Final thesis

Active Behavior in a Configurable Real-Time Database for Embedded Systems

by

Ying Du

LITH-IDA-EX--06/015--SE

2006-03-24
Final thesis

Active Behavior in a Configurable Real-Time Database for Embedded Systems

by Ying Du

LITH-IDA-EX--06/015--SE

Supervisor: Aleksandra Tešanović and
Thomas Gustafsson
Department of Computer Science
Linköping University

Examiner: Jörgen Hansson
Department of Computer Science
Linköping University
Abstract

An embedded system is an application-specific system that is typically dedicated to performing a particular task. Majority of embedded systems are also real-time, implying that timeliness in the system need to be enforced. An embedded system needs to be enforced efficient management of a large amount of data, including maintenance of data freshness in an environment with limited CPU and memory resources. Uniform and efficient data maintenance can be ensured by integrating database management functionality with the system. Furthermore, the resources can be utilized more efficiently if the redundant calculations can be avoided. On-demand updating and active behavior are two solutions that aim at decreasing the number of calculations on data items in embedded systems.

COMET is a COMponent-based Embedded real-Time database, developed to meet the increasing requirements for efficient data management in embedded real-time systems. The COMET platform has been developed using a novel software engineering technique, AspeCtual COMponent-based Real-time software Development (ACCORD), which enables creating database configurations, using software components and aspects from the library, based on the requirements of an application. Although COMET provides uniform and efficient data management for real-time and embedded systems, it does not provide support for on-demand and active behavior.

This thesis is focusing on design, implementation, and evaluation of two new COMET configurations, on-demand updating of data and active behavior. The configurations are created by extending the COMET component and aspect library with a set of aspects that implement on-demand and active behavior. The on-demand updating aspect implements the ODDFT algorithm, which traverses the data dependency graph in the depth-first manner, and triggers and schedules on-demand updates based on data freshness in the value domain. The active behavior aspect enables the database to take actions when an event occurs and a condition coupled with that event and action is fulfilled.

As we show in the performance evaluation, integrating on-demand and active behavior in COMET improves the performance of the database system, gives a better utilization of the CPU, and makes the management of
data more efficient.

**Keywords**: embedded systems, real-time databases, on-demand updating, active behavior, concurrency control, aspect-oriented software development
Acknowledgements

First of all, I would like to thank my supervisors, Aleksandra Tešanović and Thomas Gustafsson. They gave me great help and encouragement during the last six months. Then I would like to thank my examiner, Dr. Jörgen Hansson. Finally, thanks to all my friends in Linköping and my family in Beijing.
Contents

1 Introduction 1
   1.1 Motivation ........................................... 1
   1.2 Thesis Outline ...................................... 2

2 Background 5
   2.1 Real-Time and Embedded Systems .................. 5
   2.2 Database Systems .................................. 7
   2.3 Software Engineering ............................... 8
      2.3.1 Component-Based Software Development ...... 9
      2.3.2 Aspect-Oriented Software Development ...... 9
      2.3.3 ACCORD ........................................ 10
   2.4 COMET .............................................. 13

3 Problem Statement 15
   3.1 Data Management Issues in Embedded Systems .... 15
   3.2 Data Freshness and On-demand Updating .......... 18
   3.3 Extension of COMET ................................ 20
   3.4 Aims and Objectives ................................. 21

4 Advanced Preliminaries 23
   4.1 COMET Components ................................. 23
   4.2 Concurrency Control Aspect ........................ 27
   4.3 Transaction Flow .................................. 29
   4.4 COMET Data and Transaction Model ............... 31
4.5 On-Demand Depth-First Traversal Algorithm . . . . . . . . 32
4.6 Active Behavior in Real-time Databases . . . . . . . . . . . 33

5 Design and Implementation 37
5.1 On-demand Updating Configuration . . . . . . . . . . . . . 37
  5.1.1 ODDFT Data and Transaction Model . . . . . . . . . . 38
  5.1.2 ODDFT Algorithm Aspect . . . . . . . . . . . . . . . 41
  5.1.3 ODDFT Execution Flow . . . . . . . . . . . . . . . . 49
5.2 Active Behavior Configuration . . . . . . . . . . . . . . . . 55
  5.2.1 Rules of Active Behavior in COMET . . . . . . . . . . 55
  5.2.2 Aspect Structure . . . . . . . . . . . . . . . . . . . . 60
  5.2.3 Active Behavior Data and Transaction Model . . . . 63
  5.2.4 Implementation of Active Behavior Aspect . . . . . . 64

6 Performance Evaluation 67
6.1 Simulator Setup . . . . . . . . . . . . . . . . . . . . . . . . 67
6.2 Experiments . . . . . . . . . . . . . . . . . . . . . . . . . . 71

7 Conclusion 75
7.1 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
7.2 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . 76

Bibliography 79
Chapter 1

Introduction

This chapter gives the motivation for the work done in this thesis in section 1.1, and then introduces the structure of the thesis in section 1.2.

1.1 Motivation

An embedded system is a special computer system with limited resources, in terms of CPU and memory. The embedded system maintains a large amount of data and ensures that the data are fresh when they are used for calculations and diagnosis of the system. Most embedded systems have real-time constraints. This implies that the tasks, i.e., concurrent programs in the system, should be completed within a predefined time, called a deadline. An example of an embedded system is an Electronic Engine Control Unit (EECU), which is widely used to control an engine in vehicle systems. Data managed by an EECU are in order of thousands, while the EECU memory is limited to 64Kb RAM and 512Kb Flash, and 32-bit CPU runs at 16.67MHz.

In recent years, the functionality of embedded systems has increased in complexity, while the cost of embedded system development is required to be relatively low. Therefore how to efficiently manage data and guarantee data freshness in embedded systems with limited resources becomes
a challenge. One solution is to use embedded real-time database systems to provide efficient and uniform data management in embedded systems, because database systems are designed to store and manage large numbers of data. An example of such a database suitable for embedded and real-time systems is a COMponent-based Embedded real-Time database (COMET) [1], which is a configurable database platform under ongoing development. COMET has been developed using a novel software engineering technique, AspeCtual COMponent-based Real-time software Development (ACCORD) [1], which combines component-based and aspect-oriented software development in the real-time domain. The COMET platform has a library of components and aspects from which appropriate subset of components and aspects can be chosen to create various configurations to meet application requirements.

Today, data freshness in embedded systems is usually guaranteed by updating all data items with fixed frequency. However, this way, resources are unnecessarily wasted, as the data item may still be valid when it is updated. A method called on-demand updating [2] is used for ensuring data freshness while providing efficient resource usage as it enables updates on data only when it is necessary, i.e., when data are no longer fresh. Hence, a number of unnecessary calculations can be avoided. Moreover, active behavior can also be used to further decrease the update frequency, by specifying event-condition-action rules. Based on these rules, a subset of data in an embedded system can be updated only when a specific event occurs and accompanying conditions are satisfied.

Currently, the COMET platform has no support for on-demand updating or active behavior, i.e., it is not possible to configure a database from available components and aspects such that active and on-demand behavior is enforced. Hence, the goal of this thesis is to extend the COMET component and aspect library to enable support for on-demand and active behavior, so that the overall system utilization is improved.

1.2 Thesis Outline

Chapter 2, Background introduces basic concepts and terminology that are needed to understand this thesis.
Chapter 3, Problem Statement identifies problems that exist in embedded systems, suggests solutions to some of the problems, and presents the objective of this thesis.

Chapter 4, Advanced Preliminaries describes the current COMET implementation and the background of on-demand updating and active behavior.

Chapter 5, Design and Implementation presents the design and implementation of both on-demand updating and active behavior in detail.

Chapter 6, Performance Evaluation presents the performance evaluation of on-demand updating and active behavior.

Chapter 7, Conclusion summarizes this thesis and proposes future work.
Chapter 2

Background

This chapter introduces basic concepts and notations needed for understanding the remainder of the thesis. Section 2.1 presents the main concepts of real-time and embedded systems. Section 2.2 introduces basic knowledge of database systems. Section 2.3 introduces software engineering methods, namely, component-based software development and aspect-oriented software development, and ACCORD. Finally, section 2.4 presents the COMET database platform.

2.1 Real-Time and Embedded Systems

A Real-Time System (RTS) is a system sensitive to time. A RTS consists of concurrent programs called tasks which have specific time constraints. For example, a deadline is the most common temporal constraint a task needs to satisfy and it denotes a time point until which a task needs to be completed. If a task does not complete before its deadline, the result of the computation may be useless or even harmful to the system. Hence, whether a task is executed correctly depends not only on the correctness of the logical result of the computation, but also on whether the task deadline is met. A typical example of a RTS is a braking system in a car. The car must be able to stop in a predefined time frame, when it receives the
instruction from the driver, otherwise an accident can happen. Depending on the consequence of a missed deadline, RTSs can be divided into three categories as follows [3].

- **Hard real-time systems**, where all tasks must complete before their deadlines. Missing a deadline can be fatal to the system. A flight controller is a hard real-time system, because missing a deadline can lead to a catastrophe.

- **Soft real-time systems**, where missing a deadline will not cause the system to crash, but the overall performance of the system may be affected negatively, e.g., in a video conferencing system, missing a deadline would degrade the video quality.

- **Firm real-time systems**, where missing a deadline could result in a useless computation, although there will not be any negative consequences to the system and environment.

Since the processor power and memory are limited in an embedded system, when multiple tasks run simultaneously, the system needs a scheduler to decide in which order the tasks should execute to meet their deadlines. For example, in a banking system, when many tasks arrive at the same time, the system must decide the order of processing the tasks. Schedulers need knowledge of timing task parameters, e.g., Worst Case Execution Time (WCET), release time and deadline, to determine the order of task execution. WCET is the maximum time a task spends for execution and release time is the earliest time at which a task begins its execution. Schedulers can be static or dynamic, and schedules can be made offline or online [4]. One of the well-known online dynamic algorithms is Earliest Deadline First (EDF), where the task with the closest deadline is assigned the highest priority and executed. In a dynamic system operating in an environment where workload characteristics are unknown, there can happen that too many tasks are ready to execute, and there is no feasible schedule that would ensure meeting all deadlines. In such a situation the system is said to be overloaded. EDF is the optimal scheduling algorithm until the overload occurs. Possible solutions to an overload are to abort certain tasks or to prevent tasks to enter the system when it is overloaded [5].
A large number of real-time systems are also embedded. An embedded system is a special-purpose system developed to perform one particular task, such as managing disc drives and electronic engine control unit. An embedded system is a combination of both hardware and software. In this thesis we focus on the software part of an embedded system. In general, an embedded system is characterized by limited hardware resources, e.g., small memory, and possibly does not have an operating system.

There are embedded systems that do not have timeliness requirements, e.g., portable music players. Similarly, real-time systems are not all embedded, e.g., stock exchange. The systems discussed in this thesis are both embedded and real-time, and are referred to simply as embedded systems.

2.2 Database Systems

A database system is designed to store and manage a large amount of information. The database system consists of a collection of interrelated data items and a set of programs that are used to manage those data. A program, called database management system, defines structures for information storage, provides mechanisms for manipulation of information, and also guarantees the safety and validity of the information [6], preventing the corruption when the system crashes and avoiding incorrect results when the information are accessed concurrently by multiple users. In this thesis the primary focus is on database management systems and the term database system refers to a database management system.

In a database system, a transaction is the atomic logic unit of work, which contains several database operations on data items, e.g., read and write operations.

Relational Databases A relational database is a type of a database system widely used in practice. In a relational database, every object in the reality is treated as an entity, and the association among entities is called a relationship. For example, in the case of a database containing records of students, classes and courses, each “student” in a class and each “course” is an entity, while “a student registers a course/courses” is a relationship between “student” and “course”. A relational database consists of a set of
tables, which are collections of relationships. A row inside a table represents a relationship among the values on this row.

