Towards Aspectual Component-Based Development of Real-Time Systems *

Aleksandra Tešanović[1], Dag Nyström[2], Jörgen Hansson[1], and Christer Norström[2]

[1] Linköping University, Department of Computer Science, Linköping, Sweden
{alete,jorha}@ida.liu.se
[2] Mälardalen University, Department of Computer Engineering, Västerås, Sweden
{dag.nystrom,christer.norstrom}@mdh.se

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 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. We anticipate that the successful applications of the proposed concept should have 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 assumption 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.

1 Introduction

Real-time and embedded systems are being used widely in today's modern society. 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) paradigm into real-time and embedded systems development offers significant benefits, namely: (i) 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 exactly the functionality needed by the system; (ii) 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 (iii) evolutionary design as components can be replaced or added to the system,

* This work is supported by ARTES, A network for Real-time and graduate education in Sweden, and CENIIT, Center for Industrial Information Technology, under contract 01.07.

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. AOSD allows encapsulating system's crosscutting concerns 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.

Thus, the integration of the two disciplines, CBSD and AOSD, into real-time systems development would enable: (i) efficient system configuration from the components in the component library based on the system's 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's configurability and components' tailorability.

However, 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 components and aspects are appropriate for the real-time and embedded environment?
- What component model can capture and adopt principles of the CBSD and AOSD in a real-time and embedded environment?

In this paper we 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 the 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 2 we present an outline of ACCORD and its design method. We present RTCOM in section 3. In section 4 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 discussed in section 5. The paper finishes (section 6) with a summary containing the main conclusions and directions for our future research.

2 ACCORD Design Method

The growing need for enabling development of configurable real-time and embedded systems that can be tailored for a specific application [1], 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. Hence, we propose ACCORD. 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. Components use interfaces for communication with the environment, i.e., other components.
- Aspects, as properties of a system affecting its performance or semantics, and crosscutting the system's functionality [2].
- A real-time component model (RTCOM) that describes how a real-time component, supporting aspects, should look like. RTCOM is specifically developed: (i) to enable an efficient decomposition process, (ii) to support the notion of time and temporal constraints, and (iii) to enable efficient analysis of components and the composed system.

The design of a real-time system using ACCORD method 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. 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. In the final phase, components and aspects are implemented based on RTCOM. As non-functional requirements are among the most important issues in real-time system development, we first focus on the aspectual decomposition, and then discuss RTCOM.

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

2.1 Aspects in Real-Time Systems

We classify aspects in a real-time system as follows: (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 real-time system's structure, we view memory optimization as an application aspect of a real-time system. Security is another application aspect that influences system's behavior and structure, e.g., the system must be able to 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 are commonly mentioned aspects in AOSD [2]. 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 the example of the COMET system (see section 4.3). Application aspects enable tailoring of the components for a specific application, as they change code of the components.

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.

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).

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.

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 4.3). As we show in the next section, this step is especially useful for the next phase of the design, where each component is modeled and designed to take into account identified application aspects.

3 Real-Time Component Model (RTCOM)

In this section we present RTCOM, which allows easy and predictable weaving of aspects, i.e., integrating aspects into components, thus reflecting decomposition of the system into components and aspects. 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 1): (i) the functional part, (ii) the run-time system dependent part, and (iii) the composition part.

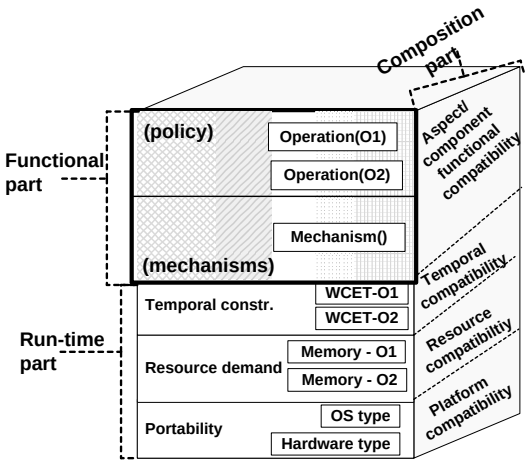

Fig. 1. A real-time component model (RTCOM)

The functional part represents the actual code that implements the component functionality. RTCOM assumes the following for the functional part of the component.

