuses on applying aspect-oriented and component-based software development to real-time system development by introducing a novel concept of aspectual component-based real-time system development (ACCORD). In this paper we presented ACCORD and its elements, which we have applied in the development of a real-time database system, called COMET. ACCORD introduces the following into real-time system development: (i) a design method, which enables improved reuse and configurability of real-time and database systems by combining basic ideas from component-based and aspect-oriented communities with real-time concerns, thus bridging the gap between real-time systems, embedded systems, database systems, and software engineering, (ii) a real-time component model, called RTCOM, which enables efficient development of configurable real-time systems, and (iii) a new approach to modeling of real-time policies as aspects improving the flexibility of real-time systems. In the COMET example we have shown that applying ACCORD could have an impact on the real-time system development in providing efficient configuration of real-time systems, improved reusability and flexibility of real-time software, and modularization of crosscutting concerns.

There are 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 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. 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.

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 real-time component model with respect to mechanisms in the component that enable tailorability by aspect weaving.

Acknowledgments

The authors would like to thank Dr. Simin Nadjm-Tehrani for comments and discussions on the formalization of the real-time component model. This work is financially supported by the Swedish Foundation for Strategic Research (SSF) via the SAVE project and the ARTES network, and the Center for Industrial Information Technology (CENIIT) under contract 01.07.

References

[1] 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)*, ser. Lecture Notes in Computer Science, vol. 2211. Tahoe City, CA, USA: Springer-Verlag, October 2001, pp. 390–402.

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

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, “Aspect-oriented programming,” in *Proceedings of the ECOOP*, ser. Lecture Notes in Computer Science, vol. 1241. Springer-Verlag, 1997, pp. 220–242.

[4] H. Ossher and G. Kiczales, Eds., *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*. ACM Press, 2002.

[5] W. G. Griswold and M. Aksit, Eds., *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*. ACM Press, 2003.

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

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

[8] 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.

[9] 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)*. Saint-Malo, France: IEEE Computer Society Press, May 1999, pp. 75–84.

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

[11] 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)*. Montreal, Canada: IEEE Computer Society Press, August 1999, pp. 351–359.

[12] OMG, "The common object request broker: Architecture and specification," OMG Formal Documentation (formal/01-02-10), February 2001, Available at: ftp://ftp.omg.org/pub/docs/formal/01-02-01.pdf.

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

[14] L. Freidrich, J. Stankovic, M. Humphrey, M. Marley, and J. Haskins, "A survey of configurable, component-based operating systems for embedded applications," *IEEE Micro*, vol. 21, no. 3, pp. 54–68, May/June 2001.

[15] B. Meyer and C. Mingins, "Component-based development: From buzz to spark," *IEEE Computer*, vol. 32, no. 7, pp. 35–37, July 1999, guest Editors' Introduction.

[16] 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.

[17] 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: Australian Computer Society, February 2002, AspectC++ can be downloaded from: http://www.aspectc.org.

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

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

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

[21] A. Tešanović, D. Nyström, J. Hansson, and C. Norström, "Aspect-level WCET analyzer: a tool for automated WCET analysis of a real-time software composed using aspects and components," in *Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis (WCET 2003)*, Porto, Portugal, July 2003.

[22] A. Tešanović, D. Nyström, J. Hansson, and C. Norström, "Aspect-level worst-case execution time analysis of real-time systems compositioned using aspects and components," in *Proceedings of the 27th IFAC/IFIP/IEEE Workshop on Real-Time Programming (WRTP'03)*. Poland: Elsevier, May 2003.

[23] 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.

[24] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in hard real-time traffic environment," *Journal of the Association for Computing Machinery*, vol. 20, no. 1, pp. 46–61, January 1973.

[25] 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.

[26] H. Lu, Y. Y. Ng, and Z. Tian, "T-tree or b-tree: Main memory database index structure revisited," in *Proceedings of the 11th Australian Database Conference*, 2000, pp. 65–73.

[27] 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*, vol. 19, no. 3, pp. 283–302, November 2000, Kluwer Academic Publishers.

[28] J. Kienzle, Y. Yu, and J. Xiong, "On composition and reuse of aspects," in *In Proceedings of the Workshop on Foundations of Aspect-Oriented Languages (FOAL 2003)*, Boston, USA, March 2003.

[29] H. Sipma, "A formal model for cross-cutting modular transition systems," in *In Proceedings of the Workshop on Foundations of Aspect-Oriented Languages (FOAL 2003)*, Boston, USA, March 2003.