The user queries the database for data using a query language. SQL is a well-known query language and is used for data accesses. More information about SQL and relational database can be found in [7] and [6], respectively.

**Real-Time Databases for Embedded Systems**  A real-time database system is different from traditional databases, because the transactions in a real-time database have timeliness requirement, i.e., they should be completed before their deadlines. When a real-time database resides in an embedded system which has no secondary memory, all data items are stored in main memory.

**Active Characteristics**  Active databases are able to react to external or internal events. The rules of active behavior are formulated as an *event-condition-action* (*ECA*) model [6]:

\[
\text{ON event IF condition THEN action}
\]

An *event* is a change to the database, e.g., a read or write operation, or a special time point like “11:59:00”. Once an *event* occurs, the database system checks the rules for this *event*. If a *condition* in the rule is satisfied, the corresponding *action* is executed. In the following example:

\[
\text{ON read}(x) \text{ IF } x \leq 0 \text{ THEN update}(y)
\]

the *event* is whenever operation *read*(x) occurs, and the *condition* is \( x \leq 0 \). If the condition is fulfilled, an *action* denoted as *update*(y) is executed.

In an active real-time database system, where the timeliness needs to be taken into account, the *action* is constrained by temporal parameters, hence extending the ECA model. The way ECA rules are extended to be suited for real-time databases is discussed in detail in section 4.6.

2.3  **Software Engineering**

This section first introduces two emerging software methodologies, component-based software development and aspect-oriented software development. Then
a combination of the two methodologies targeted for real-time systems, AC-CORD, is discussed.

2.3.1 Component-Based Software Development

Component-Based Software Development (CBSD) enables developing a complex system by using a set of pre-existing components, which are developed independently for multiple usages. A system can be upgraded with new functionality by plugging in a component containing this new functionality.

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 and real-time [8]. In the CORBA component framework [9], a component is assumed to be a CORBA object with standardized interfaces. Although there is no common definition for every component-based system, all component-based systems have one thing in common: components are for composition [10].

The component implements a set of functions, and has well-defined interfaces [11]. Since only the interfaces could be seen from outside, the component is considered to have the black box property [12, 11].

Components communicate with each other and with the external environment through the interfaces, which are generally divided into three types: provided, required and configuration interfaces [8]. The first two are used by a component to communicate with other components, while the third one is used by users when configuring a component.

2.3.2 Aspect-Oriented Software Development

Aspect-Oriented Software Development (AOSD) is another new software engineering technique. In contrast to CBSD, components in a system developed using AOSD have the white box property [13]. Namely, implementation details and functionality of the component are completely open and can be modified by component users. Besides components, a system developed using AOSD has another important constituent, an aspect. Aspects are commonly considered to be a property of a system that affects its performance or semantics, and that crosscuts the functionality of the system.
In a system developed using AOSD, components are written in general programming languages, such as C/C++ and Java, but aspects are written in a specific aspect language. Several aspect languages have been developed, e.g., Aspect C++ [15] for components written in C/C++, and AspectJ [16] for components written in Java. A special compiler called aspect weaver is used to combine components and aspects. Namely, the task of an aspect weaver is to transform the original aspect and component code into a weaved source code of the system.

In the aspect language, join points, pointcuts and advices are three essential concepts for designing and implementing aspects. Generally, an aspect includes two parts: pointcut expressions and advices [17]. Join points are well-defined points in the component code, referring to the position where an aspect should be weaved. Join points are bound with pointcuts in pointcut expressions. A pointcut can consist of more than one join point, and is always expressed by a pointcut function, such as execution(), call(), or that(). An advice is a piece of code executed when the join points are matched on the pointcut expression. There are three types of advices: (i) before advice, runs before the join point declared in the pointcut expression is reached; (ii) after advice, runs after the join point in the pointcut expression is reached; (iii) around advice, runs instead of the original component code that is declared in the pointcut expression.

Figure 2.1 shows a piece of AspectC++ code, illustrating how an aspect can be defined. Line 2 is a pointcut expression, binding a join point void havingDinner() with a pointcut exec(). Lines 3-11 define three advices that are executed before, around, and after the pointcut, respectively.

2.3.3 ACCORD

AspeCtual COmponent-based Real-time system Development (ACCORD), is a combination of CBSD and AOSD, applied to real-time system development in order to improve the reusability and flexibility of real-time software. This design methodology includes four parts [1]:

- A decomposition process, where a real-time system is decom-
Background

```
aspect dinner {
  pointcut exec() = execution('void havingDinner()');
  advice exec(): before() {
    Printf('Before dinner: Cooking');
  }
  advice exec(): around() {
    Printf('In dinner: Eating');
  }
  advice exec(): after() {
    Printf('After dinner: Having a dessert');
  }
}
```

Figure 2.1: A simple example of aspect

posed into a set of components and a set of aspects.

- **Components**, which implement a set of functions and have well-defined interfaces that are used for communication with other components and the external environment.

- **Aspects**, which describe a crosscutting system property, and influence the performance and behavior of the system when weaved into the components.

- **A real-time component model (RTCOM)**, which is a model describing how to design and implement components in a real-time system to support aspects. 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 process using ACCORD is done in three phases [1]:

- **Phase 1**: A real-time system is decomposed into a set of components. Each component should implement a well-defined functionality. Com-
ponents should be loosely coupled, but with strong cohesion. In the context of software engineering, loosely coupling and strong cohesion are desirable design attributes for components [8].

- **Phase 2:** A real-time system is decomposed into a set of aspects. In this phase, the aspects crosscutting the system functionality should be identified.

- **Phase 3:** The components and aspects are implemented based on the RTCOM model.

In RTCOM, a component provides a set of mechanisms and a set of operations. In this context, mechanisms are fine granule methods or function calls, and operations are coarse granule methods or function calls. Mechanisms are fixed parts of the component, providing basic functionality. Operations are flexible parts of the component, representing the behavior of the component. Operations are implemented using mechanisms and can be changed by weaving of different aspects.

In a real-time system, aspects can be classified into three types as shown in figure 2.2.

- **Application aspects** can change the internal behavior of components as they crosscut the code of the components. An application in this context refers to the application toward which a real-time and embedded system should be configured. The aspects implemented in this thesis fall into this type.

- **Run-time aspects** give information needed when integrating the system into the run-time environment, in order to guarantee that the timeliness requirements of the system are fulfilled.

- **Composition aspects** provide composition information for the component, including information with which components it can combined, the version of the component, and possibilities of extending the component with additional aspects.
2.4 COMET

COMET is a COMponent-based Embedded real-Time database system, which supports development of different configurations of database systems for different-purpose embedded systems. The current version of COMET is COMET v3.0. As COMET is designed by using the ACCORD method, it is decomposed into six components [1].

- The user interface component (UIC) enables users to interact with the database.
- The scheduling manager component (SMC) schedules transactions coming to the system and maintains a list of all active transactions in the system.
- The locking manager component (LMC) deals with locking of data, providing mechanisms for lock manipulation and maintaining lock records.
- The indexing manager component (IMC) is in charge of indexing of data and maintains the index structure.
- The transaction manager component (TMC) executes the incoming execution plans, thereby performing the actual manipulation of data.
The TMC contains a subcomponent, the buffer manager component (BMC), which manages the buffers used when running transactions.

- The memory management component (MMC) manages the access to data in the physical storage.

In COMET v3.0, three aspects packages have been implemented [16]. An aspects package is a set of aspects with/without a set of components, providing a specific functionality for the database system.

- Concurrency control aspects package consists of one component (the LMC) and two application aspects: concurrency control policy aspects and concurrency control transaction model aspects;

- Index aspects package consists of one component (an alternative IMC, IMC_B_tree) and the GUARD policy aspect;

- QoS aspects package consists of two components, the QoS actuator component and the feedback controller component, and three aspects — QoS management policy aspect, QoS transaction and data model aspect and QoS composition aspect.
Chapter 3

Problem Statement

Since the target application domain of this thesis are embedded systems, this chapter starts in section 3.1 with a discussion on the problems in the existing embedded systems with respect to data maintenance. Section 3.2 explains the concept of data freshness and presents examples of on-demand updating algorithms both in the time and value domain. Section 3.3 presents the needed extension of on-demand updating and active behavior for COMET. In section 3.4 the aim and objective of this thesis are identified.

3.1 Data Management Issues in Embedded Systems

In order to understand the characteristics of embedded systems, we first introduce a typical real-time embedded system, the Electronic Engine Control Unit (EECU). Then we discuss data and transaction model in this class of systems.

The EECU is used in vehicle systems, to control the engine such that the air-fuel mixture is optimal for the catalyst, the engine is not knocking, and the fuel consumption is as low as possible [2]. Since the EECU executes tasks in a best effort way, it can be classified as a soft real-time system.
Typically, the memory of an EECU is limited to 64Kb RAM and 512Kb Flash, and it has the 32-bit CPU that runs at 16.67MHz.

Generally, embedded systems need to manage a large amount of data items. For example, the number of data items in EECU software is in order of thousands. Hence, a database functionality is needed to maintain and manage these data items. The data items in an embedded system that monitors the physical environment can be classified into two types [2]:

- **Base items** \((B)\), which are read from sensors or communication links directly. The base items reflect the state of the environment, e.g., temperature and engine speed.

- **Derived items** \((D)\), which are derived from base items and can only be changed when base items are changed. They include actuator values and intermediate computation values.

The relationship of base items and derived items can be illustrated by a data dependency graph \(G = (V, E)\), which is a directed acyclic graph (see figure 3.1). \(V\) denotes a set of nodes and \(E\) denotes a set of directed edges from one node to another. A node represents a data item. A directed edge from node \(x\) to \(y\) represents that \(x\) is used for deriving the value of \(y\). All the data items that need to be read in order to derive an item \(d\) are denoted the read set of \(d\), \(R(d)\). The read set can include both base items and derived items.

Figure 3.1 gives an example of the data dependency graph. A node is represented by a circle, and inside the circle is the name of a data item. \(b_1\) to \(b_3\) represent base items, and \(d_1\) to \(d_5\) represent derived items. For instance, the read set of \(d_1\), \(R(d_1)\), includes \(b_1\) and \(b_2\), and the read set of \(d_4\), \(R(d_4)\), includes \(d_1\) and \(d_2\).

For a base item, the transaction has only one write operation, which writes the value of the item to the database. For a derived item, the transaction has several read operations, which derives the values of items in its read set, and one write operation. Therefore, it is assumed that each transaction has zero or more read operations and one write operations. A transaction can be represented by one node in \(G\) that is the item the transaction writes to. Transactions can be divided into three sets [2]:

\(^{1}\) The data is taken from an EECU in a SAAB 9-5.
- **Sensor transactions (ST)**, which update base data items, keeping them consistent with the external environment. This type of transactions are write-only transactions.

- **User transactions (UT)**, which are generated by applications. They have several read operations and one write operation.

- **Triggered transactions/Updates (TU)**, which are generated by a database system to update data items when some special event happens, such as a data item is to be read by a UT.

Current solutions for maintaining data in embedded systems are using traditional data structures and are implemented in an ad hoc way, causing the following problems [2]:

- **Problem 1**: Data items are partitioned into a number of different data areas, i.e., global and application-specific. This makes it difficult to keep track of which data items exist in the system. Also, a data item can accidentally exist in several data areas. This increases CPU and memory usage for deriving several copies.

- **Problem 2**: A number of calculations are unnecessary because the data item is updated when its value is still valid. These unnecessary calculations increase the resource-consumption, and thereby the costs due to more expensive hardware.
- **Problem 3**: Most embedded systems are time-triggered, and all data items in the systems are updated with a fixed period. In this way, some data items are updated more frequently than needed.

In order to reduce the CPU utilization and the memory consumption, the intermediate results of computations should be stored only once and calculated only when necessary. Moreover, the system is likely to be more efficient if the update frequency of data items can be decreased, which can be done by adding active behavior to the system and defining ECA rules. Hence, a part of data items can be updated on specified events if specified conditions are fulfilled. In addition, the freshness and timeliness of the data items are crucial to real-time embedded systems. It is, for these reasons, argued that an embedded real-time database with on-demand updating and active behavior is a good choice for data management.