– Each component provides a set of mechanisms, which are basic and fixed parts of the component infrastructure. Mechanisms are fine granule methods or function calls.

– Each component provides a set of operations to other components and/or to the system. Implementation of the operations determines the behavior of the component, i.e., component policy. Operations are represented by coarse granule methods or function calls. Operations are flexible parts of the component as their implementation can change by applying different application aspects. Operations are implemented using the underlying mechanisms, which are fixed parts of the component.

In order 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, the mechanisms, as basic blocks of the component, are implemented. Here, particular attention is given to identified application aspects, and the table reflecting the crosscutting effects of application aspects to different components is used to determine which application aspects are likely to use which component mechanisms. Next, the operations of the component are implemented using component mechanisms. Note, the implemented operations provide an initial component policy, i.e., basic and somewhat generic component functionality. This initial policy we call a *policy framework* of the component. The policy framework could be modified by applying different application aspects, and as such it provides a way of tailoring a component by changing its behavior, i.e., application aspects change the component policy. If all crosscutting application aspects are considered when implementing operations and mechanisms, then the framework is generic and highly flexible. However, if the system evolves such that new application aspects (not considered when developing the policy framework) need to be implemented into component code, then new mechanisms can be defined within the application aspect.

The development process of the functional part of a component results in the component that is colored with application aspects. Therefore, in the graphical view of RTCOM in figure 1, application aspects are represented as vertical layers in the functional part of the component, as they influence component behavior, i.e., change component policy.

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, and are represented as horizontal parallel layers to the functional part of the component as they describe component behavior (see figure 1). In the run-time part of the component, run-time aspects are expressed as attributes of operations, mechanisms, and application aspects, as those are the elements of the component functional part, and thereby influence the temporal behavior of the component.

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 1). Given that there are different application aspects that can be weaved 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.

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 run-time aspects are implemented, followed by the composition part and rules for composing different components and application aspects.

In the following sections we give a close-up of the application aspects and the run-time aspects within the RTCOM, followed by interfaces supported by RTCOM.

3.1 Application Aspects in RTCOM

Existing aspect languages can be used for implementation of application aspects, and their integration into components. The weaving is done by the aspect weaver corresponding to the aspect language [2]. All existing aspect languages, e.g., AspectJ [4], AspectC [5], AspectC++ [6], are conceptually very similar to AspectJ, developed for Java.

Each application aspect declaration consists of advices and pointcuts. A *pointcut* consists of one or more join points, and is described by a pointcut expression. A *join point* in an aspect language refers to a method, a type (struct or union), or any other point from which component code can be accessed. In RTCOM, the pointcut model is restricted to the mechanisms and the operations in the component, and a type (struct). This restriction is necessary for obtaining predictable aspect weaving, i.e., enabling the temporal analysis[4] of the resulting code. 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*, executed before the join point, (ii) *after advice*, executed immediately after the join point, and (iii) *around advice*, executed in place of the join point. In RTCOM, the advice model is also restricted for the reasons of enabling temporal analysis of the code. Hence, the advices are implemented using only the mechanisms of the components, and each advice can change the behavior of the component (policy framework) by changing one or more operations in the component.

