Mälardalen University Press Dissertations
No.21

Data Management in Vehicle Control-Systems

Dag Nyström

October 2005

Department of Computer Science and Electronics
Mälardalen University
Västerås, Sweden

E-mail: dag.nystrom@mdh.se
ISSN 1651-4238
ISBN 91-88834-97-2
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

As the complexity of vehicle control-systems increases, the amount of information that these systems are intended to handle also increases. This thesis provides concepts relating to real-time database management systems to be used in such control-systems. By integrating a real-time database management system into a vehicle control-system, data management on a higher level of abstraction can be achieved.

Current database management concepts are not sufficient for use in vehicles, and new concepts are necessary. A case-study at Volvo Construction Equipment Components AB in Eskilstuna, Sweden presented in this thesis, together with a survey of existing database platforms confirms this. The thesis specifically addresses data access issues by introducing; (i) a data access method, denoted database pointers, which enables data in a real-time database management system to be accessed efficiently. Database pointers, which resemble regular pointers variables, permit individual data elements in the database to be directly pointed out, without risking a violation of the database integrity. (ii) two concurrency-control algorithms, denoted 2V-DBP and 2V-DBP-SNAP which enable critical (hard real-time) and non-critical (soft real-time) data accesses to co-exist, without blocking of the hard real-time data accesses or risking unnecessary abortions of soft real-time data accesses. The thesis shows that 2V-DBP significantly outperforms a standard real-time concurrency control algorithm both with respect to lower response-times and minimized abortions. (iii) two concepts, denoted substitution and subscription queries that enable service- and diagnostics-tools to stimulate and monitor a control-system during run-time.

The concepts presented in this thesis form a basis on which a data management concept suitable for embedded real-time systems, such as vehicle control-systems, can be built.

Swedish summary - Svensk sammanfattning

Ett modernt fordon är idag i princip helt styrt av inbyggda datorer. I takt med att funktionaliteten i fordonen ökar, blir programvaran i dessa datorer mer och mer komplex. Komplex programvara är svår och kostsam att konstruera. För att hantera denna komplexitet och underlätta konstruktion, satsar nu industrin på att finna metoder för att konstruera dessa system på en högre abstraktionsnivå. Dessa metoder syftar till att strukturera programvaran i dess olika funktionella beståndsdelar, till exempel genom att använda så kallad komponentbaserad programvaruutveckling. Men, dessa metoder är inte effektiva vad gäller att hantera den ökande mängden information som följer med den ökande funktionaliteten i systemen. Exempel på information som skall hanteras är data från sensorer utspridda i bilen (temperaturer, tryck, varvtal osv.), styrdata från föraren (t.ex. rattutslag och gaspådrag), parameterdata, och loggdata som används för servicediagnostik. Denna information kan klassas som säkerhetskritisk eftersom den används för att styra beteendet av fordonet. På senare tid har dock mängden icke säkerhetskritisk information ökat, exempelvis i bekvämlighetssystem som multimedia-, navigations- och passagerarergonomisystem.

Denna avhandling syftar till att visa hur ett datahanteringssystem för inbyggda system, till exempel fordonssystem, kan konstrueras. Genom att använda ett realtidsdatabashanteringssystem för att lyfta upp datahanteringen på en högre abstraktionsnivå kan fordonssystem tillåtas att hantera stora mängder information på ett mycket enklare sätt än i nuvarande system. Ett sådant datahanteringssystem ger systemarkitekterna möjlighet att strukturera och modellera informationen på ett logiskt och överblickbart sätt. Informationen kan sedan läsas och uppdateras genom standardiserade gränssnitt anpassade för olika typer av funktionalitet. Avhandlingen behandlar specifikt problemet hur information i databasen, med hjälp av en concurrency-control algoritm, skall kunna delas av både säkerhetskritiska och icke säkerhetskritiska systemfunktioner i fordonet. Vidare avhandlas hur information kan distribueras både mellan olika datorsystem i fordonet, men också till diagnostik- och serviceverktyg som kan kopplas in i fordonet.

TO INFINITY AND BEYOND.

Buzz Lightyear - Space ranger

Acknowledgements

Finishing a journey like this requires the support and inspiration of many people. In my case I have been blessed with a whole bunch of great friends who deserves, at least, many warm thanks. Above all, I sincerely would like to thank my supervisors at the department, Professor Christer Norström and Associate Professor Mikael Nolin. Christer has always been a great source of inspiration, both on a technical and a personal level. I owe you big time! Mikael, which joined the research group sometime after my licentiate degree, has an astonishing ability to detect weaknesses in otherwise "perfect" solutions. It's a pleasure writing papers with you.

I would also like to thank the Linköping part of the COMET-project, Dr. Jörgen Hansson whose long experience in real-time database management has been a good source of knowledge. Last, but not least, Aleksandra Tesanovic, co-researcher and good friend. We have had many interesting discussions over the years. In the end, our research partnership has really shown that sometimes one plus one indeed is three.

My gratitude goes to the Swedish foundation for strategic research (both through the ARTES programme and the SAVE project) for funding this work. A special thanks to Professor Hans Hansson who, apart from being a good friend, is and has been highly devoted to these projects.

The cooperation of industry has been a vital part of this project and I would like to thank the staff at Volvo Construction Equipment Components AB, in Eskilstuna, Sweden, for the two successful weeks that we spent there. A special thanks to Nils-Erik Bånkestad for his support during the entire project. Furthermore, many thanks to Bengt Gunne at Upright Database Technology, Uppsala Sweden, for sharing his deep knowledge of embedded databases with us.

Life at the department would not be the same without my dear colleagues at the department, many thanks to you all. I would specially want to mention a few persons: Thanks to Jukka Mäki-Turja who, apart from being my personal "Salubrin-style" golf-teacher[1], is a really good friend of mine. We sure have had a few high moments through the years, not to mention a few good beers. Many thanks go to Thomas Nolte who has been a good friend and companion for many years now. After a few years of working in separate areas during our research, it sure felt great finishing up with a joint paper. I wish you the best of luck. Thanks also to Daniel Sundmark, Anders Pettersson, Jonas Neander, Ewa Hansen, Prof. Ivica Crnkovic, Harriet Ekwall, Monica Wasell and Malin Ekholm for many good laughs. I must also mention Kristian Sandström and Anders Wall (the old guard of Ph.D. students at the department) which were good inspirations when I started as a Ph.D. student.

Finally, tons of love goes to my wife Jenny and my daughter Liv for putting up with me for all these years. Your love and support is invaluable.

Dag Nyström, Västerås in October 2005

[1] Salubrin - a solution that relieve minor skin irritations such as sunburn and insect bites, and whose slogan is "It smells, it stings, but it helps"

Contents

List of publications

Publications included in the thesis

Dag Nyström, Aleksandra Tešanović, Christer Norström, Jörgen Hansson & Nils-Erik Bånkestad ***Data Management Issues in Vehicle Control Systems: a Case Study*** In proceedings of the 14th Euromicro Conference on Real-Time Systems, IEEE, Vienna, Austria, June 2002.

Dag Nyström, Aleksandra Tešanović, Mikael Nolin, Christer Norström & Jörgen Hansson ***COMET: A Component-Based Real-Time Database for Automotive Systems*** In proceedings of the Workshop on Software Engineering for Automotive Systems, Edinburgh, Scotland, May 2004.

Dag Nyström, Mikael Nolin, Aleksandra Tešanović, Christer Norström & Jörgen Hansson ***Database Pointers: Efficient and Predictable Data Access in Real-Time Control-Systems*** Article submitted for journal publication. Based upon two conference papers: (i)Database Pointers: a Predictable Way of Manipulating Hot Data in Hard Real-Time Systems In proceedings of the 9th International Conference on Real-Time and Embedded Computing systems and Applications, pages: 623 -634, Tainan, Taiwan, February 2003. (ii) Pessimistic Concurrency Control and Versioning to Support Database Pointers in Real-Time Databases In proceedings of the 16th Euromicro Conference on Real-Time Systems, Catania, Sicily, June 2004.

Dag Nyström, Mikael Nolin & Christer Norström ***Snapshots in Real-Time Database using Database Pointer Transactions*** In proceedings of the 11th IEEE International Conference on Real-Time and Embedded Computing Systems and Applications, Hong Kong, China, August 2005

Thomas Nolte & Dag Nyström ***Introducing Substitution-Queries in Distributed Real-Time Database Management Systems*** In Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation, Catania, Sicily, September 2005

Publications by the author, not included in the thesis

Journal publications

Aleksandra Tešanović, Dag Nyström, Jörgen Hansson & Christer Norström ***Aspects and Components in Real-Time System Development: Towards Reconfigurable and Reusable Software*** In Journal of Embedded Computing, Cambridge International Science Publishing, October. 2004

Conference publications

Aleksandra Tešanović, Dag Nyström, Jörgen Hansson & Christer Norström ***Towards Aspectual Component-Based Development of Real-Time Systems*** In proceedings of the 9th International Conference on Real-Time and Embedded Computing systems and Applications, pages: 279 -298, Tainan, Taiwan, February 2003.

Dag Nyström, Aleksandra Tešanović, Christer Norström & Jörgen Hansson ***Database Pointers: a Predictable Way of Manipulating Hot Data in Hard Real-Time Systems*** In proceedings of the 9th International Conference on Real-Time and Embedded Computing systems and Applications, pages: 279 -298, Tainan, Taiwan, February 2003.

Dag Nyström, Mikael Nolin, Aleksandra Tešanović, Christer Norström & Jörgen Hansson ***Pessimistic Concurrency Control and Versioning to Support Database Pointers in Real-Time Databases*** In proceedings of the 16th Euromicro Conference on Real-Time Systems, Catania, Sicily, June 2004.

Workshop publications

Aleksandra Tešanović, Jörgen Hansson, Dag Nyström & Christer Norström ***Aspect-Level WCET Analyzer: a Tool for Automated WCET Analysis of a Real-Time Software Composed using Aspects and Components*** In proceedings of 3rd International Workshop on Worst-Case Execution Time Analysis, Porto, Portugal, July 2003.

Aleksandra Tešanović, Dag Nyström, Jörgen Hansson & Christer Norström ***Integrating Symbolic Worst-Case Execution Time Analysis with Aspect-Oriented System Development*** In proceedings of OOPSLA 2002 Workshop on Tools for Aspect-Oriented Software Development, Seattle, USA, November 2002.

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

Technical reports

Aleksandra Tešanović, Dag Nyström, Jörgen Hansson & Christer Norström ***Embedded Databases for Embedded Real-Time Systems: A Component-based Approach*** MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-43/2002-1-SE Mälardalen Real-Time Research Centre, Mälardalen University, January 2002.

Dag Nyström, Aleksandra Tešanović, Christer Norström & Jörgen Hansson ***The COMET BaseLine Database Management System*** MRTC Report 98/2003, ISSN 1404-3041 ISRN MDH-MRTC-98/2003-1-SE Mälardalen Real-Time Research Centre, Mälardalen University, April 2003.

Theses

Dag Nyström ***COMET: A Component-Based Real-Time Database for Vehicle Control-Systems*** Licentiate Thesis, Department of Computer Science and Engineering, Mälardalen University, May 2003

Part I

Thesis

Chapter 1

Introduction

Most functionality in modern vehicles, such as cars, is in one way or another controlled by computers. Mechanical systems are increasingly replaced by software residing in the *vehicle control-system*. As these control-systems grow larger and larger, they become increasingly more complex to develop and maintain. To handle this growing complexity, high-level software paradigms are introduced, e.g., Autosar [1], a joint project within the automotive industry.

Hand in hand with the increasing amount of control functionality demanded comes the increasing amount of information, or data, that these systems must manage and thereby the increasing complexity of the software required. In today's systems, this data is handled in an ad hoc fashion, using internal data structures, i.e., shared variables.

The current data management approach is becoming increasingly inadequate as systems become more complex and a need for data management on a higher level of abstraction has emerged. This problem is not unique to vehicular systems but is, and has been, apparent in many types of computer systems which involves managing information, such as banking and airline reservation systems, word-processors, and e-mail/calendar applications. The solution to data management for many of these systems has been to adopt a high-level data management approach through the use of a *database management system* (DBMS). DBMSs are used to structure data into databases, and can provide a powerful means of access to data in a controlled fashion. This thesis investigates how a DBMS could be introduced into vehicle control-systems.

Performing this integration is not possible without taking into consideration the specific requirements of such a control-system. As this thesis will

show, using a general-purpose off the shelf DBMS is not feasible. Reasons why general purpose DBMSs are not applicable in such systems include; (i) Traditional DBMSs are not suitable, since the onboard computers (or *electronic control units - ECUs*) are too resource-constrained with respect to memory capacity. (ii) A traditional DBMS is intended to maximize the average throughput of data queries, while a DBMS for use in a vehicle control-system must favor guaranteeing predictability of data accesses, such as worst case-response. A DBMS used in a vehicle control-system must both be small enough to fit in a small environment, and have real-time capabilities in order to provide a time-deterministic behavior, i.e., a *real-time database management system* (RTDBMS) must be used.

1.1 Problem formulation

As current DBMSs and RTDBMSs do not suit the particular requirements of a vehicle control-system, new concepts for data management are needed. These concepts must take the two most important aspects of embedded real-time systems into consideration, namely; (i) Predictability with respect to timing. The RTDBMS must guarantee that data in the database can always be accessed within a given time. (ii) Resource efficiency with respect to memory and CPU utilization. Since embedded systems most often have limited hardware resources this issue has a high priority.

This thesis investigates how a data management concept for embedded real-time systems, such as vehicle control-systems, could be designed and implemented. In particular it will investigate the following questions:

- What are the specific data management requirements for vehicle control-systems, and how do these influence the characteristics of a suitable data management concept? These requirements are used as a basis for the development of the data management concepts performed in this work.
- How can information in an RTDBMS be accessed in a resource-efficient and deterministic way? Introducing an RTDBMS generally increases the computational overhead compared with using internal data structures, therefore resource efficiency and determinism are crucial.
- How can critical and non-critical data accesses be mixed without introducing unpredictable blocking and transaction abortions? More and more non-critical functionality, such as multimedia services, passenger-comfort, and navigation systems, are introduced in vehicles. Accessing

non-critical data must not affect the accessing of critical data, i.e., have a negative effect on the determinism of the system.

- How does the integration of an RTDBMS affect the data distribution in distributed embedded real-time systems? This question is a consequence of that most modern vehicle-control-systems are distributed among multiple hardware nodes. For data management to be useful, it must thus also involve data distribution.

1.2 The research work and method

The research presented in this thesis has been performed within a joint project between Mälardalen University, Sweden (Ph.D. student Dag Nyström, Professor Christer Norström and Associate Professor Mikael Nolin) and Linköping University, Sweden (Ph.D. student Aleksandra Tesanovic and Doctor Jörgen Hansson). The research has been centered around a jointly developed experimental real-time database management platform, denoted COMET RTDBMS.

The research work within the project has been divided into two separate sections, namely, (i) data management issues for embedded real-time control-systems (Mälardalen University), and (ii) reconfigurability and real-time system development (Linköping University). The work and contributions presented in this thesis describe the data management issues and concepts developed within the COMET project.

One central aspect of the research performed within the project has been close interaction with industry. This has enabled us to work on the solution of research problems relevant to industrial systems in practice.

The research performed on data management for embedded real-time systems has been conducted in three phases:

1. **Investigation of the current state of the art and practices.** In particular, the current state of arts and practices in the area of embedded and real-time database management systems has been studied [2]. An extensive survey of commercially available embedded DBMSs was performed and a handful of systems were investigated on the basis of a number of criteria. In addition to this survey, a second survey was performed, this time, of experimental RTDBMSs, developed in academia. The latter survey also provided documentation of basic database management systems theory.

Furthermore, industrial data management requirements, assessed in a case study performed at Volvo Construction Equipment Components AB [3] (Paper A), provided a good foundation for continued research.

2. **Formulation of initial data management concepts.** Initial ideas of how a RTDBMS could be integrated into a control-system were formulated (Partly presented in [3] (Paper A) and further developed and concretized in [4] (Paper B)).

3. **Development and evaluation of data management concepts.** In order to make an RTDBMS suitable for use in an embedded real-time control-system, several new data management concepts were needed, by; (i) developing a number of data management concepts involving efficient data access (Paper C), concurrency-control algorithms (Paper C & D), and data distribution mechanisms (Paper E). In some cases, proof-of-concept using model-checking was performed to validate the behavior of the proposed algorithms. (ii) implementation of the concepts, some in our experimental platform, denoted COMET RTDBMS, and some as test implementations. (iii) evaluation of the concepts implemented by means of simulation

1.3 Thesis outline

The thesis is outlined as follows:

Part I:

Chapter 2 presents the areas of general purpose, real-time, and embedded database management systems and puts these into perspective with respect to vehicle control-systems. Related work in relevant areas is also presented.

Chapter 3 discusses the research contributions of this work, and presents a brief overview of the papers included.

Chapter 4 concludes the thesis and proposes some possible future directions for continued work.

Part II:

Chapter 5-9 presents Papers A to E, which in detail discuss the different contributions of this work.

Chapter 2

Background and related work

This chapter aims at giving a background to the work performed in this thesis, as well as an account of some related work both in academia and in practice. The chapter gives a brief overview of the areas of database management systems and real-time database management systems and describes these areas from the view point of vehicle control-systems.

2.1 Vehicle control-systems

The functionality of modern cars is to a large extent controlled by computer-based control-systems, so called *vehicle control-systems*. Vehicle control-systems handle a wide range of different functionality ranging from safety-critical control such as engine control, transmission control, chassis and brake control (anti-lock brake control and anti spin control) to diagnostics (warning and errors detected during run-time), driver comfort control such as climate control, and multimedia services.

A vehicle control-system typically consists of a number of onboard computer nodes, designated *electronic control units* (ECUs), which are interconnected via a network, e.g., the controller area network (CAN) [5].

Each ECU contains a number of *tasks*, which are executables used to manage some functionality. Examples of typical tasks in vehicle control-systems are (i) I/O-tasks which are used to communicate with the system under con-

trol (e.g., reading sensor-values or updating actuators), (ii) control-tasks which are used to take control decisions, and (iii) management-tasks which are used to perform more administrative activities such as the system diagnostics and logging of system events.

Figure 2.1 shows a set of tasks that together manage a control-functionality. To the left, two I/O-tasks sense the environment (i.e., read hardware sensors) to determine the current state of the vehicle, such as the current vehicle speed and the position of the accelerator pedal. These values are then passed to the control-task which contains the control algorithms. In this case, the control-task might use the current speed, and the position of the gas pedal to calculate the amount of fuel to be injected into the engine. To the right in Figure 2.1, another I/O-task is responsible for feeding this value to the fuel injector.

Figure 2.2 shows as an example, an I/O-task that is responsible for reading a temperature-sensor and writing it to a shared variable. The shared variable is protected by a semaphore to ensure its integrity, i.e., so that it cannot be read while being updated.

Since vehicle control-systems (and most other control-systems) control constantly changing environments, the control-system must support *real-time* properties, i.e., it must be a *real-time system*.

A real-time system is a system in which the time at which the output is produced is significant. This is usually because the input corresponds to some event in the physical world, and the output must relate to this event. The lag (delay) from the input time to output time must be sufficiently small for acceptable timeliness[1].

One means of enforcing timeliness in a system is to introduce the notion of deadlines. In Figure 2.1 an *end-to-end deadline* is given, that is; the result must be produced within a given time. Real-time systems can be divided into, at least, two categories, namely:

- **Hard real-time systems** in which a missed deadline result in a system failure, potentially involving the loss of human lives. These systems are often referred to as *safety-critical systems*. Many vehicle control functions have hard real-time requirements.

- **Soft real-time systems** in which the missing of deadlines merely degrades the quality of service of the system. For vehicle control-systems, management tasks could be viewed as soft real-time (even though not always treated as such in practice).

[1] Definition partly taken from the Oxford Dictionary of Computing.

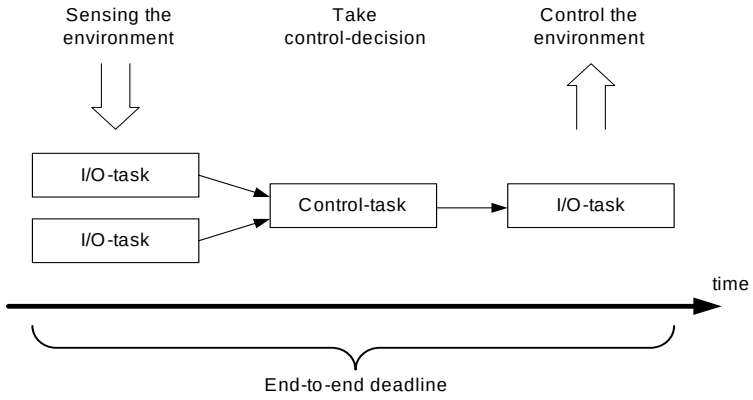


Figure 2.1: A set of tasks executing a functionality

Many real-time systems, including many vehicle control-systems, are *off-line-scheduled* and *periodic*. In an offline-scheduled system, all tasks and their timing properties are known at design time. A scheduling-tool is used to create a schedule consisting of the start times of all tasks, such that all timing requirements are satisfied. This schedule is then executed cyclically, hence the system is periodic. For some safety-critical domains, such as the automotive and avionic domains, offline-scheduled systems are considered safer than on-line scheduled systems since a proof-of-concept can be established beforehand.

A central requirement of real-time systems is predictability (or determinism), i.e., that the system must be constructed so that its behavior is always predictable. This implies that it must always (at least for hard real-time systems) be possible to calculate the system's worst case timing behavior. If all timing requirements are fulfilled under worst-case conditions, the system meets the real-time requirements.

Vehicle control-systems, apart from being real-time systems, are also *embedded systems*, which means that the computerized control-system resides in a larger system (in this case a vehicle). Characteristic of most (not all) embedded systems is that they are *resource-constrained* which means that hardware resources are limited, often with respect to both memory capacity and processor performance. The reasons for using resource-constrained systems in vehicles are many, but the main motivation is to reduce hardware costs.

Based on this, we can conclude that data management for vehicle control-

```
//Global data
struct {
    int oilPressure;
    int oilTemperature;
    int waterPressure;
    int waterTemperature;
} engine_t;

struct engine_t engine;
semaphore engine_semaphore;
...
//End global data

1 TASK OilTempReader(void){
2   int s;
3   while(1){
4     s=read_sensor();
5     wait(engine_semaphore);
6     engine.oilTemperature=s;
7     signal(engine_semaphore);
8     waitForNextPeriod();
    }
  }
```

Figure 2.2: I/O-task using a shared variable

systems must adhere to at least;

- **Predictability with respect to timing.** It should always be possible to access and manipulate data within a certain time.
- **Minimizing resource overhead.** The data management mechanisms used in a vehicle control-system must be sufficiently efficient, with respect to both memory requirement and CPU usage to be suitable in a resource-constrained environment.

2.2 Database management systems

Database systems have been developed to be a fundamental part of most larger computer applications and systems which involve the management of information. Database systems are used in software systems that handle massive amounts of data, such as hotel and ticket reservation systems, libraries, and e-commerce applications. Such database systems are referred to as *enterprise databases*. Database systems are also used as an integrated part of smaller applications, i.e., *application-embedded database systems*, such as word processors, e-mail clients, and personal organizers. Finally, *device-embedded database systems* which are normally simply referred to as embedded databases, are databases embedded in hardware products, such as mobile phones, toys, and vehicles

Even though the requirements these database systems must satisfy might differ greatly, as we will elaborate on in this chapter, database systems have some basic functionality and requirements in common.

As can be seen in Figure 2.3, users and user applications use *database queries* to retrieve information from a database. Database queries are questions presented to the database using some query language. One common database query language is the *structured query language* (SQL) [6], which provides a powerful high-level language that allows relevant information to be easily extracted from large quantities of data.

Figure 2.3 shows the architecture of a database system. From the figure it can be seen that a database system is divided into two subsystems, namely:

1. the *database* which is the collection of data that is stored in the database system, and
2. the *database management system* (DBMS) which is the software component responsible for handling the integrity of and access to the database.

Figure 2.3 shows that the DBMS is further divided into three levels, which each provides a specific service. The three levels are:

The external level which provides services to the user of the database system. The external level transforms requests formulated using a query language into *execution plans* that are understandable by the conceptual level.

The conceptual level which provides services to the external level. The main service provided is the processing of execution plans, in which the execution plan is transformed into individual read and write requests for data records (tuples) in the database. The conceptual level also ensures that the concurrency of multiple database transactions is monitored.

The physical (internal) level is responsible for organizing the physical storage of data elements in the database. The physical level provides service to the conceptual level by enabling it to perform index-lookups to locate data in the database, and by giving it access to data elements.

2.2.1 Database transactions

A database transaction is a set of database queries and operations bundled into one atomic unit of work. This implies that a database transaction is either executed entirely or not all. To clearly identify the start and completion of a database transaction, the following execution sequence is often used for transactions:

Begin of transaction	This marks the beginning of a database transaction.
Reads and writes	The execution of the database transaction could ultimately be broken down to a number of reads and writes to the database.
Commit|Rollback	This marks the end of a transaction, a **Commit** or a **Rollback** indicating its successful or unsuccessful completion.

It is only in the last step (the commit or rollback step) that any updates performed by the transaction are made visible to other transactions (in case of a commit).

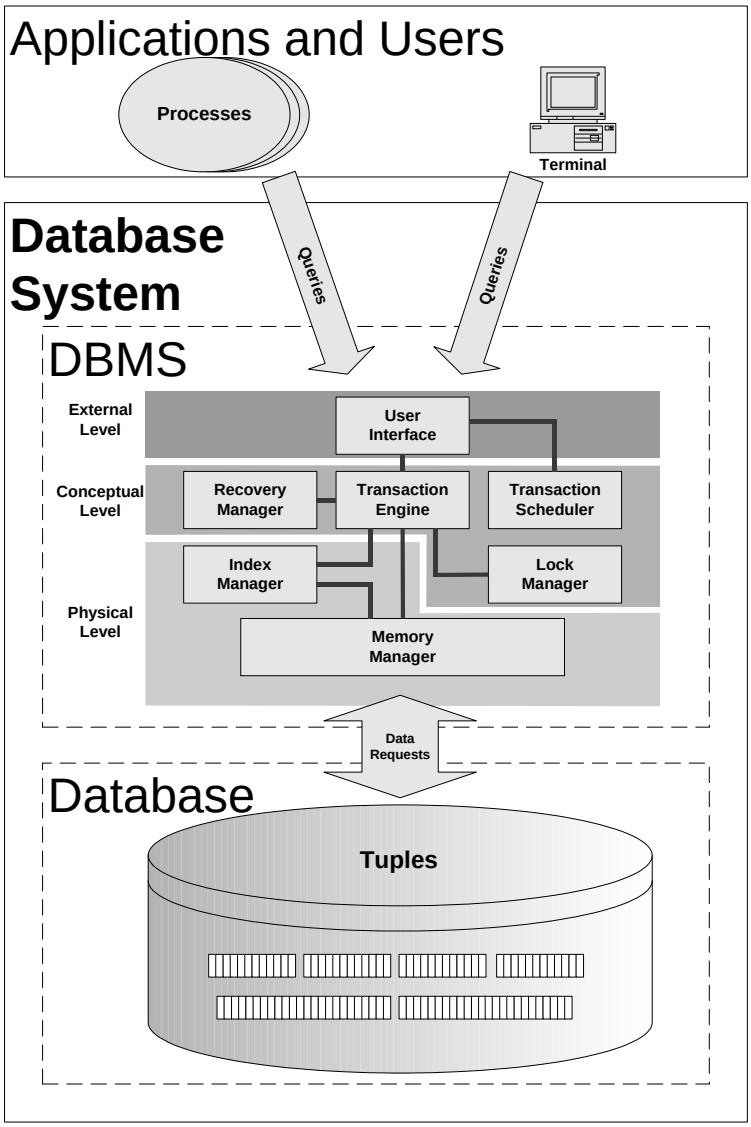


Figure 2.3: The architecture of a database system

To ensure the integrity of a database transaction it must have four properties. These properties, which are referred to as the *ACID properties*, are the following:

- **Atomicity:** A database transaction is indivisible, either it is run to completion or it is not run at all.
- **Consistency:** It must not violate logical constraints enforced by the system. For example, a bank transaction must follow the law of conservation of money. This means that after a money transfer between two accounts, the sum of the money in the accounts must be unchanged.
- **Isolation:** A database transaction must not interfere with any other concurrently executing database transaction. This is also referred to as *serialization* of database transactions, i.e., it should always be possible to establish a logical order of transaction execution of any set of database transactions.
- **Durability:** A database transaction is, once committed, written permanently into the database.

Given that a transaction is formulated such that it does not violate any consistency criteria in the database, database transactions executed in a non-concurrent system automatically have the ACID properties[2]. Since no other transactions can concurrently access the database, atomicity and isolation and durability are maintained.

The real difficulty in maintaining the ACID properties arises when multiple transactions are allowed to execute concurrently. Consider for example that transaction T_a updates a value x in the database and before it commits, transaction T_b reads x, the isolation property is violated since t_b was able to read *uncommitted*, i.e., not yet committed, data. Circumstances such as this are usually referred to as *database transaction conflicts* .

To be able to avoid or handle transaction conflicts, some form of *concurrency-control* is normally used. Concurrency-control mechanisms restrict the way transactions access and update the database. In general, concurrency-control can be obtained using four different approaches, namely *pessimistic concurrency-control*, *optimistic concurrency-control*, *multiversion concurrency-control*, and *concurrency-control with timestamps*. The following sections

[2]This assuming that a recovery system handles any abnormal terminations of database transactions

	Read-lock	**Write-lock**
Read-lock	Compatible	Incompatible
Write-lock	Incompatible	Incompatible

Table 2.1: Compatibility matrix for database locks

will briefly cover the first three of these approaches. Concurrency-control with timestamps will not be discussed further in this thesis.

Pessimistic concurrency-control

One of the most common approaches to the handling of database transaction conflicts is pessimistic concurrency-control (PCC). PCC uses the concept of *database locks* to lock data elements in the database to enforce serialization.

The most common pessimistic algorithm is the two-phase-locking (2PL) algorithm proposed in 1976 by Eswaran et al. [7]. This algorithm consists, as the name indicates, of two phases. In the first phase all locks are collected, no reading or writing to data can be performed before a lock has been obtained. When all locks are collected and the updates have been performed, the locks are released in the second phase.

Most pessimistic algorithms use, at least, two kinds of locks, *read-* and *write-locks*. The former permit data to be accessed in a read-only fashion, and the latter permit the manipulation of data. Table 2.1 shows the compatibility of these locks.

One problem with PCC is that it can cause deadlocks, in the same way as the uncontrolled use of semaphores. This problem can be overcome by either using a deadlock detection mechanism or by further restricting the concurrency-control algorithm. *Conservative 2PL*, for example, requires the transaction to obtain all its locks before the transaction can be executed by pre-declaring the locks needed [8].

Optimistic concurrency-control

Optimistic concurrency-control (OCC) was first proposed by Kung and Robinson [9] in 1981. This strategy takes advantage of the fact that conflicts in general are rather rare. The basic idea is to read and update data disregarding possible conflicts. All updates are, however, performed on temporary data. At

	Read-lock	**Write-lock**	**Certify-lock**
Read-lock	Compatible	Compatible	Incompatible
Write-lock	Compatible	Incompatible	Incompatible
Certify-lock	Incompatible	Incompatible	Incompatible

Table 2.2: Compatibility matrix for database locks used in 2V-2PL

commit-time a conflict detection is performed and the data is only written permanently to the database if no conflict has been detected. The conflict detection (verification phase) and the update phase must however be atomic, implying some form of locking mechanism. Since these two phases take much less time than a complete database transaction, locks that involve multiple data, or the whole database, can be used. Since it is an optimistic approach, performance degrades when congestion in the system increases.

Multiversion concurrency-control

Multiversion concurrency-control handles the serialization by maintaining multiple versions of data. Multiversioning permits transactions, that otherwise should have been aborted or blocked, to read old versions of data elements.

An obvious disadvantage with multiversioning algorithms is the added memory overhead. In a worst-case scenario, an unbounded amount of memory would have to be used to store all versions. However, one common solution to this is to restrict the number of versions of data elements. This of course limits the concurrency of the transactions.

A well known multiversion algorithm is the *two-version two-phase locking* (2V2PL) algorithm proposed by Lai and Wilkinson in 1984 [10]. This algorithm eliminates read-write conflicts by permitting read- and write-locks to be compatible with each other (see Table 2.2). This is achieved by maintaining 2 versions of each data element. Prior to commit, all write-locks must be converted to *certify-locks*. Certify-locks are locks that are incompatible with all other locks, and are only used during the commit-phase of a transaction.

2.3 Real-time database management systems

A *real-time database management system* (RTDBMS) is by definition a database system, since it has queries, transactions, concurrency-control and com-

mit-protocols. However, the correctness of a transaction in an RTDBMS is not only determined by logical correctness, but also by temporal correctness. Temporal properties for real-time transactions might be completion deadlines, start times, and periodic invocations [11].