### 3.2 Data Freshness and On-demand Updating

In order to keep data fresh and make the system work more efficiently, i.e., avoid unnecessary updates, on-demand updating algorithms can be used. On-demand updating means that an update of a data item is only triggered when this data item is derived in a user transaction and a special criterion is fulfilled. The criteria in these algorithms may be different according to various definitions of data freshness. However, all criteria are defined to check whether the data item is stale. If the value of the data item is stale, according to the criterion, then a related update is triggered and executed before the triggering transaction. Otherwise, if the value of the data item is not stale, no update is needed. In this way, the overall performance of the system can be improved [18, 19, 20, 21, 22, 23, 2].

Data freshness can be defined either in time domain or in value domain, and consequently, the updating algorithms can also be divided into two types.
Data Freshness in Time Domain

In the time domain, the freshness of data is defined as follows [24]:

**Definition 1**  Let $x$ be a data item (base or derived), $\text{timestamp}(x)$ be the time when $x$ was created, and $\text{avi}(x)$ be the absolute validity interval, i.e., the allowed age of $x$. Data item $x$ is absolutely consistent, i.e., fresh, when:

$$\text{current_time} - \text{timestamp}(x) \leq \text{avi}(x) \quad (3.1)$$

The timestamp in definition 1 is a physical timestamp, which is a real time point. When a data item is requested by a user transaction, equation (3.1) is checked. If the result shows that the data item is stale, an update on this data item is triggered. This simple algorithm for updating stale data items is denoted as On-Demand (OD) algorithm.

Using the freshness definition in the time domain is not efficient enough, because when choosing an $\text{avi}(x)$ one should consider the worst case change of the data item’s value. This in turn may result in too frequent and unnecessary updating of $x$. Namely, even if $x$ was created a long time ago, i.e., $\text{avi}(x)$ has expired, its value may be still valid within an acceptable bound. In this type of situation, updating of $x$ is unnecessary. The problem can be solved by using the freshness definition in value domain which can decrease the updating frequency to that of the current changes in the external environment.

Data Freshness in Value Domain

In the value domain, the concept of similarity [25] is used to check the data freshness. Similarity is specified by application designers. For example, the similarity can be a value interval, if the difference between two values falls into this interval, the two values are considered to be similar. Data validity bounds are defined as follows [23].

**Definition 2**  Each pair $(d, x)$, where $d$ is a derived data item and $x$ is an item from the read set of $R(d)$, has a data validity bound, denoted $\delta_{d,x}$, that states how much the value of $x$ can change before the value of $d$ is affected.
If $d$ is not affected by any $x$ in $R(d)$, a transaction accessing $x$ does not need to recalculate the value of $d$, because the value is considered to be similar or fresh, even if it has been in the database for a very long time. The freshness of a data item, respective to one and to all items in the read set, is defined by definition 3 and definition 4 respectively [2].

**Definition 3** Let $d$ be a derived data item, $x$ be a data item from $R(d)$, and $v^t_x$, $v^{t_0}_x$ be values of $x$ at times $t_0$ and $t$, respectively. $d$ is fresh with respect to $x$ when

$$|v^{t_0}_x - v^t_x| \leq \delta_{d,x}$$

**Definition 4** Let $d$ be a data item derived at time $t_0$ using values of data items in $R(d)$. $d$ is considered to be fresh at time $t$, if it is fresh with respect to all data items $x \in R(d)$, i.e.,

$$\bigwedge_{x \in R(d)} \{|v^{t_0}_x - v^t_x| \leq \delta_{d,x}\}$$

 evaluates to be true.

For example, in figure 3.1, updating $d_4$ requires $d_1$ and $d_2$, which are in the read set of $d_4$, $R(d_4)$. Checking whether $d_4$ is fresh requires to check whether it is fresh with respect to both $d_1$ and $d_2$. Based on the freshness definitions in the value domain, various on-demand updating algorithms can be used, such as On-Demand Depth-First Traversal (ODDFT), On-Demand Breadth-First Traversal (ODBFT) and On-Demand Top-Bottom traversal with relevance check (ODTB).

### 3.3 Extension of COMET

As mentioned in section 2.4, the current COMET implementation, COMET v3.0, has a set of basic functionalities, and also possibility of making concurrency control and QoS configurations when using relevant aspect packages. However, the COMET implementation does not have mechanisms for dealing with problems of data freshness encountered in software for embedded systems. Furthermore, COMET does not support active behavior. Hence there is a need to extend COMET to support on-demand updating and active behavior, and create two new COMET configurations.
3.4 Aims and Objectives

The objective of this thesis is to make two configurations of the COMET database by extending the COMET library. The goal is to enable maintenance of data freshness by applying the on-demand updating algorithm, and support active behavior for embedded systems. The path to achieve this goal includes the following activities.

1. Design of the on-demand algorithm ODDFT, making it suitable for COMET.

2. Design of active behavior, i.e., ECA rules, for COMET.

3. Using the ACCORD method, implement the algorithms in the COMET database system as aspects, woven into relevant components.

4. Design and construct a set of applications in order to test the implementation of the on-demand and active behavior and evaluate the performance of the two COMET configurations.
Chapter 4

Advanced Preliminaries

This chapter, in section 4.1, describes current COMET components in more depth. Section 4.2 briefly introduces one concurrency control aspect, high priority 2-phase locking with similarity, since it is used as concurrency control method for the algorithms discussed in the thesis. Section 4.3 presents the transaction execution flow in the COMET with concurrency control. Section 4.4 presents the data and transaction model used in COMET. The last two sections, section 4.5 and section 4.6, explain the on-demand updating algorithm and active behavior in detail.

4.1 COMET Components

COMET contains six components that implement the basic functionality of an embedded real-time database system. In figure 4.1, the relationships among these components are illustrated. Each component is presented by a rectangle. A shadowed rectangle means the component is affected by the concurrency control aspect. An arrow from component A to component B implies that component A requires operations of component B, i.e., component A has function calls to B [1].
User Interface Component (UIC)

The UIC is the interface between users and the database system, which hides the data manipulation details. The relationship between applications and COMET is shown in figure 4.2. The UIC provides a set of operations for applications to access and manage data items in the database. From applications’ point of view, all tasks in COMET are performed by transactions. The pseudo code in figure 4.3 shows the routine for creating and submitting a transaction from the application. An application creates a transaction and calls `RUIC_Op_beginTransaction()` to initialize it. Initialization means creating a `DBTrans` for the transaction and specifying the information of the transaction in `DBTrans`. Then, a query is constructed and bound with the transaction using `RUIC_Op_Query()`. The real execution of the transaction is started by calling `RUIC_Op_startTransaction()`. After the transaction is completed, `RUIC_Op_cleanup()` is called to free the resources. In order to provide this functionality, the UIC requires operations from both SMC and TMC.
Figure 4.2: Relationship between applications and COMET

```c
DBTrans * transaction;
RUIC_Op_beginTransaction(&transaction);
sprintf(query,"SELECT * FROM STUDENTS");
RUIC_Op_Query(transaction, query);
RUIC_Op_startTransaction(transaction);
RUIC_Op_cleanup();
```

Figure 4.3: Routine for executing a transaction
4.1. COMET Components

Scheduling Manager Component (SMC)

The SMC is the component in charge of scheduling incoming transactions. The SMC maintains two queues: a ready queue holding transactions that are ready and waiting for execution, and an active queue holding transactions that are currently executing. The SMC maintains a set of threads called a thread pool. The SMC assigns a thread to each transaction in the active queue, so that the transaction can be executed by the system. The number of threads is configurable and determined by the designer.

When the SMC gets the signal from the UIC that a transaction is ready to start, it places this new transaction into the ready queue and tries to execute it. Three possible scenarios emerge at this point [13].

1. **The thread pool contains at least one thread.** This implies that there is at least one thread available to be assigned to the incoming transaction. Hence, the transaction is moved from the ready queue to the active queue and assigned a thread to start executing.

2. **The thread pool is empty and transactions currently executing have at least the same priority as the incoming transaction.** In this case, the incoming transaction waits in the ready queue. All transactions in the ready queue are sorted with respect to their priorities, which are determined by an EDF algorithm. The next transaction that is chosen for execution is always the one with the highest priority in the ready queue.

3. **The thread pool is empty and at least one of the currently executing transactions has lower priority than the incoming transaction.** In this case, the executing transaction with the lower priority is rolled back and its thread is released and assigned to the incoming transaction, which has a higher priority. The rolled-back transaction is moved from the active queue back to the ready queue.

Transaction Manager Component (TMC)

The TMC implements all manipulations on data on behalf of transactions. It parses the execution trees (queries) by recursively calling
RTMC_Mech_Result(). This function is a main mechanism in the TMC that is used to execute transactions. For each execution tree of a transaction, the TMC creates a buffer in which it loads relevant data. Operations and mechanisms on the buffer are specified in the Buffer Manager Component (BMC), which is a part of the TMC. The BMC requires operations of the MMC and IMC to locate the requested data in the memory and load them into the buffer. If the data items are changed by a transaction (e.g., by updating), the result is written back from the buffer to the memory.

**Locking Manager Component (LMC)**

The LMC supports the operations on locking data items. It provides an initial locking policy in which all locks are granted. However, this policy can be changed by weaving various concurrency control aspects into the LMC.

**Indexing Manager Component (IMC)**

The IMC indexes all data items in the memory. The IMC is used to find tuples in relations when reading or writing data items based on meta-data it maintains. Currently, COMET has two different versions of the IMC. The default one is based on T-tree index structure [26], while the alternative one uses B-tree index structure [27].

**Memory Manager Component (MMC)**

The MMC is in charge of the manipulation of the memory storage. The memory storage is a physical device where the data items are actually stored. The MMC operations are called by the TMC and the IMC to allocate or deallocate memory when inserting or deleting tuples, and read or write data when selecting or updating tuples.

### 4.2 Concurrency Control Aspect

In order to implement on-demand updating and active behavior configurations, a prerequisite is that the database system has appropriate concur-
Concurrency control. Concurrency control resolves the conflict between transactions and guarantees the consistency when more than one transaction is attempting to access the same data resource. In this thesis, we use High Priority 2 Phases Locking (HP-2PL) with similarity [28], which is designed as an aspect in the COMET concurrency control package.

The 2PL protocol requires that each transaction issues lock and unlock requests in two phases: the growing phase and the shrinking phase. In the growing phase, a transaction obtains all locks as needed, but it cannot release any locks. Once a transaction releases a lock, it comes into the shrinking phase and cannot obtain any more locks. The 2PL protocol uses two types of locks, read (shared) locks and write (exclusive) locks. The compatibility between these locks is shown in table 4.1, where a "√" means that the locks are compatible and a "×" indicates a conflict.

If a transaction requests a lock in a conflicting mode, HP-2PL resolves this conflict based on the priorities of transactions. Two scenarios may occur: (i) the requesting transaction has the highest priority among all the conflicting transactions, in which case all transactions holding the lock are aborted and restarted, and the lock is issued to the requesting transaction; (ii) the requesting transaction’s priority is not the highest, in which case it waits for the lock until its priority becomes the highest.