3.2 Run-time Aspects in RTCOM

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 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 analysis of aspects, components, and the resulting aspect-oriented software (when aspects are weaved into components) is based on symbolic WCET analysis [7]. Applying symbolic WCET analysis to ACCORD implies the following: (i) WCETs of the mechanisms are obtained by symbolic WCET analysis, (ii) the WCET of every operation is determined based the WCETs of the mechanisms used for implementing the operation, and the internal WCET of the function or the method that implements the operation, i.e., manages the mechanisms, (iii) the WCET of every advice that changes

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

the implementation of the operation is based on the WCETs of the mechanisms used for implementing the advice, and the internal WCET of the advice, i.e., code that manages the mechanisms. Figure 2 shows the WCET specification for mechanisms in the com-

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

Fig. 2. Specification of the WCET aspect of component mechanisms

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

Fig. 3. Specification of the WCET aspect of a component policy framework

ponent, where for each mechanism the WCET is declared and assumed to be known. Similarly, figure 3 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 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. The resulting WCET of the component (or one operation within the component), colored with application aspects, is computed using the algorithm presented in [8]. The algorithm utilizes the knowledge of WCETs of all mechanisms involved, and WCETs of all aspects that change a specific operation. The detailed explanation of the algorithm and the discussion on computing WCETs of components modified by aspects can be found in [8].

```
aspect{
  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. 4. Specification of the WCET aspect of an application aspect

3.3 RTCOM Example

We now give a brief and simple example of one component and one application aspect. The purpose of this simple example is to provide guidance through the process of RTCOM implementation, and provide a clear understanding of RTCOM internals, introduced so far (a more complex and detailed example of RTCOM using COMET is discussed in section 4.4).

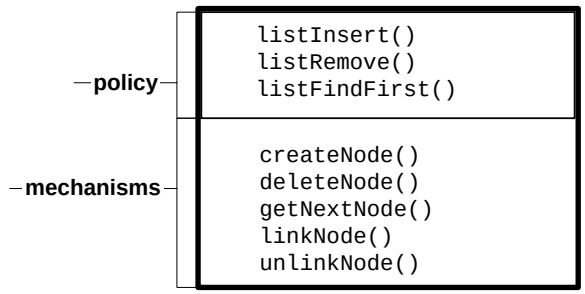

Fig. 5. The functional part of the linked list component

In this example, we consider a component implementing an ordinary linked list. The mechanisms needed are the ones for the manipulation of nodes in the list, i.e., `createNode`, `deleteNode`, `getNextNode`, `linkNode`, `unlinkNode` (see figure 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.

```
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. 6. The `listPriority` application aspect

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 6 shows the `listPriority` application aspect, which consists of one pointcut `listInsertCall`, identifying `listInsert` as a join point in the component code (lines 2-3). When this join point is reached, 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). 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) and for the application aspects, so we can determine the run-time aspects of the component colored with different application aspects.

Figure 7 presents the specification of the WCET aspect for the policy framework of the liked list component. Each operation in the framework is named and its internal WCET (`intWcet`), and the number of times it uses a particular mechanism, are declared (see figure 7). The WCET specification for the application aspect `listPriority` that changes the policy framework is shown in figure 8. Temporal information of the application aspect includes the internal WCET of an advice that modified the operation, and the information of the mechanisms used by the advice, as well as the number of times (upper bound) the advice has used a particular mechanism. Hence, the information provided in the run-time part of the component enables temporal analysis of any combinations of the component policy frameworks and application aspects.

```
operations(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;
  }

....
}
```

Fig. 7. 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. 8. The WCET specification of the `listPriority` application aspect

3.4 RTCOM Interfaces

RTCOM supports three different types of interfaces (see figure 9): (i) functional interface, (ii) configuration interface, and (iii) composition interface.

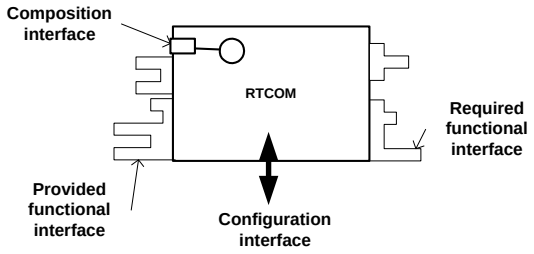

Fig. 9. Interfaces supported by the RTCOM

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.