In this section, an overview of the different issues that must be considered for real-time database management systems is presented. A recent paper by Ramamritham, Son and Dipippo provides an excellent categorization of these issues [12]. These categories are presented here with vehicle control-systems in mind. The categories pointed out in [12] are the following:

1. Data, database transactions and system characteristics
2. Scheduling and database transaction processing
3. Distribution
4. Quality of service and quality of data[3]

2.3.1 Real-time data

Just as for most software requirements, real-time requirements originate from the surrounding environment. For real-time systems, we recall from Section 2.1, that these requirements mostly include timing, such as meeting deadlines. For real-time database management systems, real-time requirements are extended to also include factors such as *data freshness* and *temporal consistency*.

By data freshness we mean the degree to which data to be read has been produced as recently as possible. For data derived from other data, the freshness of all the data used in the derivation is also important. For control-systems in general, thus also vehicle control-systems, data freshness is of the utmost importance. This is true since control-systems often reside in environments that are rapidly changing. To achieve a high degree of data freshness in RTDBMSs it sometimes must be required, in a controlled fashion, to relax the ACID properties of real-time database transactions [14].

Temporal consistency and data freshness are closely related. There are two types of temporal consistencies [15], namely *absolute consistency* and *relative consistency*. Absolute consistency is more stringent than data freshness, since

[3] Even though quality of service including feedback control are important issues for RTDBMS, it is beyond the scope of this thesis and will therefore be omitted. However, work on implementing feedback control in the COMET RTDBMS has been performed [13].

it specifies the maximum acceptable age of a data element by using an absolute validity interval. Data elements older than their absolute validity interval are as incorrect as a logically incorrect data element in any DBMS. Relative consistency is important when deriving new data from existing data elements Relative consistency stipulates the maximum permitted difference in age of the data elements used to derive the new data. Real-time database management systems might explicitly incorporate temporal consistency by integrating validity intervals and current ages of temporal data in the database schema. On the other hand, for offline-scheduled or periodic real-time systems, temporal properties can be analyzed pre run-time by making sure that the schedule is constructed such that all temporal validity intervals are maintained.

For real-time systems, and especially resource-constrained embedded systems, finding a good balance between temporal consistency (thus also data freshness) and computational resource usage is important. A higher degree of freshness requires more frequently executed database transactions. The general rule of thumb, also used in control-theory, is that updating database transactions should have half (or less) the periodicity of the absolute validity interval (the sampling theorem) [15].

2.3.2 Database transaction processing

Just as for general purpose database management systems, access to the database is obtained through database transactions. However, the main difference between database transactions and real-time database transactions is that the main goal of a DBMS is to generate as high average throughput of database transactions, while for an RTDBMS all database transactions must be executed as predictably as possible in order to keep as many deadlines as possible. Just as for real-time systems, real-time database transactions can be divided into soft- and hard real-time. For RTDBMS's these two classes impose fundamental differences.

Hard database transactions

Hard database transactions must meet their timing constraints at all costs, in particular their deadline. To enforce an absolutely predictable execution of a database transaction most often implies limiting its behavior. Typical limitations used for hard database transactions are:

Only allow precompiled queries Database transactions can be either formulated and compiled pre run-time (precompiled) or be freely formulated

during run-time (ad hoc queries). This limitation defines a set of the data elements in the database that possibly could be accessed by the database transaction. Thus, an upper bound of the number of accessible data elements is also set. The result of this is that the worst case response time for hard database transactions can be analyzed off-line [3].

Only allow periodic and sporadic transactions Similarly, as for hard real-time tasks, an upper bound on the rate of invocations of the database transactions must be established to permit calculation of the system utilization.

However, even for precompiled queries, calculating accurate worst case execution times is difficult since so many factors such as database size (locating a particular data element generally takes more time in a large database), and blocking caused by other database transactions accessing common data, see Section 2.3.2 influence their execution. This means that a database transaction, predicted to meet its deadline, might in fact miss it. To cope with such situations, *milestone monitoring* and *contingency plans* might be used. A milestone can be seen as a deadline for a part of the database transaction. If a milestone is passed and it is clear that the database transaction will not make its deadline, a contingency plan can be activated and the database transaction is aborted. Milestones and contingency plans are discussed further in [16]. The contingency plan might in its turn activate a database transaction that uses *imprecise computing* [17] to return a fairly good result based on a faster algorithm. Take for example, a query that calculates the average value of hundreds of values. If an adequate amount of values has been calculated at the time of the deadline, the result could be considered imprecise but meaningful and it is therefore returned to the client in any case.

For vehicle-control systems that utilize a real-time database, hard database transactions would be used primarily in critical I/O and vehicle control-tasks. Figure 2.4 shows the I/O task from Figure 2.2 now extended to use a database query to access its data element (located in the relation `engine`). It is worth noting that the calls to the semaphore used in Figure 2.2 have been removed since data accesses are now handled by the RTDBMS through its concurrency-control mechanisms. The following steps are needed to execute the query (apart from query parsing since the query is precompiled):

- Execution of relational operations, i.e., the `update` and the `restrict` (to locate the correct tuple) operations.

- Obtaining and releasing the appropriate database locks (if pessimistic concurrency-control is used).
- Copying of data to the database transaction's local working buffer.
- Committing the changes in the database.

It is easy to see that all these activities together make up a larger overhead as compared with obtaining a semaphore and directly writing the data to the shared variable. Given the needs of, and requirements on, hard I/O- and control-tasks in a vehicular system, the means of accessing the database as efficiently and predictably as using a shared variable is needed to successfully integrate a RTDBMS into such a system. However, some databases, especially commercially available embedded databases have mechanisms to shortcut precompiled queries, by keeping direct pointers to data elements in the database, see Section 2.6. These mechanisms share one weakness though, they limit the way a database can be reorganized during run-time.

Soft database transactions

Soft real-time database transactions may miss their deadlines. However, the general goal is to keep a quality level as high as possible. The quality of soft database transactions can be classified in two categories, namely *quality of timeliness* and *quality of DBMS service*. Quality of timeliness can be defined in a number of ways, such as keeping as many deadlines as possible and completing as many highly prioritized database transactions as possible. It is not crucial that all timing requirements are met, but the system suffers from degraded performance for each missed deadline. Considerable research has been performed in the area of soft real-time database transaction management and an extensive overview is available in [12].

However, equally important as timeliness is the quality of the DBMS service provided by soft database transactions. Since keeping deadlines is not a non-negotiable factor, the level of expressiveness can be raised. Strict periodicity or sporadicity are no longer required, and, above all, ad hoc queries can be permitted.

For vehicle control-systems using an RTDBMS, quite a number of activities can be assigned to soft management tasks and thus to soft real-time database transactions. Some of these activities include [3]:

- Logging of system states and faults

engine

subsystem	temperature	pressure
hydraulics	42	27
oil	**103**	10
cooling water	82	3

```
1 TASK OilTempReader(void){
2   int s;
3   DB_transaction t;
4   while(1){
5     s=read_sensor();
6     t=beginOfTransaction(...)
7     query(t,"UPDATE engine SET temperature=%d
                WHERE subsystem=oil;",s);
8     commit(t);
9     waitForNextPeriod();
    }
  }
```

Note: For readability reasons the query is formulated as an ad hoc query, in reality it would probably be precompiled.

Figure 2.4: I/O-task using a relational query

- Parts of the instrument-board not displaying critical information
- Interactive menu-systems

In current vehicle control-systems which do not utilize an RTDBMS, these activities are often implemented as hard real-time tasks. However, by using an RTDBMS to handle a clean and predictable separation of hard and soft data accesses, these activities can be moved to soft real-time. A study performed at Volvo Construction Equipment Components AB showed that significant savings in CPU utilization can be achieved by moving non-critical activities from hard tasks to soft tasks [18, 19].

Mixing hard and soft database transactions

One problematic issue which must be addressed is how to mix hard and soft database transactions in the same system. Unfortunately, this problem cannot be solved in the same way as when scheduling real-time systems, for example by executing soft tasks as background service or by using some server algorithm, such as the constant bandwidth server [20]. For database transactions, the problem mainly involves the sharing of resources, i.e., the data elements that are common to both soft and hard database transactions.

Consider a vehicle control-system in which a large number of critical hard real-time tasks containing hard database transactions are executing at high frequencies together with soft sporadic tasks which consist of soft database transactions with execution-times long in comparison with those of hard database transactions. Furthermore, soft and hard database transactions share common data elements. If no special considerations are given to the mixing of these two types of database transactions, the following two possible scenarios are to be expected:

Unacceptable blocking of hard database transactions. This might be the case if a standard pessimistic lock-based concurrency-control is used. As soon as a soft database transaction obtains a database lock for a data element, it efficiently blocks any hard database transaction that attempts to access that particular data element. Given that the life-span of a soft database transaction might be arbitrarily long, this blocking might jeopardize the safety of the vehicle.

Unacceptable abortion of soft database transactions. This scenario is probable if an optimistic concurrency-control approach is used. During the life-span of a soft database transaction it might access a great number

> of data elements. The longer the life-span of the soft database transaction, the more likely hard database transactions will, successfully, access and update data elements used by the soft database transaction. Thus, when the soft database transaction reaches its validation phase, it will be aborted due to data conflicts. It is a well-known fact that database transactions with a long life-span are affected negatively by shorter transactions [21]

One approach to increasing the throughput of database transactions with a long life-span is the concept of SAGAs [22] which divides a database transaction into a number of sub-transactions. If the database transaction is divided so that the individual sub-transactions could be interleaved arbitrarily with other database transactions, a logical consistency could still be maintained. However, SAGAs are not atomic but should be executed as a unit.

Real-time concurrency-control

To achieve a deterministic execution of database transactions in RTDBMSs, specialized real-time concurrency-control algorithms must be used. These algorithms attempt to establish a good balance between missed deadlines and temporally inconsistent database transactions. Much effort has been invested in finding algorithms suited to real-time systems. However, typically, no single real-time concurrency-control approach seems to suit the needs of all types of real-time systems. Real-time concurrency-control algorithms must therefore be specialized to suit the particular requirements of the real-time system for which they are intended.

Real-time concurrency-control algorithms are often based on some general concurrency-control strategy, i.e., the pessimistic, optimistic, or multiversion strategy.

One of the best known pessimistic real-time concurrency-control approaches is the *two phase locking with high priority abort* (2PL-HP) [23]. 2PL-HP is, as the name suggests, a prioritized variant of the two phase locking approaches. (see Section 2.2.1). In 2PL-HP, transactions with lower priority are always aborted in favor of those with a higher priority in case of a transaction conflict. One potential problem with this approach to vehicle control-systems is the potential starvation of lower prioritized transactions [24].

A different pessimistic real-time concurrency-control algorithm is the *read/-write priority ceiling protocol* (RWPCP) [25] in which 2PL is combined with the *priority ceiling protocol* (PCP) [26]. RWPCP supplements the exclusive

locks used in PCP with shared locks. RWPCP is intended for hard periodic real-time transactions.

A well known optimistic real-time concurrency-control is the *optimistic concurrency-control with broadcast channel* (OCC-BC) [27] in which a validating transaction informs all conflicting transactions so that they can abort immediately. The advantage of this enhancement of the classic optimistic concurrency-control algorithm [9] is that the commitment of all transactions that reach the validation phase is guaranteed. OCC-BC does not, however, take priorities into consideration and can therefore cause the abortion of critical activities.

Priorities are taken into consideration in the *optimistic concurrency-control with priority wait* (OPT-WAIT) algorithm [28] by incorporating a *priority wait* mechanism that forces transactions to wait for the completion of transactions with higher priority if they are in conflict with each other. A further improvement of OPT-WAIT is the *WAIT-50* algorithm [28] which works in exactly the same way as OPT_WAIT except that it must wait only if *at least* 50% of the conflicting transactions have higher priority than the transaction itself. The WAIT-50 algorithm has been shown to have excellent performance [12].

For real-time databases, a number of variants of these algorithms that suit such databases better have been proposed [29, 30]. Song and Liu showed that OCC algorithms performed poorly with respect to temporal consistency in real-time systems that consist of periodic activities [31], while they performed very well in systems in which database transactions had random parameters, e.g., event-driven systems. However, it has been shown that the strategies and the actual implementation of the locking and abortion algorithms significantly determine the performance of OCC [21]. Furthermore, it has been established that due to the conservative approach taken in pessimistic concurrency-control, it is more suited for resource constrained real-time systems (such as vehicle control-systems) than optimistic approaches [32].

2.4 Embedded database management systems

Embedded databases are becoming more and more common in embedded systems, especially in the telecommunication area and in PDAs. A number of commercial products have been available for a number of years, and the market is growing. For traditional enterprise database systems, the main objectives are throughput, flexibility, scalability, functionality etc. Physical size, resource usage, and processor usage are not of equal importance since hardware is now

relatively inexpensive. In embedded systems these issues are much more important, therefore the main issues for embedded database systems are quite different and can be summarized as [33, 34, 35]:

- **Minimizing the memory footprint.** The memory demands of an embedded system are most often, mainly for economical reasons, kept as low as possible. A typical footprint for an embedded database is within the range of some kilobytes to a 2 megabytes.

- **Minimizing the CPU usage.** In an embedded system, the database management system and the application are most often run on the same processor. This requires the database process to allocate strictly, minimum CPU bandwidth to leave as much CPU capacity as possible for the application.

- **Support for multiple operating systems.** In an enterprise database system, the DBMS is typically run on a dedicated server using a normal operating system. The clients, desktop computers, other servers, or even embedded systems, connect to the server using a network connection. Because a database most often runs on the same hardware as the application in an embedded system, and because embedded systems often use specialized operating systems, the database system must support these operating systems.

- **High availability.** In contrast to a traditional database system, most embedded database systems do not have a system administrator present during run-time. Therefore, an embedded database must be able to run independently.

2.5 Real-time and embedded databases in practice

This section aims at providing an overview of the major existing real-time and embedded database platforms, both those developed as research platforms and those commercially available. For a more detailed survey on this subject, see [2]

2.6 Commercial embedded platforms: a survey

2.6.1 Databases investigated

In this survey we have selected a handful of systems that together give a general picture of the types of products on the market. This list of systems is not to be considered complete in any way, but represents a cross-section of the commercially available platforms.

- Pervasive.SQL by Pervasive Software Inc. This database has three different versions for embedded systems: Pervasive.SQL for smart-card, Pervasive.SQL for mobile systems, and Pervasive.SQL for embedded systems. All three versions integrate well with each other and also with their non-embedded versions of Pervasive.SQL. Their classification of systems and the fact that they have, compared with most embedded databases, very small memory requirements were reasons for investigating this database [36][4].
- Polyhedra by Enea AB. This database was selected for three reasons, (i) it is claimed to be a real-time database, (ii) it is a main memory database, and (iii) it is an active DBMS [37].
- RDM by Birdstep Technology, Inc. In the same way as Polyhedra, RDM claims to be a real-time database. It is however fundamentally different from the Polyhedra system, by, for example, being an embedded library and, thus, does not incorporate the client/server model [38].
- Berkeley DB by Sleepycat Software Inc. This database, which also is implemented as a library, was selected for the survey because it is distributed as open source and therefore interesting from a research point of view [39].
- TimesTen by TimesTen Performance Software. This relational database is, like the Polyhedra system, a main memory real-time database system [40].

2.6.2 Survey criteria

There are a number of criteria that could be used to evaluate and classify an embedded database management system. In this survey, a set of criteria have

[4] Unfortunately, the DBMS suite for embedded, mobile and smart-card systems is no longer commercially available.

been selected that can be related to the system's feasibility as a data manager for a vehicle control-system. The selected criteria are:

DBMS model There are basically two different DBMS models supported. The first model is the client/server model, in which the database server can be considered to be an application running separately from the real-time application. The server is typically accessed either using inter-process communication (IPC) or some network. The second model is to compile a DBMS-library together with the application into one executable system. When a task wants to access the database it only needs a function call to make the request. Both approaches have advantages and disadvantages. A client/server model can be more easily used in a distributed environment, using standardized interfaces. However, performing all database requests using IPC is expensive due to frequent context-switches between application and database server. For embedded DBMS libraries, database requests are much more efficient for requests from the local application. Distributed access needs to be handled by the control-application since embedded libraries usually lack standardized interfaces [34].

Data model The data model concerns the logical structuring of data. The most common model is the relational model in which data is organized in tables consisting of columns and rows. Databases implementing relational data model are referred to as relational databases (RDBMS). One advantage with the relational model is its ability to handle complex logical queries that can be used to extract a specific selection of data, i.e. a view. However, one disadvantage with the relational model is a fairly high computational overhead, even when few data are to be retrieved. This can be a serious problem for databases used in time-critical applications such as vehicle control-systems. The object-oriented data model is a different kind of data model, which can be viewed as an extension of the semantics of an object-oriented language. This model allows class instances to be created and then stored in the database. A third data model, which has evolved from both the relational and the object-oriented model, incorporates objects in relations, and these are thus designated object-relational databases (ORDBMS).

Memory requirements The memory requirement of the database is an important issue in the case of embedded databases residing in environments with small memory resources. For mass-produced embedded computer

systems, e.g., such as ECU's, minimizing hardware is usually a significant factor in reducing product costs. There are two interesting properties to consider for embedded databases, first of all the memory footprint size, which is the size of the database with no data content. Secondly, data overhead, i.e., the number of bytes required to store a data element apart from the size of the data element itself, is of interest.

Concurrency-control Depending on which concurrency-control approach used, different systems might be more or less suitable for use in a vehicle control-system, see Section 2.3.2.

2.6.3 DBMS model and memory requirements

As can be seen from Table 2.4, the client/server platforms are typically larger than the platforms implemented as an embedded library, with the Pervasive-.SQL suite as an exception. This discrepancy in size might be the result of the amount of included functionality rather than a direct consequence of the client/server arhictecture. Adding standardized interfaces, such as ODBC, OLE-DB and JDBC directly in the DBMS kernel increases the memory requirements. A difference in the focus of target applications might instead explain this. Typically, Client/Server applications target larger embedded applications, such as traffic light control, and mobile-phone base-stations, while embedded libraries focus more on smaller embedded (control-) systems.

In the case of the Pervasive.SQL systems, these more specifically target smaller applications, such as mobile devices and smart-card applications. It might seem odd that a smart-card application uses a client/server architecture for the DBMS, but this has natural causes. When a smart-card, incorporating Pervasive.SQL for smart-cards, is inserted into its host, e.g., an automatic teller machine, a java applet containing the application is downloaded from the card to the host. The host then executes the applet, which calls the DBMS executing on the smart-card. This approach, along with appropriate encryption, ensures the security of the data, since it cannot be accessed directly by the host.

2.6.4 Data model

As shown in Table 2.4, all systems in the survey except Berkeley DB are relational. For a vehicle control-system, a relational model might be useful for

Criteria		DBMS platforms						
		P.SQL for Mobile Sys.	*P.SQL for Embedded Sys.*	*P.SQL for Smart-Cards*	*Polyhedra*	*RDM*	*Berkeley*	*TimesTen*
DBMS Model	Client/Server	x	x	x	x			x
	Library					x	x	
Data Model	Relational	x	x	x	x	x		x
	Object-Relational				(x)			
	Other						x	
Memory Req.[a]	Footprint	50-400k	50kb	8kb	1.5-2Mb	400-500kb	175kb	5Mb
Concurrency-Control	Pessimistic	x	x	x		x	x	x
	Optimistic				x			
	None						x	

Table 2.4: Characteristics of the surveyed commercial embedded database management systems

[a]The values given in the table are the ”footprint”-values made available by the vendors when the survey [2] was written

dynamic management tasks used for diagnostics and driver information. However, due to its computational overhead, it might be considered too slow for time-critical vehicle-control.

Most systems have ways to "shortcut" access to data and therefore bypass the index lookup routine. Pervasive.SQL, for example, can access data using the Btrieve transactional engine that bypasses the relational engine. The Btrieve engine accepts commands such as `getNextTuple` and `getFirst-Tuple` etc. The RDM system uses a different approach; In many real-time systems data items are accessed in a predefined order (for example in a control-system in which some sensor values are read from the database and the result is written back to the database). Fast accesses can be achieved by inserting shortcuts between data elements so that they are directly linked in the order they are accessed,

However, one problem with pointing directly to data elements from the application is that reorganizing the database becomes much more complex since a large number of pointers will become stale during the reorganization.

The Polyhedra DBMS system is fundamentally different from the other relational systems in this survey, because of its active behavior. This is achieved through active queries. An active query is formulated and launched in the same way as a normal query, but the query remains in the database until explicitly aborted. If a change in the database would alter the result of the query, the query is automatically executed again. For overload and predictability reasons, a minimum interarrival time between executions can be specified for active queries.

As mentioned above, Berkeley DB is the only non-relational system in this survey. It uses a key-data relationship instead. A data element is associated with a key. There are three ways to search for data, from a key, from a part of a key or as a sequential search. The data can be of any volume and of virtually any structure. Since the keys are plain ASCII strings, the data can contain other keys so that complex relations can be built up. In fact it would be possible to implement a relational engine on top of the Berkeley DB database.

2.6.5 Concurrency-control

All databases except the Polyhedra DBMS use pessimistic concurrency-control (see Table 2.4). For a vehicle-control system, this might introduce problems with respect to blocking and possibly the abortion of database transactions. This is because of the mix of hard high-frequency control-tasks executing on high priorities, and soft low-frequency (or event-driven) management tasks.

Using plain pessimistic concurrency-control in such a system might introduce unacceptable blocking of database transactions in hard tasks, or a high degree of abortions (and possible starvation) of database transactions executing in the soft tasks.

Pervasive.SQL has two kinds of database transactions: exclusive and concurrent. An exclusive database transaction locks a complete database file for the entire duration of the database transaction, allowing only concurrent non-transactional clients to perform read-only operations on the file. A concurrent database transaction, however, uses read and write locks with much finer granularity, e.g., page or single data locks.

The Berkeley DB has three configurations: (i) The non-concurrent configuration allows only one thread at a time to access the database, removing the need for concurrency-control. (ii) The concurrent version allows concurrent readers and writers to access the database simultaneously. (iii) The concurrent transactional version allows for full transactional functionality with concurrency-control, such as fine-grain locking and database atomicity.

TimesTen also has a "no concurrency" option, in which only a single process can access the database. This option is suitable for non-preemptive systems. In addition, TimesTen supports two lock sizes: data-store level and record level locking.

2.7 Real-time research platforms: a survey

A number of databases can be classified as pure real-time databases. However these databases are the results of research projects and are not yet on the commercial market. We have selected a number of real-time database systems and compared them with respect to the following criteria:

- **Real-time properties.** This criterion enables us to discuss real-time aspects of the systems and how they are implemented.

- **Distribution.** The criterion enables us to consider different aspects with respect to distributing the database.

- **Database transaction workload characteristics.** The criterion enables us to discuss how the database transaction system is implemented.

2.7.1 Platforms investigated

The systems selected in this survey are representative of the more recent real-time database platforms developed. The systems surveyed all have some support for hard (or in one case firm) real-time database transactions, there are other systems intended primarily for soft real-time systems [41, 42, 43]. For a more detailed survey which includes these systems also, the reader is referred to [2].

- The Distributed active, real-time Database System (DeeDS) [44] supports both hard and soft real-time database transactions. DeeDS is developed at the University of Skövde, Sweden. It uses extended ECA [16] rules to achieve an active behavior with real-time properties. It is designed to take advantage of a multiprocessor environment.
- The RODAIN system [45] developed at the University of Helsinki, Finland, is a firm real-time database system primarily intended for telecommunication. It is designed for high degree of availability and fault-tolerance. It is tailored to fit the characteristics of telecommunication transactions identified as short queries and updates, and long massive updates.
- The ARTS-RTDB system [46] developed at the University of Virginia, Charlottesville, US, supports both soft and hard real-time database transactions. It uses imprecise computing to ensure timeliness of database transactions. It is built on top of the ARTS real-time operating system [47].

2.7.2 DeeDS

DeeDS [44] is a main-memory database intended for both hard and soft real-time systems. It is built for the OSE delta real-time operating system, developed by ENEA DATA [48], a distributed hard real-time operating system designed for both uniprocessor and multiprocessor systems. The DeeDS system consists of two parts, one part that handles non-critical system services and one that handles critical system services. All critical system services are executed on a dedicated processor to simplify overhead cost and increase the concurrency in the system.

DeeDS supports sporadic (event-triggered) and periodic database transactions. There are two classes of database transactions: critical (hard database transactions) and non-critical (soft database transactions). Milestone monitoring and contingency plans are used to ensure that a deadline for a critical

database transaction is kept. The timing requirements for a database transaction are passed to the system as a parameterized value function, reducing the computational cost and the storage requirements of the system.

Database transactions are scheduled online in two steps. First, a sufficient schedule is produced. This schedule meets the minimum requirements, for example all hard deadlines and 90 percent of the soft deadlines are met. Secondly, the event monitor's worst-case execution time is subtracted from the remaining allowed time for the scheduler during this time slot, and this time is used to refine and optimize the database transaction schedule [49].

The DeeDS [44] system has less strict criteria for global consistency (i.e., consistency among the nodes in a distributed system), in order to enforce timeliness. They guarantee that each node has a local consistency, while the distributed database might be inconsistent due to different views of the system at the different nodes. This approach might be suitable for systems that mostly rely on data gathered locally, but occasionally use data imported from other subsystems. Consider an engine that has all engine data, e.g., ignition control, engine speed and fuel injection, stored in a local node of a distributed database. The timeliness of these local data items is essential in order to run the engine. To further increase the efficiency of the engine, remotely gathered data, such as data from the transmission with less critical timing requirements can be distributed to the local node.

2.7.3 RODAIN

The RODAIN system is a firm real-time database system intended for telecommunication applications. A firm database transaction has a level of criticality between the criticality levels of hard and soft database transactions. When a firm database transaction misses its deadline, its result is rendered useless, and the database transaction must be discarded. The telecommunication environment is a dynamic and complex environment that deals with both temporal data and non-temporal data. Thus, RODAIN supports both soft and firm real-time database transactions [45] The developers believe, however, that hard real-time databases will not be used in telecommunication in the near future because generally, they are too expensive to develop or use to suit the market [45].

One of the key issues for RODAIN is availability. Therefore a fault-tolerant system is necessary. The RODAIN system is designed to have two nodes, thus giving full database replication. This is especially important since the database, as the DeeDS platform, is a main-memory system. It uses a primary and a mirror node. The primary database sends logs to the mirror database which in

turn acknowledges all calls. The communication between the nodes is assumed to be reliable and have a bounded message transfer delay. A watchdog monitor detects any failing subsystem or database transaction and can instantly take actions to handle the situation, e.g., swap the primary node and the mirror node in case of a failure or restart a database transaction. The failed node always recovers as a mirror node and loads the database image from permanent storage.

Five different areas for databases in telecommunication can be identified [50]:

1. Retrieval of persistent customer data.
2. Modifications of persistent customer data.
3. Authorization of customers, e.g., PIN codes.
4. Sequential log writing.
5. Mass-calling and tele-voting. This can be performed in large blocks.

From these five areas, three different types of database transactions can be derived [51]: short simple queries, simple updates, and long massive updates. Short simple queries are used when retrieving customer data and authorizing customers. Simple updates are used when modifying customer data and writing logs. Long massive updates are used when tele-voting and mass-calling is performed.

The concurrency-control in RODAIN supports different kinds of serialization protocols. One protocol is the τ -serialization in which a database transaction may be allowed to read old data provided the update is not older than a predefined age [45].

Apart from the deadline, an isolation level can also be assigned to a database transaction. A database transaction executing on a low isolation level accepts that database transactions running on a higher isolation level are accessing locked objects. However, it cannot access objects belonging to a database transaction with a higher degree of isolation.

The RODAIN system supports data distribution. The database on the nodes can be anything between fully replicated and completely dispersed. The assumption is that only a few requests in telecommunications need access to more than one database node and that the request distribution of requests to the databases in the system can be arranged to be almost uniform [45].

2.7.4 ARTS-RTDB

The relational database ARTS-RTDB [46] incorporates imprecise computing. ARTS-RTDB is superimposed on top of the distributed real-time operating system ARTS, developed by Carnegie Mellon University [47]. ARTS schedules tasks according to a value function varying with time, which specifies their criticality. It supports both soft and hard real-time tasks.

In ARTS-RTDB, the INSERT, DELETE, UPDATE and SELECT operations have been identified as the most critical data operations. Efforts have therefore been made to optimize the system to increase the efficiency of these four operations. According to [46] many real-time applications use these operations during run-time almost exclusively.

ARTS-RTDB uses 2PL-HP and to avoid the costly process of rolling back the aborted database transaction, all data writing is performed on copies. At the point of commit, the database transaction asks the lock-manager if a commit can be allowed. If this is the case, the database transaction invokes a subsystem that writes all the data into the database.

ARTS-RTDB has been extended to support distribution. The database nodes use a shared file which contains information that binds all relations to the server responsible for a particular relation. Since the file is shared between the nodes, it is to be treated as a shared resource and must therefore be accessed using a semaphore. It is believed that if relations are uniformly distributed between the nodes and if no hot-spot relations exist, there will be an increase in performance. A hot-spot relation is a relation used by many database transactions and such a relation can lead to a performance bottleneck in the system.

2.8 Observations

This chapter has presented work performed within a number of different research areas related to data management for vehicle systems. Furthermore, two surveys have been presented, covering database platforms, both research-oriented and commercially available.

One problem when investigating commercial products as well as research platforms, is to obtain unbiased and detailed technical information. With respect to commercial platforms, the information has been collected mainly from so called, ”white-papers” and correspondence with technical staff at the respective company. This material is not acceptable as convincing from a research point of view. Information regarding the research platforms investigated

has mainly been collected from scientific publications which have been peer-reviewed and therefore can be said to have scientific validity. Unfortunately, most of these publications cover individual aspects and concepts developed within the research group behind the research platform. Only a few ”position-style” papers specifically describing the research platform are available[5].

Nevertheless, two conclusions relevant to data management for vehicle systems can be drawn:

- **The commercial products do not have real-time properties strict enough to be used in vehicle control-systems.** No real-time guarantees are given for any of the products. Even though some of the products offer means of directly accessing data (Similar to the concept proposed in Paper C), the use of general purpose concurrency-control algorithms still adds unpredictability.
- **The research platforms are not intended to be used in embedded environments.** The aim of these platforms has not been to be resource efficient, the DeeDS platform requires two processors, one for hard and one for soft real-time transactions. Similarly, the RODAIN platform uses a mirror-node for replication. In the case of ARTS-RTDB, the use of a centralized database-file, protected by a distributed semaphore is not a satisfactory solution.

[5] In retrospect, this also applies to the papers produced within the COMET project.

Chapter 3

Thesis contributions

3.1 Research contributions

In the problem formulation (Section 1.1), a number of research questions was formulated (Restated below for convenience).

- What are the specific data management requirements for vehicle control-systems, and how do these influence the characteristics of a suitable data management concept?
- How can information in an RTDBMS be accessed in a resource-efficient and deterministic way?
- How can critical and non-critical data accesses be mixed without introducing unpredictable blocking and transaction abortions?
- How does the integration of an RTDBMS affect the data distribution in distributed embedded real-time systems?

The work presented in this thesis proposes data management concepts that identify and target these problems. The specific research contributions of this thesis are:

- A case-study investigating data management requirements for an industrial vehicle control-system. Knowledge of true data management requirements is necessary when developing concepts and algorithms to be used in such systems.

- A high-level data management concept, denoted COMET, intended for embedded real-time systems, such as vehicle control-systems. COMET consists of a reconfigurable RTDBMS and a number of analysis and configuration tools.
- A resource-efficient and predictable data access method for RTDBMSs, denoted database pointers. This access method, which points out individual data elements, is needed since limited computational power in the hardware requires that accessing critical data must be performed as efficient as possible.
- Two concurrency-control algorithms, denoted 2V-DBP and 2V-DBP-SN-AP, which enables data in the database to be shared between critical hard real-time tasks and non-critical soft real-time tasks. As more and more non-critical functionality is introduced in the systems, longer and more dynamic non-critical queries are being introduced. Even though these queries in themselves are not critical they will influence the critical queries, since they might access common data. Maintaining determinism for hard real-time queries, while still allowing soft real-time queries to be executed without being aborted or starved is important.
- A data distribution mechanism for distributed embedded real-time systems, such as vehicle control-systems. The concept introduces the notion of subscription- and substitution-queries which can be used by service- and diagnostics-tools to stimulate and monitor a system during run-time.

These contributions are presented in the papers appended to the thesis (Chapter 5 to Chapter 9). A short summary of each of the main ideas of the papers are presented below.

3.2 Paper A

Dag Nyström, Aleksandra Tešanović, Christer Norström, Jörgen Hansson & Nils-Erik Bånkestad ***Data Management Issues in Vehicle Control Systems: a Case Study*** *In proceedings of the 14th Euromicro Conference on Real-Time Systems, IEEE, Vienna, Austria, June 2002.*

Paper A presents a case-study performed at Volvo Construction Equipment Components AB (Volvo CE), at Eskilstuna, Sweden. This case-study assesses the requirements of the data in their vehicle control-system during 2002, and

their practice of data management. The paper suggests how the system could be redesigned to incorporate an RTDBMS. Suggestions on how this RTDBMS could be designed are also presented.