When the similarity is taken into account, the conflict is resolved based on the similarity of transactions in the value domain. Let \( v(\tau, x) \) be the value that transaction \( \tau \) wants to write to data item \( x \). Let \( v(x) \) be the original value of \( x \) before any transaction acquires a lock on \( x \). \( \tau_i \) denotes any transaction that holds a lock on item \( x \), and \( \tau_r \) denotes an incoming transaction that requests a lock on item \( x \). If \( \tau_r \) is an update transaction, the conflicting transactions are considered similar and the lock is granted to \( \tau_r \) if the following conditions are fulfilled:

<table>
<thead>
<tr>
<th></th>
<th>read</th>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>√</td>
<td>×</td>
</tr>
<tr>
<td>write</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

Table 4.1: Conflict table for HP-2PL
1. $v(\tau_r, x)$ is similar to $v(x)$;

2. $v(\tau_r, x)$ is similar to $v(\tau_i, x)$, $\forall i: 1 \leq i \leq n$  

If $\tau_r$ is a read-only transaction, the lock is granted when the following condition holds:

1. $v(x)$ is similar to $v(\tau_i, x)$, $\forall i: 1 \leq i \leq n$  

If involved transactions are not similar, the conflict is handled by the conventional HP-2PL. The semantics of “similar” [25] is the same concept as used in data freshness definition in the value domain (see section 3.2).

4.3 Transaction Flow

In the COMET configured with concurrency control, all transactions follow a sequence of steps during execution. The execution flow is represented in figure 4.4 and explained below.

1. A transaction is initialized through the UIC as follows. The application creates a new transaction, constructs a SQL query as a string, and binds it to the transaction.

2. The query is parsed by the UIC and a corresponding execution tree is generated.

3. The transaction waits to be scheduled by the SMC. The SMC schedules the transaction using concurrency control, assigns a thread to the transaction, and starts executing it.

4. From this step, the transaction actually starts to execute operations. The execution is handled mainly by the TMC, cooperating with the MMC and the IMC. The TMC loads the needed relations into buffers as follows. First, the IMC is used to locate the meta-data, which contain information about the relation, e.g., the names and types of its attributes. Second, the TMC reads the meta-data into the buffer by calling the MMC. Finally, the TMC finds the actual data items, i.e., the tuples, using the IMC, and reads them from the memory into the buffer one by one via the MMC.
5. The tuples not needed by the query are deleted from the buffer.

6. The TMC performs the operations required by the query, e.g., reads or writes the value of an attribute. All these operations are done on the tuples in the buffer in this step. If the transaction only has reading operations, it leaves the TMC at this point, following the route to step 8 as shown in figure 4.4.

7. If the tuples are changed in the previous step, they are written back to the memory. The TMC uses the IMC to find the correct address of these tuples, and then uses the MMC to write tuples into the memory.

8. The result of the transaction execution is returned to the UIC via the TMC. The UIC presents the result to the user.

9. Finally, the thread assigned to the transaction is released back to the thread pool by the SMC.
4.4 COMET Data and Transaction Model

Data items in COMET are stored and managed as in relational databases (see section 2.2). Each transaction in COMET contains a number of transaction operations, such as update or select, etc., which are implemented as SQL queries. A transaction can have more than one query, and each query is parsed into a parse tree (also called an execution tree). The transaction model is implemented by a struct \texttt{DBTrans} in the UIC. It holds the necessary information about a transaction, namely:

- \texttt{transactionID} identifies a transaction uniquely,
- \texttt{hasresult} is a boolean flag of each query in a transaction, indicating whether this query has a result (a query is a standard SQL-like query \cite{7});
- \texttt{result} is a buffer containing the result of a query, and a buffer in COMET is a struct used to store data temporarily for performing read and write operations.
- \texttt{noftree} is the number of parse trees (queries),
- \texttt{treelist} is a list of parse trees which contain queries,
- \texttt{deadline} is the deadline of a transaction, and
- \texttt{inUse} is a flag indicating whether a transaction is completed.

Besides \texttt{DBTrans}, a struct in the SMC, \texttt{SMC_Data}, also holds a part of transaction information needed for scheduling, as follows:

- \texttt{ThreadID} is the unique identifier of a thread,
- \texttt{ThreadHandle} is the handle of a thread that a transaction is running in, and
- \texttt{deadline} is the deadline of a transaction, same as in \texttt{DBTrans}. 
4.5 On-Demand Depth-First Traversal Algorithm

The ODDFT algorithm [2] is based on data freshness in the value domain (see section 3.2). It traverses the data dependency graph (presented in section 3.1) in depth-first order, and schedules the on-demand update transactions. A scheme for marking affected data items is needed for ODDFT to be able to schedule updates of data items. To that end, every data item is associated with a logical timestamp \( p_a \), indicating the latest logical time that the data item was found to be (potentially) affected by a change in another item. The logical timestamp is not a physical time, rather it is a unique integer value relative to an initial value, which increases monotonically. The marking scheme is described as follows.

**Step 1:** Update each base data item periodically. Upon update, check the freshness of each base item’s children according to definition 3 (see section 3.2).

**Step 2:** When a data item \( d \) (base or derived) is found to be stale due to a change in its parent \( x \), mark \( d \) and all its descendants as potentially affected. This is done by setting \( d \) and its descendants’ \( p_a \) to \( \max(p_a, ts(\tau)) \), where \( ts(\tau) \) is the logical timestamp of transaction \( \tau \) updating \( x \). Therefore, \( p_a \) always holds the latest timestamp of a transaction that makes the data item potentially affected.

**Step 3:** Before a user transaction starts, determine which data items should be updated. Schedule these updating transactions before the user transactions. Since the user transaction has updated the data item \( d \), \( d \)'s \( p_a \) is set to zero, meaning that \( d \) is fresh.

In Step 3, the ODDFT algorithm is implemented to generate a schedule and thereby guarantee that all on-demand updates are going to be executed before the user transaction. The routine of the algorithm contains three main steps (see figure 4.5).

1. **Traverse.** Assume \( d \) is a derived data item that a user transaction \( UT(d) \) wants to derive. If \( p_a(d) > 0 \), traverse the data dependency graph bottom-up from \( d \) following the depth-first order, recursively.
2. **Check the freshness.** For each $x \in R(d)$, if $pa(x) = 0$, $x$ is fresh and no update is needed; if $pa(x) > 0$, $x$ is potentially affected, then check whether $x$ is really stale using equation 4.1:

$$error(x, freshness\_deadline) > \delta_x$$

where $error(x, freshness\_deadline)$ is a worst-case value change of $x$ before $freshness\_deadline$. $freshness\_deadline$ is the latest time a data item should be valid, and is set to the deadline of $UT(d)$, because all data items should be fresh before the user transaction completes. $\delta_x$ is the validity bound of $x$. Therefore, when equation 4.1 is fulfilled, $x$ is considered to be stale. The value of $error$ is calculated as follows:

$$error(x, t) = (\text{maximum change of } x \text{ per time unit}) \times (t - ts(x))$$

where $ts(x)$ is the timestamp of $x$’ current value, i.e., the physical time when $x$ was updated to this value.

3. **Schedule.** Create a new transaction for updating $x$, whose deadline is set to the release time of the user transaction. Put this transaction into the schedule queue, a last-in-first-out queue. If the queue has any duplicates, remove them. Go to step 1 and find the next item $x$ until all $x$’s in $R(d)$ have been traversed.

For example, in figure 3.1 assume $pa$ of $d_1$ to $d_5$ are all larger than 0. When a user transaction wants to derive $d_5$, the ODDFT finds $d_4$ first and calculates $error(d_4, freshness\_deadline)$. If it is larger than $\delta_{d_4}$, a new transaction updating $d_4$ is created and placed into the schedule queue. Then the algorithm traverses $d_1$ and $d_2$ respectively. Assume both of them are found to be stale, hence three updates are scheduled by ODDFT, which are $update(d_1)$, $update(d_2)$ and $update(d_4)$.

### 4.6 Active Behavior in Real-time Databases

Many real-time database systems are active databases, because they must respond to external events. Considering time constraints, the traditional ECA model can be changed to [29]:
4.6. Active Behavior in Real-time Databases

Figure 4.5: The ODDFT algorithm
ON event IF condition THEN action within $t$

In the above model, a time constraint is added to the action part, which constrains the action to be completed within a time $t$. Event-condition coupling specifies when to evaluate the condition relative to the triggering event. Condition-action coupling specifies when to execute the triggered action relative to condition evaluation. There are three types of coupling modes: immediate, deferred and detached:

*Immediate coupling*, the condition evaluation (action execution) is performed immediately after event detection (condition evaluation).

*Deferred coupling*, the condition evaluation (action execution) is not performed immediately, but within the same transaction.

*Detached coupling*, the condition evaluation (action execution) is performed in a separate transaction from the triggering transaction.
4.6. Active Behavior in Real-time Databases
Chapter 5

Design and Implementation

This chapter presents the design and implementation of on-demand updating and active behavior in COMET in section 5.1 and section 5.2, respectively.

5.1 On-demand Updating Configuration

On-demand updating configuration is added to COMET with concurrency control. Two aspects are designed and implemented for this configuration, namely the ODDFTdatamodel aspect and the ODDFTalgorithm aspect. In order to explain the functionality and role of these aspects, we first discuss data and transaction models needed for the algorithm. Then we present the design and implementation of the ODDFT algorithm in detail. Finally, we present the resulting execution flow of the COMET with ODDFT configuration.
5.1.1 ODDFT Data and Transaction Model

As introduced in section 4.4, COMET currently does not support a dedicated struct for the data model, rather data items are managed in the format of tables and attributes. Furthermore, all manipulations on data are done by modifying and manipulating these tables and attributes. If a new data struct could be introduced for data items suitable for more applications in embedded systems, e.g., data items described by the data dependency graph, manipulation on data can be implemented more straightforward. If the positions of data are held in a struct, transactions could fetch them directly rather than constructing a SQL query for each read/write operation. However, all components and aspects developed so far use a table-attribute format, and they would not work on a new data model struct unless a large amount of work is invested to change all these components and aspects. Therefore, in this thesis, the data model is also implemented by using tables and attributes.

In the remainder of this section, the attributes of data items and new structs for data and transactions are explained. Then the join points and advices of the ODDFTdatamodel aspect are discussed.

Data Model

The entire set of data, including base items and derived items, is stored in one table, where one tuple stores one data item. Figure 5.1 illustrates how a set of data is stored in COMET tuple by tuple. Every tuple has the following attributes, characterizing a data item:

- **ID** is the unique integer identifier of a data item and the key of the tuple.
- **name** is the name of a data item, represented as a string, e.g., $b_0$, $d_4$.
- **type** is the type of a data item, which can be represented as an integer, using 0 for base items and 1 for derived items.
- **initValue** is the initial value of a data item. It is assumed that all the data items’ values are float numbers.
## Design and Implementation

<table>
<thead>
<tr>
<th>ID</th>
<th>NAME</th>
<th>TYPE</th>
<th>INITVALUE</th>
<th>VALUE</th>
<th>PTS</th>
<th>VALIDBOUND</th>
<th>PA</th>
<th>AVI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>b0</td>
<td>0</td>
<td>1.12</td>
<td>3.24</td>
<td>1200</td>
<td>2.00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>b1</td>
<td>0</td>
<td>42.00</td>
<td>50.00</td>
<td>1200</td>
<td>5.00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>d0</td>
<td>1</td>
<td>15.33</td>
<td>12.24</td>
<td>1544</td>
<td>1.50</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>d1</td>
<td>1</td>
<td>6.78</td>
<td>7.90</td>
<td>1089</td>
<td>3.00</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>d2</td>
<td>1</td>
<td>4.05</td>
<td>5.15</td>
<td>946</td>
<td>0.50</td>
<td>9</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 5.1: A set of data stored in COMET

- **value** is the current value of a data item.
- **pts** is the physical timestamp of a data item’s current value.
- **validBound** is the validity bound of a data item. For example, if the **validBound** of \( d_2 \) is 0.5, this would imply \( d_2 \)’s children are valid with respect to \( d_2 \) until the change of \( d_2 \) is larger than 0.5, e.g., a change of \( d_2 \) from 0 to 0.7.
- **pa** is the logical timestamp of a data item. When **pa** is larger than zero, this means that the item is potentially affected by a change in another data item at the latest logical time **pa**.
- **avi** is the absolute validity interval, only used for base data items.