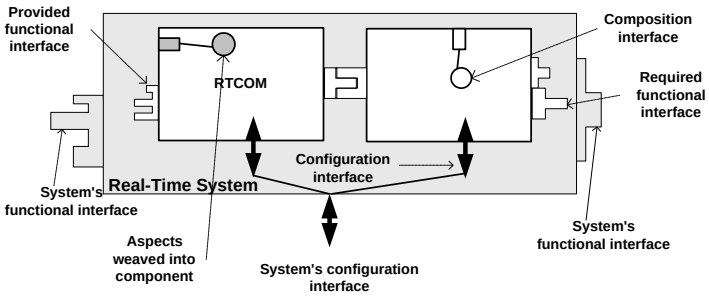

Fig. 10. Interfaces and their role in the composition process

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 the configuration interface, and enables the

designer to inspect the behavior of the system towards the run-time environment (see figure 10).

Composition interfaces, which correspond to join points, are embedded into the functional component part. 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, and are "activated" 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.

Explicit separation of software component interfaces into composition interfaces and functional interfaces is introduced in [9].

4 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.

4.1 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, 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.

4.2 COMET Components

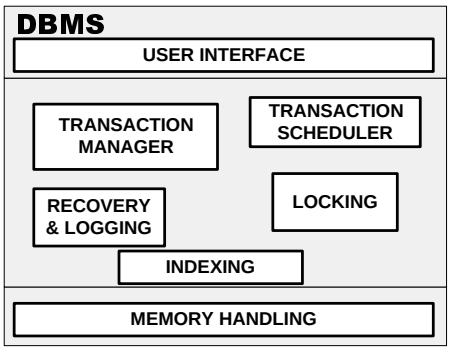

Fig. 11. COMET decomposition into a set of components

Following the ACCORD design method presented in section 2 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 11): 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.

The *transaction scheduler component* (TSC) provides mechanisms for performing scheduling of transactions coming into the system, based on the scheduling policy chosen. COMET supports a variety of scheduling policies, e.g., EDF and RM [10]. 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 [11]. 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 maintains lock records in the database.

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 [12] and multi-versioning suitable for applications with a large number of read-only transactions [13]. Additionally, it is possible to customize 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 is 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 *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.

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.

4.3 COMET Aspects

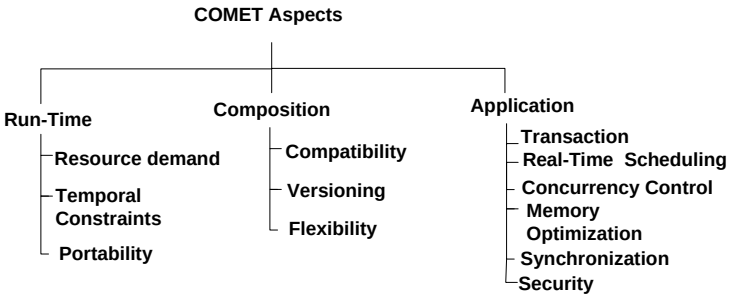

Fig. 12. Classification of aspects in an embedded real-time database system

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 12, and it fully corresponds to the ACCORD decomposition (given in section 2.1) 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 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 1. As can be seen from the table, all identified application aspects crosscut more than one component.

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.

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				X
Memory optimization	X	X	X	X	X		X
Synchronization		X	X	X	X		X
Security	X		X	X		X	X

Table 1. Crosscutting effects of different application aspects on the COMET components

4.4 COMET RTCOM

Components and aspects in COMET are implemented based on RTCOM (discussed in section 3). 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: assigns locks to requesting transactions, and maintains a lock table, thus it records all locks obtained by transactions in the system. As can be seen from the table 1, 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 concurrency control (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 13): `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