The paper concludes that for the two ECUs investigated, one instrumental ECU for an articulated hauler and one vehicle ECU for a wheel loader, data currently is scattered in the system in different data storages. These data storages are implemented using internal data structures. Some of these structures are so complex that index mechanisms are used to locate data elements.

A common property of these systems is that they have an offline scheduled non-preemptive task model so that all data conflicts are avoided, i.e., no critical sections or semaphores are used since mutual exclusion is upheld by the non-preemptive task model. The paper generalizes the problem by sketching how the RTDBMS could be extended to support a preemptive task model.

Dag Nyström's contributions: The data management issues (jointly with Aleksandra Tešanović), database requirements, and database integration issues.

3.3 Paper B

Dag Nyström, Aleksandra Tešanović, Mikael Nolin, Christer Norström & Jörgen Hansson ***COMET: A Component-Based Real-Time Database for Automotive Systems*** *In proceedings of the Workshop on Software Engineering for Automotive Systems, Edinburgh, Scotland, May 2004.*

Paper B presents a data management concept, denoted *the COMET development suit* suitable for embedded real-time systems. The paper describes how COMET could be used to model the data, configure the RTDBMS, and analyze its behavior. The paper furthermore sketches how the COMET development suit is used to configure the RTDBMS based on the requirements of the system. Finally, the paper presents the highly configurable architecture of the COMET RTDBMS. The aim of the paper is to place the concepts developed and implemented within the COMET project in a context to demonstrate how they could be used together.

Dag Nyström's contributions: The basic architecture of the COMET RTDBMS, the database pointer concept, and the concurrency-control algorithm.

3.4 Paper C

Dag Nyström, Mikael Nolin, Aleksandra Tešanović, Christer Norström & Jörgen Hansson ***Database Pointers: Efficient and Predictable Data Access in Real-Time Control-Systems*** *Article submitted for journal publication. Based upon two conference papers: (i) "Database Pointers: a Predictable Way of Manipulating Hot Data in Hard Real-Time Systems" In proceedings of the 9th International Conference on Real-Time and Embedded Computing systems and Applications, pages: 623 -634, Tainan, Taiwan, February 2003. (ii) "Pessimistic Concurrency Control and Versioning to Support Database Pointers in Real-Time Databases" In proceedings of the 16th Euromicro Conference on Real-Time Systems, Catania, Sicily, June 2004.*

Paper C introduces the concept of database pointers, an efficient and predictable way of accessing data in real-time database management systems. The concept, which is designed to be used in conjunction with a traditional relational database model, permits the creation of variables that point directly to individual data elements in the database. This enables the application to access data within the database in much the same way as by using shared variables while still retaining all the benefits of using a database (such as allowing advanced concurrency-control and allowing reorganization of the database during run-time). In Figure 3.1, the example I/O-task from Figure 2.2 and Figure 2.4 is augmented with database pointer access. From the figure, it can be seen that the database pointer `ptr` is bound to the oil temperature in the initialization part of the task (line 4) using a high level SQL query, while data access in the control-loop is performed directly to the data element (line 7). By using a database pointer, the overhead resulting from traditional database queries is removed while still allowing data to be published in the database.

Paper C further presents a concurrency-control algorithm, denoted 2-Version Database Pointer Concurrency Control (2V-DBP). The concurrency algorithm allows hard database pointer transactions and soft relational transactions to be executed without blocking (or aborting) each other. 2V-DBP is suited to resource-constrained, safety critical, real-time systems that have a mix of hard real-time control applications and soft real-time management, maintenance, or user-interface applications. The correctness of 2V-DBP is shown by using model checking, and its performance is compared with a standard real-time concurrency-control algorithm.

Dag Nyström's contributions: Main contributor of the technical ideas. Main author of paper.

```
1 TASK OilTempReader(void){
2   int s;
3   DBPointer *ptr;
4   bind(&ptr,"SELECT temperature
              FROM engine WHERE
              subsystem=oil;");
5   while(1){
6     s=read_sensor();
7     write(ptr,s);
8     waitForNextPeriod();
    }
  }
```

Figure 3.1: I/O-task using a database pointer

3.5 Paper D

Dag Nyström, Mikael Nolin & Christer Norström ***Snapshots in Real-Time Database using Database Pointer Transactions*** *In proceedings of the 11th IEEE International Conference on Real-Time and Embedded Computing Systems and Applications, Hong Kong, China, August 2005*

Paper D presents an extension to the 2V-DBP algorithm from paper C. The new algorithm, denoted 2-version database concurrency-control with snapshots (2V-DBP-SNAP), enables hard tasks to access the database using transactions semantics, i.e., to access multiple data elements in an atomic fashion. 2V-DBP-SNAP introduces the concept of *snapshot sets* which permit sets of data elements to be accessed atomically, both by hard and soft tasks, using snapshot and soft transaction respectively.

This approach is useful for hard control-activities requiring a set of data elements to be viewed and controlled in one atomic operation.

Dag Nyström's contributions: Main contributor of the technical ideas. Main author of paper.

3.6 Paper E

Thomas Nolte & Dag Nyström ***Introducing Substitution-Queries in Distributed Real-Time Database Management Systems*** *In Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation, Catania, Sicily, September 2005*

Paper E proposes a data distribution mechanism suitable for embedded real-time control-systems. The mechanism uses a high level communication protocol to implement (i) distributed ad hoc queries, (ii) subscription queries, and (iii) substitution queries. These three mechanisms are developed with diagnostic- and service-tools in mind. These tools are connected to the vehicle through the network bus, and can be used to calibrate the system (e.g. engine calibration) or diagnose the current state of the system. Current commercially available systems used for calibration are limited in the sense that they only permit predefined sets of data to be read or updated. Using a RT-DBMS equipped with database pointers, any data elements in the database can be queried, subscribed upon, or substituted during run-time.

Dag Nyström's contributions: Joint contributor of the technical ideas, mainly database and data management contributions. Joint author of paper.

Chapter 4

Conclusions and future work

4.1 Conclusions

Integrating database technologies in vehicle control-systems is considered controversial by many. The general conception of database systems is that they are highly resource demanding both with respect to memory consumption and computational overhead. Furthermore, database systems are considered too non-deterministic to be used in vehicle control-systems. This is true for many general purpose database management systems. However, commercially embedded solutions exist today, with database engines as small as a few kilobytes. Furthermore, many years of research in real-time database management systems shows that these systems can be made deterministic.

This thesis has presented real-time database concepts developed specially with embedded hard real-time control-systems in mind. The focal point of this work has been the requirements of the system in which the database is intended to be integrated, and not primarily the database technology itself. Instead of asking the question ”How predictable can we make a real-time database system?”, this work was conducted asking the question ”Given the system requirements, how can we find the most suitable DBMS model?”. One concrete example of this is blocking. While most concurrency-control algorithms aim at minimizing blocking (or transaction abortions for that matter) in the system, the 2V-DBP concurrency-control algorithm presented in this thesis limits instead the permitted behavior of transactions so that blocking for hard transactions is eliminated, or in the case of 2V-DBP-SNAP, limited.

It is my belief that, due to the gradually increasing volume of information

managed by vehicle control-systems, they will evolve to become information-centric, just as so many other types of software systems. This will become even clearer when comfort electronics, telematics, navigation systems, and vehicle networking is introduced as integrated parts of vehicles.

4.2 Future work

The work in this thesis consists of a number of run-time concepts intended for application in vehicle control-systems. These concepts enable the introduction of a higher level data management. The COMET RTDBMS in its current form consists of a fairly mature library of run-time components that implements an RTDBMS that can be configured to suit the particular needs of different control-systems, as well as tools for its configuration and analysis. However, methods and theories on how to integrate and use COMET in a development setting are not yet available. By obtaining these, an *information-centric approach*, in which information itself is elevated to become a design entity, could be achieved.

One possible approach to achieving an information-centric approach is to integrate the RTDBMS as an integrated part of a *component-framework*, to introduce data management naturally in the design and development process. An argument for this approach is that the automotive industry is currently targeting component-based software engineering as a means of reducing complexity in their software systems. Creating this information-centric view, together with adequate tools, would ensure:

- means of obtaining a logical, rather than a physical, view of the data in the system. This logical view is made possible with the tools used to organize the data according to some data modeling paradigm, e.g., entity/relation-modeling [8]. This information-centric view enables data modeling and management to be detached from the modeling and management of the software architecture of the control-system. The advantage of this is that developers and architects of the control-system are given data management at a higher level of abstraction, but need no in-depth knowledge of database management. Even though logic modeling of data in itself is not new, the special requirements and characteristics of embedded real-time systems, such as temporal consistency [15] predictability [34] and resource management, must be considered in this information-centric view.

- that data can be made available where needed in a predictable, transparent, and efficient manner. By explicitly defining what data is required and provided by each component, mechanisms in the component framework obtain the necessary data before the execution of the component, and write back data produced by the component to the database.
- that data is transparently distributed between the electronic control units (ECUs) in a distributed control-system. We have already shown in this thesis that an RTDBMS can provide valuable support in this distribution (Paper E).
- that the data management requirements and characteristics of the different nodes are already captured at design-time, so that the RTDBMSs supporting the needed functionality and requirements can be configured.

Apart from the above, somewhat ambitious, goal, certain more threads remain loose after this thesis. Work worth consideration could be:

- **Incremental updates.** The 2V-DBP and the 2V-DBP-SNAP algorithms slightly relax the serialization criteria (see Paper C). This relaxation can, under some circumstances, cause what is called *lost updates*, i.e., updates not becoming visible since other ”later” updates have already been performed. This approach is feasible for updates that are not dependent on each other, e.g., sensor values that are periodically updated. However problems are encountered with data with incremental transactions, i.e., transactions that increase or decrease the value instead of writing absolute values. To ensure integrity, 2V-DBP(-SNAP) does not permit these data to be updated by both hard and soft transactions.
- **Verification of the serialization.** The formal verification performed on 2V-DBP in Paper C verifies some important properties of the versioning algorithm used in the algorithm. However, there has been no full formal verification of the properties of the relaxed serialization. It is my opinion that model checking is not an adequate approach for this verification, instead a more analytical approach might be needed.
- **Integration of concepts in an industrial setting.** To test fully the concepts presented in this thesis, a case study in which the COMET RTDBMS is integrated in an existing control-systems should be performed. Such a case-study would show the impact, in practice, of the proposed concepts.

Bibliography

[1] H. Heinecke, K.P. Schelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J.L. Mat, K. Nishikawa, and T. Scharnhorst. Automotive open system architecture - and industry-wide initiative to manage the complexity of emerging automotive e/e-architectures. Technical report, AUTOSAR Partnership, http://www.autosar.org/, 2004.

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

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

[4] Dag Nyström, Aleksandra Tešanović, Mikael Nolin, Christer Norström, and Jörgen Hansson. COMET: A Component-Based Real-Time Database for Automotive Systems. In *Proceedings of the Workshop on Software Engineering for Automotive Systems*, pages 1–8. The IEE, June 2004.

[5] Robert Bosch GmbH. BOSCH's Controller Area Network. http://-www.can.bosch.com/.

[6] Stephen Cannan and Gerhard Otten. *SQL - The Standard Handbook*. MacGraw-Hill International, 1993.

[7] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *The communications of the ACM*, 19(11):624–633, November 1976.

[8] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley Longman Publishing Co., 2000.

[9] H. T. Kung and J. T. Robinsson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[10] Ming-Yee Lai and W. Kevin Wilkinson. Distributed transaction management in jasmin. In Umeshwar Dayal, Gunter Schlageter, and Lim Huat Seng, editors, *Tenth International Conference on Very Large Data Bases, August 27-31, 1984, Singapore, Proceedings*, pages 466–470. Morgan Kaufmann, 1984.

[11] J. Stankovic, S. Son, and J. Hansson. Misconceptions About Real-Time Databases. *IEEE Computer*, 32(6):29–36, June 1999.

[12] Krithi Ramamritham, Sang H. Son, and Lisa Cingiser Dipippo. Real-Time Databases and Data Services. *Journal of Real-Time Systems*, 28(2/3):179–215, November/December 2004.

[13] A. Tešanović, M. Amirijoo, M. Björk, and J. Hansson. Empowering configurable qos management in real-time systems. In *Proceedings of the Fourth ACM SIG International Conference on Aspect-Oriented Software Development (AOSD'05)*, Chicago, USA, March 2005. ACM Press.

[14] Tei-Wei Kuo and Aloysius K. Mok. SSP: a Semantics-Based Protocol for Real-Time Data Access. In *Proceedings of 14th IEEE Real-Time Systems Symposium*, pages 76–86. IEEE Computer Society, December 1993.

[15] K. Ramamritham. Real-Time Databases. *International Journal of distributed and Parallel Databases*, 1(2):199–226, 1993.

[16] J. Eriksson. Real-Time and Active Databases: A Survey. In *Proceedings of the Second International Workshop on Active, Real-Time and Temporal Databases*, nr. 1553 in Lecture note series., pages 1–23. Springer-Verlag, December 1998.

[17] K.J. Lin, S. Natarajan, and J. W. S. Liu. Imprecise Results: utilizing partial results in real-time systems. In *Proceedings of the Real-Time Systems Symposium*. IEEE, 1987.

[18] Jukka Mäki-Turja, Kaj Hnninen, and Mikael Nolin. Efficient development of real-time systems using hybrid scheduling. In *International conference on Embedded Systems and Applications (ESA)*, 6 2005.

[19] Kaj Hänninen and Toni Riutta. Optimal Design. Master's thesis, Mälardalens Högskola, Dept of Computer Science and Engineering, 2003.

[20] Abeni and Buttazzo. Integrating multimedia applications in hard real-time systems. In *19th IEEE Real-Time Systems Symposium*, 1998.

[21] J. Huang, J.A. Stankovic, K. Ramamritham, and D.F. Towsley. Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 35–46. Morgan Kaufmann, September 1991.

[22] H. Garcia-Molina and K. Salem. SAGAS. In *Proceedings of the 1987 ACM SIGMOD Conference on Management of Data*, pages 249–259. ACM Press, 1987.

[23] R.K Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *ACM Transactions on Database Systems*, 17, September 1992.

[24] Dag Nyström, Mikael Nolin, Aleksandra Tešanović, Christer Norström, and Jörgen Hansson. Pessimistic Concurrency Control and Versioning to Support Database Pointers in Real-Time Databases. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 261–270. IEEE Computer Society, June 2004.

[25] L. Sha, R. Rajkumar, S. H. Son, and C.-H. Chang. A real-time locking protocol. *IEEE Transactions on Computers*, 40(7):793–800, September 1991.

[26] L. Sha, R. Rajkumar, and J.P Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9), September 1990.

[27] D. Menasce and T. Nakanishi. Optimistic versus pessimistic concurrency control mechanisms in database management systems. *Information Systems*, 7(1):13–27, 1982.

[28] J.R. Haritsa, M. Carey, and M. Livney. Dynamic real-time optimistic concurrency-control. In *Prooceedings of the 11th Real-Time Systems Symposium*. IEEE Computer Society Press, December 1990.

[29] P. S. Yu, K. Wu, K. Lin, and S. H. Son. On Real-Time Databases: Concurrency Control and Scheduling. *Proceedings of the IEEE*, 82(1):140–157, January 1994.

[30] F. Baothman, A. K. Sarje, and R. C. Joshi. On Optimistic Concurrency Control for RTDBS. In *Proceedings IEEE Region 10 International Conference on Global Connectivity in Energy, Computer, Communication and Control*, volume 2, pages 615–618. IEEE Computer Society, December 1998.

[31] X. Song and J. Liu. Maintaining Temporal Consistency: Pessimistic vs. Optimistic Concurrency control. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):786–796, October 1995.

[32] R. Agrawal, M. Carey, and M. Livney. Concurrency-control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems*, 12(4):609–654, December 1987.

[33] S. Ortiz. Embedded Databases Come Out of Hiding. *IEEE Computer*, 33(3):16–19, March 2000.

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

[35] Raima Corporation. Databases in Real-time and Embedded Systems. http://www.raimabenelux.com/, February 2001.

[36] Pervasive Software Inc. http://www.pervasive.com.

[37] Enea AB. http://www.enea.se.

[38] Birdstep Technology ASA. http://www.birdstep.com.

[39] Sleepycat Software Inc. http://www.sleepycat.com.

[40] TimesTen Performance Software. http://www.timesten.com.

[41] J. A. Stankovic, S. H. Son, and J. Liebeherr. *Real-Time Databases and Information Systems*, chapter BeeHive: Global Multimedia Database Support for Dependable, Real-Time Applications, pages 409–422. Kluwer Academic Publishers, 1997.

[42] B. Adelberg, B. Kao, and H. Garcia-Molina. Overview of the STanford Real-time Information Processor (STRIP). *SIGMOD Record*, 25(1):34–37, 1996.

[43] J. Zimmermann and A. P. Buchmann. REACH. In *N. Paton (ed): Active Rules in Database Systems, Springer-Verlag*, 1998.

[44] S. F. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring. DeeDS Towards a Distributed and Active Real-Time Database System. *ACM SIGMOD Record*, 25(1):38–40, 1996.

[45] J. Taina and K. Raatikainen. RODAIN: A Real-Time Object-Oriented Database System for Telecommunications. In *Proceedings of the workshop on Databases: active and real-time*, pages 10–14. ACM Press, November 1996.

[46] Y-K. Kim, M. R. Lehr, D. W. George, and S. H. Song. A Database Server for Distributed Real-Time Systems: Issues and Experiences. In *Proceedings of the Second IEEE Workshop on Parallel and Distributed Real-Time Systems*, pages 66–75. IEEE Computer Society, April 1994.

[47] H. Tokuda and C. Mercer. ARTS: A Distributed Real-Time Kernel. *ACM SIGOPS Operating Systems Review*, 23(3):29–53, July 1989.

[48] ENEA Data. OSE Real-time system. http://www.enea.se.

[49] J. Mellin, J. Hansson, and S. Andler, editors. *Real-Time Database Systems: Issues and Applications*, volume 396 of *The Kluwer International Series in Engineering And Computer Science*, chapter Refining Timing Constraints of Application in DeeDS. Kluwer Academic Publishers, 1997.

[50] K. Raatikainen and J. Taina. Design Issues in Database Systems for Telecommunication Services. In *Proceedings of IFIP-TC6 Working conference on Intelligent Networks*, pages 71–81. Kluwer Academic Publishers, August 1995.

[51] T. Niklander and K. Raatikainen. RODAIN: A Highly Available Real-Time Main-Memory Database System. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium*, pages 271–278, September 1998.

Index

Part II

Included papers

Chapter 5

Paper A: Data Management Issues in Vehicle Control Systems: a Case Study

Dag Nyström, Aleksandra Tešanović, Christer Norström, Jörgen Hansson, and Nils-Erik Bånkestad

In Proceedings of 14th EUROMICRO Conference on Real-Time Systems, Vienna, Austria, June 2002

Abstract

In this paper we present a case study of a class of embedded hard real-time control applications in the vehicular industry that, in addition to meeting transaction and task deadlines, emphasize data validity requirements. We elaborate on how a database could be integrated into the studied application and how the database management system (DBMS) could be designed to suit this particular class of systems.

5.1 Introduction

In the last ten years, control systems in vehicles have evolved from simple single processor systems to complex distributed systems. At the same time, the amount of information in these systems has increased dramatically and is predicted to increase further with 7-10% per year [1]. In a modern car there can be several hundreds of sensor values to keep track of. Ad hoc techniques that are normally used for storing and manipulating data objects as internal data structures in the application result in costly development with respect to design, implementation and verification of the system. Further, the system becomes hard to maintain and extend. Since the data is handled ad hoc, it is also difficult to maintain its temporal properties. Thus, the need for a uniform and efficient way to store and manipulate data is obvious. An embedded real-time database providing support for storage and manipulation of data would satisfy this need.

In this paper we study two different hard real-time systems developed at Volvo Construction Equipment Components AB, Sweden, with respect to data management. These systems are embedded into two different vehicles, an articulated hauler and a wheel loader. These are typical representative systems for this class of vehicular systems. Both systems consist of a number of nodes distributed over a control area network (CAN).

The system in the articulated hauler is responsible for I/O management and controlling of the vehicle. The system in the wheel loader is, in addition to controlling the vehicle, responsible for updating the driver display. We study structures of the systems and their data management requirements to find that today data management is implemented as multiple data storages scattered throughout the system. The systems are constructed out of a finite number of tasks. Each task in the system is equipped with a finite amount of input and output ports, through which inter-task communication is performed. Due to intense communication in both systems, several hundred ports are used. These ports are implemented as shared memory locations in main memory, scattering the data even more.

We study temporal properties of the data in the systems and conclude that the they could benefit from a real-time database (RTDB). Furthermore, we discuss how the current architecture could be redesigned to include a RTDB. The important feature of a RTDB in these systems is to guarantee temporal consistency and validity [2] rather than advanced transaction handling. In a typical vehicular system, nodes vary both in memory size and computation and, hence, there is a need for a scalable RTDB that can be tailored to suit different kinds

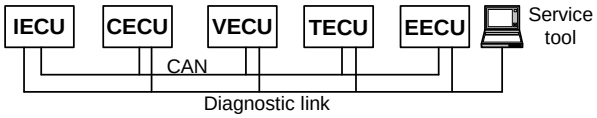


Figure 5.1: The overall architecture of the vehicle controlling system.

of systems. In this paper transactions refer to a number of reads and/or updates of data in a database. Thus, tasks can contain transactions.

The contribution of this paper is a detailed case-study of the two Volvo applications. Furthermore, we elaborate on how the existing hard real-time system could be transformed to incorporate a RTDB. This architectural transition would allow data in the system to be handled in a structured way. In this architecture, the database is placed between the application and the I/O management. We elaborate on why concurrency control, for this transformed system, is not necessarily needed for retaining the integrity of transactions. Moreover, we argue that a hard real-time database that would suit this system could be implemented using passive components only, i.e., a transaction is executed on the calling task's thread of execution. This implies that the worst-case transaction execution time is added to the worst-case execution time of the task, retaining a bounded execution time for all tasks.

In section 2 we study the existing vehicle systems and their data management requirements in detail. In section 3 we discuss: how the systems could be redesigned to use a RTDB, the implications for the application and the RTDB, and how existing real-time database platforms would suit the studied application. We conclude our work and present future challenges in section 4.

5.2 The Case Study

The vehicle control system consists of several subsystems called electronic control units (ECU), connected through two serial communication links: the fast CAN link and the slow diagnostic link, as shown in the figure 5.1. Both the CAN link and the diagnostic link are used for data exchange between different ECUs. Additionally, the diagnostic link is used by diagnostic (service) tools. The number of ECUs can vary depending on the way functionality is divided between ECUs for a particular type of vehicle. For example, the articulated hauler consists of five ECUs: instrumental, cabin, vehicle, transmission and

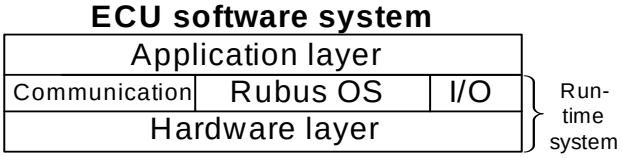


Figure 5.2: The structure of an ECU.

engine ECU, denoted IECU, CECU, VECU, TECU, and EECU, respectively. In contrast, the wheel loader control system consists of three ECUs, namely IECU, VECU, and EECU.

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

Both VECU and IECU software systems consist of two layers: a run-time system layer and an application layer (see figure 5.2). The run-time system layer on the lower level contains all hardware-related functionality. The higher level of the run-time system layer contains an operating system, a communication system, and an I/O manager. Every ECU uses the real-time operating system Rubus. The communication system handles transfer and reception of messages on different networks, e.g., CAN. The application is implemented on top of the run-time system layer. The focus of our case study is data management in the application layer. In the following section we briefly discuss the Rubus operating system. This is followed by sections where functionality and a structure of the application layer of both VECU and IECU, are discussed in more detail (in following sections we refer to the application layer of the VECU and IECU as the VECU (software) system and the IECU (software) system).

5.2.1 Rubus

Rubus is a real-time operating system designed to be used in systems with limited resources [3]. Rubus supports both off-line and on-line scheduling, and consists of two parts: (i) red part, which deals with hard real-time; and (ii) blue part, which deals with soft real-time.

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

5.2.2 VECU

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

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

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

The *sensor/actuator raw data* is a set of data elements that are either read from sensors or written to actuators. The data is stored in the same format as they are read/written. This data, together with the *sensor/actuator parameter data*, is used to derive the *sensor/actuator engineering data*, which can be

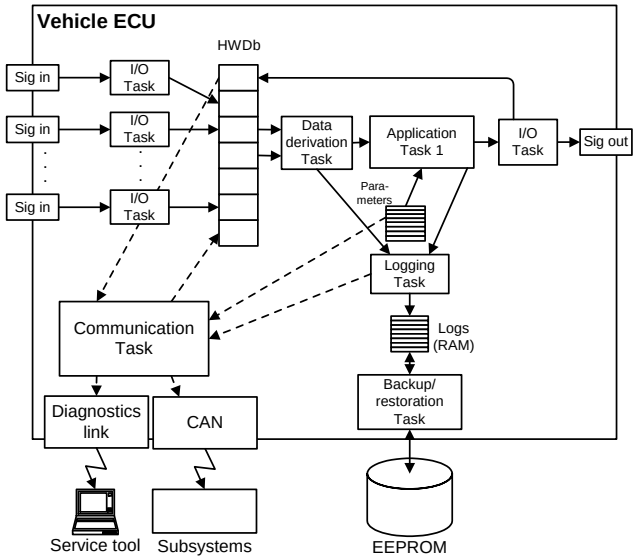


Figure 5.3: The original architecture of the VECU.

used by the application. The sensor/actuator parameter data contains reference information about how to convert raw data received from the sensors into engineering data. For example, consider a temperature sensor, which outputs the measured temperature as a voltage T_{volt}. This voltage needs to be converted to a temperature T using a reference value T_{ref}, e.g., $T = T_{volt} \cdot T_{ref}$.

In the current system, the sensor/actuator (raw and parameter) data are stored in a vector of data called a hardware database (HW Db), see figure 5.3. The HW Db is, despite its name, not a database but merely a memory structure. The engineering data is not stored at all in the system but is derived "on the fly" by the data derivation tasks. Apart from data collected from local sensors and the application, sensor and actuator data derived in other ECUs is stored in the HW Db. The distributed data is sent periodically over the CAN bus. From the application's point of view the locality of the data is transparent in the sense that it does not matter if the data is gathered locally or remotely.

Some of the data derived in the applications is of interest for statistical and maintenance purposes and therefore the data is logged (referred to as *logging data*) on permanent storage media, e.g., EEPROM. Most of the logging data

is cumulative, e.g., the vehicle's total running time. These logs are copied from EEPROM to RAM in the startup phase of the vehicle and are then kept in RAM during runtime, to finally be written back to EEPROM memory before shutdown. However, logs that are considered critical are copied to EEPROM memory immediately at an update, e.g., warnings. The *parameter data* is stored in a parameter area. There are two different types of parameters, permanent and changeable. The permanent parameters can never be changed and are set to fulfill certain regulations, e.g., pollution and environment regulations. The changeable parameters can be changed using a service tool.

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

5.2.3 IECU

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

The IECU differs from the VECU in several ways. Firstly, the data volume in the system is significantly higher since the IECU controls displays and, thus, works with a large amount of images and text information. Moreover, the data is scattered in the system and depending on its nature, stored in a number of different data structures as shown in figure 5.4. Similarly to the HW Db, data structures in the IECU are referred to as databases, e.g., image databases, menu databases and language databases. Since every text and image information in the system can be displayed in thirteen different languages, the interrelationships of data in different data storages are significant.

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

Data from the HW Db in the IECU is periodically pushed on to the CAN

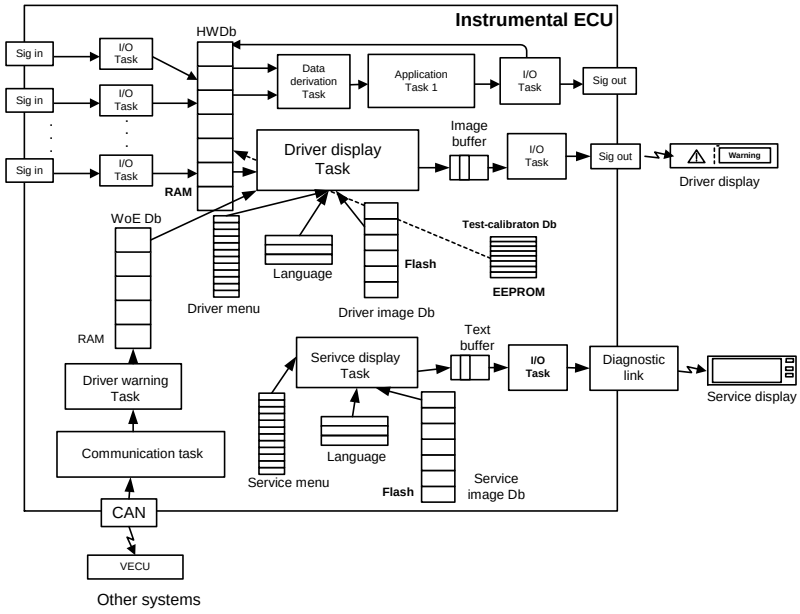


Figure 5.4: The original architecture of the IECU.

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

The driver display is updated as follows (see figure 5.4). The driver display task periodically scans the databases (HW Db, WoE Db, menu Db) to determine the information that needs to be displayed on the driver display. If any active WoE exists in the system, the driver display task reads the corresponding image, in the specified language, from the image database located in a persistent storage and then writes the retrieved image to the image buffer. The image is then read by the blue I/O task, which then updates the driver display with an image as many times as defined in the WoE Db. Similarly, the driver display task scans the HW Db and menu database. If the hardware database has been updated and this needs to be visualized on the driver display, or if data in the menu organization has been changed, the driver display task reads the corresponding image and writes it to the driver display as described previously. In case the service tool is plugged into the system, the service display task updates the service display in the same way as described for the driver display, but using its own menu organization and image database, buffer, and the corresponding blue I/O task.

5.2.4 Data Management Requirements

The table 5.1 gives an overview of data management characteristics in the VECU and IECU systems. The following symbols are used in the table: As can be seen from the table 5.1, all the data in both systems are scattered in groups of different flat data structures referred to as databases, e.g., HW Db, image Db, WoE Db and language Db. These databases are flat because data is structured mostly in vectors, and the databases only contain data with no support for DBMS functionality.

The nature of the systems put special requirements on data management (see table 5.1): (i) static memory allocation only, since dynamic memory allocation is not allowed due to the safety-critical aspect of the systems; (ii) small memory consumption, since production costs should be kept as low as possible; and (iii) diverse data accesses, since data can be stored in different storages, e.g., EEPROM, Flash, and RAM.

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

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

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

Table 5.1: Data management characteristics for the systems.

Most data, from different databases and even within the same database, is logically related. These relations are not intuitive, which makes the data hard to maintain for the designer and programmer as the software of the current system evolves. Raw values of sensor readings and actuator writings in the HW Db are transformed into engineering values by the data derivation task, as explained in section 5.2.2. The engineering values are not stored in any of the databases, rather they are placed in ports (shared memory) and given to application tasks when needed.

The period times of updating tasks ensure that data in both systems (VECU and IECU) are correct at all times with respect to absolute consistency. Furthermore, task scheduling, which is done off-line, enforces relative consistency of data by using an off-line scheduling tool. Thus, data in the system is temporally consistent (we denote this data property in the table as temporal validity). Exceptions are permanent data, e.g., images and text, which is not temporally constrained (see table 5.1).

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

Some data is associated with a range of valid values, and is kept logically consistent by tasks in the application (see table 5.1). The negative effect of enforcing logical consistency by the tasks is that programmers must ensure consistency of the task set with respect to logical constraints.

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

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

Data in the VECU is organized in two major data storages, RAM and Flash. Logs are stored in EEPROM and RAM (one vector of records), while 251

items structured in vectors are stored in the HW Db. Data in the IECU is scattered and interrelated throughout the system even more in comparison to the VECU (see table 5.1). For example, the menu database is related to the image database, which in turn is related to the language Db and the HW Db. Additionally, data structures in the IECU are fairly large. HW Db and WoE Db resides in RAM. HW Db contains 64 data items in one vector, while WoE Db consists of 425 data items structured as 106 records with four items each. The image Db and the language Db reside in Flash. All images can be found in 13 different languages, each occupying 10Kb of memory. The large volume of data in the image and language databases requires indexing. Indexing is today implemented separately in every database, and even every language in the language Db has separate indexing on data.

The main problems we have identified in existing data management can be summarized as follows:

- all data is scattered in the system in a variety of databases, each representing a specialized data store for a specific type of data;
- engineering values are not stored in any of the data stores, but are placed in ports, which enlarges maintenance complexity and makes adding of functionality in the system a difficult task;
- application tasks must communicate with different data stores to get the data they require, i.e., the application does not have a uniform access or view of the data;
- temporal and logical consistency of data is maintained by the tasks, increasing the level of complexity for programmers when maintaining a task set; and
- data from different databases exposes different properties and constraints, which complicates maintenance and modification of the systems.