As mentioned in section 3.1, the relationships of data items can be illustrated by a data dependency graph. This data dependency graph is stored in an adjacent matrix, implemented as a two-dimensional array in the ODDFT aspects. The row number and column number of an element
are both the ID of a data item. Each element of the matrix contains two fields:

- \textit{edge} is an integer type flag, which equals to 1 if there is an edge between two data items whose IDs are the row number and the column number, or equals to 0 if there is no edge between two items, i.e., they have no dependency relationship.

- \textit{updateFunction} is a pointer to a function that defines how to calculate and update the item’s value.

For example, assume that data item \(d_2\) is a parent of data item \(d_4\), ID of \(d_2\) is 2, ID of \(d_4\) is 4, and \(d_4 = d_2 \times 0.85\). In the adjacent matrix, item \(\text{matrix}[2][4]\) specifies the relationship between \(d_2\) and \(d_4\) by setting \textit{edge} to 1 and \textit{updateFunction} pointing to an updating function. \textit{edge} = 1 implies that there exists an edge from \(d_2\) to \(d_4\) in the data dependency graph, i.e., \(d_2\) is a parent of \(d_4\). The updating function pointed by \textit{updateFunction} contains the procedure for calculating \(d_4 = d_2 \times 0.85\).

\textbf{Transaction Model}

The ODDFT transaction model is implemented by extending the structs \texttt{DBTrans} of the UIC and \texttt{SMC_Data} of the SMC. The additional attributes introduced by the ODDFTDatamodel aspect are listed below.

- \textit{TransType} is the type of a transaction and can be \textit{ST}, \textit{UT}, \textit{TU} and \textit{OT}. \textit{ST} denotes sensor transactions, \textit{UT} denotes user transactions, \textit{TU} denotes triggered transactions and \textit{OT} denotes all other types of transactions, e.g., transactions performing \textit{create} or \textit{insert} operations when a set of data are created in the database.

- \textit{releaseTime} is the physical time when a transaction is initialized.

- \textit{arrivalTime} is the physical time when a transaction arrives to the system.

- \textit{dataID} is the ID of the data item which a transaction wants to access and update.
- \( \text{pts} \) is the physical timestamp of a transaction.
- \( \text{lts} \) is the logical timestamp of a transaction.

5.1.2 ODDFT Algorithm Aspect

The ODDFT algorithm aspect implements the functionality of the ODDFT algorithm in the COMET platform. Namely, the marking scheme for updating base data items and annotating the derived ones is constructed by inserting functions before and after write operations on the buffer. The actual ODDFT algorithm is carried out by inserting a series of functions before the \texttt{SMC\_Mech\_Start} operation, which schedules transactions. When this aspect is weaved into COMET, the UIC, TMC and SMC are crosscut.

Implementation of the ODDFT Marking Scheme

The marking scheme is implemented at the point when write operations are performed, i.e., at Step 6 of the overall COMET transaction flow described in section 4.3. The routine of the scheme is as follows.

1. The old value of a data item is saved before an update, which is done for both base and derived items.

2. After the new value of the data item is written into the buffer, the comparison of the old and new value is performed. If the difference exceeds the validity bound Step 3 is carried out; otherwise Step 4 is performed.

3. The data dependency graph is traversed top-bottom, and \( \text{pa} \) timestamps of the descendants of the data item are changed to the updating transaction’s logical timestamp. This is done by a recursive function call shown in figure 5.2. In figure 5.2, argument \( d \) is a data item, and argument \( \text{newPA} \) is the new \( \text{pa} \) value which is the logical timestamp of the transaction updating \( d \). In lines 2-3, the function finds every child of \( d \) by checking the adjacency matrix. If there is an edge from \( d \) to an item, this item is a child of \( d \) and its ID is \( i \). In line 4, the \textit{child} is located in the database by its ID. In lines 5-6, the following
5.1. On-demand Updating Configuration

```plaintext
markPA(d, newPA) {
    for (i = 0; i<numOfData; i++) {
        if (matrix[d.id][i]->edge==1) {
            findChild(child, i);
            if (newPA>child.pa)
                child.pa = newPA;
            markPA(child, newPA);
        }
    }
}
```

Figure 5.2: Function markPA()

is performed. If newPA is larger than child’s old pa, it is assigned to child.pa since pa is the latest logical time that a data item was found to be (potentially) affected. Finally, in line 7, the function recursively calls itself, to mark other descendents.

4. The item’s pa stamp is reset to zero, because its value is fresh now.

5. The item’s physical timestamp is changed to correspond the physical timestamp of the updating transaction.

**Implementation of the ODDFT Algorithm**

The ODDFT algorithm is run before a user transaction is scheduled by the SMC, i.e., in Step 3 in the overall COMET transaction flow.

Before we describe the main steps of the algorithm, we introduce the following notation.

- `userTrans(d)` denotes a user transaction that wants to derive a data item `d`.

- `triggeredTrans(x)` denotes a triggered transaction that updates data item `x ∈ R(d)`.

- affecting_queue is a queue of transactions, defined in the ODDFT algorithm aspect. It records all triggered transactions, in order to check duplicates.

The following steps represent the routine of the ODDFT algorithm for COMET. Recall that userTransaction(d) is waiting in the ready queue when the ODDFT algorithm starts to execute.

1. userTransaction(d) is removed temporarily from ready queue, before the SMC moves it to the active queue and starts executing. This is done to ensure that triggered updates can be inserted before the user transaction.

2. freshnessDeadline is set to the deadline of userTransaction(d), because all \( x \in R(d) \) should be valid before userTransaction(d) completes.

3. The affecting_queue is created in order to record the triggered updating transactions.

4. The data dependency graph is traversed in a depth-first manner using adjacent matrix and a data item \( x \) in \( R(d) \) is found. This is depicted in figure 5.3, lines 2-4.

5. \( x \) is checked for freshness as follows (see figure 5.3, line 5). If \( pa \) of \( x \) is not larger than zero and \( error \) is not larger than its validity bound (see equation 4.1), \( x \) is fresh and algorithm continues execution from Step 8. Otherwise, the algorithm continues onto Step 6.

6. triggeredTrans(x) is created and the transaction type is set to TU (triggered updating) as shown in line 6 of figure 5.3.

7. The release time of triggeredTrans(x), \( rt(x) \), and the arrival time of userTransaction(d), \( at(d) \), are compared. If \( rt(x) < at(d) \) which means triggeredTrans(x) is too late to be executed, triggeredTrans(x) is released, and then Step 8 is performed. Otherwise, the next \( x \) in \( R(d) \) is looked up by a recursive function call, i.e., Step 4 is performed until there is no more \( x \) in \( R(d) \). This step is illustrated in figure 5.3, lines 7-11.
8. Duplicate transactions are checked by checking if \( \text{triggeredTrans}(x) \) is already in the \textit{affecting_queue}. If not, \( \text{triggeredTrans}(x) \) is placed into the \textit{affecting_queue} and into the SMC ready queue, assigned an available thread and started. This step is executed when all ancestors of \( x \) have been traversed and on-demand updating transactions for them have been created, by recursively executing Steps 4-8, as illustrated in figure 5.3, lines 12-16.

9. The \textit{affecting_queue} is cleared.

10. \( \text{userTransaction}(d) \) is placed back to the SMC ready queue, and is started, and the normal COMET transaction execution flow is resumed.

\textbf{Aspect Structure}

The ODDFT data structures are declared in the ODDFTdatamodel aspect, while the ODDFT algorithm is implemented in the ODDFTalgorithm aspect. These two ODDFT aspects crosscut three basic components in COMET, the UIC, SMC, and TMC. They also use several operations and mechanisms from the UIC, SMC, TMC, MMC, and IMC components. The relationships between the ODDFT aspects and basic COMET components are illustrated in figure 5.4. A shadowed component means that aspects are weaved into it. A dotted arrow from an aspect to a component means that the aspect uses operations and/or mechanisms of the component. Table 5.1 also shows which components are crosscut and used by ODDFT aspects: a “√” in the crosscut column means that the component is crosscut by the aspects, while a “√” in the used column means that the component is used by the aspects. A short description of why the component is crosscut and/or used is also given. Figure 5.5 shows pseudocode of the ODDFTAlgorithm aspect, and illustrates how the ODDFT aspects crosscut the components. In the pseudocode, a join point \texttt{SMC\_Mech\_Start} in the SMC is declared in a pointcut expression, and then an advice \texttt{ODDFT\_beforeSchedule} corresponding to this pointcut is declared.
```c
oddft(d, freshness_deadline) {
    for (i = 0; i<numOfData; i++) {
        if (matrix[i][d.id]->edge==1) {
            get x whose id = i;
            if x.pa>0 and error(x,freshness_deadline)>x.validBound {
                create triggeredTrans(x);
                if (triggeredTrans(x).releaseTime<
                    userTransaction(d).arriveTime) {
                    release triggeredTrans(x);
                    break;
                } else {
                    oddft(x, freshness_deadline);
                    if no duplicates{
                        put triggeredTrans(x) into affectingQueue;
                        put triggeredTrans(x) into readyQueue;
                        start executing triggeredTrans(x);
                    }
                }
            }
            oddft(x, freshness_deadline);
            if no duplicates{
                put triggeredTrans(x) into affectingQueue;
                put triggeredTrans(x) into readyQueue;
                start executing triggeredTrans(x);
            }
        }
    }
}
```

Figure 5.3: Function oddft()
5.1. On-demand Updating Configuration

![ODDFT aspects in COMET](image)

Figure 5.4: ODDFT aspects in COMET
<table>
<thead>
<tr>
<th>Components</th>
<th>Crosscut</th>
<th>Used</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UIC</td>
<td>√</td>
<td>√</td>
<td>The UIC is crosscut to extend the transaction model when transactions are initialized and started. An operation in the UIC is replaced by the aspects to ensure support for user and triggered transactions. The aspects use the UIC to start a triggered updating transaction and to look up a transaction.</td>
</tr>
<tr>
<td>SMC</td>
<td>√</td>
<td>√</td>
<td>The SMC is crosscut to extend the transaction model, to enable the ODDFT algorithm, and to insert on-demand updates. The SMC mechanisms are used when scheduling on-demand updating transactions before the user transaction starts.</td>
</tr>
<tr>
<td>TMC</td>
<td>√</td>
<td>√</td>
<td>The TMC is crosscut to build the scheme needed for the ODDFT algorithm (see section 4.5). The mechanisms of its subcomponent BMC are used when the aspects perform read and write operations on the buffer.</td>
</tr>
<tr>
<td>MMC</td>
<td></td>
<td>√</td>
<td>The MMC is used by the aspects to access the memory when updating data items.</td>
</tr>
<tr>
<td>IMC</td>
<td></td>
<td>√</td>
<td>The IMC is used by the aspects to allocate data items in the memory.</td>
</tr>
</tbody>
</table>

Table 5.1: The components crosscut and used by the ODDFT aspects
aspect ODDFTalgorithm {
    public:
    /*
       * SMC pointcut
    */
    pointcut ODDFT_beforeSchedule(int transactionID, void * lpStartAddress) = execution("bool SMC_Mech_Start(...)") && args(transactionID, lpStartAddress);

    . . . . . .

    advice ODDFT_beforeSchedule(transactionID, lpStartAddress): before (int transactionID, void * lpStartAddress) {
        DBTrans * trans = NULL;
        // Find the transaction
        trans = RUIC_Op_lookupTransaction(transactionID);
        // Check if the transaction is a user transaction, call oddft
        if (trans->transType == UT) {
            ODDFT_Op_oddft(trans);
        }
    }
    . . . . . .
}

Figure 5.5: A piece of code of the ODDFTalgorithm aspect
5.1.3 ODDFT Execution Flow

When the ODDFT aspects are weaved into the COMET components, the on-demand updating algorithm works as explained in the steps below. This execution flow is also illustrated in figure 5.6. For each step in the figure, a short description is given in an oval box, and the components and aspects used in this step are listed under the box. A dotted arrow means that some steps in the whole COMET transaction execution routine are omitted. The flow of execution is as follows.