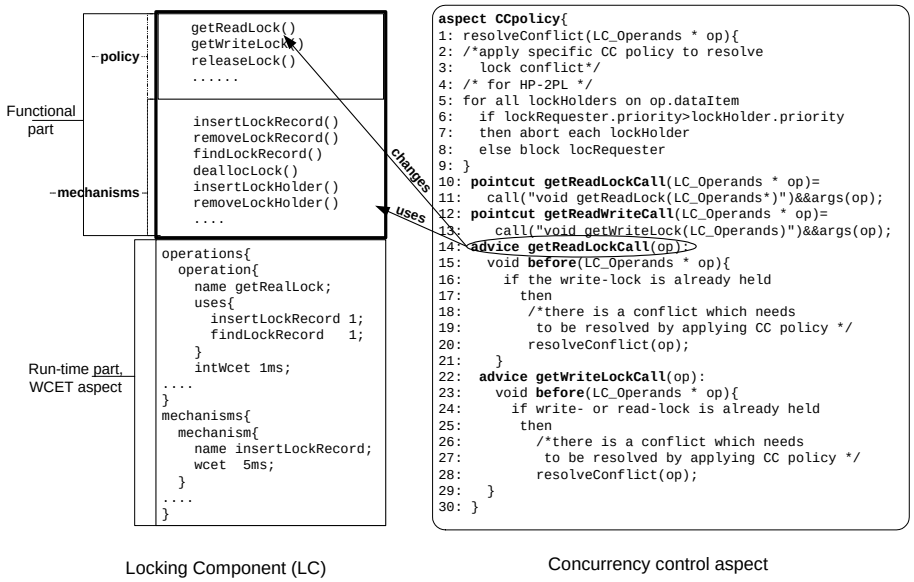

Fig. 13. The locking component and the concurrency control aspect

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 [14] and RWPCP [15]. 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 [16] and OCC-APR [17].

The right part of the figure 13 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. 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 1). In the example of the CC aspect implementing pessimistic HP-2PL protocol (see figure 13), 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.

The benefits of applying ACCORD to the development of COMET platform are the following (in the context of the given example).

- Enabling clean separation of concurrency control as an aspect that crosscuts the LC, which allows high code reusability as the same component mechanisms are used in almost all CC aspects.
- Weaving of a CC aspect into the LC changes the policy of the component by changing the component code, and provides an efficient way of tailoring the component and the system to fit a specific requirement (in this case specific CC policy), leaving the configuration of COMET unchanged.
- Having the LC functionality encapsulated into the 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.
- Having a run-time part of the components and aspects enables analysis of the temporal behavior of the resulting code (see the run-time part of the LC in the left of the figure 13).

The drawback of applying ACCORD to real-time system development is an explosion in possible combinations of components and aspects. This is common 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 [18, 19, 9].

5 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 [20], and port-based object component [21]. Therefore, analysis of real-time components in these solutions addresses the problem of temporal scopes at a component level as task attributes [20, 1, 21]: 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 [20, 1, 21], enables support for multidimensional separation of concerns and allows integration of aspects into the component code. VEST [1] also 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 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) [22]. To the best of our knowledge, KIDS [23] 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 [24, 25], 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 [26], 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.

In contrast to traditional methods for design of real-time systems [27, 28], which focus primarily on the ways of decomposing the system into tasks and handling temporal requirements, ACCORD design method focuses on the ways of decomposing a real-time system into components and aspects to achieve better reusability and flexibility of real-time software.

6 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.

Several research questions remain to be resolved, including:

- developing rules for checking compatibility of aspects and components,
- analyzing component and aspect behavior on different hardware and software platforms in real-time environments to identify trade-offs in applying aspects and components in a real-time setting,
- studying performance of the real-time system with different configurations of components and aspects, and
- providing automated tool support for the proposed development process.

Currently we are focusing on enabling predictable aspect weaving and component composition, and providing tools for automatized temporal analysis of aspects, components, and the resulting system.