5.3 Modeling the System to Support a RTDB

To be able to implement a database in the real-time system, the system needs to be redesigned to support a database. For the studied application, this could be done by separating I/O management from the application.

As mentioned in section 5.2.2 and shown in figure 5.3, the data flow goes from the I/O tasks, via the HW Db and application tasks to the I/O tasks to the

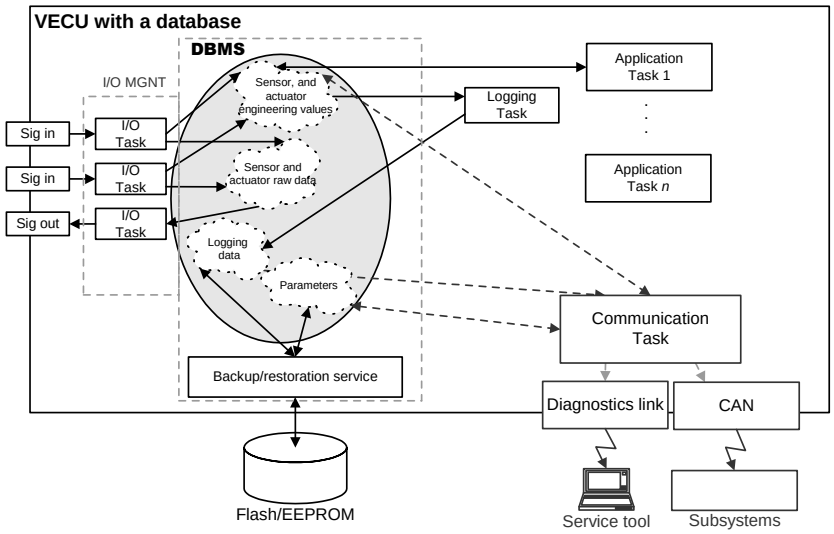


Figure 5.5: The new architecture of the VECU.

right, sending the values to the actuators. The transition of such a system could, at a high level, be performed in three steps. The first step is to separate all I/O tasks from the application. This can be viewed as "folding the architecture". By doing this an I/O management is formed that is separated from the control application. The second step is to place the real-time database between the I/O management and the control application as shown in figure 5.5. In the Volvo case, the HW Db is replaced by a RTDB which is designed using a passive library. The desired properties of this RTDB are described more in detail in section 5.3.1. The I/O tasks are modified to communicate with the database instead of the data derivation tasks. The application is, analogue to the I/O tasks, also modified to communicate with the database only. At this stage the database splits two domains, the I/O domain and the application domain. The last step is to collect additional data that might be scattered in the system into the database, e.g., parameter and logging data. The tasks that communicate with these data stores are, similar to the I/O and application tasks, modified to communicate with the database only. With this architecture we have separated the application from the I/O management and the I/O ports. The database could be viewed as a layer between the application and the operating system,

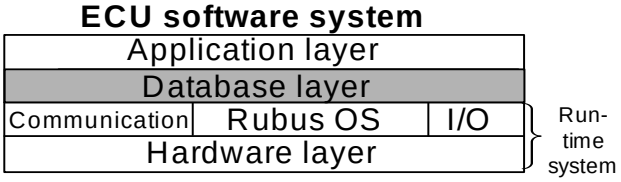


Figure 5.6: The structure of an ECU with an embedded database.

extending the real-time operating system functionality to embrace data management, see figure 5.6. All data in the system is furthermore collected in one database, satisfying the need for a uniform and efficient way to store data. Another important issue, shown in figure 5.5, is that both the raw sensor data and the engineering data, previously derived by the data derivation task, are now included in the database. The actual process of deriving the engineering values could be performed in multiple ways. The I/O tasks could be modified to embrace this functionality, so that they write both the raw value and the engineering value to the database. Another, perhaps more elegant, way of solving this is to use database rules, where a rule is triggered inside the database as soon as a data item is updated. This rule would execute the code that derive the engineering value.

5.3.1 Data Management Implications

When designing a system running on the described hardware, one of the main goals is to make it run with as small processor and memory footprint as possible. Traditionally, for data, this is achieved by using as small data structures as possible. A common misconception is that a database is a very large and complex application that will not fit into a system such as this [4]. However, there are, even commercial DBMSs that are as small as 8Kb, e.g., Pervasive.SQL. It should be added, though, that even if the size of the DBMS is very small, the total memory used for data storage can increase because of the added overhead for each data element stored in the database. This is because memory is used to store the database indexing system, data element locks, etc. Clearly, there is a trade-off between functionality and memory requirements. The most important issue in this application is timeliness. The system cannot be allowed to miss deadlines and behave unpredictable in any way. It is off-line scheduled

with non-preemptable tasks. This fact provides some interesting implications. No task, except the driver display task (see section 5.2.3), can preempt another task. Thus, database conflicts are automatically avoided since the tasks themselves are mutually exclusive. This makes database concurrency control and locking mechanisms unnecessary because only one transaction can be active in such a system at any given time, thus serialization of transactions are handled "manually". This is similar to why semaphores are not needed for non-preemptive real-time systems [5].

Implementing a database into the existing system will have benefits. All data, regardless of on which media it is stored, can be viewed as one consistent database. The relations between the data elements can be made clearer than today. For example, currently an image retrieval in the IECU is performed by first looking in the image Db, then in the language Db, and finally in the HW Db. A database query asking for an image, using the current language and the correct value from the HW Db, can be done in one operation. Furthermore, constraints on data can be enforced centrally by the database. If a data element has a maximum and a minimum value, the database can be aware of this and raise an exception if an erroneous value is inserted. Today, this is performed in the application, implying a responsibility that constraints are made consistent between all tasks that use the data.

In this system the transaction dispatching delay is removed since a database scheduler is not needed. Also, conflict resolution is removed since no conflicts will occur because only one transaction is running at any given time. Regarding the data access time, it will increase as the database grows larger. However, this can be tolerated since the increase can be controlled in two ways. First of all, as the database is a main-memory database, any access to data will be significantly shorter than the execution times of the transactions. To decrease the transaction response times various indexing strategies especially suited for main-memory databases can be used, e.g., t-tree [6] and hashing algorithms [7].

The application investigated in this paper consists of, as previously mentioned, primarily non-preemptable tasks, hence no concurrency control is needed. One interesting question is how this approach would fit into a preemptable off-line scheduled system. This would call for some kind of concurrency control in the database, thus possibly resulting in unpredictable response times for transactions due to serialization conflicts. However, this could be avoided by solving all conflicts off-line. Since all transactions in the system are known a priori, we know all data elements that each transaction touches. This allows us to feed the off-line scheduler with information about which transactions might

cause conflicts if preempted by each other. The scheduler can then generate a schedule where tasks containing possibly conflicting transactions do not preempt each other.

5.3.2 DBMS Design Implications

If we can bound the worst case response time for a specific transaction, we can add this time to the calling tasks worst-case execution time (WCET) without violating the hard real-time properties of the system.[1] Execution of the transaction on its task's thread instead of having separate database tasks, decreases the number of tasks in the schedule, making it easier for the off-line scheduling tool to find a feasible schedule. However a question one should ask is: How do we find the worst case response time for a transaction? There are basically four different circumstances that define the response time of a transaction, namely: (i) the time it takes from the instant a transaction is released until the instant it is dispatched; (ii) the actual execution time of the code that needs to be executed; (iii) the time it takes to access the data elements through the indexing system; and (iv) the time it takes to resolve any serialization conflicts between transactions. For an optimistic concurrency control this would imply the time it takes to run the transaction again, and for a pessimistic concurrency control it would be the time waiting for locks.

In this system the transaction dispatching delay is removed since a database scheduler is not needed in this system. Also, conflict resolution is removed since no conflicts will occur because only one transaction is running at any given time. Regarding the data access time, it will increase as the database grows larger. However this can be tolerated since the increase is bounded if a suitable indexing structure is used, such as the T-tree [6] or the hashing [7] algorithms.

In future versions of this application, it is expected that some of the functionality is moved to the blue part, thus requiring concurrency control and transaction scheduling since we cannot predict the arrival times of blue tasks. Moving parts of the application to the blue part could imply restructuring the data model if a database is not used. If new functionality from the database will be needed in the future, the database schema can be reused. Still, this would not allow non-periodic transactions. Furthermore, it would not allow tasks scheduled online, e.g., blue tasks. However, an extension that would allow this is shown in figure 5.7. A non-preemptable scheduler task is placed in

[1] The response time is defined as the time from transaction initiation to the completion of the transaction.

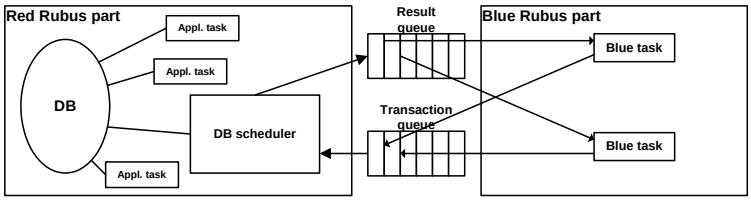


Figure 5.7: A database that supports non-periodic transactions via an external scheduler.

the red part of Rubus. Since this task is non-preemptable it is mutually exclusive towards all other tasks and can therefore have access to the entire database. If this task is scheduled as a periodic task, it acts like a server for transaction scheduling. Thus, the server reads all transactions submitted to the transaction queue, process them and return the results in the result queue (blue tasks are preemptable and, hence, their execution can be interleaved).

From the blue tasks' perspective, they can submit queries, and since we know the periodicity of the scheduler task we can determine the worst-case execution time for these transactions. From the red tasks' perspective, nothing has changed, they are still, either as in the current system, non-preemptive resulting in no conflicts, or they are scheduled so that no conflicts can occur. It is important to emphasize that this method is feasible only if any transaction processed by the scheduler task can be finished during one instance of the scheduling task. If this requirement cannot be met an online concurrency control is needed.

5.3.3 Mapping Data Requirements to Existing Database Platforms

Today there are database platforms, both research and commercial platforms, which fulfill a subset of the system requirements. The DeeDS [8] platform, for example, is a hard real-time research database system that support hard periodic transactions. It also has a soft and a hard part. Furthermore, the DeeDS system uses milestones and contingency plans. These hard periodic transactions would suit the red Rubus tasks and would, if used with milestones and contingency plans, suit the Volvo application. The milestones would check that no deadlines are about to be missed, and the contingency plans would ex-

ecute alternate actions if that is the case. DeeDS is, as the STRIP system [9] a main memory database that would suit this application. The Beehive [10] system implements the concept of temporal validity, that would ensure that temporal consistency always exists in the database. These platforms are designed as monolithic databases with the primary intent to meet multiple application requirements with respect to real-time properties, and on a lesser extent the embedded requirements. As such, they are considered to provide more functionality than needed, and as a consequence, they are not optimal for this application given the need to minimize resource usage as well as overall system complexity.

On the commercial side, embedded databases exist that are small enough to fit into the current system, e.g., the Berkeley DB by Sleepycat Software Inc. and the Pervasive.SQL database for embedded systems. There are also pure main-memory databases on the market, e.g., Polyhedra and TimesTen. Polyhedra, DeeDS, STRIP, and REACH [11] are active database systems, which can enforce consistency between the raw values and the engineering values, and thereby removing the need for the data derivation task. However, integrating active behavior in a database makes timing analysis of the system more difficult. The Berkeley DB system allows the user to select between no concurrency control and an pessimistic concurrency control [12]. If Volvo should decide upon moving part of the functionality to the blue part, concurrency in the database would be necessary. The option of choosing whether or not to use concurrency control would enable the use of the same DBMS, database scheme, and database interface regardless of the strategy being used. Unfortunately, none of the commercial systems mentioned have any real-time guarantees and are therefore not suitable for this type of application.

5.4 Conclusions

We have studied two different hard real-time systems from the vehicular industry with respect to data management, and we have found that data is scattered throughout the system. This implies that getting a full picture of all existing data and its interrelations in the system is difficult.

Further, we have redesigned the architecture of the system to support a real-time database. In this new architecture all tasks communicate through the database instead of using ports, and the database provides a uniform access to data. This application does not need all the functionality provided by existing real-time database research platforms, and issues like concurrency and schedul-

ing have been solved in an easy way. Currently the application is designed so that all tasks are off-line scheduled. All tasks, except the driver display task, are non-preemptive. However, future versions of the application are expected to embrace preemption as well as online scheduled tasks.

Finally, we have discussed mapping the data management requirements to existing databases. Some of the database platforms, both research and commercial, offer functionality that is needed by the system, but at the same time they introduce a number of unnecessary features.

Our future work will focus on the design and implementation of a tailorable real-time embedded database [13]. This includes: (i) developing a set of real-time components and aspects, (ii) defining rules for composing these components into a real-time database system, and (iii) developing a set of tools to support the designer when composing and analyzing the database system. A continuation of this case study where we will implement our database in the Volvo system is planned.

Bibliography

[1] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano - a Revolution in On-Board Communications. Technical report, Volvo Technology Report, 1998.

[2] K. Ramamritham. Real-Time Databases. *International Journal of distributed and Parallel Databases*, 1(2):199–226, 1993.

[3] Rubus OS - reference manual. Articus Systems, 1996.

[4] J. Stankovic, S. Son, and J. Hansson. Misconceptions About Real-Time Databases. *IEEE Computer*, 32(6):29–36, June 1999.

[5] J. Xu and D. L. Parnas. Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations. *IEEE Transactions on Software Engineering*, 16(3):360–369, 1990.

[6] H. Lu, Y. Ng, and Z. Tian. T-Tree or B-Tree: Main Memory Database Index Structure Revisited. In *Proceedings of the 11th Australasian Database Conference*, pages 65–73. IEEE Computer Society, January 2000.

[7] W. Litwin. Linear Hashing: A New Tool for File and Table Addressing. In *Proceedings of the 6th International Conference on Very Large Databases*, pages 212–223. Springer, October 1980.

[8] S. F. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring. DeeDS Towards a Distributed and Active Real-Time Database System. *ACM SIGMOD Record*, 25(1):38–40, 1996.

[9] B. Adelberg, B. Kao, and H. Garcia-Molina. Overview of the STanford Real-time Information Processor (STRIP). *SIGMOD Record*, 25(1):34–37, 1996.

[10] J. A. Stankovic, S. H. Son, and J. Liebeherr. *Real-Time Databases and Information Systems*, chapter BeeHive: Global Multimedia Database Support for Dependable, Real-Time Applications, pages 409–422. Kluwer Academic Publishers, 1997.

[11] A. P. Buchmann, A. Deutsch, J. Zimmermann, and M. Higa. The REACH active OODBMS. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 476–476. ACM Press, May 1995.

[12] X. Song and J. Liu. Maintaining Temporal Consistency: Pessimistic vs. Optimistic Concurrency control. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):786–796, October 1995.

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

Chapter 6

Paper B: COMET: A Component-Based Real-Time Database for Automotive Systems

Dag Nyström, Aleksandra Tešanović, Mikael Nolin, Christer Norström, and Jörgen Hansson

In Proceedings of the Workshop on Software Engineering for Automotive Systems (SEAS04), Edinburgh, Scotland, May 2004
In conjunction with the International Conference on Software Engineering 2004 (ICSE04)

Abstract

With the increase of complexity in automotive control systems, the amount of data that needs to be managed is also increasing. Using a real-time database management system (RTDBMS) as a tightly integrated part of the software development process can give significant benefits with respect to data management. However, the variability of data management requirements in different systems, and the heterogeneousness of the nodes within a system may require a distinct database configuration for each node. In this paper we introduce a software engineering approach for generating RTDBMS configurations suitable for resource-constrained automotive control systems, denoted the COMET development suit. Using software engineering tools to assist developers with design and analysis of the system under development, different database configurations can be generated from pre-fabricated components. Each generated COMET database contains only functionality required by the node it is executing on.

6.1 Introduction

In recent years, automotive control systems have evolved from simple single processor systems to complex distributed systems. At the same time, the amount of data that needs to be managed by these systems is increasing dramatically; data volume managed by automotive systems is predicted to increase 7-10% per year [1]. Current techniques for storing and manipulating data objects in automotive systems are ad hoc in the sense that they normally manipulate data objects as internal data structures. This lack of a structured approach to data management results in a costly development process with respect to design, implementation, and verification of the system [2]. It also makes the system difficult to maintain and develop while preserving consistency with the environment, e.g., maintaining temporal properties of data. As data complexity is growing the need for a uniform, efficient, and persistent way to store data is becoming increasingly important. Using a real-time database management system (RTDBMS) as a tightly integrated part of an automotive control system has the potential to solve many of the problems that application designers have to consider with respect to data management, e.g., locking of the data, persistency and deadlock situations. More importantly, incorporating an RTDBMS into an automotive control system can reduce development costs, result in higher quality of the design of the systems, and consequently yield higher reliability [3].

The variability of data management requirements in different automotive control systems requires distinct RTDBMS configurations specially suited for the particular system [4]. Since an automotive control system is heterogeneous, consisting of several nodes (called electronic control units, ECUs), see figure 6.1, the ability to configure the RTDBMS to suit the requirements of an individual node is crucial. For instance, an automotive system could consist of a small number of resource adequate ECUs responsible for the overall performance of the vehicle, e.g., 32bit CPUs with a few Mb of RAM, and a large number of ECUs responsible for controlling specific subsystems in the vehicle, which are significantly resource-constrained, e.g., an 8bit micro-controller and a few kb of RAM [2]. ECUs with greater amount of resources usually have real-time operating systems support, which is not affordable in small resource-constrained ECUs. Although different in their characteristics and available resources, all nodes in an automotive control system are exchanging, sharing and manipulating data, thereby requiring a uniform way of data management, e.g., via a RTDBMS.

The heterogeneous characteristics of nodes in an automotive control system

result in a need to have distinct RTDBMS configurations suited for a particular node [2]. In safety-critical nodes, tasks are often non-preemptive and scheduled off-line, implying that a RTDBMS configuration for that node could be made small in size and provided functionality, since the majority of the RTDBMS's functionality, such as synchronization and concurrency-control, could be handled off-line. Less critical and larger nodes have preemptable tasks, requiring a RTDBMS configuration with run-time concurrency control mechanisms, and support for database queries formulated during run-time (ad-hock queries). A configurable RTDBMS supporting different types of nodes would, from the application's point of view, provide uniform access to the data regardless of the size and characteristics of an ECU.

Today, there exists a number of commercial databases suitable for embedded systems, e.g., Pervasive.SQL [5], Polyhedra [6], Berkeley DB [7], and TimesTen [8]. Although small in size and therefore suitable for resource-constrained automotive control systems, these databases do not incorporate real-time behavior. This in turn implies that their behavior cannot be analyzed, which makes them unsuitable for deployment in an automotive system. Research projects that are building real-time database platforms, such as DeeDS [9], RODAIN [10], STRIP [11], and BeeHIVE [12], mainly address real-time requirements, are monolithic, and targeted towards a larger-scale real-time application, which makes them unsuitable for use in embedded resource-constrained environments.

In this paper we propose a software engineering approach for generating RTDBMS configurations suitable for resource-constrained automotive control systems. This approach is supported by the **COMET development suit**. The suit consists of a set of data management, analysis and configuration tools, as well as a library of pre-defined software artifacts providing specific RTDBMS functionality. The library of artifacts and the possible configurations of the RTDBMS are referred to as the COMET RTDBMS platform. With the COMET development suit we aim at providing software developers an automated way of tailoring and analyzing the data management for a particular automotive control system, or a node in the system. The COMET RTDBMS platform, a part of the COMET development suit, is developed using an approach to aspectual component-based software development (ACCORD) [13]. ACCORD enables us to utilize the benefits of component-based software development (CBSD) [14] by developing components that encapsulate specific real-time database functionalities. ACCORD also enables us to exploit the benefits of aspect-oriented software development (AOSD) [15] by providing a way of encapsulating, managing, and implementing crosscutting concerns

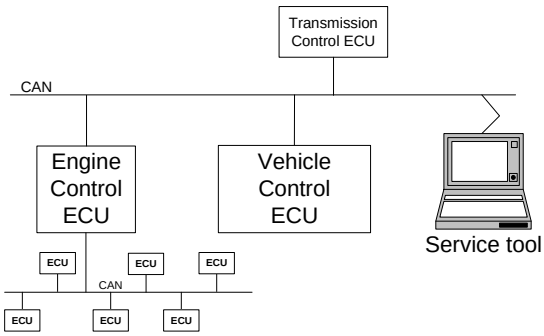


Figure 6.1: An heterogeneous automotive control system

in a RTDBMS in a predictable manner; crosscutting concerns include concurrency control, logging, and recovery. In AOSD, a crosscutting concern is a functionality or non-functional feature that cannot cleanly be encapsulated in a procedure, function, object or a class [15].

The paper is organized as follows. In section 6.2 we present the COMET development suit. We present the key concepts used in the COMET development suit in section 6.3, including COMET aspects and components and possible COMET RTDBMS configurations. We conclude the paper and discuss our future work in section 6.4.

6.2 The COMET development suit

To successfully and efficiently generate systems from a library of pre-defined artifacts, the development process should be supported by appropriate tools. In this section we present our view of the overall development process to obtain system-specific RTDBMS configurations. Figure 6.2 shows the constituents of this process.

As shown in figure 6.2, the development of a RTDBMS configuration starts with specifying the requirements of an automotive control system, which are then used as input for making a model of the system. This model consists of the nodes, their interconnections and the individual run-time properties, e.g., the scheduling policy of the node, if the tasks are preemptive or not, available memory, and CPU resources. The goal of making the model of the underly-

ing system is to derive required database functionality for each of the node in the system. Examples of functionality are support for ad-hoc queries (queries dynamically created during run-time), and to enable the data organization to be changed during run-time (i.e., provide a dynamic database schema). Next, a model of the database, i.e., the actual data, and any precompiled queries are derived with the help of the data engineering tool. This step also involves specifying which parts of the database should be available on which node, and the temporal properties of the data, such as temporal consistency [16]. This information, i.e., desired database functionality for each node, data model, and database schema, is then used by the configuration tool to select a set of aspects and components from the library to form a database configuration suitable for each of the nodes (see figure 6.2). The overall decomposition of the database functionality into aspects and components, and the development of components and aspects, is done according to the ACCORD design principle (see section 6.3.1). The obtained COMET configurations can then be analyzed with respect to run-time properties, e.g., worst case execution time, memory requirements, and response time analysis, by the analysis tools. If the analysis indicates that the configuration is unfeasible, the configuration step and analysis step could be further iterated until an acceptable solution is found.

The resulting RTDBMSs are configured to contain no more than the needed functionality, thus reducing both computational costs and memory requirements.

6.3 The COMET key concepts

As mentioned, different nodes in the automotive control system may require distinct RTDBMS configurations. Component-based databases [17, 18, 19, 20, 21, 22, 7] using the component-based software development paradigm [14] can be partially or completely assembled from a pre-defined set of components with well-defined interfaces. Therefore, these are suited for tailoring a database system towards an application. However, there are aspects of database systems that are difficult to encapsulate into components with well-defined interfaces; typical examples include synchronization, transaction models, and database policies such as concurrency control [23]. These aspects are crosscutting concerns that permeate the whole system and affect multiple components. Hence, using traditional component-based approach is necessary but not sufficient to enable efficient development of configurable RTDBMSs. Therefore, in COMET we use an approach to aspectual component-based real-time system

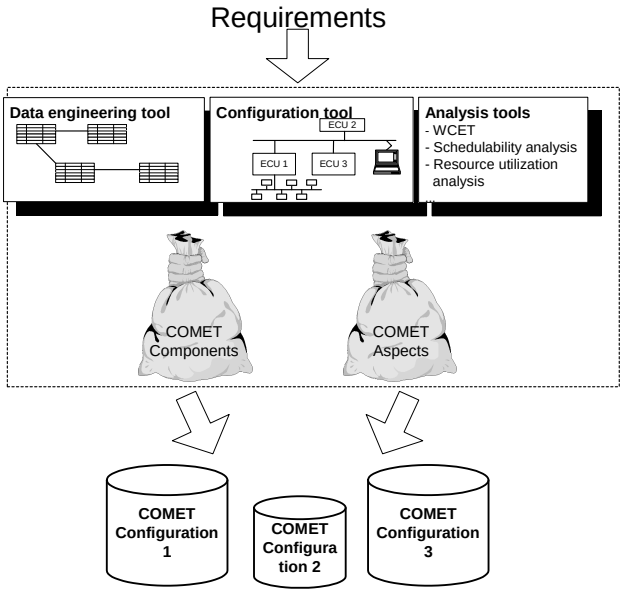


Figure 6.2: The COMET development suit

development (ACCORD) [13, 24] (discussed in section 6.3.1) that provides a notion of a reconfigurable component, and thereby enables both encapsulation of RTDBMS functionality into components and efficient handling and implementation of crosscutting concerns via aspects.

Using the ACCORD approach, different COMET components and aspects can be developed, and then used for assembling COMET configurations suitable for a specific automotive control system. Existing COMET components and aspects are discussed in section 6.3.2. We illustrate the COMET concepts introduced in this section with an example of the COMET configuration suitable for a particular node in the automotive control system in section 6.3.3.

6.3.1 Aspects and components in RTDBMSs

ACCORD utilizes notions from both component-based and aspect-oriented software development, integrating them into real-time system development. While CBSD traditionally use black box as an abstraction metaphor for the components, AOSD utilizes the white box metaphor to emphasize that all details of the implementation should be revealed. ACCORD supports the notion

of a reconfigurable real-time component model (RTCOM) [25, 13, 24]. Components built using RTCOM are grey boxes as they are encapsulated in interfaces but changes to their behavior can be performed in a predictable way using aspects. Aspects are allowed to modify the code of the components in predefined, explicitly declared, reconfiguration points. In this section we briefly review RTCOM and its configurability via aspects, while detailed descriptions of ACCORD and RTCOM can be found in [25, 13, 24].

Aspects are programming-language level constructs encapsulating crosscutting concerns that invasively change the code of the component and correspond to the traditional aspects in existing aspect languages. The main constituents of aspects are: (i) components, written in a component language, e.g., C, C++, and Java; (ii) aspects, written in a corresponding aspect language, e.g., AspectC [26], AspectC++ [27], and AspectJ [28]; and (iii) an aspect weaver, which is a preprocessor that inserts code from the aspects into the reconfiguration points of the components.

An aspect in an aspect language consists of pointcuts and advices. Next we give a brief review of a typical syntax and semantics used in an aspect language; figure 6.5 shows a concrete example of an aspect woven into a component. A *pointcut* in an aspect language consists of one or more join points, and it is described by a pointcut expression. A *join point* refers to a point in the component code where aspects should be woven, e.g., a method, a type (struct or union). In RTCOM join points are explicitly declared in the component interfaces as reconfiguration points, and these are declared such that temporally predictable weaving in the component code can be done. An *advice* is a declaration used to specify the code that should run when the join points, specified by a pointcut expression, are reached. Different kinds of advices can be declared, such as: (i) *before advice* code is executed before the join point, (ii) *after advice* code is executed immediately after the join point, and (iii) *around advice* code is executed in place of the join point.

6.3.2 The COMET RTDBMS platform

A central goal with COMET is to enable configurability so that it can handle a variety of different application requirements; COMET has an architecture that allows this [29]. Following the ACCORD design method described in section 6.3.1, the architecture of COMET consists of a number of components and a number of aspects. Each component provides a well-defined service through operations that are defined in a component's interface. Aspects and components that together provide a specific functionality are denoted as aspect

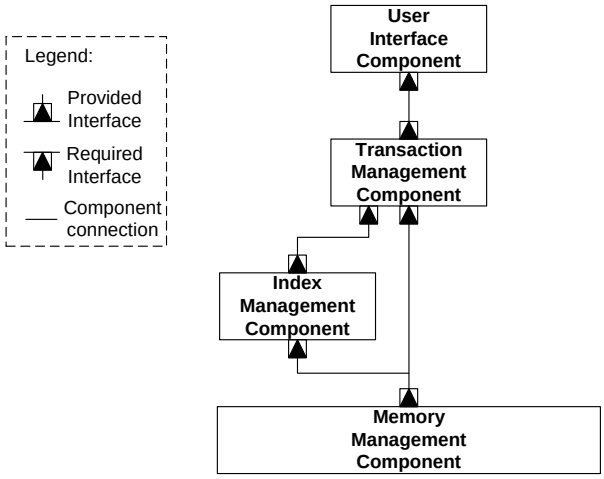


Figure 6.3: The basic architecture of COMET

packages.

The foundation of COMET consists of a basic architecture in which components can be instantiated (see figure 6.3). A fully instantiated basic architecture is referred to as a basic configuration. The basic configuration builds a fully functional RTDBMS capable of storing, manipulating and querying data using some high level database query language. Even though a basic configuration is considered to be a RTDBMS, it has limited functionality, e.g., it cannot handle concurrent transactions, and it has no database crash recovery mechanisms. A basic COMET configuration consists of the following four components:

1. **The user interface component** (UIC) provides a database interface to the application. This interface consists of a data manipulation language, in which the user (application) can query and manipulate data elements. Furthermore, the interface consists, if configured so, of a data definition language which enables the user to manipulate the database schema, e.g., creating and dropping relations (tables). Application requests are parsed by the UIC, and are then converted into an execution-plan.

2. **The transaction management component** (TMC) is responsible for ex-

ecuting incoming execution-plans, thereby performing the actual manipulation of the data in the database.

3. **The index management component** (IMC) is responsible for maintaining an index of all tuples in the database. This is normally done through hash-tables or index-trees. The IMC is capable of transforming a database key into the memory address of the tuple correspondent to the database key. Furthermore, the IMC maintains the database schema in its index of meta-data.

4. **The memory management component** (MMC) is responsible for memory allocation of tuples, metadata, and database indexes.

By selecting versions of these components, different basic COMET configurations can be derived.

In addition to these, mandatory, components, it is possible to add optional components to the architecture, such as the scheduling management component (SMC), which is responsible for scheduling transactions. This is useful when the application is preemptive and multiple transactions can be issued simultaneously. However, noteworthy is that a basic configuration of COMET cannot execute multiple transactions concurrently. In this case the SMC maintains the list of transactions in a ready queue and releases the next transaction when the previous is completed.

The services described above are all well defined and their activities are to a high degree isolated, i.e., it would be possible to exchange each one of these services with a different implementation, as long as they interact with other services in the same way. This makes these services suitable for encapsulation into components.

However in a RTDBMS there are concerns which cannot be divided into isolated activities, but rather crosscut multiple components in the system. These crosscutting concerns are, in COMET, encapsulated into aspect packages, which can contain both aspects and components. In figure 6.4, three such aspect packages can be seen, namely:

1. **The concurrency control aspect package** (CCA) allows multiple transactions to be executed concurrently. Managing concurrent transactions requires some form of concurrency control. The CCA consists of a locking management component (LMC) and a concurrency control aspect. The LMC allows transactions to obtain read- and write-locks on data elements. The concurrency control aspect contains the code for obtaining and releasing the locks, as well as a transaction conflict resolution

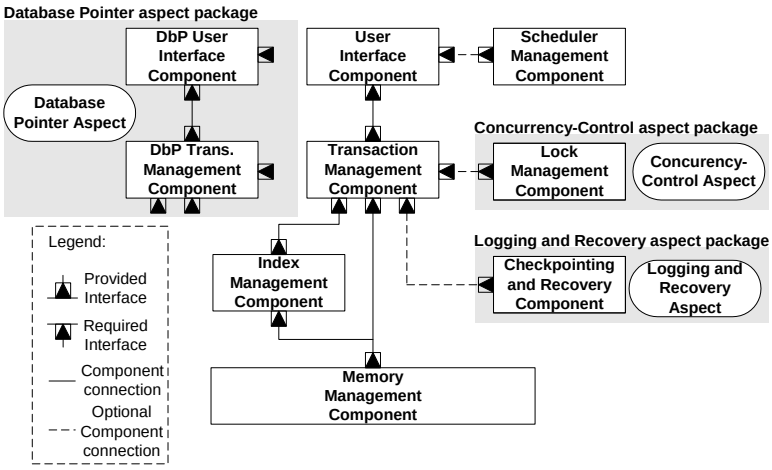


Figure 6.4: The architecture of COMET with aspect packages

method. The code of the concurrency control aspect is woven into (i) the TMC to force transactions to obtain locks before accessing data elements, and to release them when finished, (ii) the SMC to enable it to handle graceful termination of aborted transactions, and (iii) the LMC to adapt its behavior according to the conflict resolution policy used.

2. **The database pointer aspect package** (DBPA) enables the application to access individual data elements within the database in an efficient and predictable way. A database pointer [30] is a pointer that is first bound to a specific data element, which then can be read and written with a minimum overhead. Database pointers are used together with the relational data model, and they do not place limitations on the RTDBMS with respect to reorganizing the database schema during run-time. The database pointer concept is developed with automotive control in mind and is a fundamental part when integrating a RTDBMS into an automotive control system. The DBPA consists of two components, namely the database pointer user interface component that provides the application with the database pointer user interface, and the database pointer transaction management component that executes the database pointer operations. Furthermore, the DBPA consists of a database pointer aspect

which is woven into the TMC and the IMC, adapting them to co-exist with database pointers.