1. An application is initialized by the UIC. The ODDFTdatamodel aspect initializes the logical timestamp, and saves the physical initial time, by crosscutting the UIC component.

2. When a transaction begins, the ODDFTdatamodel aspect extends \texttt{DBTrans} and \texttt{SMC\_Data} structs, by adding necessary attributes (as discussed in section 5.1.1) for the transaction model, by crosscutting the UIC and SMC.

3. If the transaction is a user transaction updating item \(d\), before the SMC places the transaction into the active queue and assigns it a thread, the ODDFTalgorithm aspect performs the ODDFT algorithm (see section 5.1.2 for details), and schedules all the needed updating transactions before this user transaction. This is accomplished by weaving the ODDFTalgorithm aspect into the SMC component (as illustrated in figure 5.5).

4. Before a write operation on the buffer, if a new value of a data item is going to be written, the ODDFTalgorithm aspect saves this item’s old value, for the use in next step, by crosscutting the TMC.

5. After the write operation, the ODDFTalgorithm aspect compares the new and old values of data item \(d\), and takes actions in order to build the marking scheme of the ODDFT algorithm. This step is done also by crosscutting the TMC.
5.1. On-demand Updating Configuration

Join points and Advices

The ODDFTdatamodel aspect affects two components of COMET, the UIC and the SMC. As shown, this aspect extends the transaction model with additional information about transactions by adding extra members into DBTrans and SMC_Data structs. It also generates the logical timestamp and stores the original physical time when the database system is initialized. The ODDFTdatamodel aspect is stored in an aspect file, ODDFTdatamodel.ah. The functions this aspect uses are defined in a separate C file named ODDFTdatamodel.c. Figure 5.7 gives a pseudocode illustrating the outlook of the aspect. Tables 5.2 to 5.7 describe join points and advices declared in the ODDFTdatamodel aspect. The tables also give information stating components affected by the join point, components used by the aspect, the type of the advice at this join point (before/after/around), and a short description of the advice.

The ODDFTalgorithm aspect is stored in an aspect file, ODDFTalgorithm.ah. The functions it uses are defined in a C file, ODDFTalgorithm.c.

Figure 5.6: ODDFT execution flow
aspect ODDFTdatamodel {
    VARIABLES
    private:
        int TransType;
        Time releaseTime;
        Time arrivalTime;
        float pts;
        int dataID;
    
    POINTCUTS
    public:
        /*
         * UIC pointcut
         * Save the initial time point, in order to generate timestamps
         */
        pointcut ODDFT_initializeTimestamp(UIC_SystemParameters systemParameters) =
            execution("bool RUIC_Op_init(...)") && args(systemParameters);

        ADVICES
        /*
         * Extends the DBTrans struct, adding some transaction information for
         * on-demand
         * updates.
         *
         * introductions
         */
        advice "DBTrans" : int transType;
        advice "DBTrans" : Time releaseTime;
        advice "DBTrans" : Time arrivalTime;
        advice "DBTrans" : float pts;
        advice "DBTrans" : int lts;
        advice "DBTrans" : int dataID;

        ..............
};

Figure 5.7: The ODDFTdatamodel aspect
### 5.1. On-demand Updating Configuration

<table>
<thead>
<tr>
<th>Join point</th>
<th>DBtrans</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component affected</td>
<td>UIC</td>
</tr>
<tr>
<td>Component Used</td>
<td>UIC</td>
</tr>
<tr>
<td>Type</td>
<td>Introduction</td>
</tr>
<tr>
<td>Advice</td>
<td></td>
</tr>
<tr>
<td>int transType</td>
<td>the type of transactions: system transactions, user transactions, triggered updates or other transactions</td>
</tr>
<tr>
<td>Time releaseTime</td>
<td>the physical time when a transaction is released</td>
</tr>
<tr>
<td>Time arrivalTime</td>
<td>the physical time when a transaction arrives</td>
</tr>
<tr>
<td>float pts</td>
<td>the physical timestamp of a transaction</td>
</tr>
<tr>
<td>int lts</td>
<td>the logical timestamp of a transaction</td>
</tr>
<tr>
<td>int dataID</td>
<td>the ID of the data item that a transaction accesses</td>
</tr>
</tbody>
</table>

Table 5.2: Introduction DBTrans

<table>
<thead>
<tr>
<th>Join point</th>
<th>SMC_Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component affected</td>
<td>SMC</td>
</tr>
<tr>
<td>Component Used</td>
<td>SMC</td>
</tr>
<tr>
<td>Type</td>
<td>Introduction</td>
</tr>
<tr>
<td>Advice</td>
<td></td>
</tr>
<tr>
<td>int transType</td>
<td>the type of transactions: system transactions, user transactions, triggered updates or other transactions</td>
</tr>
<tr>
<td>int dataID</td>
<td>the ID of the data item that a transaction accesses</td>
</tr>
</tbody>
</table>

Table 5.3: Introduction SMC_Data
### Design and Implementation

**Join point**

execution

(RUIC\_Op\_beginTransaction())

<table>
<thead>
<tr>
<th>Component affected</th>
<th>Component Used</th>
<th>Type</th>
<th>Advice</th>
</tr>
</thead>
<tbody>
<tr>
<td>UIC</td>
<td>UIC</td>
<td>before</td>
<td>ODDFT_beginTransaction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Saves the information, such as the additional members introduced in DBTrans. Because in RUIC_Op_beginTransaction(), it allocates a position in TransList (a list maintains all transactions) for a transaction, copies all original information in DBTrans to this new position, then repoints the DBTrans pointer to it. If additional members are not saved, they will be lost after RUIC_Op_beginTransaction() operation.</td>
</tr>
</tbody>
</table>

Table 5.4: Advice ODDFT\_beginTransaction (before)

<table>
<thead>
<tr>
<th>Component affected</th>
<th>Component Used</th>
<th>Type</th>
<th>Advice</th>
</tr>
</thead>
<tbody>
<tr>
<td>UIC</td>
<td>UIC</td>
<td>after</td>
<td>ODDFT_beginTransaction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Restores the information in TransList to values before DBTrans pointer was repointed in the previous advice.</td>
</tr>
</tbody>
</table>

Table 5.5: Advice ODDFT\_beginTransaction (after)
5.1. On-demand Updating Configuration

<table>
<thead>
<tr>
<th>Join point</th>
<th>call (SMC_createData())</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component affected</td>
<td>SMC</td>
</tr>
<tr>
<td>Component Used</td>
<td>SMC</td>
</tr>
<tr>
<td>Type</td>
<td>after</td>
</tr>
<tr>
<td>Advice</td>
<td>ODDFT_SMCcreateData</td>
</tr>
<tr>
<td></td>
<td>Transfers additional transaction information in DBTrans to SMC_Data, such as dataID and TransType. Because this information is used frequently by other advices, in this way the information can be accessed directly from SMC_Data.</td>
</tr>
</tbody>
</table>

Table 5.6: Advice ODDFT_SMCcreateData

<table>
<thead>
<tr>
<th>Join point</th>
<th>execution (RUIC_Op_init())</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component affected</td>
<td>UIC</td>
</tr>
<tr>
<td>Component Used</td>
<td>UIC</td>
</tr>
<tr>
<td>Type</td>
<td>after</td>
</tr>
<tr>
<td>Advice</td>
<td>ODDFT_initializeTimestamp</td>
</tr>
<tr>
<td></td>
<td>When the database is initialized, the advice initializes the logical timestamp to zero and saves the initial physical time point in order to generate physical timestamps.</td>
</tr>
</tbody>
</table>

Table 5.7: Advice ODDFT_initializeTimestamp
Join point execution
(RUIC\_Op\_commitTransaction())

<table>
<thead>
<tr>
<th>Component affected</th>
<th>UIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component Used</td>
<td>UIC</td>
</tr>
<tr>
<td>Type</td>
<td>after</td>
</tr>
<tr>
<td>Advice</td>
<td>ODDFT_commitTrans</td>
</tr>
</tbody>
</table>

If the transaction is a user transaction or a triggered updating, the advice replaces RUIC\_Op\_commitTransaction with ODDFT\_Op\_commitTransaction. Because in ODDFT\_Op\_commitTransaction, it looks up the updateFunction for the data item in the adjacent matrix.

Table 5.8: Advice ODDFT\_commitTrans

The join points and advices are listed in tables 5.8 to 5.11.

5.2 Active Behavior Configuration

Active behavior functionality is added to COMET on top of the configuration containing concurrency control and on-demand updating. Active behavior is designed and implemented as an aspect, the activeBehavior aspect. This section first presents the rules of active behavior specified for embedded systems and adjusted for COMET. Then, the aspect structure is described and the data model and new data structures are introduced. Finally, the implementation of the active behavior aspect is explained in detail.

5.2.1 Rules of Active Behavior in COMET

COMET configurations, including concurrency control and on-demand update, do not support active behavior, since it is assumed that the system
5.2. Active Behavior Configuration

<table>
<thead>
<tr>
<th>Join point</th>
<th>call (BMC_Mech_floatSetXY()) within(bool RTMC_Mech_update())</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component affected</td>
<td>TMC(BMC)</td>
</tr>
<tr>
<td>Component Used</td>
<td>TMC(BMC)</td>
</tr>
<tr>
<td>Type</td>
<td>before</td>
</tr>
<tr>
<td>Advice</td>
<td>ODDFT_writeFloatData</td>
</tr>
<tr>
<td></td>
<td>Before updating a data item, saves its old value, in order to compare with the new value after updating.</td>
</tr>
</tbody>
</table>

Table 5.9: Advice ODDFT_writeFloatData (before)

<table>
<thead>
<tr>
<th>Join point</th>
<th>call (BMC_Mech_floatSetXY()) within(bool RTMC_Mech_update())</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component affected</td>
<td>TMC(BMC)</td>
</tr>
<tr>
<td>Component Used</td>
<td>TMC(BMC), UIC, IMC, MMC</td>
</tr>
<tr>
<td>Type</td>
<td>after</td>
</tr>
<tr>
<td>Advice</td>
<td>ODDFT_writeFloatData</td>
</tr>
<tr>
<td></td>
<td>After writing a data item’s value to the buffer, check whether the change exceeds the validity bound. If exceeding, marks all its descendants’ pa to the transaction’s logical timestamp. Then resets this updated item’s pa to zero, and updates its value’s physical timestamp corresponding to the transaction’s arrival time.</td>
</tr>
</tbody>
</table>

Table 5.10: Advice ODDFT_writeFloatData (after)
Join point execution (SMC_Mech_Start())

<table>
<thead>
<tr>
<th>Component affected</th>
<th>SMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component Used</td>
<td>SMC, TMC(BMC), UIC, IMC, MMC</td>
</tr>
<tr>
<td>Type</td>
<td>before</td>
</tr>
<tr>
<td>Advice</td>
<td>ODDFT_beforeSchedule</td>
</tr>
</tbody>
</table>

Table 5.11: Advice ODDFT_beforeSchedule

is time triggered, i.e., all data items are derived (updated) periodically. In such a case, each data item has its own updating interval. For example, in an EECU, the engine temperature (a base item) is updated from the sensor every second, and the temperature compensation factor (a derived item and a child of the engine temperature) is updated every five seconds.

If the active behavior could be part of COMET, then it would not be necessary to periodically update all data items. A subset of data items, including all base items and some derived ones is still updated based on their updating intervals. However, the remaining items can be updated only when appropriate conditions are satisfied on specific events, i.e., these updates can be triggered. Figure 5.8 shows an example of a set of data. When the database does not support active behavior, all data items are updated periodically. When the database is active, all base data items and derived data items in the left part are still updated periodically, while updates of derived data items in the right part are only triggered on appropriate events. Based on the above assumption, the ECA rules are defined as follows.