[30] R. K. Abbott and H. Garcia-Molina, "Scheduling real-time transactions: a performance evaluation," *ACM Transactions on Database Systems*, vol. 17, no. 3, pp. 513–560, September 1992.

[31] L. Sha, R. Rajkumar, S. H. Son, and C.-H. Chang, "A real-time locking protocol," *IEEE Transactions on Computers*, vol. 40, no. 7, pp. 793–800, September 1991.

[32] 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.

[33] 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*, vol. 49, no. 5, pp. 482–502, May 2000.

[34] 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," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-2000-TR-008, 2000.

[35] 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.

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

[37] 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: IEEE Computer Society Press, May 2003.

[38] A. Rashid and E. Pulvermueller, "From object-oriented to aspect-oriented databases," in *Proceedings of DEXA 2000*, ser. Lecture Notes in Computer Science, vol. 1873. Springer-Verlag, 2000, pp. 125–134.

[39] A. Geppert, S. Scherrer, and K. R. Dittrich, "KIDS: Construction of database management systems based on reuse," Department of Computer Science, University of Zurich, Tech. Rep. ifi-97.01, September 1997.

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

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

[42] "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.

[43] H. Gomaa, "A software design method for real-time systems," *Communications of the ACM*, vol. 27, no. 9, pp. 938–949, September 1984.

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

[45] 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*, vol. 6, no. 3, pp. 72–82, 1991.

[46] A. Burns and A. Wellings, *HRT-HOOD: a Structured Design Method for Hard Real-Time Ada Systems*, ser. Real-Time Safety Critical Systems. Elsevier, 1995, vol. 3.

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

[48] I. Crnkovic, B. Hnich, T. Jonsson, and Z. Kiziltan, "Specification, implementation, and deployment of components," *Communications of the ACM*, vol. 45, no. 10, pp. 35–40, October 2002.

[49] A. Dogac, C. Dengi, and M. T. Öszu, "Distributed object computing platform," *Communications of the ACM*, vol. 41, no. 9, pp. 95–103, 1998.

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

[51] 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*, ser. Lecture notes in computer science, vol. 821. Springer-Verlag, 1994, pp. 386–407.

Aleksandra Tešanović received the B.Sc. degree in electrical engineering from University of Banja Luka, Bosnia and Herzegovina, in 1999, and the Licentiate degree in computer science from Linköping University, Sweden, in 2003. She is currently a Ph.D. student at the Department of Computer Science, Linköping University, Sweden. Her current research interests include software engineering methods, composition techniques, and tools for component-based real-time and embedded systems.

Dag Nyström received his M.Sc. in computer engineering during 2001 and his Licentiate degree in 2003, both from Mälardalen University, Sweden. He is currently employed as a Ph.D. student at the Department of Computer Science and Engineering, Mälardalen University, Sweden. His current research interest is mainly data management in vehicular control-systems.

Jörgen Hansson received the B.Sc. and M.Sc. degree from University of Skövde, Sweden, in 1992 and 1993 respectively. He received his Ph.D. degree in 1999 from Linköping University, Sweden. He is an Assistant Professor at the Department of Computer Science in Linköping University. He has authored/co-authored 30 papers and edited two books in these areas. His research has focused on techniques for ensuring robustness and timeliness in complex real-time applications that are prone to transient overloads. He has been involved in the design and construction of the DeeDS system, a distributed active real-time database system suitable for large complex real-time systems. His current research interests include techniques and methodologies for repositories functioning in real-time, adaptive overload management, and component-based software architectures for embedded and real-time systems. Dr. Hansson serves as the Director of the National Graduate School in Computer Science (CUGS) in Sweden. Dr. Hansson has served as Program and General Chair for the International Workshop on Active and Real-Time Database Systems (ARTDB-95, ARTDB-97).

Christer Norström is professor in Computer Engineering at Mälardalen University. He is Dean for the faculty of Science and Technology at Mälardalen University. He is one of the founding members of the Department of Computer Science and Engineering. Previously he was working as manager for future technology at ABB Automation Technology Products/ Robotics. He has also worked as a consultant, in particular for the automotive industry. His research interests are design of complex real-time systems, system and software engineering for real-time systems. Christer is very interested in technology transfer from academia to industry and he has manifested that through several successful transfers to the automotive industry. Christer was instrumental to the forming of a dynamic innovation system Robotdalen, which was granted 100 MSEK 2003. Christer has given numerous courses on real-time system for industry both in Sweden and in Europe. He received a Ph.D from Royal Institute of Technology (KTH), Stockholm in 1997, became Docent at KTH in 2001, and Professor at Mälardalen University 2002. In year 2001 he was awarded best teacher at Mälardalen University.