3. **The logging and recovery aspect package** (LRA) ensures that the database is consistent after a system crash. Logging and recovery is performed through periodic checkpoints, where an image of the database is saved to a persistent storage and all intermediate changes to the database are logged. The aspect package consists of one component, the checkpointing and recovery component (CRC), which contains methods defining how to checkpoint and log changes to the database, and one aspect, the logging and recovery aspect (LRA). The LRA is woven into the MMC, the TMC, and the CRC.

Hence, the COMET RTDBMS platform contains a set of components and aspect packages that are suitable for automotive control systems.

COMET components discussed in this section are suitable for configuring RTDBMSs for use in ECUs requiring the relational data model that can be manipulated using ad-hoc queries. Currently these components support the most common database query commands, namely the `select`, `insert`, `update`, `delete`, `create table` and `drop table`. However, there are also COMET components that only allow static database schemas and precompiled queries, which are suitable for nodes that cannot afford, or do not require, ad-hoc queries. In a configuration generated from such components most of the functionality is handled off-line using the COMET tools.

6.3.3 A configuration example

To illustrate how the COMET RTDBMS can be configured to suit a particular ECU we present the following example in which we create a suitable COMET configuration based on a number of requirements. Note that the given requirements are typical data management requirements that can be found in an engine ECU of a modern car [3, 2]. Consider the engine ECU with the following data management requirements:

R1: The application performs computations using data obtained from sensors. Sensor data should reflect the state of the controlled environment and, hence, are associated with hard real-time temporal requirements and data freshness requirements.

R2: The application performs diagnostics on the system in order to analyze the system behavior. The diagnostics should be performed both in the

steady state of the vehicle and in the transient states, in order to get the full spectrum of data and be able to analyze the vehicle behavior under all situations. Diagnostic operations performed on the system are not critical to the operational safety of the vehicle and, therefore, are associated with soft real-time temporal requirements.

R3: The set of data in the system is fixed at compile time and is never changed during run-time.

R4: The ECU uses preemptive fixed-priority scheduling, implying that multiple tasks can execute concurrently. This in turn implies that the same data items can be read and written by different tasks (which could result in inconsistent data values in the system).

R5: It should be possible to connect a service tool to retrieve system data.

When configuring a RTDBMS for such a system, we begin by modeling the ECU based on the requirements. The configuration tool is then used to provide a suitable basic COMET configuration.

In this case, the basic configuration could be based on the components providing a relational data model, since creating views and complex queries is a required feature of the RTDBMS configuration for this ECU (requirement **R2**). Furthermore, the relational data model provides support for ad-hoc queries (requirement **R5**). However, since a dynamic database schema is not necessary (requirement **R3**), the MMC from the basic COMET configuration in figure 6.3 can be replaced with components providing static data management and database indexing.

When a suitable basic configuration has been selected, the configuration tool proceeds by adding suitable aspect packages. First, the database pointer aspect package is selected to provide fast access of individual data elements in the ECU (requirement **R1**). Then, in order to fulfill requirement **R4** and enable concurrent execution of transactions such that data values in the database are kept consistent, the concurrency control aspect package is selected. In this case, a concurrency control algorithm 2V-DBP [31] could be selected. 2V-DBP combines locking with two-versions of selected parts of the database to enable hard database pointer transactions to execute without being blocked by soft relational transactions. Using 2V-DBP enables the application to support both hard control tasks and soft diagnostics task in the ECU. Now, all the requirements are fulfilled and the RTDBMS configuration is complete.

The next step is to enter the database schema into the data engineering tool. Since we have chosen a static database schema, the necessary data structures

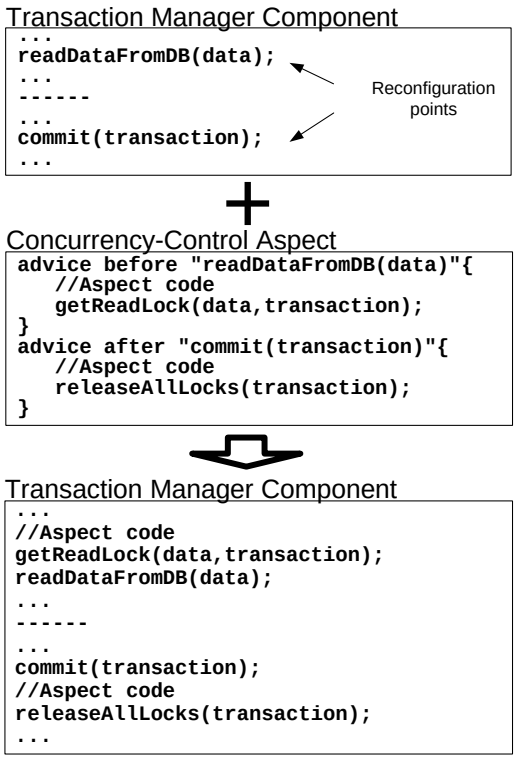


Figure 6.5: A simplified example of the weaving process

for the database are created. In this step, precompiled queries can also be created and optimized.

Finally, the analysis tools are used to check if the generated configuration of the RTDBMS is feasible. The analysis tool is used to determine the worst case execution time of the RTDBMS so that schedulability analysis can be performed. Furthermore, the memory requirements of the RTDBMS and physical storage needed for storing data in the system are analyzed.

Given that the configuration and analysis can be done on the models of the components, aspects, and the RTDBMS configuration, the actual weaving and component composition process can be performed after the database configu-

ration is found feasible by the tools. Figure 6.5 illustrates the weaving process and its constituents for the TMC and the concurrency control aspect. The concurrency control aspect contains two advices: (1) an advice of type before that defines the code that should be inserted into TMC reconfiguration point `readDataFromDB()` to ensure that a transaction obtains the lock on a data item, and (2) an advice of type after that defines the code to be inserted immediately after the transaction commits to ensure that the transaction releases all the locks it has obtained while executing in the database. The result of the weaving is the TMC modified at the reconfiguration points, such that every read of the data item is now preceded by locking, while every commit of the transaction is followed by unlocking (see figure 6.5). Note that this example is simplified to show main constituents of the aspects, components, and their possible interaction. However, in the actual implementation, the concurrency control aspect is more complex and contains advices that crosscut the behavior of the SMC and LMC components as well. Moreover, the concurrency control aspect code provides an efficient way of handling possible deadlocks in the database. The overall benefit of having aspects for tailoring database components is the ability to use reconfiguration points in the syntax of the pointcuts of advices. This in turn enables us to identify not only the places in the code of the program that have the same signature as the reconfiguration points (used in the presented example), but also define pointcuts that refer to the execution of the reconfiguration points (i.e., after the call has been made and a function started to execute), and to match any reconfiguration point that has values of a specified type. Also, operators &&, ||, and ! can be used to logically combine or negate pointcuts. Furthermore, separation of concerns into aspects enables us to have both components without aspects, and reconfigured components with aspects as aspect weaving results in a new component weaved with aspect code, but leaving the code of the original component unchanged and available for future reuse, i.e., now we can also reuse already reconfigured components or use original components with different aspects in other reuse contexts. For extensive discussion on benefits of having aspects for tailoring components in the RTDBMSs we refer interested readers to [25, 24].

Finally, if the obtained final configuration of the RTDBMS is found to meet the original requirements, as well as the timing requirements, the RTDBMS is compiled and made ready for deployment into the application.

By this simple example we have shown how a set of requirements can be used when configuring a COMET RTDBMS in order to get a RTDBMS specially suited for a particular ECU in an automotive control system.

6.4 Conclusions

In this paper we have presented a configurable real-time database platform, called COMET, which provides support for efficient data management in heterogeneous automotive systems. The COMET platform consists of a library of components and aspects, and is supported by a tool suite. The COMET tool suit assists system designers in configuring and analyzing different COMET configurations based on the specific requirements of the targeting automotive system and its nodes. While components encapsulate distinct functionalities of a database system, aspects allow efficient tailoring of the components and the database system based on the requirements of the underlying automotive system or its node. Our approach in providing different COMET configurations by using components in the library together with aspects can also be viewed as efficient product-line architectures of real-time database systems in the automotive domain.

We have showed the differences of the provided properties of the commercially available embedded databases, as well as real-time databases, compared with the needs of automotive control systems. To the best of our knowledge, no previous work exists that takes a holistic approach to data management in automotive systems. Given the increase of data complexity in automotive systems it is our experience that a more structured form of data management will be necessary in a near future, in order to keep time to market as well as development and maintenance costs down.

Although we have presented and discussed distinct configurations of COMET suitable for different nodes in the automotive systems, these were not developed using the full automated support of the tool suite. Rather, the automation done in the development process of COMET configurations so far has been focused on the analysis tools, where we developed the tool for analyzing different configurations of aspects and components with respect to their temporal properties [32, 33]. The remaining part of the COMET tool suit is currently under development. Further work on integrating the database into the component framework, to allow components to be easily distributed over multiple nodes is also planned.

Bibliography

[1] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano - a Revolution in On-Board Communications. Technical report, Volvo Technology Report, 1998.

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

[3] T. Gustafsson and J. Hansson. Data management in real-time systems: a case of on-demand updates in vehicle control systems. In *Proceedings of the Real-Time Application Symposium (RTAS 2004)*. IEEE Computer Society Press, May 2004.

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

[5] Pervasive Software Inc. http://www.pervasive.com.

[6] Enea AB. http://www.enea.se.

[7] Sleepycat Software Inc. http://www.sleepycat.com.

[8] TimesTen Performance Software. http://www.timesten.com.

[9] S. F. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring. DeeDS Towards a Distributed and Active Real-Time Database System. *ACM SIGMOD Record*, 25(1):38–40, 1996.

[10] J. Lindstrom, T. Niklander, P. Porkka, and K. Raatikainen. A Distributed Real-Time Main-Memory Database for Telecommunication. In *Proceedings of the Workshop on Databases in Telecommunications*, pages 158–173. Springer, September 1999.

[11] B. Adelberg, B. Kao, and H. Garcia-Molina. Overview of the STanford Real-time Information Processor (STRIP). *SIGMOD Record*, 25(1):34–37, 1996.

[12] J. A. Stankovic, S. H. Son, and J. Liebeherr. *Real-Time Databases and Information Systems*, chapter BeeHive: Global Multimedia Database Support for Dependable, Real-Time Applications, pages 409–422. Kluwer Academic Publishers, 1997.

[13] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Towards aspectual component-based real-time systems development. In *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA'03)*, February 2003.

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

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

[16] K. Ramamritham. Real-Time Databases. *International Journal of distributed and Parallel Databases*, 1(2):199–226, 1993.

[17] M. J. Carey, L. M. Haas, J. Kleewein, and B. Reinwald. Data access interoperability in the IBM database family. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Interoperability*, 21(3):4–11, 1998.

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

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

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

[21] Universal data access through OLE DB. OLE DB Technical Materials. OLE DB White Papers, 12 April 2001. Available at http://www.microsoft.com/data/techmat.htm.

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

[23] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(4):355–398, 1992.

[24] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Aspects and components in real-time system development: Towards reconfigurable and reusable software. *Journal of Embedded Computing*, February 2004.

[25] A. Tešanović. Towards aspectual component-based real-time system development. Technical report, Department of Computer Science, Linkping University, June 2003. Licentiate Thesis, ISBN 91-7373-681-3.

[26] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, 2002.

[27] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: an aspect-oriented extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, February 2002. Australian Computer Society.

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

[29] Dag Nyström. COMET: A Component-Based Real-Time Database for Vehicle Control-Systems. Licentiate Thesis ISBN 91-88834-46-8, Department of Computer Science and Engineering, Mälardalen University, Sweden, May 2003.

[30] Dag Nyström, Aleksandra Tešanović, Christer Norström, and Jörgen Hansson. Database Pointers: a Predictable Way of Manipulating Hot Data in Hard Real-Time Systems. In *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications*, pages 623–634, February 2003.

[31] Dag Nyström, Mikael Nolin, Aleksandra Tešanović, Christer Norström, and Jörgen Hansson. Pessimistic Concurrency Control and Versioning to Support Database Pointers in Real-Time Databases. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 261–270. IEEE Computer Society, June 2004.

[32] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Integrating symbolic worst-case execution time analysis into aspect-oriented software development. OOPSLA 2002 Workshop on Tools for Aspect-Oriented Software Development, November 2002.

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

Chapter 7

Paper C: Database Pointers: Efficient and Predictable Data Access in Real-Time Control-Systems

Dag Nyström, Mikael Nolin, Aleksandra Tešanović, Christer Norström, and Jörgen Hansson

Article submitted for journal publication. Based upon the following two conference paper:

Database Pointers: a Predictable Way of Manipulating Hot Data in Hard Real-Time Systems
Dag Nyström, Aleksandra Tešanović, Christer Norström, and Jörgen Hansson
In Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA03), Tainan, Taiwan, February 2003

Pessimistic Concurrency Control and Versioning to Support Database Pointers in Real-Time Databases
Dag Nyström, Mikael Nolin, Aleksandra Tešanović, Christer Norström ,
and Jörgen Hansson
In Proceedings of 16th EUROMICRO Conference on Real-Time Systems, Catania, Sicily, June 2004

Abstract

This paper introduces the concept of database pointers, an efficient and predictable way of accessing data in real-time database management systems. The concept, which is designed to be used in conjunction with a traditional relational database model, allows the creation of variables that point directly to individual data elements in the database. This enable the application to access data within the database in a similar fashion as using shared variables, while still retaining all the benefits of using a database (such as allowing advanced concurrency-control and allowing reorganization of the database during run-time). Furthermore, the paper presents a concurrency-control algorithm, denoted 2-Version Database Pointer Concurrency Control (2V-DBP). The concurrency algorithm allow hard database pointer transactions and soft relational transactions to be executed without blocking (or aborting) each other. 2V-DBP is suited for resource-constrained, safety critical, real-time systems that have a mix of hard real-time control applications and soft real-time management, maintenance, or user-interface applications. The concept presented in this paper is validated, both by formal verification and performance evaluation. The evaluation show that the concept clearly outperforms 2-phase locking with high priority abort (2PL-HP) with respect to minimizing transaction abortions and response times.

7.1 Introduction

The complexity of embedded real-time control-systems is ever increasing in many product domains, this is especially true for automotive systems [1]. These systems have evolved from simple electro-mechanical systems, through single processor systems, to complex distributed systems. At the same time, the amount of data that needs to be managed is increasing dramatically; data volume managed by automotive systems is predicted to increase 7-10% per year[1]. Current techniques for storing and manipulating data objects in automotive systems are ad hoc in the sense that they normally manipulate data objects as internal data structures. This lack of a structured approach to data management results in a costly development process with respect to design, implementation, and verification of systems [2]. It also makes the system difficult to maintain or extend while preserving key properties, e.g., maintaining temporal properties of data. As data complexity is growing the need for a uniform, efficient, and persistent way to store data is becoming increasingly important. Using a Real-Time DataBase Management System (RTDBMS) as a tightly integrated part of an automotive control system has the potential to solve many of the problems that application designers have to consider with respect to data management, e.g., locking of the data, deadlock situations and persistency [3]. More importantly, incorporating an RTDBMS into an automotive control system reduces development costs, result in higher quality of the design of the systems, and consequently yield higher reliability [4].

However, integrating an RTDBMS into a real-time control-system also have potential drawbacks, in terms of increased unpredictability and added execution overhead. Executions of traditional relational (e.g. SQL [5]) transactions are costly operations, both with respect to execution times and memory consumption. Relational transaction processing include, parsing of queries, locating and fetching data tuples using the RTDBMS index structure, as well as the execution of relational operators (such as products, joins etc). There exists a trade-off between flexibility and performance. However, a large part of the software controlling the vehicle does not need this flexibility, nor can it afford the overhead costs.

To resolve this problem, we propose the concept of database pointers [6] to complement the relational query processing in a RTDBMS. Database pointers have the efficiency of a shared variable and can be combined with the relational data model. They allow a fast and predictable way of accessing data in a database without the need of going through the expensive relational query processing. Furthermore database pointers provide an interface that uses a

"pointer-like" syntax. The concept of database pointers is straightforward to apply by an engineer as it resembles current techniques (shared variables) used to access individual sensor data. Database pointers allow fast and predictable accesses of data without violating temporal, logical data-consistency or transaction serialization. It can be used together with the relational data model without violating the database integrity.

Furthermore, we propose a concurrency control algorithm, called 2-Version DataBase Pointer concurrency control (2V-DBP) [7] that allow efficient coexistence of soft real-time database transactions (denoted *soft transactions*) and hard real-time database transactions (denoted *hard transactions*) in a RT-DBMS. To support both types of transactions while avoiding long delays for hard transactions and eliminating soft transaction abortions caused by hard transactions, we propose the use of a versioning algorithm that uses traditional pessimistic concurrency control [8] for soft, relational, transactions but allow hard, database pointer, transactions to execute regardless of any locks held by soft transactions.

2V-DBP is suited for resource-constrained safety-critical, real-time systems that have a mix of hard real-time control applications and soft real-time management, maintenance, or user-interface applications. The algorithm, which combines the concept of database pointers and relational transaction management, satisfies the need for predictable and time-efficient hard real-time control-applications, while allowing relational soft management transactions access to the database without being starved by the hard transactions. The algorithm uses a versioning technique for the hard transactions and *two-phase locking high priority* (2PL-HP) [8] for the soft transactions. In order to support these two concurrency control methods we introduce a simplified form of *versioning*, i.e., we maintain two versions of selected data elements. Our algorithm overcomes the widely recognized problem that transactions with low priority and long execution times are penalized due to the likeliness of data conflicts [9].

The contributions of this paper include a database access concept which: (i) allow data to be efficiently accessed without having to involve costly transaction management, (ii) provides efficient, and time-deterministic, execution of hard transactions, regardless of any database locks held by other transactions; (iii) bounds the maximum memory overhead caused by adding versions of data elements; (iv) allows soft transactions to be executed even though the database is read and updated by hard transactions.

We also present an evaluation of the concept; showing significant benefits for both hard and soft transactions.

The costs of using our proposed concept is an added (although predictable) memory overhead, since all data used by the hard transactions is stored in two versions, and a relaxation of the serialization criteria for soft management transactions.

In section 7.2, our system model and transaction models are presented, as well as some related work. We present the database pointer concept, assuming a pessimistic concurrency-control approach in section 7.3. We improve the performance of the database pointer concept in section 7.4 where we present our proposed concurrency control algorithm 2V-DBP. Furthermore, we provide a formal verification of the algorithm. In section 7.5, we show an evaluation of our proposed concept. We conclude the paper in section 7.6.

7.2 Background

7.2.1 Related Work

Today, there exists a number of commercial databases in the embedded systems market, e.g., Pervasive.SQL [10], Polyhedra [11], Berkeley DB [12], and TimesTen [13]. Although small in size and therefore suitable for resource-constrained automotive control systems, these databases do not incorporate real-time behavior. This in turn implies that their behavior cannot be analyzed, which makes them unsuitable for deployment in an automotive system. Research projects that are building real-time database platforms, such as DeeDS [14], RODAIN [15], STRIP [16], and BeeHIVE [17], mainly address real-time requirements, are monolithic, and targeted towards a larger-scale real-time application, which makes them unsuitable for use in embedded resource-constrained environments.

To the best of our knowledge our experimental RTDBMS, denoted COMET [3], is the only RTDBMS addressing modular configurability, resource conservation, and real-time predictability.

Other approaches regarding concurrency-control for real-time systems exists [18, 19, 20, 21, 22]. However, none of these approaches are focussed towards the specific difficulties concerning vehicle-control systems. Either they address, *bounding* blocking times and abortions, or they consume vast amount of memory (typically versioning algorithms). Our proposed algorithm, 2V-DBP, *eliminates* blocking and abortions for hard transactions, while still consuming a fairly small (and predictable) of memory.

7.2.2 System Model

This paper focuses on real-time applications used to control a process, e.g., critical control-functions in a vehicle such as engine or brake control. The basic flow of execution in such a system is [2]: (i) periodic scanning of sensors, (ii) execution of control algorithms, such as PID-regulators, and (iii) propagation of the result to the actuators. Typically, the application is structured into multiple tasks executed by a preemptive real-time operating system. The tasks can use a RTDBMS to access and manipulate shared data. Hence, the RTDBMS needs some form of concurrency control to maintain consistency given multiple concurrent accesses. In traditional relational databases, data manipulation is performed using queries formulated in a special purpose language such as SQL. Such queries can either be created dynamically (during run-time), so called ad-hoc queries, or be created before run-time and stored in a precompiled format. The latter is the common case in real-time systems, since precompiling a query saves both time and memory during run-time.

7.2.3 Application and task model

We classify the tasks in the system into three categories, namely, I/O-tasks, control-tasks, and management-tasks [2]. The I/O-tasks are typically executed periodically, often at high frequencies. There are two types of I/O-tasks; (i) tasks that read a sensor, and write the value to the database using a write only transaction , and (ii) tasks that read a value from the database, using a read only transaction, and then write it to an actuator. Table 7.2 summarizes the properties of the three types of tasks. I/O-tasks touch very few, in most cases only one, data element in the database, and their transactions are always precompiled.

Control tasks take a set of data values and derive new actuator values, thus performing update transactions on the database, i.e., performing a number of read operations followed by a number of write operations. For most control tasks in a real-time control system, reading the freshest data values available is sufficient (and preferable). Note that this desire to read fresh data is not always adhered to by traditional RTDBMSs, since they focus on preserving transaction ordering rather than providing data freshness.

Management tasks are the only tasks running soft transactions. An example of a management task might be a task presenting statistical information about the current state of the vehicle to the user. A management transaction might also be constructed during run-time, for example by a service technician using a service tool connected to the vehicle.

	Transaction property					
Task type	*Hard RT*	*Soft RT*	*Frequency*	*Trans. type*	*Precompiled*	*Ad hoc*
Control tasks	x		H	U	x	
I/O tasks	x		H	RW	x	
Management tasks		x	L	RWU	(x)	(x)

Legend:
x - the property is true for the task type
(x) - the property is true for some tasks of the task type
H/L - indicates high, or low frequency
RWU - indicates read only, write only, or update transaction type

Table 7.2: Transaction properties for the system's task types.

7.2.4 Relational Query Processing

Relational queries provide a flexible way of viewing and manipulating data. The backside of this flexibility is performance loss.

Figure 7.1 shows a typical architecture of a RTDBMS. A query, requesting value x, passed to a relational RTDBMS typically will go through the following steps:

1. The query is passed from the application to the SQL interface.
2. The SQL interface requests that the query should be scheduled by the transaction scheduler.
3. The relational query processor parses the query and creates an execution plan.
4. All tuples possibly containing x are located by the index manager.
5. The locks needed to gain access to these tuples are obtained by the concurrency controller.
6. The tuples are then fetched from the database, and are then processed by the relational engine.

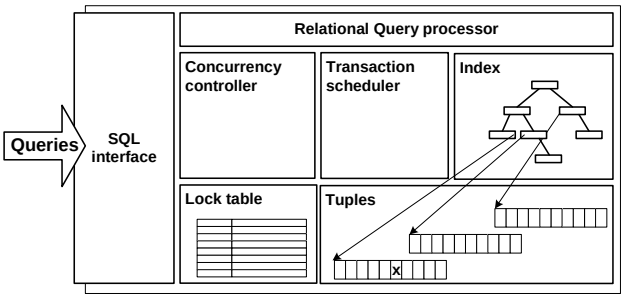


Figure 7.1: Architecture of a typical Database Management System.

7. All locks are released by the concurrency controller.

8. The result is returned to the application.

According to the relational data model a database query on a set of relations in itself always returns a relation. In this case it might be a relation with a cardinality and degree of one (one row and one column), containing only x. In order for the application to read the value of x, it must extract it from the resulting relation using some API call, e.g. `getXY(1,1)`.

In this example we assume a pessimistic concurrency control policy. However, the flow of execution is analogous if a different policy is used.

7.2.5 Transaction models

All tasks in the system that interact with the RTDBMS do this through database transactions. A database transaction consists of a set of database operations, e.g., read and write operations.

In this paper, we consider two different transactions types, namely: (i) Soft transactions which use the relational data model. These transactions provide flexible and dynamic access to data in the database. (ii) Hard transactions which use the database pointer model. Database pointers allow only one operation on one data element per transaction. This operation can either be a read or a write operation. Hard transactions cannot be aborted and will always complete successfully. All transaction has atomic semantics, i.e., either they are fully executed or not executed at all.

7.3 Database pointers with pessimistic concurrency control

The concept of database pointers consists of four different components:

- The `DBPointer` data type, used by the application to access data elements in the database.
- The data pointer entry table, which contains all information needed by the pointers.
- The database pointer interface, which provides a number of operations on the database pointer.
- The database pointer flag, which is used to ensure consistency in the database.

Using the concept of database pointers, the architecture of the DBMS given in figure 7.1, is modified to include database pointer components, as shown in figure 7.2. To illustrate the way database pointers work, and its benefits, we use the example presented in section 7.2.4, i.e., the request for retrieving the data x from the database.

Using the database pointer interface, the request could be made significantly faster and more predictable:

1. A read operation, specifying which database pointer to read is passed from the application to the database pointer interface.
2. The data pointer entry shown in figure 7.3, which is directly pointed out be the database pointer, consists of three fields: the physical address of data element x, information about the data type of x, and eventual locking information that shows which lock x belongs to. Since a pessimistic concurrency is used, the lock for x would be obtained.
3. The value of x is obtained from the database using the direct pointer in the data pointer entry.
4. The lock is released by the concurrency controller.
5. The result is returned to the application.

The four components of the database pointer and its operations are described in detail in sections 7.3.1 to 7.3.4.

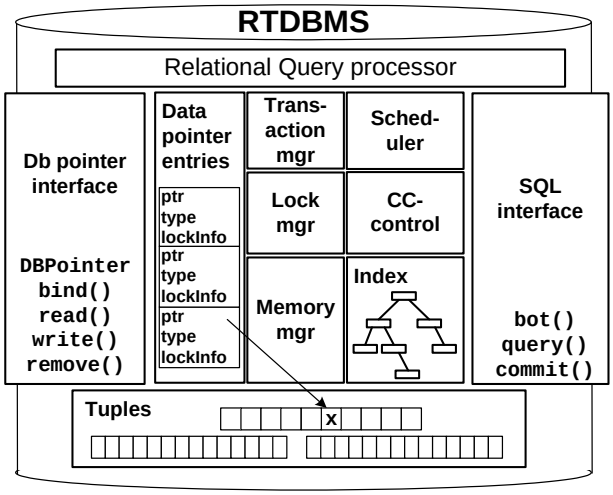


Figure 7.2: Architecture of a DBMS with database pointers.

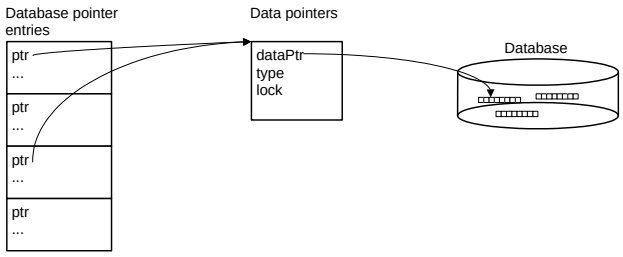


Figure 7.3: The data structures used by the database pointer

7.3.1 The Database Pointer Interface

The database pointer interface consists of four operations:

1. **`bind(ptr,q)`** This operation initializes the database pointer *ptr* by binding it to a data pointer entry, which in turn points to the physical address of the data. The physical binding is done via the execution of the query *q*, which is written using a logical data manipulation language, e.g., SQL. The query should be formulated in such a way that it always returns the address of a single data element. By using the bind operation, the binding of the data element to the database pointer is done using a logical query, even though the result of the binding is physical, i.e., the physical address is bound to the database pointer entry. This implies that no knowledge of the internal physical structures of the database is required by the application programmer.

2. **`remove(ptr)`** This operation deletes a data pointer entry.

3. **`read(ptr)`** This operation returns the value of the data element pointed by *ptr*. It uses locking if necessary.

4. **`write(ptr,v)`** This operation writes the value *v* to the data element pointed by *ptr*. It also uses locking if necessary. Furthermore, the type information in the database pointer entry is compared with the type of *v* so that a correct type is written.

The pseudo codes for the `write` and `read` operations are shown in figure 7.4. The `write` operation first checks that the types of the new value matches the type of the data element (line 2), and then obtains a write lock for the corresponding lock (line 4), i.e., locks the relation that the data element resides in. The data element is then updated (line 5), and finally the lock is released (line 6). The `read` operation obtains the corresponding read lock (line 10), reads the data element (line 11), releases the lock (line 12), and then returns the value to the application (line 13).

7.3.2 The `DBPointer` Data Type

The `DBPointer` data type is a pointer declared in the application task. When the pointer is initialized using the **`bind(ptr,q)`** operation, it points to its corresponding data pointer entry, which in its turn points to the actual data element, see figure 7.3.

```
1  write(DBPointer dbp, Data value){
2     if(DataTypeOf(value) != dbp->type)
3        return DATA_TYPE_MISMATCH;
4     DbGetWriteLock(dbp->lockInfo);
5     *(dbp->ptr) = value;
6     DbReleaseLock(dbp->lockInfo);
7     return TRUE;
   }

8  read(DBPointer dbp){
9     Data value;
10    DbGetReadLock(dbp->lockInfo);
11    value = *(dbp->ptr);
12    DbReleaseLock(dbp->lockInfo);
13    return value;
   }
```

Figure 7.4: The pseudo codes for the `write` and `read` operations

7.3.3 The Data Pointer Entry

The data pointer entry contains information needed by the database pointer, namely:

1. A pointer to the physical memory location of the data element inside the tuple. Typically, the information stored is the data block the tuple resides in, an offset to the tuple, and an offset to the data element within the tuple.

2. The data type of the data element pointed by the database pointer. This is necessary in order to ensure that any write to the data element matches its type, e.g., it is not feasible to write a floating point value to an integer.

3. Lock information describing the lock that corresponds to the tuple, i.e., if locking is done on relation granules, the name of the relation should be stored in as lock information. Note, if locks are not used in the DBMS, i.e., if optimistic concurrency control is used, some other serialization information can be stored in the data pointer entry instead of the lock information.

7.3.4 The Database Pointer Flag

The database pointer flag enables tuples to be restructured and moved by the RTDBMS during run time.

For example, if an additional attribute is inserted into a relation, e.g., a column is added to a table, it would imply that all tuples belonging to the relation need to be restructured to contain the new data element (the new column). Hence, the size of the tuples changes, relocation of the tuples to new memory locations is most probable. Since a schema change is performed via the SQL interface, it will use and update the index in the index manager. If one of the affected tuples is also referenced from a database pointer entry, the entry have to be updated with the new physical location of the tuple.

Each database pointer flag that is set in the index structure indicates that the tuple flagged is also referenced by a database pointer. This informs the index manager that if this tuple is altered, e.g., moved, deleted, or changed, the corresponding data pointer entry must be updated accordingly.

7.4 The 2-version database pointer algorithm (2V-DBP)

In section 7.3, it has been assumed that a pessimistic concurrency-control algorithm is used, both for soft (relational) and hard (database pointer) transactions. This approach, as shown in section 7.5, punishes both hard and soft transactions severely. Using a pessimistic concurrency-control algorithm, will either cause hard transactions to be blocked by the soft transactions due to lock conflicts, or cause excessive abortions of the soft transactions (especially under 2PL-HP).

The 2V-DBP algorithm allows hard database transactions to execute without being blocked by soft database transactions. Furthermore, soft transactions, using the relational part of the RTDBMS are allowed to execute without being blocked or aborted by the hard database transactions. To achieve this behavior, two versions of all data elements pointed out by database pointers must exist in the database in a similar way as in the two-version priority ceiling protocol proposed by Kuo, Kao, and Shu [19].

The behavior, at a high level of abstraction, of 2V-DBP is discussed in sections 7.4.1 to 7.4.4, while the underlying versioning algorithm that ensures the desired behavior is presented in sections 7.4.5 to 7.4.6.

7.4.1 Soft transactions

The soft transactions utilize the relational part of the RTDBMS, and use an extended form of 2PL-HP [8]. Soft transactions pass through the following steps throughout their executions:

1. **The Begin of Transaction step (BOT)** in which the transaction becomes active.
2. **The lock-obtaining step** in which the transaction obtains all locks necessary to complete. In 2V-DBP, the set of locks does not have to be defined prior to the BOT of a transaction, i.e., 2V-DBP allow ad hoc queries.
3. **The committing step** in which the transaction starts to write back the updated data elements to the database. Before this step, the transaction might be aborted due to some data conflict. However, when the transaction enters the committing step it cannot be aborted any longer.
4. **The End Of Transaction (EOT) step** in which the transaction releases all locks, and the transaction is completed. When the EOT step has been executed, all changes to the database made by the transaction are made visible to other transactions.