Event

In embedded systems, for example, events are changes of data that reflect the changes in the external environment. Therefore, events can be defined as occurrences of updating transactions. More specifically, events can be
further divided into two categories: before updating event and after updating event. Recall that updating transactions include system transactions (ST), user transactions (UT) and also triggered updating transactions (TU). On an event, zero or more than zero actions can be triggered, depending on a particular condition which we describe next.

**Condition**

Due to different types of events, conditions are also divided into two types.

*Before updating condition* is the on-demand updating condition. Namely, before a user transaction starts, on-demand updating transactions are triggered by using an on-demand updating algorithm in order to guarantee the freshness of data.

*After updating condition* is declared according to the change of data values. Namely, after an update, the change of the item’s value is checked, and if it exceeds a value threshold or falls into a specified interval, a corresponding action is triggered. For example, in figure 5.9, when a transaction updating $d_4$ to a value larger than 99, an action
on EVENT after update(d_4)
if CONDITION d_4 > 99
then ACTION update(d_{10})
if CONDITION d_4 ∈ {70, 80}
then ACTION update(d_7)

Figure 5.9: An example for condition after updating

update(d_{10}) is triggered. When d_4 is updated to a value between 70 to 80, another action update(d_7) is triggered.

Action

An action is an updating transaction, separated from the triggering transaction. This action-related updating transaction is time constrained and must complete before its deadline. The deadline of an action updating data item d is specified to be the end point of the current updating interval t_i of the data item d.

ACTION update(d) within t_i
The updating interval is used to update the data item periodically when the database is not active. If an action cannot meet the deadline, it will be postponed to the next updating interval of \( d \). For example, in figure 5.10, the updating interval of \( d_7 \) is 20 time units, and the execution time of the action \( update(d_7) \) is 10 time units. If \( update(d_7) \) is triggered at time point 8, its deadline is set to 20.

\[
\text{ACTION } update(d_7) \text{ within } 20 \\
\]

if \( update(d_7) \) is triggered at time point 15, it cannot complete in this period, so \( update(d_7) \) is postponed to the next updating period.

\[
\text{ACTION } update(d_7) \text{ within } 40 \\
\]

5.2.2 Aspect Structure

The activeBehavior aspect crosscuts two components, the UIC and TMC, and uses operations and mechanisms from the UIC, TMC, SMC and the ODDFT aspects. The relationships among these aspects and components are illustrated in figure 5.11. A shadowed component indicates that it is crosscut by the activeBehavior aspect. A dotted arrow from an aspect to a component/aspect indicates that the aspect uses some operations and/or mechanisms of the component/aspect. Table 5.12 also gives a brief description of the relationships. The symbols used in this table are analogous as in table 5.1.
Figure 5.11: The active behavior aspect in COMET
### 5.2. Active Behavior Configuration

<table>
<thead>
<tr>
<th>Components / Aspects</th>
<th>Crosscut</th>
<th>Used</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UIC</td>
<td>✓</td>
<td>✓</td>
<td>The UIC is crosscut to insert events and condition evaluations. The aspect uses the UIC to start triggered actions (transactions).</td>
</tr>
<tr>
<td>SMC</td>
<td></td>
<td>✓</td>
<td>The SMC mechanisms are used to schedule triggered actions (transactions).</td>
</tr>
<tr>
<td>TMC</td>
<td>✓</td>
<td>✓</td>
<td>The TMC is crosscut to trace the change of data values for condition evaluations. The mechanisms of its sub-component BMC are used when the aspect performs read and write operations on the buffer.</td>
</tr>
<tr>
<td>ODDFT</td>
<td></td>
<td>✓</td>
<td>An ODDFT operation is used to generate physical timestamps for triggered actions (transactions).</td>
</tr>
</tbody>
</table>

Table 5.12: The components crosscut and used by the activeBehavior aspect.
5.2.3 Active Behavior Data and Transaction Model

The active behavior configuration uses the same data model and transaction model as the ODDFT configuration. In addition, two data structs are added to specify the ECA rules as follows.

**struct CAcouple** is a struct recording the information of one condition-action coupling. Figure 5.12 gives an example of CAcouple. CAcouple contains two members:

- **condFunction** is a pointer to a function, which specifies how to evaluate a condition. The function returns *true* if the condition is fulfilled, and returns *false* if the condition is not fulfilled. In figure 5.12, condFunction points to `d4condition0()`, which checks the value of $d_4$. When $d_4$ is larger than 99, `d4condition0()` returns *true*, otherwise returns *false*.

- **targetID** is the ID of a data item, which will be updated if the condition is satisfied, i.e., *condFunction* returns *true*. Still in figure 5.12, the targetID corresponds to $d_{11}$, when function `d4condition0()` returns *true*, an action `update(d_{10})` will be triggered.

**struct ABrule** is a struct storing the information of ECA rules for a data item. Figure 5.13 gives an example of ABrule for item $d_4$. Struct ABrule contains the following members:
5.2. Active Behavior Configuration

Figure 5.13: An example of struct ABrule

```c
struct d4ABrue {
    dataValue = 0;
    interval  = 5;
    postponed = false;
    calist[0] = d4CAcouple;
}
```

- `dataValue` is the current value of a data item. It is kept the same as the current value stored in the database. When performing a condition evaluation in COMET, the value can be fetched from this struct directly, other than from the database by constructing a SQL query, consequently making the implementation more efficient. In figure 5.13, `dataValue` of `d_4` is 0 currently.

- `interval` is the updating interval for a data item. The interval of `d_4` is 5.

- `postponed` is a boolean flag. When an action updating a data item is postponed to the next updating interval, `postponed` is set to `true`, otherwise it is `false`. In figure 5.13, the postponed flag is false for `d_4`.

- `calist` is an array contains a number of `condition-action` coupling by using the `CAcouple` struct. Each data item can have several conditions according to the change of its value, and each condition is bound with a distinct action. In figure 5.13, the calist of `d_4` contains one `CAcouple`, `d_4CAcouple`.

5.2.4 Implementation of Active Behavior Aspect

The implementation of the active behavior consists of two parts. One is carried out by the ODDFT configuration and is related to the functionality of condition evaluation and action execution before updating. The other is
a part of the active behavior aspect, giving the functionality of condition evaluation and action execution after updating. In the overall COMET transaction execution flow (see section 4.3 and figure 4.4), the functionality implemented by the active behavior aspect is added after Step 7 and Step 8, before Step 9, i.e., in the event a transaction has completed and cannot be rollbacked.

Before describing the implementation of active behavior, we introduce the following notation. Let $d$ denote a data item and $update(d)$ denote a transaction updating $d$. The active behavior is implemented by the following steps.

1. When $update(d)$ is completed, if postponed flag of $d$ is $true$, reset the flag to $false$, as the postponed action has completed.

2. Check ECA rules of $d$. Evaluate the condition specified in each CAcouple one by one. If the condition is $true$, continue to Step 3, otherwise check the next condition until there is no more conditions.

3. If the action is $update(x)$ in this condition-action coupling, then calculate the time left in the current updating interval of $x$.

4. Compare the time left with the worst case execution time of $update(x)$. If there is enough time to execute $update(x)$, create a transaction for $update(x)$ and start it. Otherwise, set the postponed flag of $x$ to $true$ indicating that this action is postponed to the next updating interval.

5. Return to Step 2 and check the next rule until all conditions have been evaluated.

**Join points and Advices**

The activeBehavior aspect is stored in an aspect file, `activeBehavior.ah`. The functions it uses are defined in a c file, `activeBehavior.c`. The join points and advices are listed in tables 5.13 to 5.14, as in the former section.
### 5.2. Active Behavior Configuration

#### Table 5.13: Advice ab_commitpoint

<table>
<thead>
<tr>
<th>Join point</th>
<th>call (RUIC_Op_rollbackTransaction())</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component affected</td>
<td>UIC</td>
</tr>
<tr>
<td>Component Used</td>
<td>UIC, SMC</td>
</tr>
<tr>
<td>Type</td>
<td>before</td>
</tr>
<tr>
<td>Advice</td>
<td>ab_commitpoint</td>
</tr>
</tbody>
</table>

When a transaction is completed, which means a data item has been updated and cannot be rollback, the advice checks the conditions. If the postponed flag is true, resets it to false, because the updating transaction has been executed.

#### Table 5.14: Advice ab_saveNewValue

<table>
<thead>
<tr>
<th>Join point</th>
<th>call(BMC_Mech_floatSetXY())</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component affected</td>
<td>TMC (BMC)</td>
</tr>
<tr>
<td>Component Used</td>
<td>TMC (BMC)</td>
</tr>
<tr>
<td>Type</td>
<td>after</td>
</tr>
<tr>
<td>Advice</td>
<td>ab_saveNewValue</td>
</tr>
</tbody>
</table>

After a new value is written into the buffer, the advice also saves the new value to dataValue in the ABrule struct.
Chapter 6

Performance Evaluation

This chapter describes the performance evaluation of the active behavior COMET configuration. Section 6.1 presents the setup of a simulated database, while in section 6.2 the experiments which are used to evaluate the performance are presented and the results are discussed.

6.1 Simulator Setup

COMET runs on a Sun Ultra-10 workstation in a command window as a user application. The RAM of the workstation is 256 Mb, and the operating system is Solaris 10. To simulate the environment for the experiments in this thesis, COMET is configured with the concurrency control, HP-2PL with similarity, and the on-demand updating algorithm, ODDFT. COMET uses EDF algorithm to schedule the transactions and assigns each transaction a thread for execution. Then, the threads are scheduled by the operating system.

The simulator consists of a small set of 15 data items. These data items simulate the data items in an embedded system which can be described in a data dependency graph. Although the number of data items is much less than in the real situation, the logical relationship of the data items is similar with the real situation. Therefore, the small set of data items is
6.1. Simulator Setup

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Value Interval</th>
<th>Validity Bound</th>
<th>Updating Interval</th>
<th>Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$b_1$</td>
<td>{0,2}</td>
<td>0.2</td>
<td>4 sec</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$b_2$</td>
<td>{0,4}</td>
<td>0.5</td>
<td>4 sec</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$b_3$</td>
<td>{2,5}</td>
<td>0.5</td>
<td>4 sec</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$b_4$</td>
<td>{0,3}</td>
<td>0.5</td>
<td>4 sec</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$b_5$</td>
<td>{0.5,1.5}</td>
<td>0.1</td>
<td>4 sec</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$d_1$</td>
<td>{0,6}</td>
<td>0.5</td>
<td>12 sec</td>
<td>$d_1 = b_1 + b_2$</td>
</tr>
<tr>
<td>6</td>
<td>$d_2$</td>
<td>{4,10}</td>
<td>1</td>
<td>8 sec</td>
<td>$d_2 = b_3 \times 2$</td>
</tr>
<tr>
<td>7</td>
<td>$d_3$</td>
<td>{0,18}</td>
<td>1</td>
<td>16 sec</td>
<td>$d_3 = d_1 \times 3$</td>
</tr>
<tr>
<td>8</td>
<td>$d_4$</td>
<td>{0,60}</td>
<td>3</td>
<td>20 sec</td>
<td>$d_4 = d_1 \times d_2$</td>
</tr>
<tr>
<td>9</td>
<td>$d_5$</td>
<td>{0,30}</td>
<td>2</td>
<td>24 sec</td>
<td>$d_5 = d_4/2$</td>
</tr>
<tr>
<td>10</td>
<td>$d_6$</td>
<td>{1,4}</td>
<td>0.6</td>
<td>8 sec</td>
<td>$d_6 = b_4 + 1$</td>
</tr>
<tr>
<td>11</td>
<td>$d_7$</td>
<td>{2,8}</td>
<td>1</td>
<td>20 sec</td>
<td>$d_7 = d_6 \times 2$</td>
</tr>
<tr>
<td>12</td>
<td>$d_8$</td>
<td>{2/3,8/3}</td>
<td>0.3</td>
<td>24 sec</td>
<td>$d_8 = d_7/3$</td>
</tr>
<tr>
<td>13</td>
<td>$d_9$</td>
<td>{0,1}</td>
<td>0.1</td>
<td>16 sec</td>
<td>$d_9 = b_5 - 0.5$</td>
</tr>
<tr>
<td>14</td>
<td>$d_{10}$</td>
<td>{0,0.25}</td>
<td>0.02</td>
<td>12 sec</td>
<td>$d_{10} = d_9 \times 0.25$</td>
</tr>
</tbody>
</table>

Table 6.1: Specified information of data items

appropriate to be used in the experiments. The data items are stored in one table, where each tuple represents one data item. The data dependency graph of the items is shown in figure 5.8, where $b_1$ to $b_5$ are base data items, and $d_1$ to $d_{10}$ are derived data items. Table 6.1 gives specified information of the data items, including data ID, data name, value interval, validity bound, updating interval and the calculation associated with each data item.