References

1. Stankovic, J.: 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 (2000)
2. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Proceedings of the ECOOP. Volume 1241 of Lecture Notes in Computer Science., Springer-Verlag (1997) 220–242
3. Crnkovic, I., Larsson, M., eds.: Building Reliable Component-Based Real-Time Systems. Artech House Publishers (2002) ISBN 1-58053-327-2.
4. Xerox: The AspectJ programming guide (2002)
5. Coady, Y., Kiczales, G., Feeley, M., Smolyn, G.: 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)
6. Spinczyk, O., Gal, A., Schröder-Preikschat, W.: AspectC++: An aspect-oriented extension to C++. In: Proceedings of the TOOLS Pacific 2002, Australian Computer Society (2002)
7. Bernat, G., Burns, A.: An approach to symbolic worst-case execution time analysis. In: Proceedings of the 25th IFAC Workshop on Real-Time Programming, Palma, Spain (2000)
8. Tešanović, A., Nyström, D., Hansson, J., Norström, C.: Integrating symbolic worst-case execution time analysis into aspect-oriented system development. OOPSLA 2002 Workshop on Tools for Aspect-Oriented Software Development (2002)
9. Aßmann, U.: Invasive Software Composition. Springer-Verlag, Universität Karlsruhe (2002)
10. Liu, C.L., Layland, J.W.: Scheduling algorithms for multipprogramming in hard real-time traffic environment. Journal of the Association for Computing Machinery **20** (1973) 46–61
11. Nyström, D., Tešanović, A., Norström, C., Hansson, J.: Data management issues in vehicle control systems: a case study. In: Proceedings of the 14th EUROMICRO International Conference on Real-Time Systems, Vienna, Austria (2002) 249–256
12. Lu, H., Ng, Y., Tian, Z.: T-tree or b-tree: Main memory database index structure revisited. 11th Australasian Database Conference (2000)

13. Rastogi, R., Seshadri, S., Bohannon, P., Leinbaugh, D.W., Silberschatz, A., Sudarshan, S.: Improving predictability of transaction execution times in real-time databases. Real-time Systems **19** (2000) 283–302 Kluwer Academic Publishers.
14. Abbott, R.K., Garcia-Molina, H.: Scheduling real-time transactions: A performance evaluation. ACM Transactions on Database Systems **17** (1992) 513–560
15. Sha, L., Rajkumar, R., Son, S.H., Chang, C.H.: A real-time locking protocol. IEEE Transactions on Computers **40** (1991) 793–800
16. Lee, J., Son, S.H.: Using dynamic adjustment of serialization order for real-time database systems. In: Proceedings of the 14th IEEE Real-Time Systems Symposium. (1993)
17. Datta, A., Son, S.H.: 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** (2000) 482–502
18. Bosch, J.: Design and Use of Software Architectures. Addison-Wesley (2000)
19. Bachmann, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R., Wallnau, K.: Technical concepts of component-based software engineering. Technical Report CMU/SEI-2000-TR-008, Carnegie Mellon University (2000)
20. Isovic, D., Lindgren, M., Crnkovic, I.: System development with real-time components. In: Proceedings of ECOOP Workshop - Pervasive Component-Based Systems, France (2000)
21. Stewart, D.S.: Designing software components for real-time applications. In: Proceedings of Embedded System Conference, San Jose, CA (2000) Class 408, 428.
22. Rashid, A., Pulvermueller, E.: From object-oriented to aspect-oriented databases. In: Proceedings of the DEXA 2000. Volume 1873 of Lecture Notes in Computer Science., Springer-Verlag (2000) 125–134
23. Geppert, A., Scherrer, S., Dittrich, K.R.: KIDS: Construction of database management systems based on reuse. Technical Report ifi-97.01, Department of Computer Science, University of Zurich (1997)
24. Oracle: All your data: The Oracle extensibility architecture. Oracle Technical White Paper (1999)
25. Informix: Developing DataBlade modules for Informix-Universal Server. Informix DataBlade Technology (2001) Available at http://www.informix.com/datablades/.
26. (Papers, O.D.W.)
27. Gomaa, H.: Software development of real-time systems. Communications of the ACM **29** (1986) 657–668
28. Kopetz, H., Zainlinger, R., Fohler, G., Kantz, H., Puschner, P., Schütz, W.: The design of real-time systems: from specification to implementation and verification. Software Engineering Journal **6** (1991) 72–82