The following rules are applicable for soft transactions:

Rule 1. *Soft transactions can read a data element from the database after having successfully obtained either a read lock or a write lock.*

Rule 2. *Soft transactions can change a value of a data element in the database after having successfully obtained a write lock.*

Rule 3. *All locks needed for completing a soft transaction must be obtained prior to the transaction entering the committing step.*

Rule 4. *Read locks on a particular data element in the database can be held by multiple soft transactions simultaneously, thus read locks are compatible with other read locks for the same data element.*

Rule 5. *A write lock on a particular data element in the database grants a soft transaction exclusive access to the data element, so that no other soft transactions can hold, or obtain, any type of lock on the data element in question during the time the write lock is held. Thus write locks are incompatible with any other lock for the same data element.*

Rule 6. *A transaction takes the database from a consistent state to a new consistent state. This means that during the execution of a soft transaction no changes to the database, caused by the transaction, are visible to other transactions until it finishes EOT.*

Rule 7. *If two soft transactions attempt to obtain a read lock or a write lock, which violate the lock compatibility stated in rule 4 and 5, result in that the transactions are considered to be in conflict with one another.*

7.4.2 Hard transactions

All hard transactions use database pointers. Even though hard transactions can access the same data elements as soft transactions, hard transactions are never blocked by database locks. However, hard transactions take database locks in consideration and access the database differently if the data element is locked, see section 7.4.5.

The following rules are applicable to all hard transactions:

Rule 8. *A hard transaction can either read or write a data element, even if the data element is locked by a soft transaction.*

Rule 9. *A hard transaction can never come in conflict with any other transaction.*

Rule 9 is enforced by making hard transactions non-preemptable, see section 7.4.5.

7.4.3 Transaction conflicts

Since soft transactions might be in conflict with other soft transactions, as stated in rule 7, a policy on how to resolve these conflicts is needed. The following rules are applicable to solve transaction conflicts:

Rule 10. *A soft transaction that has not yet entered the committing step will be aborted by the concurrency control algorithm iff it is in conflict, according to rule 7, with a soft transaction executing with a higher priority.*

Rule 11. *A soft transaction that is in conflict, according to rule 7, with a soft transaction executing at a lower priority which has entered the committing step, will be blocked from execution until the committing transaction has finished its execution.*

Theorem 1. *A database transaction can never enter a state of deadlock caused by conflicts with any other database transaction.*

Proof Since a hard database transaction can never be in conflict with any other transaction (rule 9), conflicts can thus only occur among soft transactions. Transaction conflicts among soft transactions are resolved in two ways; (i) If the conflicting transaction is executing at a lower priority than any other conflicting transaction, and has not yet entered the committing step it is aborted (rule 10), thus resolving the conflict. (ii) If the conflicting transaction is executing at a lower priority than any other conflicting transaction, and has entered the committing step, any conflicting transaction will be blocked until the transaction is completed (rule 11), and thus releasing all it locks. Since a transaction, which has entered the committing step, cannot obtain any further locks (rule 3), it cannot cause any further conflicts with any other transaction. $\therefore$

7.4.4 Transaction serialization and relaxation

The goal of a concurrency control algorithm is to resolve data conflicts between concurrent transactions so that it appears that they are run in sequence, hence transactions are serialized. The traditional notion of serialization is to serialize transactions in the order that they commit, i.e., in the order their updates are visible to other transactions. However, it has been recognized that this notion of serialization is not ideal for accessing real-time data [23], since freshness of data often is more important than maintaining the traditional serialization order.

In 2V-DBP, the following serialization rules apply to transactions:

Rule 12. *A set of executing soft transactions are serialized in the order they perform EOT, thus making their changes visible to other transactions.*

Rule 13. *A hard transaction, reading or writing the value of a data element x, is serialized before all hard transactions reading or writing the value x at a later time. Furthermore, the transaction is serialized before any soft database transaction obtaining a lock on x at a later time.*

Rule 14. *A hard transaction, updating the value of a data element currently locked by a soft transaction, is serialized after that transaction.*

Intuitively, rule 14 implies that if, during the execution of a soft transaction, a hard transaction updating the database is serialized after the soft transaction,

the soft transaction must not update the data element in question. This is because the hard transaction is serialized after the soft transaction, and thus the value produced by the soft transaction is already overwritten by the hard transaction, since logically it was executed after the soft transaction.

Rule 13 and 14 imply a relaxation of the serialization order. Consider the example given in figure 7.5 where transactions T_1 to T_3 execute in the following way:

	Event	Data State	Comment
T_1	BOT	$\{x, y\}$	T_1 starts
T_1	$W\_lock(x)$		T_1 obtains write lock on x, thus obtained a local copy of x.
T_2	$Write(x) \rightarrow x'$	$\{x', y\}$	T_2 pre-empts T_1 and updates x. Since x is write locked by T_1, T_2 is serialized after T_1, according to rule 14.
T_3	$Write(y) \rightarrow y'$	$\{x', y'\}$	T_3 updates y. Since y is not yet write-locked by T_1, T_3 is serialized before T_1, according to rule 13.
T_1	$W\_lock(y')$		T_1 obtains write lock in y, thus obtaining a local copy of x'.
T_1	Execute query		T_1 derives x'' and y''.
T_1	committing		T_1 enters the committing step.
T_1	$\neg(Upd(x''))$		T_1 does not update $x- > x''$, according to rule 14.
T_1	$Upd(y') \rightarrow y''$		T_1 updates y, however this update is not yet visible to other transactions.
T_1	EOT	$\{x', y''\}$	T_1 ends and releases its locks. y'' is now visible to other transactions.

From the example we see that the resulting serialization order is T_3, T_1, and T_2, even though the actual order of commit is T_2, T_3, and T_1. This relaxation of the serialization does, however, not imply that soft transactions read inconsistent data since all transactions, according to rule 6, take the database from one consistent state to another, see section 7.4.6. This serialization approach trades a relaxation of serialization for freshness of data.

7.4.5 Realizing 2V-DBP using versioning

As stated earlier, the RTDBMS maintains two versions of data items that have data pointers to them. These versions are used to realize 2V-DBP, as described by rules 8, 9, and 12–14. The data structures used by 2V-DBP are as follows (depicted in figure 7.6):

- A list of the active soft transactions, where each entry contains the current state (`state`) of the transaction, and a local working copy of the data element (`localVer`).

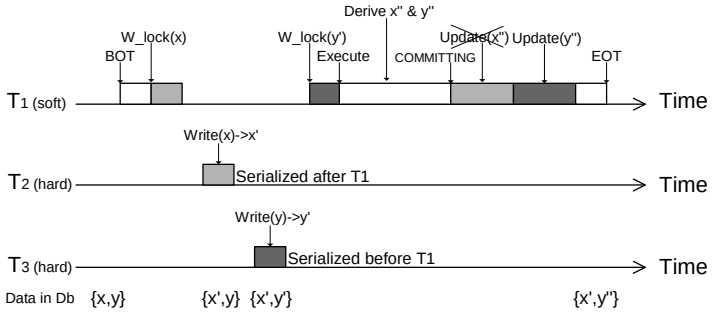


Figure 7.5: An execution-trace of three transactions

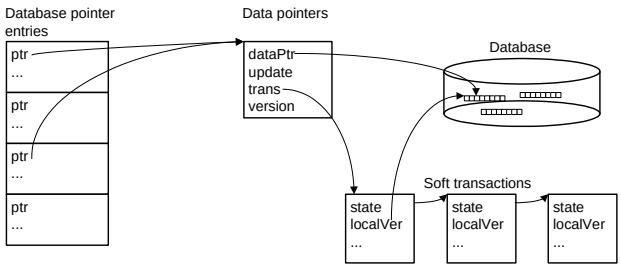


Figure 7.6: The data structures used for versioning

```
1  trans.state=EXECUTING; //BOT
2  For each tuple loop
3    obtainLock(tuple);
4    For each element in tuple loop
5      if (HasDbP(element) and isWriteLocked(tuple))
6         beginATOM();
7           DbP.version=Database.element;
8           DbP.trans=currentTrans();
9           DbP.update=CLEAN;
10          localVer=Database.element;
11        endATOM();
12     else
13        localVer=Database.element;
14     End if
15   End loop
16 End loop
17 //Manipulate tuples
18 trans.state=COMMITTING
19 For each manipulated tuple loop
20   For each element in tuple loop
21     if (HasDbP(element))
22       beginATOM();
23         if (DbP.update==CLEAN)
24           Database.element=localVer;
25         End if
26       endATOM();
27     else
28       Database.element=localVer;
29     End if
30   End loop
31 End loop
32 releaseAllLocks(trans); //EOT
33 trans.state=NO_TRANS;  //EOT
```

Figure 7.7: A soft transaction using 2V-DBP

- A second version of the data element (`version`) as well as a flag (`update`) are added to the data pointer. The `update` flag can have the values `clean` and `dirty`, where the latter indicates that the data has been updated by a hard transaction since it was previously write locked by a soft transaction. Since hard transactions do not use database locks, the `lockInfo` entry presented in figure 7.2 is removed.
- A pointer (`trans`) to any soft transaction holding a write lock on the data element is added to the data pointer.

The implementation of a soft transaction is presented in figure 7.7. First the BOT step is executed (line 1), by setting the state of the transaction to `executing`. The next step, the lock obtaining step, is then executed (lines 2-16) by obtaining a lock for each tuple (line 3). When the lock is granted,

the data element in the tuple is fetched to the local version. If a write locked data element is also pointed out by a database pointer, lines 6-11 are atomically executed, i.e., without being preempted. This atomicity is ensured by the `beginATOM()` and `endATOM()` functions, e.g., by temporarily disabling all interrupts. When fetching the data element, the `version` and the `trans` in the data pointer are also updated. Furthermore, the `update` flag is set to `clean` (line 9), to indicate that no hard transaction has altered the data element since the locking of the tuple. Finally, the data element is read from the database (line 10).

When all data elements of all tuples needed by the transactions are locked and copied to local versions, the transaction executes the query, in which all local versions of the write locked data elements can be manipulated.

The next step, the committing step, is entered (line 18) by changing the state of the transaction. Now the transaction can write all manipulated data elements back to the database (line 19-31) The data elements pointed out by database pointers are only updated if the `update` flag still indicates `clean`. Note that the second version (`version`) is not updated.

In the last step, the EOT step, the transaction releases all its obtained locks (line 32), and changes state to `no_trans` (line 33). The algorithm ensures that no data produced by the transaction is visible to any other transaction prior to this final step. We show formally, in section 7.4.6, that this property is indeed satisfied.

Hard transactions execute entirely in one atomic operation, this implies that BOT and EOT coincide in time. This atomicity is, again, provided by `beginATOM()` and `endATOM()`, see lines 1 and 7 in figures 7.8 and 7.9. Note that in this implementation does not contain any calls to the lock manager as in the original database pointer write and read operations, see figure 7.4. A hard transaction will read the data element in the database if the data element is not write locked by a transaction. Otherwise, it will read the value from the `version` in the data pointer. A hard transaction always writes directly to the data element in the database. However, if it is locked by a write lock, it will also update the `version` in the data pointer, as well as setting the `update` flag to `dirty` to indicate that the data element is now updated after it was write locked by the transaction.

7.4.6 Formal verification of the versioning algorithm

In order to formally show the correctness of the versioning algorithm used in 2V-DBP, we have chosen to use the tool UPPAAL [24] to verify important

```
1  beginATOM();  //BOT
2    if (DbP.trans->state!=NO_TRANS)
3      version=NEW_VALUE;
4      update=DIRTY;
5    End if
6    *(DbP.dataPtr)=NEW_VALUE;
7  endATOM();    //EOT
```

Figure 7.8: A hard write transaction using 2V-DBP

```
1  beginATOM();  //BOT
2    if (DbP.trans->state!=NO_TRANS)
3      localVer=version;
4    else
5      localVer=*(DbP.dataPtr);
6    End if
7  endATOM();    //EOT
```

Figure 7.9: A hard read transaction using 2V-DBP

properties of the algorithm.

One important property to verify is whether or not transactions always read the correct version of a data element, i.e., the value produced by the last serialized transaction updating that particular data element. We refer to this as the durability of transactions. Another equally important property is to verify that no intermediate results produced by executing transactions are visible to other transactions, e.g., verify the consistency of transactions. Finally, we verify that the versioning algorithm is deadlock-free.

UPPAAL is a toolbox for modeling, verification and validation of real-time systems. It is appropriate for systems that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels and (or) shared variables. UPPAAL is designed mainly to check invariant and reachability properties by exploring the state space of a system [24]. UPPAAL provides a graphical interface in which the user graphically models the system using timed automatas. Each transition in an automaton can have guards assigned to them, preventing the system to perform the transition if the condition stated in the guard is not fulfilled. Guards use the same syntax as conditions in `C`, e.g., `op == op`, `op < op`, and `op != op`. If a transition is performed, a (possibly empty) set of assignments is executed, e.g., `op := op`. All operations, e.g., guards, assignments and synchronization, performed during one transition are considered to be one atomic operation which cannot be preempted by other transitions. It is also

possible to use communication channels in which two automatas can perform synchronized transitions. When the real-time system has been modeled, the system can be model-checked using requirement specification queries.

The automatas depicted in figures 7.10 to 7.13 show the behavior of transactions (for a data element pointed out by a data pointer), as modeled in UPPAAL. In the modeling of the system, two states (AFTERREAD_VERIFICATION_STATE in figure 7.10 and 7.12) and one variable (lastCommittedTransaction) have been added. These do not affect the behavior of the model, but are added for verification purposes. Three of the states are marked with the letter "C", which, in UPPAAL, indicates that the state is committed, i.e., the automata must immediately move to the next state.

The hard write, the hard read, and the soft read operations are a direct translation of the pseudo programs presented in section 7.4.5. The soft write transactions presented in figure 7.10, however, deserve further explanation:

1. The transition from BOT to EXECUTING (The AFTERREAD_VERIFICATION_STATE is disregarded for now) corresponds to the lock obtaining step of the transaction.

2. The transition from EXECUTING to COMMITTING_NOT_WRITTEN_DATA correspond with the transaction entering the committing state.

3. In the COMMITTING_NOT_WRITTEN_DATA the decision whether or not to update the database is taken based upon the value of the update flag.

4. The transition from COMMITTING_HAS_WRITTEN _DATA to EOT corresponds to lines 32-33 in figure 7.7.

In our verification we parallel compose the four automatons. This configuration is minimalistic but sufficient in order to capture all possible types of interactions where hard and soft transactions can interfere with each other (i.e., soft read vs. hard write, soft write vs. hard write, soft write vs. hard read, and all possible orderings of these pairs of interactions).

In this verification, three properties are verified, (i) the algorithm is deadlock-free, (ii) the value written by the last committed transaction is the value read by other transactions, i.e., the durability of transactions, and (iii) the consistency of transactions. The syntax of the queries in UPPAAL are not explained in this paper, instead we refer to [24] for a detailed description of the UPPAAL syntax. In the verification we denote soft read transactions as SR, and hard read transactions as HR.

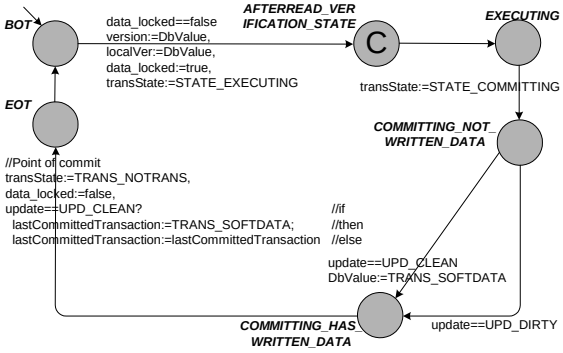


Figure 7.10: Automata for a soft write transaction

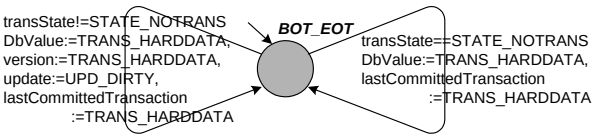


Figure 7.11: Automata for a hard write transaction

The versioning algorithm is deadlock-free This property is trivial to check, since UPPAAL already has a mechanism to verify this. The result of the query `A[] not deadlock` showed that the algorithm is deadlock-free.

Durability of transactions To verify this property, four queries are used, namely:

1. `A[] ((SR.AFTERREAD_VERIFICATION_STATE and SR.localVer==TRANS_HARDDATA) imply (lastCommittedTransaction==TRANS_HARDDATA))`
   The query can be interpreted as "If a soft read transaction has read a value produced by a hard transaction, then it is always the case that the latest serialized transaction was a hard transaction?"

2. `A[] ((SR.AFTERREAD_VERIFICATION_STATE and SR.localVer==TRANS_SOFTDATA) imply (lastCommittedTransaction==TRANS_SOFTDATA))`
   The query can be interpreted as "If a soft read transaction has read a value produced by a soft transaction, then it is always the case that the latest serialized transaction was a soft transaction?"

3. `A[] ((HR.EOT and SR.localVer==TRANS_HARDDATA) imply (lastCommittedTransaction==TRANS_HARDDATA))`
   The query can be interpreted as "If a hard read transaction has read a value produced by a hard transaction, then it is always the case that the latest serialized transaction was a hard transaction?"

4. `A[] ((HR.EOT and SR.localVer==TRANS_SOFTDATA) imply (lastCommittedTransaction==TRANS_SOFTDATA))`
   The query can be interpreted as "If a hard read transaction has read a value produced by a soft transaction, then it is always the case that the latest serialized transaction was a soft transaction?"

The verification showed that all four queries were fulfilled.

Consistency of transactions This property was implicitly verified when the durability property was verified, since if the value visible to transactions always originate from the last committed transaction, no data can be visible from uncommitted transactions.

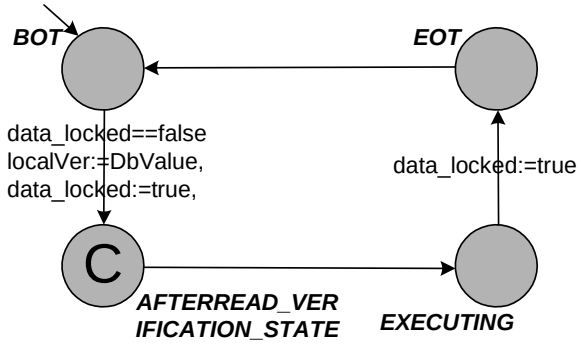


Figure 7.12: Automata for a soft read transaction

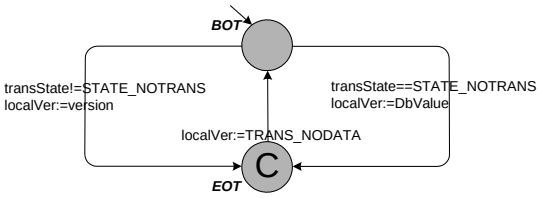


Figure 7.13: Automata for a hard read transaction

7.5 Performance evaluation

We have performed a performance evaluation of database pointers. The goal of the evaluation is to illustrate, for a synthetic but realistic scenario, the positive impact 2V-DBP has, comparing it to traditional pessimistic concurrency control used in section 7.3. Specifically, the goal is to demonstrate that 2V-DBP provides significant benefits for *both* soft and hard transactions, illustrating that 2V-DBP do not represent a tradeoff between good service for either soft or hard transactions.

To evaluate the performance of the 2V-DBP algorithm, we compared it with the 2PL-HP algorithm for both soft and hard transactions. 2PL-HP is a well-known pessimistic concurrency control algorithm which can be implemented on all priority-based operating systems. To realize this evaluation, a real-time system executing soft relational transactions and hard database pointer transactions was implemented on the Asterix real-time kernel [25]. These tests were then executed on a standard PC with an Intel Pentium 350MHz processor.

The Asterix real-time kernel is an operating system, using fixed priority scheduling, intended for small embedded applications. Since Asterix supports pre-emptable scheduling, semaphores are needed to ensure task synchronization. In Asterix, semaphores are implemented using the immediate inheritance protocol [26]. The interrupt latency of the kernel, executing on the computer used in these tests, is in the order of $20\mu s$.

The RTDBMS in the test consists of 300 tuples with four data elements each. Also 300 randomly selected data elements are pointed out by database pointers. Every $400ms$ a soft transaction is launched into the system. Each soft transaction randomly write- and/or read locks up to 200 tuples. Also, every $20ms$, a hard transaction is launched. The hard transaction executes either a read or a write operation on a randomly selected database pointer.

This transaction schedule mimics a hard real-time vehicle control system with numerous hard I/O and control tasks, as well as a number of management tasks executing data intensive soft transactions. It is, furthermore, fair to assume that for RTDBMSs residing in vehicle control systems, a significant part of the database is accessed by hard transactions, since most execution in these systems would involve the controlling of the vehicle, hence the high amount of database pointers.

In figure 7.14 the mean abortion ratios for both 2V-DBP and 2PL-HP is presented. The system's abortion rate is sampled with an interval of two seconds, and the samples (indicated by boxes and diamonds in figure 7.14) show the mean abortion ratio for each interval. The comparison shows that the to-

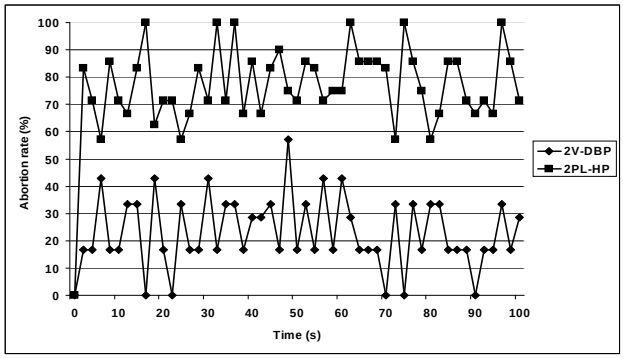


Figure 7.14: Abortion-ratios for soft transactions

tal mean abortion ratio for 2V-DBP is 25%, compared to approximately 75% for 2PL-HP. It should be noted that all abortions for 2V-DBP are induced by soft transactions aborting other soft transactions, and that no transactions are aborted because of conflicts with hard transactions.

Figure 7.15 shows the response times for hard transactions executing under 2PL-HP. Roughly, the response times can be grouped into three classes, namely:

- Transactions executed without interference from other transactions. More than 95% of the hard transactions falls into this class. These transactions have a response time similar to the 2V-DBP case, i.e., 5-8μs. However, a fraction of these have also suffered from the kernel timer interrupt.

- Transactions causing soft transactions to be aborted. This implies that the hard transaction must execute the abort transaction procedure before continuing. These transactions, which are just above 4% of all hard transactions, have an execution time of 95μs.

- Transactions suffering from priority inversion. Due to the use of a common semaphore to administrate the 2PL-HP lock tables these transactions have been blocked by soft transactions. The execution times of these transactions range up to $\sim$180μs. Only a small fraction (0.1%) of all transactions fall into this class.

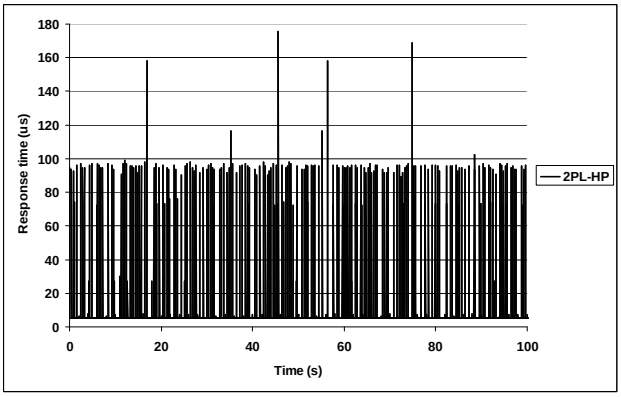


Figure 7.15: Response-times for hard transactions using 2PL-HP

Figure 7.16 shows the corresponding response times for the hard transactions executing under 2V-DBP. The figure shows that all 5000 transactions launched during the 100 second test interval, but a handful ($\sim$20 transactions) have constant execution time (5 μs). The remaining transactions have suffered from latency caused by the kernel, i.e., a timer interrupt has occurred during the execution of the transaction.

The measurements taken from these two execution cases show that 2V-DBP outperforms 2PL-HP, both with respect to a minimized amount of aborted transactions, as well as constant execution times for hard transactions. This shows that 2V-DBP is a suitable approach to manage hard transactions in real-time control systems, since it provide high throughput of soft transaction, as well as short constant execution times for hard transactions.

7.5.1 Memory overhead of 2V-DBP

Another important issue for embedded systems is the memory overhead. In most cases there is a clear trade-off between functionality and resource allocation; 2V-DBP is no exception. An important property of 2V-DBP however, is that the memory consumption of 2V-DBP is bounded and predictable. We illustrate this with the example presented in section 7.5.

This database would, for an average length of each data element of 2 bytes, add up to 2,3kb (300 tuples * 4 data elements * 2 bytes). To structure the data,

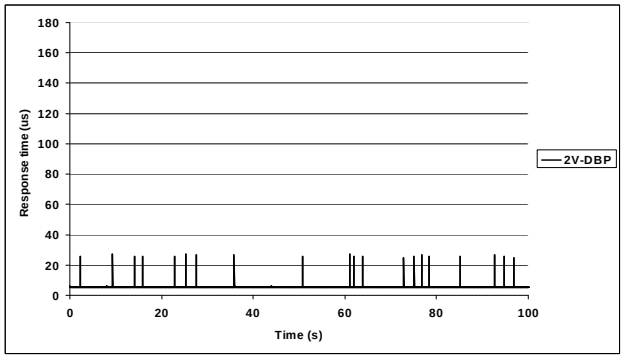


Figure 7.16: Response-times for hard transactions using 2V-DBP

a RTDBMS needs to use additional memory, e.g., overhead used to index data and to store relation information. The commercial Polyhedra embedded database management system [11] has an overhead of 28 bytes/tuple [27]. Using this RTDBMS for the database would add 8,2 KB of overhead. Adding 300 database pointers that use locking would imply an extra overhead of 2,7kb (if pointers use 4 bytes, integers use 2 bytes, and the lock information is stored in an integer). On the other hand, if 300 database pointers use 2V-DBP instead of locking, would imply an overhead of no more than 3,6kb.

This implies that for a system that uses 2PL-HP the total memory consumption would be 13,2 KB. The memory consumption for the same system using 2V-DBP would be 14,1 KB, i.e., 2V-DBP increases the overhead by 6,8 %.

7.6 Conclusions and future work

We have presented a database access concept consisting of an data access mechanism entitled database pointers, and a concurrency control algorithm that allows co-existence of soft real-time, relational database transactions (soft transactions) and hard real-time database pointer transactions (hard transactions) [6] in a Real-Time Database Management System (RTDBMS). The algorithm, called 2-Version DataBase Pointer concurrency control (2V-DBP) uses, traditional, pessimistic concurrency control for soft transactions and a simplified form of versioning, i.e., we maintain multiple (in our case two) versions of

data accessed by database pointers [6].

2V-DBP supports soft transactions with long execution times without risking that soft transactions are aborted by high priority hard transactions. Thus, 2V-DBP overcomes the recognized problem that transactions with low priority and long execution times are penalized due to the likeliness of data conflicts, resulting in frequent aborts [9].

2V-DBP supports hard transactions without risking hard transactions being delayed by long-running soft transactions. Such delays could otherwise be the case even if *high priority abort* is employed, since (i) abort of soft transactions is itself a time-consuming procedure, and (ii) once a soft transaction reaches the commit state it can no longer be aborted. Also, database pointers ensure fast and deterministic access to data elements, allowing access to the database without consulting the RTDBMS indexing system.

We have proved that 2V-DBP is free of deadlocks and formally verified that the versioning algorithm provides consistency and durability of transactions. Unlike traditional versioning algorithms for databases [20], the 2V-DBP uses a bounded number of versions: two versions for data that are accessed by database pointers, other data uses one single version.

We have implemented 2V-DBP and compared it to the pessimistic concurrency-control algorithm *two-phase locking high priority* (2PL-HP). Our comparison scenario shows that the abortion ratio was significantly decreased, from an average of 75% using 2PL-HP to 25% using 2V-DBP. Furthermore, the worst observed response-time for hard transactions was greatly reduced from about 175μs to 27μs (of which 20μ is the system interrupt latency; the actual execution times for hard transactions were always in the range 5-7μs). Thus, we conclude that both hard and soft transactions benefit from the 2V-DBP algorithm. The cost for introducing 2V-DBP is slightly increased memory overhead for maintaining internal data structures and one extra version of the data elements used by hard transactions. For an example database, the overhead of using 2V-DVP instead 2PL-HP was calculated to 6,8%.

The database access concept presented in this paper show that efficient integration of a real-time database is possible, even in safety-critical embedded real-time control-system, without adding unpredictability to the system.

Bibliography

[1] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano - a Revolution in On-Board Communications. Technical report, Volvo Technology Report, 1998.

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

[3] Dag Nyström, Aleksandra Tešanović, Mikael Nolin, Christer Norström, and Jörgen Hansson. COMET: A Component-Based Real-Time Database for Automotive Systems. In *Proceedings of the Workshop on Software Engineering for Automotive Systems*, pages 1–8. The IEE, June 2004.

[4] T. Gustafsson and J. Hansson. Data management in real-time systems: a case of on-demand updates in vehicle control systems. In *Proceedings of the Real-Time Application Symposium (RTAS 2004)*. IEEE Computer Society Press, May 2004.

[5] Stephen Cannan and Gerhard Otten. *SQL - The Standard Handbook*. MacGraw-Hill International, 1993.

[6] Dag Nyström, Aleksandra Tešanović, Christer Norström, and Jörgen Hansson. Database Pointers: a Predictable Way of Manipulating Hot Data in Hard Real-Time Systems. In *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications*, pages 623–634, February 2003.

[7] Dag Nyström, Mikael Nolin, Aleksandra Tešanović, Christer Norström, and Jörgen Hansson. Pessimistic Concurrency Control and Versioning to

Support Database Pointers in Real-Time Databases. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 261–270. IEEE Computer Society, June 2004.

[8] R.K Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *ACM Transactions on Database Systems*, 17, September 1992.

[9] J. Huang, J.A. Stankovic, K. Ramamritham, and D.F. Towsley. Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 35–46. Morgan Kaufmann, September 1991.

[10] Pervasive Software Inc. http://www.pervasive.com.

[11] Enea AB. http://www.enea.se.

[12] Sleepycat Software Inc. http://www.sleepycat.com.

[13] TimesTen Performance Software. http://www.timesten.com.

[14] S. F. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring. DeeDS Towards a Distributed and Active Real-Time Database System. *ACM SIGMOD Record*, 25(1):38–40, 1996.

[15] J. Lindstrom, T. Niklander, P. Porkka, and K. Raatikainen. A Distributed Real-Time Main-Memory Database for Telecommunication. In *Proceedings of the Workshop on Databases in Telecommunications*, pages 158–173. Springer, September 1999.

[16] B. Adelberg, B. Kao, and H. Garcia-Molina. Overview of the STanford Real-time Information Processor (STRIP). *SIGMOD Record*, 25(1):34–37, 1996.

[17] J. A. Stankovic, S. H. Son, and J. Liebeherr. *Real-Time Databases and Information Systems*, chapter BeeHive: Global Multimedia Database Support for Dependable, Real-Time Applications, pages 409–422. Kluwer Academic Publishers, 1997.

[18] F. Baothman, A. K. Sarje, and R. C. Joshi. On Optimistic Concurrency Control for RTDBS. In *Proceedings IEEE Region 10 International*

Conference on Global Connectivity in Energy, Computer, Communication and Control, volume 2, pages 615–618. IEEE Computer Society, December 1998.

[19] Tei-Wei Kuo, Yuan-Ting Kao, and LihChyun Shu. A Two-Version Approach for Real-Time Concurrency Control and Recovery. In *Proceedings of the Third IEEE International High Assurance Systems Engieering Symposium*. IEEE Computer Society, November 1998.

[20] Rajeev Rastogi, S. Seshadri, Philip Bohannon, Dennis W. Leinbaugh, Abraham Silberschatz, and S. Sudarshan. Logical and physical versioning in main memory databases. In *The VLDB Journal*, pages 86–95, 1997.

[21] X. Song and J. Liu. Maintaining Temporal Consistency: Pessimistic vs. Optimistic Concurrency control. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):786–796, October 1995.

[22] P. S. Yu, K. Wu, K. Lin, and S. H. Son. On Real-Time Databases: Concurrency Control and Scheduling. *Proceedings of the IEEE*, 82(1):140–157, January 1994.

[23] Tei-Wei Kuo and Aloysius K. Mok. SSP: a Semantics-Based Protocol for Real-Time Data Access. In *Proceedings of 14th IEEE Real-Time Systems Symposium*, pages 76–86. IEEE Computer Society, December 1993.

[24] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[25] Henrik Thane, Anders Pettersson, and Daniel Sundmark. The Asterix realtime kernel. In Eduardo Tovar and Christer Norström, editors, *Proceedings of the Work-in-progress and Industrial Session of the 13th Euromicro Conference on Real-Time Systems,Delft Netherlands*. http://citeseer.nj.nec.com/thane01asterix.html, June 2001.

[26] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*, chapter 13.10.1 Immediate Ceiling Priority Inheritance. Addison-Wesley, second edition, 1996. ISBN 0-201-40365-X.

[27] Aleksandra Tešanović, Dag Nyström, Jörgen Hansson, and Christer Norström. Embedded Databases for Embedded Real-Time Systems: A Component-Based Approach. Technical Report MRTC Report ISSN

1404-3041 ISRN MDH-MRTC-43/2002-1-SE, Dept. of Computer Engineering, Mälardalen University, January 2002.