The ECA rules are defined in table 6.2. The Condition column gives a value interval. If a value of the data item falls within the interval, the action in the Action column is taken. For example, when the value of $d_2$ is changed to 9, which falls within the interval {8,10}, the action $update(d_6)$ is triggered and the ODDFT algorithm is performed before the update. Figure 6.1 also illustrates the triggering relationships among data items,
where a dotted arrow from item \( x \) to \( y \) represents that a change of the value of \( x \) may trigger an action updating \( y \).

The test application loops 200 times, and each loop is 4 seconds long, so the whole test runs for 800 seconds. Figure 6.2 gives a pseudocode illustrating what the application does in each loop. Since the updating interval of base data items is 4 seconds, all base items are updated once in each loop. For derived data items, when the loop time can be divided by the updating interval of a derived item, the derived item is updated. For example, as the updating interval of \( d_2 \) is 8 seconds, \( d_2 \) is updated in every other loop.

In general, the arrival rate of transactions is used to change the system load. However, in the case of active behavior, a number of transactions are triggered depending on events and conditions, making the arrival rate difficult to calculate. Therefore in these experiments, we choose to change the WCET instead of the arrival rate to induce a change in the system load. Hence, all transactions are associated with WCETs and are forced to execute for entire WCET. The WCET for the transaction updating base items (ST) is fixed to 0.5 second, while the WCETs for transactions updating derived items (UT and TU) is varied from 0.2 to 1.5 seconds. For every WCET, the application runs 5 times to get 5 samples, in this way the 95% confidence interval can be calculated.

<table>
<thead>
<tr>
<th>Data Item</th>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d_1 )</td>
<td>{0,2}</td>
<td>update(( d_{10} ))</td>
</tr>
<tr>
<td>( d_2 )</td>
<td>{8,10}</td>
<td>update(( d_6 ))</td>
</tr>
<tr>
<td>( d_3 )</td>
<td>{15,18}</td>
<td>update(( d_9 ))</td>
</tr>
<tr>
<td></td>
<td>{0,5}</td>
<td>update(( d_6 ))</td>
</tr>
<tr>
<td>( d_4 )</td>
<td>{30,40}</td>
<td>update(( d_7 ))</td>
</tr>
<tr>
<td>( d_5 )</td>
<td>{0,10}</td>
<td>update(( d_8 ))</td>
</tr>
<tr>
<td></td>
<td>{10,20}</td>
<td>update(( d_{10} ))</td>
</tr>
</tbody>
</table>

Table 6.2: The ECA rules for experiments
### 6.1. Simulator Setup

b: base data
d: derived data

**Figure 6.1: The ECA rules for test**

```java
for (run=0; run<200; run++) {
    updateBaseData();
    for each derived item d {
        if (run % (interval of d)==0) {
            updateDerivedData(d);
        }
    }
    _usleep(4 seconds);
}
```

**Figure 6.2: The pseudocode of the application**
6.2 Experiments

The experiments are designed to compare the system performance between the database without the active behavior functionality, where all data items are updated periodically, and the database with the active behavior functionality. Hence, two scenarios are considered in the experiments as follows.

**Periodic Scenario** where COMET is not active. All data items are updated periodically based on their updating intervals and the ODDFT algorithm is used.

**Active Scenario** where COMET is active. All base items and the derived items in the left part in figure 6.1, $b_1$ to $b_5$ and $d_1$ to $d_5$, are still updated periodically, while updates of other items in the right part, $d_6$ to $d_{10}$, are only triggered depending on the ECA rules. The values are changed randomly, so the actions updating $d_6$ to $d_{10}$ are triggered randomly. The ODDFT algorithm is also used in this scenario.

Figure 6.3 shows the total number of all arriving UTs and TUs, including the aborted transactions that miss their deadlines. Figure 6.4 shows the number of access operations, i.e. read and write operations on values and all parameters of derived data items, e.g., $pa$ and $pts$. Figure 6.5 shows the deadline miss ratio, which is a ratio of the number of missed deadlines of transactions to the number of all transactions. For both periodic and active scenarios, every point represents the mean value of 5 samples, and the short bar at every point is the 95% confidence interval of that value.

In figure 6.3 and figure 6.4, we can see that the number of transactions and the number of accesses decrease as the WCET increases in both scenarios. That is because before a TU is scheduled, the ODDFT algorithm checks whether the TU has enough time to perform based on the release time of the TU and the arrival time of the triggering transaction (see section 5.1.2). If there is insufficient time to execute a TU, the TU is aborted. The release time of a TU equals the deadline of the TU minus WCET. When the WCET increases, more TUs are aborted based on the time limit check, and hence the number of transactions decreases. Since the number of transactions decreases, less access operations are performed. However, a part of access operations are performed on the parameters of data items,
which are accessed frequently by the ODDFT algorithm. Therefore, the difference of the numbers of transactions between two scenarios is significant, but the difference of the numbers of accesses is relatively small. Considering the system performance, reducing the number of transactions is more important than reducing the number of accesses, as CPU can have less pressure to handle and schedule arriving transactions and less memory is allocated for transactions.

In the periodic scenario, the deadline miss ratio is 0 when the WCET is 0.3 second, which means that at this point the system is not overloaded. When the WCET increases to 0.4 second, the system is overloaded and its performance degrades quickly: the number of transactions decreases, less access operations are performed and the deadline miss ratio increases. The ratio becomes 36.68% when the WCET is 1.5 seconds.

In the active scenario, the number of arriving transactions and accesses on the data base is less than in the periodic scenario, because the updates of $d_6$ to $d_{10}$ are not so frequent and are triggered depending on the ECA rules. Figure 6.6 compares the number of updates of $d_6$ to $d_{10}$ in both scenarios. The number in the active scenario is the average value calculated in all experiments in this scenario. In the active scenario, the system is overloaded when the WCET is 0.5 second, when the WCET is 1.5 second, the ratio of deadline misses is 31.64%, which is still better than in the periodic scenario.

From the above comparison, when the active behavior functionality is added to COMET, the system load is reduced, the CPU and memory can be used more efficiently, therefore the performance is improved.
Figure 6.3: Number of UTs and TUs

Figure 6.4: Number of data accesses
6.2. Experiments

Figure 6.5: Deadline miss ratio

Figure 6.6: The number of updates of $d_6$ to $d_{10}$
Chapter 7

Conclusion

This chapter gives conclusions in section 7.1, and discusses issues for the future work in section 7.2.

7.1 Summary

Nowadays, developers of software for embedded systems face the challenge of how to manage a large number of data while the resources, the CPU and memory, are limited. This thesis discusses two methods developed specifically to address data management in resource constrained environments, the on-demand updating and active behavior. The focus is on extending COMET, which is a configurable real-time database for embedded systems, with two new configurations that support on-demand updating and active behavior functionality.

In the on-demand updating COMET configuration, the ODDFT algorithm is implemented as two aspects which crosscut three COMET components. When a user transaction arrives, the ODDFT algorithm traverses the data dependency graph in depth-first manner, and triggers and schedules on-demand updates based on the data freshness in the value domain. This way, the unnecessary recalculations in updates are avoided, enabling more efficient resource manipulation, e.g., CPU and memory.
The active behavior is also implemented as an aspect, which crosscuts two COMET components. When COMET is active, which is ensured by defining event-condition-action rules, a part of data items are only updated when necessary (as opposed to periodic updates). The updates of these data items are triggered only when a particular event occurs and a condition associated with an event is satisfied. As a result, less transactions arrive at the system and less access operations are performed on the database. The usage of CPU and memory is decreased, and the system performance is improved. As we demonstrate experimentally, in the periodic scenario the system is overloaded when the WCET increases to 0.4 second, while in the active scenario the system is overloaded when the WCET increases to 0.5 second. When the WCET is 1.5 seconds, the deadline miss ratio is 36.68% in the periodic scenario and is 31.64% in the active scenario.

7.2 Future Work

A number of issues remain that can be improved in future. First, the data struct in COMET is not very appropriate for the management of data items in the embedded systems that can be represented by a data dependency graph. All information of a data item, not only the value but also the parameters, e.g., timestamp, is stored as an attribute of a tuple in the table. The operations on the parameters need to construct SQL queries and search in the memory via a complicated process, which slows down the transaction execution. It would be more beneficial to have a new data struct introduced into COMET for the embedded system data model.

Second, more work can be invested in the performance evaluation. In this thesis, the performance evaluations are done on a small data set. It would be more beneficial to have also simulators of a large data set, since the number of data in embedded systems is generally in order of thousands. Also, the experiments use a loop to enable the application to run for a long period of time. In the future work, physical time can be used for this simulation, and the result can be compared with results obtained using the loop method as using physical time may be closer to the real life implementation.

Third, the on-demand updating configuration added by this thesis in-
cludes one algorithm, the ODDFT algorithm. Besides the ODDFT algo-

rithm, there are several other on-demand algorithms, such as ODKB and

ODBFT. The on-demand updating configurations can be further extended
to support more algorithms within the COMET platform.

Finally, the stability of COMET could be further improved. Specifically,
memory management exhibits unpredictable behavior, especially when two
or more transactions try to access the same address in the memory. Fur-
thermore, how to make new aspects cooperate with the existing aspects
needs to be investigated in more depth. The aspects added by this thesis
are able to work together with the concurrency control aspects, but it has
not been tested if they can work with other aspects, such as QoS aspects.
7.2. Future Work
Bibliography


An embedded system is an application-specific system that is typically dedicated to performing a particular task. Majority of embedded systems are also real-time, implying that timeliness in the system need to be enforced. An embedded system needs to be enforced efficient management of a large amount of data, including maintenance of data freshness in an environment with limited CPU and memory resources. Uniform and efficient data maintenance can be ensured by integrating database management functionality with the system. Furthermore, the resources can be utilized more efficiently if the redundant calculations can be avoided. On-demand updating and active behavior are two solutions that aim at decreasing the number of calculations on data items in embedded systems.

COMET is a COMponent-based Embedded real-Time database, developed to meet the increasing requirements for efficient data management in embedded real-time systems. The COMET platform has been developed using a novel software engineering technique, AspeCtual COmponent-based Real-time software Development (ACCORD), which enables creating database configurations, using software components and aspects from the library, based on the requirements of an application. Although COMET provides uniform and efficient data management for real-time and embedded systems, it does not provide support for on-demand and active behavior.

This thesis is focusing on design, implementation, and evaluation of two new COMET configurations, on-demand updating of data and active behavior. The configurations are created by extending the COMET component and aspect library with a set of aspects that implement on-demand and active behavior. The on-demand updating aspect implements the ODDFT algorithm, which traverses the data dependency graph in the depth-first manner, and triggers and schedules on-demand updates based on data freshness in the value domain. The active behavior aspect enables the database to take actions when an event occurs and a condition coupled with that event and action is fulfilled.

As we show in the performance evaluation, integrating on-demand and active behavior in COMET improves the performance of the database system, gives a better utilization of the CPU, and makes the management of data more efficient.
Copyright

Svenska

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnad som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/