Chapter 8

Paper D: Snapshots in Real-Time Databases using Database Pointer Transactions

Dag Nyström, Mikael Nolin, and Christer Norström

In Proceedings of the 11th IEEE International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA05), Hong Kong, China, August 2005

Abstract

We present 2V-DBP-SNAP, an algorithm that allows hard real-time tasks in an embedded real-time control system to read a snapshot of a number of data elements in a real-time database. Furthermore, 2V-DBP-SNAP allows these data elements to be shared with soft real-time tasks, which access them using a database query language, and with other hard real-time tasks that use database pointers. 2V-DBP-SNAP allows temporal behavior and memory consumption to be accurately predicted. Introducing snapshot transactions is beneficial for embedded control-systems, such as for engine control in an automotive system, since a snapshot of the state of the environment can be collected, e.g., the state of all cylinders in the engine. 2V-DBP-SNAP is lightweight and predictable, both with respect to computational and memory overhead, and is therefore highly suited for resource constrained systems.

8.1 Introduction

As complexity in automotive control systems increases [1], the need for a structured way to handle and maintain data is needed [2]. One attractive solution is to integrate the data into a real-time database management system (RTDBMS) [3, 4, 5]. Such an RTDBMS must be able to handle concurrently executing transactions in an efficient and predictable way.

In [6], a concurrency control algorithm, denoted *2-Version Database Pointer Concurrency Control algorithm* (2V-DBP) that allows hard and soft tasks to access shared data, was introduced. The algorithm, which combines the concept of database pointers [7] and relational transaction management satisfies the need for predictable and time-efficient hard real-time control-applications. However, one drawback with 2V-DBP is that it doesn't allow hard transactions to access more than one data element per transaction, thus tasks that need to access multiple data elements cannot obtain atomic transaction semantics, since each data access would have to be in a separate transaction. For a majority of control tasks in a vehicular system, using multiple transactions is sufficient, and preferable, since such an approach favors freshness of data over atomic data access. However, in automotive systems, a minor part of the control-tasks need atomic transactions. Consider an engine-controller in which you need to have a consistent view of the state of all cylinders in the engine. To allow this, we extend the 2V-DBP algorithm to support *snapshots* [8]. We call this algorithm *2-version database pointer concurrency-control with snapshots* (2V-DBP-SNAP).

In this paper, we present and evaluate the 2V-DBP-SNAP concurrency control algorithm, which has the following benefits: (i) It allow hard tasks to access non-snapshot data in the database without being blocked by any database locks. (ii) It overcomes the widely recognized problem that soft transactions with low priority and long execution times are penalized due to the likeliness of data conflicts [9]. (iii) It allows hard tasks to access a set of data, using snapshot transactions, to be read and/or written atomically. Blocking for snapshot transactions is deterministic and bounded. (iv) It is well suited for embedded real-time systems since it is both resource-efficient and predictable.

8.2 System model

This paper focuses on real-time applications used to control a process, e.g., critical control-functions in a vehicle such as engine or brake control. The

tasks use database transactions to access and manipulate shared data stored in a database.

8.2.1 Snapshots

We define snapshots and their operations as follows:

Definition 1. *A **snapshot requirement** on a set of data elements is the requirement that the data elements must be able to be read as an consistent instant (**snapshot**) view of the data elements.*

Definition 2. *We refer to a set of data elements that have a snapshot requirement as **snapshot data**.*

Definition 3. *We define an operation as having a **snapshot property** if the operation accesses snapshot data such that the snapshot requirement for the data is maintained.*

Intuitively, Definition 1 implies that any operation that accesses data within a snapshot must be able to retrieve a snapshot of the state of the data elements, even if the data elements currently are updated by some other operation. Furthermore, Definition 3 implies that if an operation, e.g., a database transaction, accesses data elements within snapshot data (any combination of reads and writes) in such a way that the snapshot requirement is maintained (for any other operation on that particular data), the operation has a snapshot property. Introducing concurrently executing transactions often makes it impossible to handle snapshot data, even when well known traditional concurrency-control algorithms, such as 2-phase locking [10] and optimistic concurrency control [11], are used. Previous work on snapshot semantics in RTDBMSs involves keeping multiple versions of data to achieve snapshot properties for all transactions and all data in the database [12]. However, the overhead, with respect to memory and CPU utilization, for managing these versions is often unacceptable in resource constrained systems. In 2V-DBP-SNAP, individual sets of data elements with snapshot requirements are identified, and only one extra version of each data element in the set is needed.

8.2.2 Task and transaction models

From a conceptual point of view, we divide the tasks in the system into three categories, namely, I/O-tasks, control-tasks, and management-tasks [2].

I/O and control-tasks are hard real-time tasks, typically executed periodically, and often at high frequencies. I/O-tasks are used to either read sensor-values or update actuators, while control tasks are used to derive new actuator values from sensor values. These tasks normally touch very few data elements, and their transactions are always *precompiled*, (i.e., formulated at compile-time). In some cases, I/O and control tasks might use snapshot data, e.g., all cylinders of an engine may need to be read and controlled atomically. I/O and control tasks not using snapshot data interact with the database through *hard database transactions*. These transactions utilize the database pointer interface, providing an efficient and predictable access to a single data element in the database. A majority of the transactions in a vehicle, i.e., vehicle control, would fall into this class. Similarly, I/O and control-tasks that use snapshot data, interact with the database through *snapshot transactions*. These transactions also utilize the database pointer interface, but are allowed to access multiple data elements. Since snapshot transactions are more complex than hard transactions, and the need for snapshot data only applies to a limited number of tasks, one could expect few transactions of this class.

Management tasks are soft real-time tasks, which execute *soft database transactions*. An example of a management task might be a task presenting statistical information about the current state of the vehicle to the user. A management transaction might also be constructed during run-time, for example by a service technician using a service tool connected to the vehicle. We assume that all soft (i.e., management) tasks execute on lower priorities than all hard (i.e., I/O and control) tasks. Soft transactions, either precompiled or ad hoc (formulated and parsed at run-time), utilize the relational database query interface, e.g., SQL-interface [13], for access. They provide a flexible access to the data in the database to the system and are especially suited for management tasks, e.g., logging, diagnostics, and user interface (HMI) tasks, e.g., tasks controlling the instrument board.

8.3 Database pointers with versioning

Before presenting our proposed concurrency control algorithm, we will recapitulate the database pointer concept [7], and the concurrency control algorithm, denoted 2V-DBP [6], which our algorithm is extending.

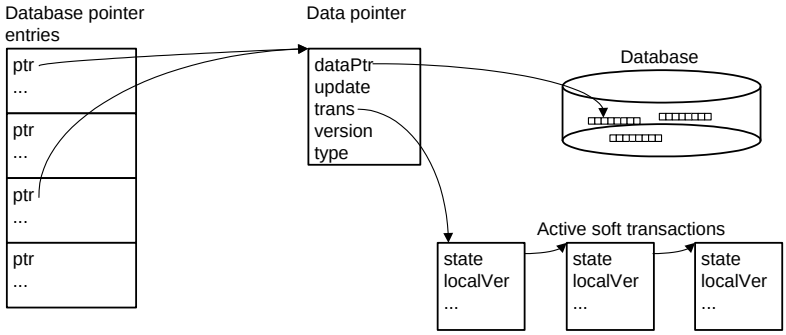


Figure 8.1: The data structures used by database pointers under 2V-DBP.

8.3.1 Database pointers

Database pointers allow individual data elements in a RTDBMS to be accessed in an efficient and predictable manner and are implemented using the following data structures (Figure 8.1):

- **The database pointer entry** which is a data-structure that belongs to each individual database pointer. The address of this structure is returned to the application when the database pointer is bound to its data element.
- **The data pointer** which contains a pointer to the data element in the database, as well as information on the type of the data, e.g., an integer or a string. The `version`, `update`, and `trans` entries are used by 2V-DBP, and will be discussed in Section 8.3.2. Using a two-level pointer indirection instead of allowing the application to directly have a pointer to the data element ensures that the database can be reorganized during run-time without creating stale pointers in the application.

8.3.2 The 2-version database pointer algorithm

The 2V-DBP algorithm allows hard database transactions to execute without being blocked by soft database transactions [6]. Furthermore, soft transactions, using the relational part of the RTDBMS, are allowed to execute without being blocked or aborted by the hard database transactions. To achieve this behavior,

two versions of all data elements pointed out by database pointers exist in the database in a similar way as in the two-version priority ceiling protocol [14], and the two-version two phase locking [15]. To maintain the versions of data elements, the following data structures are used (see Figure 8.1):

- **A second version of the data element** (`version`) and a flag (`update`) which indicates if the second version is dirty (updated by a hard transaction) or clean.
- **A list of all active soft transactions** where each entry consists of, among other information, the current state (`state`, see Section 8.3.3) of the transaction, and local working copies of the data elements used by each transaction (`localVer`).
- **The `trans` pointer**, residing in the data pointer, points out any soft transaction holding a write lock on the data element. It is used to ensure that soft transactions commit atomically.

8.3.3 Soft transactions

The soft transactions utilize the relational part of the RTDBMS, and use an extended form of the *two-phase locking high priority* 2PL-HP [16] concurrency control policy.

A soft transaction has the following execution workflow; When the query has been parsed, all database locks needed to complete the query is obtained. Arbitration of any lock-conflicts are solved according to 2PL-HP, i.e., in favor of the higher prioritized transaction. When a lock is granted, the data is immediately copied to the local working copy and the `update`-flag is set to clean. When all data is retrieved, the data is manipulated according to the query. After that the transaction commits and all manipulated data are written back to the database according to the 2V-DBP versioning scheme [6], except those data elements that are dirty (determined by the `update`-flag, see Section 8.3.4). This behavior ensures an atomic commit. When the manipulated data is written back, the transaction returns its locks.

8.3.4 Hard transactions

All hard transactions use database pointers. Hard transactions can access the same data elements as soft transactions, however hard transactions are never

blocked by database locks. Still hard transactions take database locks in consideration and access the database differently if the data element is locked.

Hard transactions that write to the database will, if the data is write locked by a soft transaction, set the `update` flag to `dirty` to indicate that the data element is not to be overwritten by the soft transaction. The execution-time of a hard transaction is short and close to constant, since it only contains a handful of sequential instructions. Hence, hard transactions are in their entirety executed non-preemptive. Experiences from implementing 2V-DBP show that the duration of these non-preemptive sections are significantly shorter than the interrupt latency of typical real-time operating systems. From a schedulability analysis perspective, the operating system itself introduces a greater blocking, and a hard transaction do therefore not influence the maximum blocking in the system.

8.3.5 Transaction serialization and relaxation

The goal of a concurrency control algorithm is to resolve data conflicts between concurrent transactions so that it appears that they are run in sequence, hence transactions are serialized. The traditional notion of serialization is to serialize transactions in the order that they commit, i.e., in the order their updates are visible to other transactions. However, it has been recognized that this notion of serialization is not ideal for real-time data [17], since freshness of data often is more important than maintaining the traditional serialization order.

In 2V-DBP, for each soft transaction, a feasible serialization of all transactions can be found. However, different soft transactions can have different perceptions of the actual serialization order. Thus, 2V-DBP introduces a relaxed serialization order that favors data freshness. As an example of our serialization method, consider the execution trace depicted in Figure 8.2. In the figure we can see that the hard transaction t_2 is serialized *after* t_1 since it updated its data element (x) *after* t_1 locked (and thus read) it. Note, that t_1 did not write x back to the database during the commit phase since it was dirty. Similarly, t_3 is serialized *before* t_1 since it updated its data element (y) *before* t_1 read it. From the example we see that the resulting serialization order is t_3, t_1, and t_2, even though the actual order of commit is t_2, t_3, and t_1.

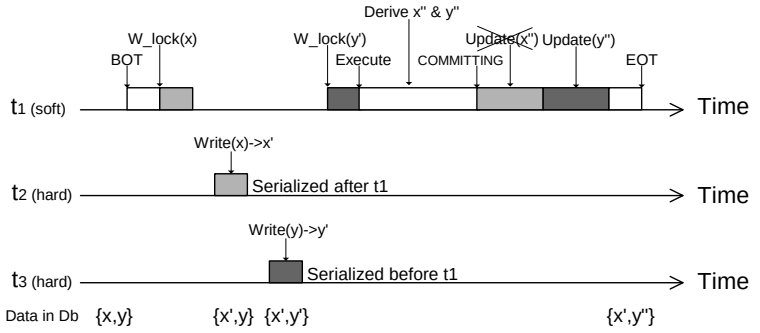


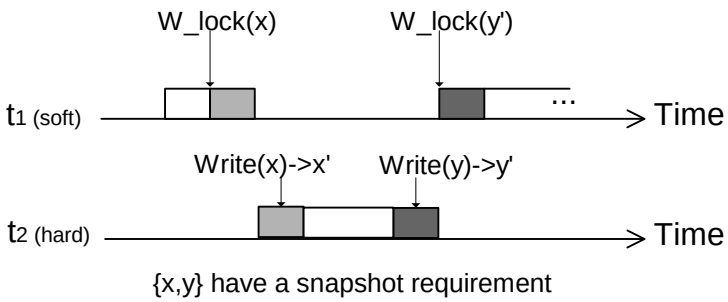
Figure 8.2: An execution-trace of three transactions

8.4 The 2-version database pointer snapshot algorithm

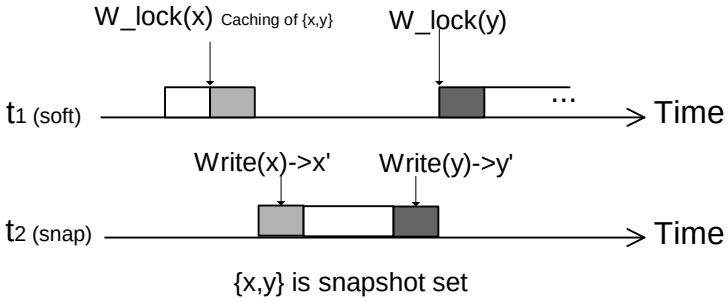
In its current form, 2V-DBP is not sufficient to support the notion of snapshot data just by allowing hard transactions to read or update multiple data elements. Consider the example depicted in Figure 8.3(a) in which the hard transaction t_2, preempting the soft transaction t_1, is allowed to update multiple data elements. In this example, t_1 reads x and y', hence t_1 only sees partial results of the updates performed by t_2. The two correct scenarios in this example would be that t_1 either reads x and y (t_1 is serialized before t_2), or that it reads x' and y' (t_1 is serialized after t_2). In this example, the snapshot requirement of $\{x, y\}$ is violated.

From the above example we see that introducing hard transactions with snapshot properties need a more elaborate handling, therefore we introduce the 2V-DBP-SNAP algorithm, which is an extension to 2V-DBP, that introduces a third transaction-class, namely the snapshot transaction. Snapshot transactions are allowed to read and/or update several database pointers during their execution, and are guaranteed to have a snapshot property.

The main idea of 2V-DBP-SNAP is to cluster data that have snapshot requirements into *snapshot sets*. A snapshot set is defined as a set of data elements with a common snapshot requirement, i.e., is used by one or more snapshot transactions. In Figure 8.4, three snapshot sets (ab, x, and y) can be seen. Note that snapshot transactions a and b have a common data element, thus they share the same snapshot set. A snapshot transaction is only allowed



(a) Faulty trace under 2V-DBP



(b) Correct trace under 2V-DBP-SNAP

Figure 8.3: Execution-traces for transactions using snapshot data

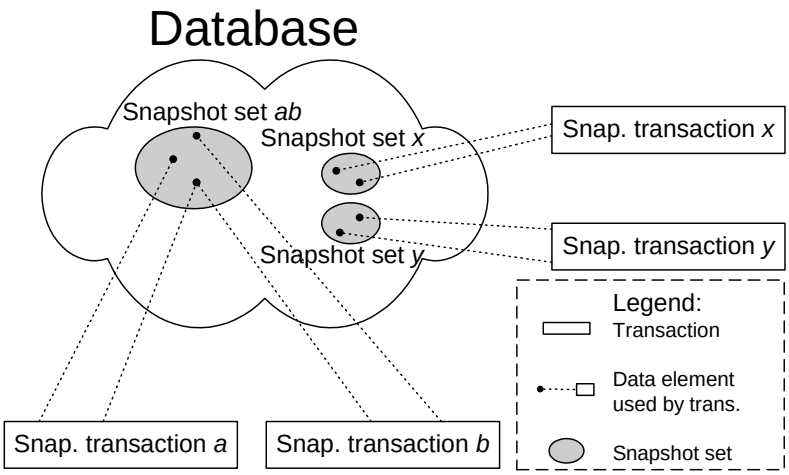


Figure 8.4: Snapshot sets in 2V-DBP-SNAP

to access data elements within its snapshot set. When a soft transaction locks a data element within a snapshot set, it must cache the entire snapshot set to its local working copy before it may continue to execute.

The result of using 2V-DBP-SNAP on the faulty example in Figure 8.3(a) is depicted in Figure 8.3(b). In this case, the soft transaction t_1 caches both x, and y, when locking x, and then use the cached value, when locking y. In the example we see that t_1 no longer sees a partial result of t_2 and the snapshot requirement of $\{x, y\}$ is now fulfilled.

8.4.1 Snapshot sets

Partitioning data into snapshot sets is a straightforward procedure, which can be easily automated. The algorithm works as follows:

Definition 4. *Let s_x denote the set of data elements accessed by snapshot transaction x.*

The algorithm passes through the following two steps:

1. Let **set** S be the set of all data elements sets, thus $S = \{s_1, s_2, ..., s_n\}$.

2. While there exist pairs of data elements sets (s_x, s_y) such that $s_x \in S, s_y \in S$ and $s_x \cap s_y \neq \varnothing$: Remove s_x and s_y from S and add a new data elements sets $s_{xy} = s_x \cup s_y$ to S.

S now contains all snapshot sets (which are all disjoint sets of data elements) of the database.

Both steps of the algorithm are straightforward since all snapshot transactions are precompiled, and thus all data elements possibly accessed are known at compile-time. Partitioning data into snapshot sets is typically performed off-line. The algorithm can, however, be applied during run-time. This would allow the system to create new snapshot transactions during run-time. However, we will not elaborate further on this in this paper.

8.4.2 The 2V-DBP-SNAP data structures

The 2V-DBP data structures from Figure 8.1 must be extended to incorporate the snapshot sets. This is done by adding a new data structure, the *snapshot pointer*, which have a similar functionality as the data pointer described in Section 8.3. In difference from the data pointer, which maintains a single data element, a snapshot pointer handles sets of data, i.e., all data within one snapshot set. The snapshot pointer contains the following entries, see Figure 8.5:

- **The** `dataPtr set`**,** `version set`**,** `update set` **and** `type set` **entries.** These entries are equivalent to the `dataPtr`, `version`, `update` and `type` entries in the data pointer, but are extended to sets with one element per data element in the snapshot set.

- **The** `trans` **entry.** This entry is similar to the `trans` entry in the data pointer, and contains a pointer to any soft transaction currently holding a *snapshot lock* on the snapshot set. Snapshot locks are ordinary database locks, maintained by the RTDBMS lock manager [5], that are used by soft transactions when accessing a snapshot. The usage of snapshot locks is further explained in Section 8.4.5.

- **The** `snapshot semaphore` **entry.** This entry is used to enforce mutual exclusion among snapshot transactions accessing the same snapshot set, The usage of snapshot semaphores is further discussed in Section 8.4.3.

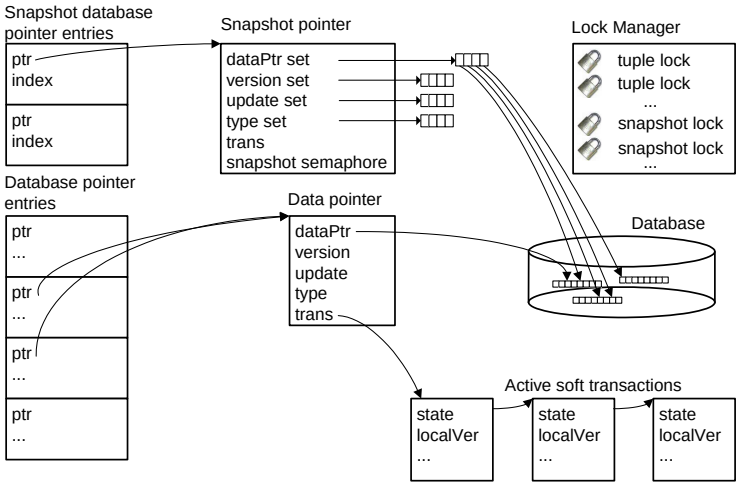


Figure 8.5: The extended data structures used by 2V-DBP-SNAP.

8.4.3 Introducing snapshot transactions

A snapshot transaction operates in a similar fashion as hard transactions, with the difference that a snapshot transaction can read and manipulate multiple data elements. Snapshot transactions, as hard transactions, use the database pointer interface to access data elements. In addition to this a `begin of transaction` and an `end of transaction` are used to indicate what part of the application code is within the transaction. The work-flow of a snapshot transaction is as follows:

- **1. Begin of transaction.** In this step, the transaction is started. It consists of two steps, namely:(i) The snapshot semaphore is obtained, in order to ensure mutual exclusion among all snapshot transactions accessing the same snapshot set. (ii) Identify if a soft transaction is currently holding a write-lock on the snapshot set, see Section 8.4.5. If this is the case, the `update`-flag must be set to `dirty` for all data elements manipulated by the transaction. This is similar as for hard transactions executing under 2V-DBP.

- **2. Execute the transaction.** That is, read and write to any data elements

within the snapshot set. Calculations to derive the transaction results can also be executed.

- **3. End of transaction.** In this step, the transaction is ended, and the snapshot semaphore is released.

The snapshot semaphore might introduce blocking on other, higher prioritized snapshot transactions executing on the same snapshot set. This blocking is, however, deterministic if a real-time semaphore protocol, such as the immediate inheritance protocol is used [18], and if the worst case execution time is bounded for each snapshot transaction. It is noteworthy that this blocking affects neither snapshot transactions accessing other snapshot sets, hard transactions, nor soft transactions.

8.4.4 Hard transactions under 2V-DBP-SNAP

Hard transactions under 2V-DBP-SNAP can only modify data elements that are not in any snapshot set. However, it is possible for a hard transaction to read a data element from a snapshot set. Not allowing hard transactions to update data in a snapshot set is not a limitation, but a consequence of the concept of snapshots. If data elements within a snapshot set would be individually updated by different transactions, the snapshot requirement would be violated. However, if a behavior where individual elements of snapshots are updated is wanted, snapshot transactions updating these data elements can instead be used.

8.4.5 Extending soft transactions

Soft transactions executing under 2V-DBP-SNAP differ from soft transactions executing under 2V-DBP in the following two ways:

1. In difference to snapshot transactions, soft transactions cannot read individual data elements from the snapshot at arbitrary points in time. If a soft transaction needs to access a data element in a snapshot set, the entire set must be cached to the local working copy. This is a non-preemptive operation. If the transaction, later in its execution, requests a different data element within the same snapshot set, the transaction uses the cached copy. Even though the soft transaction reads the complete set it might choose to only update a subset of the elements in the set.

Transaction type	**Soft**	**Hard**	**Snapshot**
Soft	L		
Hard	V	M	
Snapshot	S & V	D	M or D

Legend:
L - Database locks
V - Versioning
M - Mutual exclusion
D - Disjoint data
S - Snapshot set

Table 8.1: Serialization policies between transaction types.

2. Prior to reading the snapshot set, the corresponding snapshot lock must be obtained. Since the snapshot set is a shared resource it needs to be protected, the same way as ordinary data tuples. Just as for any lock in 2V-DBP, snapshot locks are managed by the lock manager in the RT-DBMS. Hence, they can be either read-, or write-locked, and lock conflicts are resolved using the 2PL-HP policy.

8.4.6 Serialization in 2V-DBP-SNAP

In 2V-DBP-SNAP, a set of different serialization techniques are used. Table 8.1 shows the different policies used between different transaction types. We refer to [6] for a discussion about the serialization among soft and hard transactions, and instead focus on the serialization policies used for snapshot transactions. Table 8.1 shows three possible serialization cases, namely serialization among:

1. **Two concurrent snapshot transactions.** In this case, there are two possible sub-cases; (i) If the transactions access different snapshot sets, no conflicts can occur (hence transactions are serialized in the order they commit), and (ii) if the transactions access the same snapshot set, the snapshot semaphore will ensure mutual exclusion (and thus no conflicts). The transactions are serialized in the order they obtain the snapshot semaphore.

2. **A snapshot transaction concurrent with a hard transaction.** In this case, there are three possible sub-cases; (i) Since a hard transaction is not allowed to write to data elements in a snapshot set, this case is trivial, and transactions are serialized in the order they commit. (ii) If a

hard transaction reads a data element after the snapshot transaction has updated it, the hard transaction is serialized after the snapshot transaction. (iii) If a hard transaction reads a data element before the snapshot transaction has updated it, the hard transaction is serialized before the snapshot transaction.

3. **A snapshot transaction concurrent with a soft transaction.** Since soft transactions read the entire snapshot set in one non-preemptive operation, it can be viewed as one data element (from a soft transactions perspective). Figure 8.3(b) illustrates this. If a snapshot set is manipulated by a snapshot transaction after being cached by a soft transaction, the versioning algorithm will ensure that, just as for soft transactions in 2V-DBP, only the clean data elements (determined by the `dirty` flag) are updated. Thus, a snapshot transaction is serialized before a soft transaction iff it is executed before the soft transaction caches the snapshot set, otherwise the snapshot transaction is serialized after the soft transaction.

8.4.7 Evaluation of 2V-DBP-SNAP

The aim of 2V-DBP-SNAP is to allow hard control-tasks to achieve a consistent view and control of the state of the environment. This aim must be fulfilled without the introduction of unpredictable, or significant blocking. In fact, 2V-DBP-SNAP introduces two forms of blocking for snapshot transactions.

First, the snapshot semaphore introduces blocking among concurrently executing snapshot transactions. This approach can be compared to using shared variables which are protected by a semaphore. One benefit of managing this in database transactions instead of in the application code is that minimal snapshot sets automatically can be constructed off-line, thus increasing the concurrency in the system. Furthermore, deadlocks caused by erroneous semaphore usage can be avoided since only one semaphore is used per transaction.

Second, soft transactions introduce blocking on snapshot transactions, since soft transactions needs to read (or update) a set of data elements in a non-preemptive operation. If, during the creation of the snapshot sets, the snapshot transactions overlap to a large extent, the snapshot sets might get, in fact, arbitrarily large. Note, however, that this blocking is deterministic, and the maximum blocking factor can be derived and analyzed off-line. From our experience with automotive control-systems [2], only few data elements in a system have a need for snapshot semantics, and thus the snapshot sets are not likely to be large. An implementation of 2V-DBP-SNAP under the Asterix real-time op-

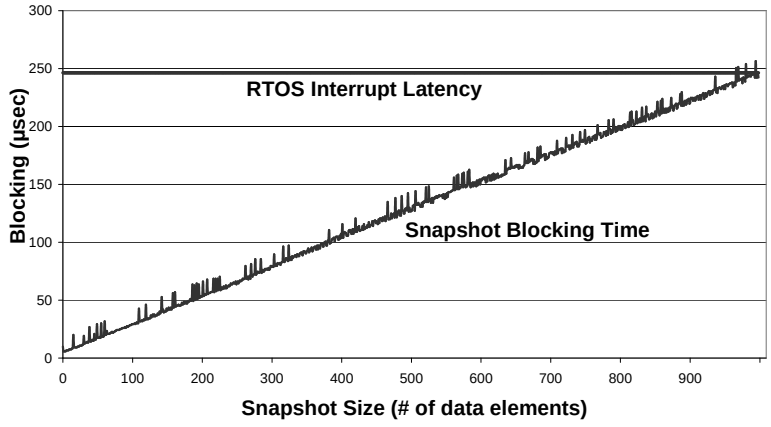


Figure 8.6: Blocking introduced by soft transactions caching a snapshot set

erating system (rtos) [19] executing on an 133Mhz Intel 486 processor shows that snapshots with up to 939 data elements can be used before reaching blocking times that exceed the interrupt latency of the rtos, see Figure 8.6. From the figure we can see that the rtos latency is approximately 250μ sec. The snapshot caching in the soft transaction reaches this time at approximately 940 data elements. The evaluation shows that, in the normal case, soft transactions that cache a snapshot set will not affect the maximum blocking of the system, since snapshot sizes will not reach this size.

Another limitation in 2V-DBP-SNAP is snapshots used in interrupts. Introducing blocking for interrupt handlers is not preferable, however, in its current form, 2V-DBP-SNAP, may introduce blocking for interrupt handlers that execute snapshot transactions. One possible solution to this is to manage this data in the application, by letting the interrupt handler write the data to a buffer, which is then handled by a task executing a snapshot transaction.

However, these limitations do not negatively influence the system, or the users of the system. Instead 2V-DBP-SNAP is designed with automotive system's requirements in mind, and its minimal overhead is to a high degree suit-

able for use in such systems.

8.5 Conclusions

We have presented a concurrency control algorithm which is based on a previous algorithm, denoted 2V-DBP [6], which allows co-existence of soft real-time, relational database transactions (soft transactions), and hard real-time database pointer transactions (hard transactions) [7].

The presented algorithm, called 2-version database pointer concurrency-control with snapshots (2V-DBP-SNAP), extends 2V-DBP by introducing snapshot support. This allows hard real-time tasks to get a consistent instant view of a set of data elements in the real-time database. Furthermore, this data can be shared with soft database transactions without violating the snapshot requirements on the data, i.e., the requirement to be able to get a snapshot of the data elements. To be able to allow transactions to have snapshot support, the concept of snapshot sets is introduced.

2V-DBP-SNAP is designed to be lightweight with respect to overhead and blocking of transactions. An implementation of 2V-DBP-SNAP shows that this is also the case. The algorithm is intended for hard real-time control applications, e.g., automotive control systems, in which hard control-tasks need a consistent view (or a consistent control) of multiple data elements.

Future work includes further evaluation of the algorithm as well as introducing wait-free or lock-free algorithms to the snapshot sets in order to allow concurrently executing snapshot transactions to access the same snapshot set.

Bibliography

[1] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano - a Revolution in On-Board Communications. Technical report, Volvo Technology Report, 1998.

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

[3] S. F. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring. DeeDS Towards a Distributed and Active Real-Time Database System. *ACM SIGMOD Record*, 25(1):38–40, 1996.

[4] J. Lindstrom, T. Niklander, P. Porkka, and K. Raatikainen. A Distributed Real-Time Main-Memory Database for Telecommunication. In *Proceedings of the Workshop on Databases in Telecommunications*, pages 158–173. Springer, September 1999.

[5] Dag Nyström, Aleksandra Tešanović, Mikael Nolin, Christer Norström, and Jörgen Hansson. COMET: A Component-Based Real-Time Database for Automotive Systems. In *Proceedings of the Workshop on Software Engineering for Automotive Systems*, pages 1–8. The IEE, June 2004.

[6] Dag Nyström, Mikael Nolin, Aleksandra Tešanović, Christer Norström, and Jörgen Hansson. Pessimistic Concurrency Control and Versioning to Support Database Pointers in Real-Time Databases. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 261–270. IEEE Computer Society, June 2004.

[7] Dag Nyström, Aleksandra Tešanović, Christer Norström, and Jörgen Hansson. Database Pointers: a Predictable Way of Manipulating Hot Data in Hard Real-Time Systems. In *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications*, pages 623–634, February 2003.

[8] Håkan Sundell and Philippas Tsigas. Simple Wait-Free Snapshots for Real-Time Systems with Sporadic Tasks. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications*. Springer-Verlag, August 2004.

[9] J. Huang, J.A. Stankovic, K. Ramamritham, and D.F. Towsley. Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 35–46. Morgan Kaufmann, September 1991.

[10] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *The communications of the ACM*, 19(11):624–633, November 1976.

[11] H. T. Kung and J. T. Robinsson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[12] Thomas Gustafsson. Maintaining data consistency in embedded databases for vehicular systems. Linköping Studies in Science and Technology Thesis No. 1138. Linköping University. ISBN 91-85297-02-X, 2004.

[13] Stephen Cannan and Gerhard Otten. *SQL - The Standard Handbook*. MacGraw-Hill International, 1993.

[14] Tei-Wei Kuo, Yuan-Ting Kao, and LihChyun Shu. A Two-Version Approach for Real-Time Concurrency Control and Recovery. In *Proceedings of the Third IEEE International High Assurance Systems Engieering Symposium*. IEEE Computer Society, November 1998.

[15] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Publishing Company , 1987.

[16] R.K Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *ACM Transactions on Database Systems*, 17, September 1992.

[17] Tei-Wei Kuo and Aloysius K. Mok. SSP: a Semantics-Based Protocol for Real-Time Data Access. In *Proceedings of 14th IEEE Real-Time Systems Symposium*, pages 76–86. IEEE Computer Society, December 1993.

[18] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*, chapter 13.10.1 Immediate Ceiling Priority Inheritance. Addison-Wesley, second edition, 1996. ISBN 0-201-40365-X.

[19] Henrik Thane, Anders Pettersson, and Daniel Sundmark. The Asterix realtime kernel. In Eduardo Tovar and Christer Norström, editors, *Proceedings of the Work-in-progress and Industrial Session of the 13th Euromicro Conference on Real-Time Systems,Delft Netherlands*. http://citeseer.nj.nec.com/thane01asterix.html, June 2001.

Chapter 9

Paper E: Introducing Substitution-Queries in Distributed Real-Time Database Management Systems

Thomas Nolte and Dag Nyström

In Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA05), Catania, Sicily, September 2005

Abstract

This paper introduces query mechanisms that allow automotive control-systems (using a distributed real-time database management system (RTDBMS)) to be queried, monitored and stimulated during run-time without violating its temporal properties. The mechanisms are completely transparent to the control application since they are handled by the RTDBMS. The COMET RTDBMS is extended with ad hoc capabilities to support the introduction of subscription and substitution queries, which are used for monitoring and stimulation. These queries are intended to be used by service and calibration tools to help in the development and maintenance of modern automotive systems. Using these queries could reduce development costs, result in higher quality of the system design and consequently yield higher reliability.

9.1 Introduction

In recent years, automotive control-systems have evolved from simple single processor systems to complex distributed systems. At the same time, the amount of data that needs to be managed by these systems is increasing dramatically; the data volume managed by automotive systems is predicted to increase 7-10% per year [1]. Current techniques for storing and manipulating data objects in automotive systems are ad hoc in the sense that they normally manipulate data objects as internal data structures. This lack of a structured approach to data management results in a costly development process with respect to design, implementation, and verification of the system [2]. It also makes the system difficult to maintain and develop while preserving consistency with the environment, e.g., maintaining temporal properties of data. As data complexity is growing, the need for a uniform, efficient, and persistent way to store data is becoming increasingly important. One way of handling this complexity is to use a real-time database management system (RTDBMS) as a tightly integrated part of the automotive control-system. RTDBMSs has the potential to solve many of the problems that application designers have to consider with respect to data management, e.g., locking of the data, persistency and deadlock situations. More importantly, incorporating an RTDBMS into an automotive control-system could reduce development costs, result in higher quality of the design of the systems, and consequently yield higher reliability [3]. Moreover, an RTDBMS enables development of advanced diagnosis tools, by providing a uniform interface to access data.

Today, there exists a number of both commercial and research databases suitable for embedded systems. Commercial embedded platforms include Pervasive.SQL [4], Polyhedra [5], Berkeley DB [6], and TimesTen [7]. Research real-time platforms include, DeeDS [8], RODAIN [9], STRIP [10], and BeeHIVE [11]. The general trend among these platforms is that commercial systems focus towards the embedded systems market, i.e., focus on flexibility, adaptability, and efficiency, while the research platforms mainly address real-time requirements. This discrepancy makes neither type of database system suitable for the automotive domain, where the adaptability and efficiency must be combined with maintaining real-time requirements. However, the real-time database management system COMET [12] aims to bridge this gap by providing both a light-weight, adaptable, and reconfigurable design paradigm [13], as well as efficient and predictable RTDBMS mechanisms [14].

This paper shows how the COMET RTDBMS can be extended with distribution mechanisms to allow service tools to query, monitor and stimulate

a control system at run-time, while still maintaining a high level of abstraction. Three types of database queries, ad hoc queries, subscription queries, and substitution queries are introduced to obtain this behavior. These queries are handled by the RTDBMSs on each node and are completely transparent to the control application and are not violating the temporal properties of the control system.

The outline of the paper is as follows; in Section 9.2, a background of automotive systems, current service tools, CAN, and COMET are given. The paper continues in Section 9.3, in which the extended data distribution is presented. Finally, in Section 9.4 the paper is summarized and concluded.

9.2 System model

In this section, typical automotive control-systems are presented together with how these systems distribute data. Furthermore, the COMET RTDBMS is presented in detail, including how it is distributed using CAN. Also, typical service tools used for automotive systems are presented, motivating the contribution of this paper, namely the database queries allowing for querying, monitoring and stimulation of data during run-time of real-time control-systems.

9.2.1 Automotive control-systems

Typical automotive control-systems are found in chassis and vehicle safety systems, such as Vehicle Dynamics Control (VDC) systems, also known as Electronic Stability Program (ESP). VDC/ESP is designed to assist the driver in over-steering, under-steering and roll-over situations [15]. This, and similar safety systems, such as the Anti-lock Brake System (ABS), all require *feedback control*.

Other safety-systems are air-bag systems [16], that control the operation of air-bags in the vehicle. Typically a vehicle contains several air-bags that are connected to sensors that detect abnormal situations, e.g., sudden acceleration or decelerations of the vehicle. Once an abnormal situation is detected the correct (depending on the type of crash) air-bags are inflated in a matter of half a millisecond.

Body and comfort electronics require both feedback and *discrete control* for subsystems such as climate control, cruise control, locks, window lifts, seat control and HMI, to mention a few. Typically body and comfort electronics rely on driver interaction and are not safety-critical, they involve hundreds of

system states and events, and they interface to physical components in the vehicle, e.g., motors and switches.

Other automotive control-systems are powertrain systems and x-by-wire systems. Powertrain is the assembly of gears by which power is transmitted from the engine of the vehicle to the driving axis. Powertrain includes *engine control* which involves the coordination of fuel injection, engine speed, valve control, cam timing etc. X-by-wire is the notation for new subsystems replacing hydraulic and mechanical parts with electronics and computer (feedback) control systems. Examples of x-by-wire systems are steer-by-wire, shift-by-wire, throttle-by-wire and break-by-wire.

During the development and maintenance of these control systems, support through hardware tools is essential. These tools are used to monitor and diagnose both the software control-system and the mechanical systems.

9.2.2 Architecture

An automotive control-system (subsystem) consists of one or several Electronic Control Units (ECUs). An automotive system, consisting of several subsystems with a total of up to 70 ECUs, has to distribute thousands of variables and signals (data) over several communication networks [17], e.g., CAN networks. This makes a modern automotive system complex.

In this paper, all ECUs are assumed to be equipped with the COMET RT-DBMS [12]. Moreover, these ECUs are assumed to be connected using the Controller Area Network (CAN) [18].

9.2.3 CAN

CAN, or the Controller Area Network [18], is a serial bus that was developed in the beginning of the eighties by Bosch. Today CAN is the most widely used vehicular network in the automotive industry. Over the years several different CAN standards have been developed and used in different applications, where the ISO 11898 [19, 20] is the most commonly used fieldbus in the European automotive industry.

A typical CAN application is any type of embedded system with soft real-time requirements and loop times of 5-50 ms. CAN transmits messages in an event-triggered fashion using frames containing 0 to 8 bytes of data and 4 to 6 bytes of header. These frames can be transmitted at speeds of 10 Kbps up to 1 Mbps.

CAN handles communication faults by retransmission, and there is no error containment or support for higher level of fault tolerance. However, it holds a strong position and will most likely continue to be the most used communication bus in the automotive application domain for a long time.

With CAN, messages are not interrupted while in transmission. Moreover, the CAN message identifier (ID) is representing the priority of the message. Hence, CAN is implementing non-preemptive *Fixed Priority Scheduling (FPS)*, and suitable analysis techniques can be used, like the FPS response-time tests to determine the schedulability of CAN message frames, presented by Tindell *et al.* [21, 22].

Using FPS, priorities are assigned to the messages before execution (offline), by the allocation of message identifiers (IDs). The message with the highest priority among all messages available for transmission is scheduled for transmission.

9.2.4 Data distribution in automotive control-systems

OSEK/VDX [23] is an effort to standardize and increase portability of automotive software. Among the OSEK/VDX specifications, OSEK/VDX COM [24] is a uniform communication environment for automotive control unit application software. OSEK/VDX COM provides communication services through a well defined API. Moreover, it specifies an Interaction Layer (IL) that provides the communications interface to the application. The application can transmit messages to other applications resident on the same ECU or on other ECUs. If the receiving application is resident on the same ECU, the IL handles the communications internally. If receiving applications are resident on another ECU, the IL packs one or more messages (signals) into Interaction Layer Protocol Data Units (I-PDUs). These I-PDUs are then sent to the Network Layer (NL), either periodically or explicitly initiated by some event. However, OSEK/VDX COM does not specify the NL other than defining some minimum requirements.

AUTOSAR [25], which aims at providing a global standard for software in automotive systems, proposes similar mechanisms for data distribution using a run-time environment to route communications both inter- and intra-ECU.

Apart from OSEK/VDX and AUTOSAR, automotive systems distribute signals over CAN in several ways, e.g., with the usage of Volcano as done by Volvo Car. The Volcano system [1] provides tools for packaging signals into messages, as well as assigning priorities for CAN messages to achieve a high utilization of the bus. Moreover, it is possible to perform timing analysis of the

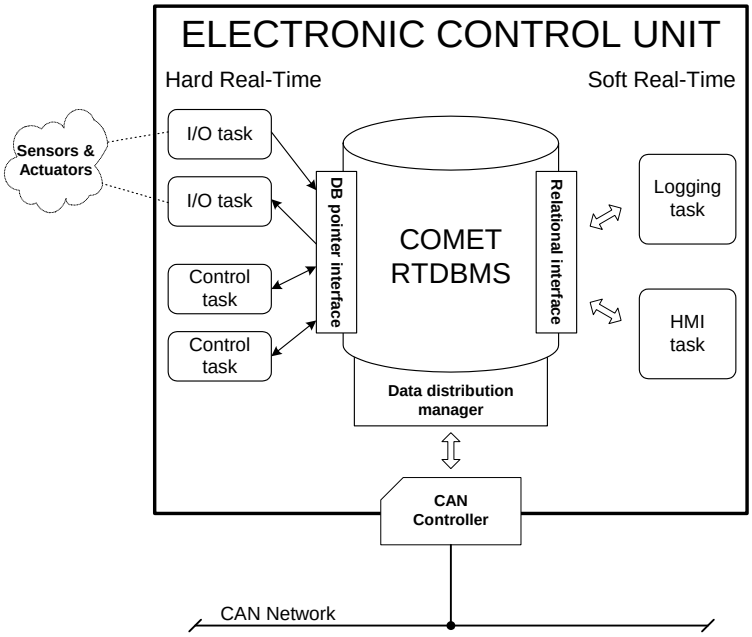


Figure 9.1: The architecture of an ECU using COMET

system using the Volcano tools. An offline schedulability test is done to ensure that all deadlines are met.

9.2.5 The COMET real-time database management system

The COMET RTDBMS [12] is a data management system intended primarily for embedded control-systems, e.g., automotive systems. COMET contains data management concepts that allow hard and soft application tasks to access and share data in a predictable and efficient way [14].

Tasks in the system interact with the RTDBMS through database transactions. Different types of tasks require different kinds of transactions. Therefore, transactions are divided into the following two classes, see Figure 9.1:

1. **Soft transactions**, either precompiled or ad hoc (formulated and parsed at run-time) reside in soft real-time tasks. These transactions utilize the

relational database query interface, such as, SQL [26], for database access. Soft transactions provide a flexible and dynamic access to the data in the database to the system and are especially suited for management tasks, e.g., logging, diagnosis, and, user interface (HMI) tasks, e.g., tasks controlling the dash board.

2. **Hard transactions**, which are precompiled, reside in hard real-time tasks. A hard transaction utilizes the database pointer interface (see Section 9.2.5) [14], providing an efficient and predictable access to individual data elements in the database. A majority of the transactions in a vehicle, i.e., transactions used for vehicle control, would fall into this class.

Database pointers

Database pointers allow individual data elements in an RTDBMS to be accessed in an efficient and predictable manner [14]. They are intended as a complement to the relational data model, without limiting the expressibility of the relational query processing.

Figure 9.2 shows an example of an I/O task that periodically reads a sensor and propagates the sensor value to the database using a database pointer, in this case the oil temperature in the `engine` relation. The task consists of two parts, an initialization part (lines 2–4) executed when the system is starting up, and a periodic part (lines 5–8) scanning the sensor. The initialization of the database pointer is first done by declaring the database pointer (line 3) and then binding it to the data element containing the oil temperature in the engine (line 4). When the initialization is completed, the task begins to periodically read the value of the sensor (line 6), then propagates the value to the RTDBMS using the database pointer (line 7), and finally awaits the next invocation of the task (line 8).

Database pointers are implemented using the data structures shown in Figure 9.3. The binding of a database pointer to a database element is performed in the following steps:

1. A new *database pointer entry* is created in the RTDBMS.

2. The SQL query is executed. It is required that the result of the query is a single data element. If it is the first time the data element is bound to a database pointer, a new *data pointer* is created in the RTDBMS. The data pointer is initialized with the address of the data element and its type.

```
1 TASK OilTempReader(void) {
2   int s;
3   DBPointer *ptr;
4   bind(&ptr, "SELECT temperature FROM engine
                WHERE  subsystem=oil;");
5   while(1){
6     s=read_sensor();
7     write(ptr,s);
8     waitForNextPeriod();
    }
  }
```

Figure 9.2: An I/O task that uses a database pointer

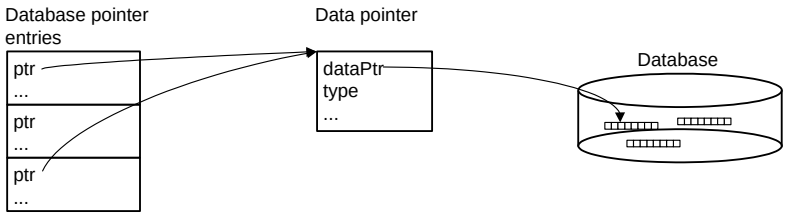


Figure 9.3: The data structures used by database pointers

3. The database pointer entry is set to point at the data pointer.
4. Finally, the pointer to the database pointer entry is returned as a `DBPointer*`.

In addition to the `bind(ptr,q)` operation, the database pointer interface consists of the `remove(ptr)` operation which deallocates a database pointer, the `write(ptr,data)`, and the `read(ptr)` operations which updates, respectively reads the data element.

Concurrency in COMET

For applications that use multiple transactions possibly executed in parallel, some form of concurrency control in the RTDBMS is needed to maintain the consistency of the database. One common way of enforcing concurrency control is to introduce database locks. Before a transaction is allowed to access a data element in the database, the appropriate lock must be obtained. Database locks are similar to semaphores in the sense that they protect a shared resource.

For real-time systems, e.g., automotive control-systems, using locks might introduce unwanted blocking. This is especially true for systems that have both hard tasks executing at high frequencies, and soft tasks that might execute transactions with long execution times.

In COMET, this problem is solved by combining database locks (for soft transactions) with a versioning algorithm (for hard transactions). The concurrency control algorithm, denoted 2-version database pointer concurrency control (2V-DBP) [14], allow hard and soft transactions to share common data elements without interfering with each other.

Data distribution in COMET

To be able to support distributed automotive control-systems, COMET needs to be equipped with a distribution manager that communicates using the Controller Area Network (CAN). The distribution manager supports periodic precompiled queries (sporadic queries are treated as periodic based on their minimum inter-arrival time). When queries are distributed between ECUs in the system, data is *mapped* onto periodic CAN frames (messages) with identifiers (IDs) assigned to fulfill timing requirements of the transactions. The mapping of query data onto CAN frames is done similar to what is explained in Section 9.2.4, i.e., using a tool where signals and data are mapped onto messages. These messages are then periodically sent (multicasted/broadcasted) on the CAN bus. For the remainder of this paper, this periodically sent traffic is defined as the *original system CAN traffic*.

Both hard and soft periodic transactions are mapped between CAN frames and the COMET database using database pointers with 2V-DBP for efficient access. Hence, both packaging of data into messages for transmission, and updating data in the database upon message reception, are fast and simple operations. Moreover, the usage of a database simplifies data access. Using, e.g., the CAN Calibration Protocol (CCP) [27], lists of data elements are used to describe which data elements that are to be mapped into specific messages. In the CCP specification, these lists represent physical memory addresses. However, using the COMET RTDBMS with the database pointer concept, data access is handled on a logical (relational) level. Hence, no direct access to the ECU memory is required, protecting the ECU while providing a clear interface to the ECU's data elements, still allowing fast access to its data elements.

Note that the mapping of data onto a set of periodic CAN frames, together with performing the schedulability test on the set of messages, is done offline hence not requiring resources during runtime. During runtime mapping is done

simply based on lookup tables, containing database pointers.

9.2.6 Service tools for automotive systems

During the development and maintenance of a modern automotive system, support through hardware tools connected to the control system is essential. These tools are mainly used to calibrate, test and diagnose both the software control system and the mechanical systems. For the remainder of this paper, these tools are simply referred to as *service tools*.

Today, a substantial effort is put into calibrating the parameters of an automotive control-system. The aim of this work is, among others, to optimize the performance of the system, and to comply with regulations regarding emissions etc. It is noteworthy that it is not the performance of the control system and its real-time properties, e.g., keeping deadlines and minimizing jitter, that is calibrated, but the performance of the mechanical system being controlled. For an automotive engine, typically several man-years are invested in calibration.

In order to perform the calibration, a *calibration tool* is used. These tools are typically connected to the vehicle via the CAN network, and then the automotive system can be monitored or updated using the tools. Even though commercial calibration tools that support the CCP exist, e.g., CAMEO for Vehicle Use [28], and CANaph Graph [29], it is not uncommon that in-house developed tools are used.

Service tools are also used to detect and diagnose potential system failures, both electrical and mechanical, during service of the vehicle. By providing service stations with powerful service tools, more efficient and accurate service can be performed. Desirable functionalities in such systems include:

- Downloading of warning- and error-logs from the vehicle. These logs contain information on system anomalies detected in the vehicle since its last service. Typical logs might include sporadic failures of sensors and abnormal sensor readings such as temperatures.

- Reading of a set of data elements in the vehicle. Such information can be used to further localize errors. An example of such a reading might be to obtain information on all current sensor values regarding the engine.

- Periodically subscribe to data elements to monitor fluctuations of their values over time, e.g., RPM- or temperature readings. Such information might be used to spot intermittent failures.

- Take control of (stimulate) a subsystem or function in the vehicle. Consider, for example taking over the acceleration pedal to be able to control the RPM of the engine. This functionality is useful for automated tests of the vehicle.

Since service tools communicate with the vehicle through its data, it is natural that the service tools communicate directly with the RTDBMSs in the ECUs, since they are responsible for managing the data in the automotive system. An advantage of this is also that the automotive application itself needs not to be aware of the existence of service tools.

So far COMET has only been discussed in the context of precompiled, periodic (and sporadic) offline scheduled data distribution (the original system CAN traffic). Hence, it must be extended to allow these new event driven ad-hoc activities to be executed. To allow the above mentioned activities, three new types of *distributed* database transactions are introduced:

1. **Soft ad hoc queries** These queries are similar to soft transactions, in the sense that they use a query language, but can now be formulated at runtime. Since it can not be foreseen which data elements an ad hoc query will access, these queries must be allowed to be distributed, i.e., gather information from different ECUs in the system.

2. **Subscription queries** These queries allow a data element, not currently distributed, to be subscribed to by a task on a different ECU or by a service tool. This query type consists of three parts, (i) a `start of subscription` in which the subscriber requests that a subscription is started, (ii) the actual subscription itself, and (iii) an `end of subscription`.

3. **Substitution queries** These queries allow current producers of data (sensors etc.) in the system to be overridden, in order for a service tool to take control of a certain subsystem. A substitution query has, as subscription queries, three parts, namely, (i) `start of substitution`, (ii) the actual substitution itself, in which the substitution data is propagated through the network, and (iii) the `end of substitution`.

9.3 Extending the COMET data distribution

To be able to incorporate the three query types introduced in Section 9.2.6, the data distribution in COMET needs to be extended to support *ad hoc CAN traf-*

fic, see Figure 9.4. From the figure, it can be seen that the ad hoc CAN traffic is added in a lower priorities segment than the original system CAN traffic. In this segment, every node (both ECUs and service tools) are assigned a unique CAN ID to transmit on. This implies that any message collisions among ad hoc messages are handled by the CAN network. Furthermore, since all ad hoc CAN messages are transmitted with lower priorities than the original system CAN traffic, the schedulability is still valid regardless of the amount of ad hoc CAN traffic. However, it must be checked whether the ad hoc CAN traffic, although unlikely, introduce longer blocking times than caused by existing traffic, affecting the timely delivery of messages. It is common to assume the blocking time to be equal to the longest possible CAN-frame, in which case this check is not needed.

9.3.1 Ad hoc queries

Ad hoc queries are distributed database queries, formulated at run-time, normally by a service tool. This type of query allows a user to view the current state of the system using a powerful high level query language, i.e., SQL.

It is however noteworthy, that ad hoc queries provide neither transaction nor snapshot semantics, i.e., the result of an ad hoc query cannot be viewed as the state of the system at a single instance in time. However, ad hoc queries follow the COMET consistency properties, namely that data consistency and transaction semantics can be guaranteed locally on each ECU, but due to the fact that freshness typically is more favored than global consistency [30], inconsistencies among nodes can be tolerated. This coincides with the consistency of most automotive systems in practice.

The execution-flow of an ad hoc query is as follows (numbers in Figure 9.4 correspond to the list below):

1. An ad hoc query is entered to the service tool. These queries follow standard SQL syntax.

2. The query parser in the service tool parses the SQL query, and creates an execution plan. To create and optimize the execution plan, the query parser has access to the metadata, i.e., information such as structure and size of the data elements and relations in the database. The metadata is stored in the service tool.

3. The query engine in the service tool starts to process the execution plan. Usually, the first step in the execution plan is to retrieve data from the

database, so therefore the distribution manager sends out a request for data on the CAN network, using the service tool's assigned CAN ID. Typically, such a request is on the form <*DATA_REQ, REL_NAME, COND*>, where all tuples (rows in a relation) for the relation *REL_NAME* which meet the boolean condition *COND* are requested.

4. All distribution managers in the ECUs will then forward this message to its local query engine, which will launch a soft transaction retrieving the requested tuples. It is however noteworthy that only the tuples that each ECU has ownership of (stated in each ECU's metadata) is retrieved. Declaring ownership for each tuple avoids several ECUs to retrieve (and thus return) the same tuple to the service tool. When all tuples are retrieved, each ECU's distribution manager packs them together in CAN messages and transmits them on its respective CAN ID. When a distribution manager is completed (possibly after sending 0 tuples), it acknowledges end of transmission.

5. Finally, the query engine in the service tool completes the execution plan and outputs the query result.

9.3.2 Subscription queries

Subscription queries are used to monitor individual internal data elements over a period of time. These queries utilize just as the ad hoc queries, low priority CAN traffic to initialize and terminate subscriptions. The level of service, with respect to frequency and Quality of Service (QoS), of the subscription can be specified. QoS is divided into two classes, either the subscription is performed as a background service (soft real-time), using ad hoc traffic, or it is guaranteed (hard real-time). Guaranteed subscriptions undergo an admission control in which a schedulability analysis (as presented in Section 9.2.3) of the original system CAN traffic together with the added subscription traffic is performed. This analysis determines whether or not to accept (admit) the subscription. It is assumed that the service tool has the full knowledge of the system, in terms of the original system's CAN traffic. If admitted, the subscription will temporarily be treated as a part of the original system CAN traffic. The execution-flow of a subscription query is as follows:

1. A subscription query is entered into the service tool. The query consists of the following:

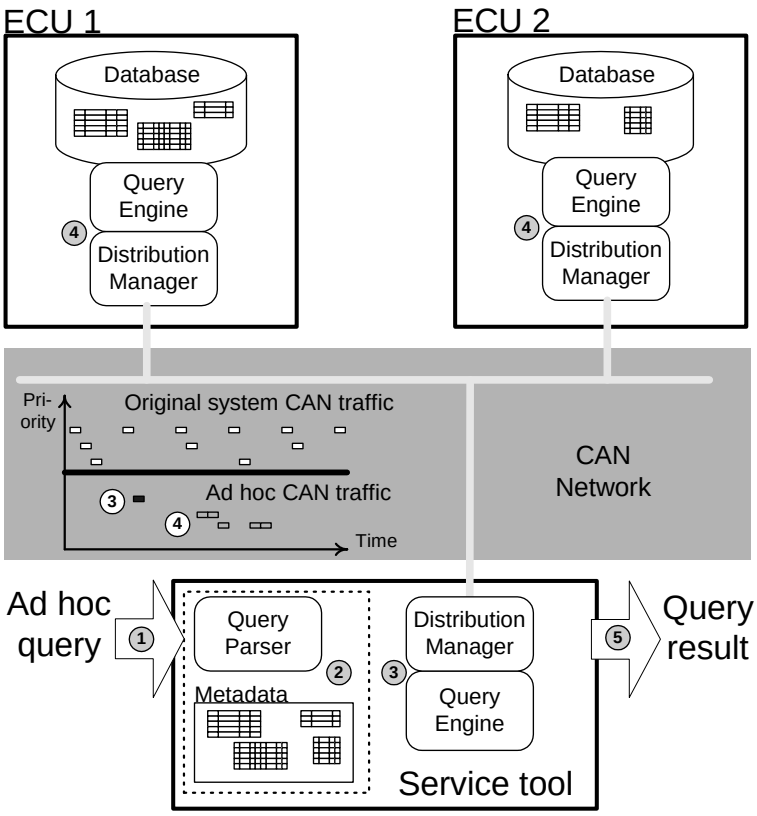


Figure 9.4: Execution of an ad hoc query

- *<NODE, REL_NAME, KEY, ATTRIBUTE>*, which corresponds to the relation name, the key and attribute (row and column) of the tuple located in the ECU pointed out by *NODE* in which the data element to subscribe upon is located. This information is enough to uniquely identify any data element in any database in the system.
- *<PERIODICITY, QoS>*, which corresponds to the periodicity and the QoS level (soft or hard) of the subscription.

2. The query engine of the service tool first checks in its metadata if the data to view already is distributed (i.e., is already in the original system CAN traffic) with at least the same level of service. In that case the service tool uses that distribution.

3. For queries with the QoS level set to hard, the query engine in the service tool performs an admission control. In the admission control, the following is determined; (i) if the subscription, given its periodicity, can be safely inserted into the original system CAN traffic without violating any system requirements (i.e., schedulability analysis is performed), and (ii) at which priority (CAN id) it can be transmitted.

4. Given that the query is admitted (or if the QoS level is soft) an ad hoc message is sent out with the following format: *<SUB_REC, NODE, REL_NAME, KEY, ATTRIBUTE, PERIODICITY, CANID>*.

5. The ECU being addressed receives the message and acknowledges it.

6. The ECU then creates a new database pointer (if not already existent) and periodically starts to transmit on the assigned CAN ID.

7. Eventually, the service tool transmits an `end of subscription`, and the subscription is terminated.

9.3.3 Substitution queries

Substitution queries are used to stimulate the system from a service tool or similar. When a substitution query is active for a data element, it overrides the producer of that element. In Figure 9.5, a substitution query for data element x is active, thus any producer in the control application is overridden. Still however, the producers will receive a normal response (e.g., `query successful` or similar) on their data updates on x. This implies that, from the control applications point of view, a substitution (or subscription) is completely transparent

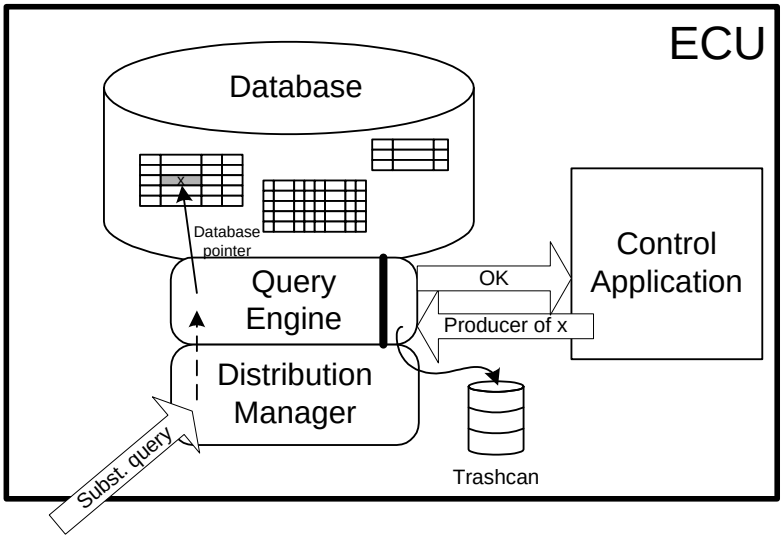


Figure 9.5: Execution of an substitution query

since it is handled by the RTDBMS. The workflow for a substitution query is the same as for a subscription query, except that data packages are sent from the service tool to the control application. Just as for subscription queries, substitution queries can be executed on both a soft and a hard QoS-level.

9.4 Summary

During the development and maintenance of an automotive system, service tools play an important role in calibration, testing and diagnosis. These tools need an intimate access to system data to be able to monitor the system behavior during run-time.

This paper presents how a real-time database management system can be used to enable this behavior. The COMET RTDBMS is extended with three new query types for querying, monitoring and stimulating data during runtime of the system, without violating the temporal properties of existing control systems. These new queries could help in the development of modern

automotive systems, reducing development costs, resulting in higher quality of the system design and consequently yield higher reliability. Furthermore, the approach presented in this paper enables any data residing in the database to be monitored and stimulated during runtime, also data that is not explicitly pre-configured for this data access.

Future work includes extending the distributed RTDBMS to also support other networks [31], such as Local Interconnect Network (LIN) [32] and Flexray [33].

Bibliography

[1] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano - a revolution in on-board communication. *Volvo Technology Report 98-12-10*, 1998.

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

[3] T. Gustafsson and J. Hansson. Data management in real-time systems: a case of on-demand updates in vehicle control systems. In *Proceedings of the Real-Time Application Symposium (RTAS 2004)*. IEEE Computer Society Press, May 2004.

[4] Pervasive Software Inc. http://www.pervasive.com.

[5] Enea AB. http://www.enea.se.

[6] Sleepycat Software Inc. http://www.sleepycat.com.

[7] TimesTen Performance Software. http://www.timesten.com.

[8] S. F. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring. DeeDS Towards a Distributed and Active Real-Time Database System. *ACM SIGMOD Record*, 25(1):38–40, 1996.

[9] J. Lindstrom, T. Niklander, P. Porkka, and K. Raatikainen. A Distributed Real-Time Main-Memory Database for Telecommunication. In *Proceedings of the Workshop on Databases in Telecommunications*, pages 158–173. Springer, September 1999.

[10] B. Adelberg, B. Kao, and H. Garcia-Molina. Overview of the STanford Real-time Information Processor (STRIP). *SIGMOD Record*, 25(1):34–37, 1996.

[11] J. A. Stankovic, S. H. Son, and J. Liebeherr. *Real-Time Databases and Information Systems*, chapter BeeHive: Global Multimedia Database Support for Dependable, Real-Time Applications, pages 409–422. Kluwer Academic Publishers, 1997.

[12] Dag Nyström, Aleksandra Tešanović, Mikael Nolin, Christer Norström, and Jörgen Hansson. COMET: A Component-Based Real-Time Database for Automotive Systems. In *Proceedings of the Workshop on Software Engineering for Automotive Systems*, pages 1–8. The IEE, June 2004.

[13] Aleksandra Tešanović, Dag Nyström, Jörgen Hansson, and Christer Norström. Aspects and Components in Real-Time System Development: Towards Reconfigurable and Reusable Software. *Journal of Embedded Computing*, February 2004.

[14] Dag Nyström, Mikael Nolin, Aleksandra Tešanović, Christer Norström, and Jörgen Hansson. Pessimistic Concurrency Control and Versioning to Support Database Pointers in Real-Time Databases. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 261–270. IEEE Computer Society, June 2004.

[15] A. T. van Zanten, R. Erhardt, K. Landesfeind, and G. Pfaff. VDC systems development and perspective. In *SAE World Congress*. SAE, 1998.

[16] Robert Boys. Safe-by-Wire: The Leading Edge in Airbag Control. In *SAE World Congress*, Detroit, MI, USA, 2003. SAE.

[17] Nicolas Navet, Yeqiong Song, Francoise Simonot-Lion, and Cedric Wilwert. Trends in Automotive Communication Systems. *Proceedings of the IEEE*, 93(6), June 2005.

[18] Robert Bosch GmbH. BOSCH's Controller Area Network. http://-www.can.bosch.com/.

[19] ISO 11898. Road Vehicles - Interchange of Digital Information - Controller Area Network (CAN) for High-Speed Communication. *International Standards Organisation (ISO)*, ISO Standard-11898, Nov 1993.

[20] ISO 11898-1. Road Vehicles - Controller Area Network (CAN) - Part 1: Data link layer and physical signalling. *International Standards Organisation (ISO)*, ISO Standard-11898-1, 2003.

[21] K. W. Tindell, A. Burns, and A. J. Wellings. Calculating Controller Area Network (CAN) Message Response Times. *Control Engineering Practice*, 3(8):1163–1169, 1995.

[22] K. W. Tindell, H. Hansson, and A. J. Wellings. Analysing Real-Time Communications: Controller Area Network (CAN). In *Proceedings of* 15^{th} *IEEE Real-Time Systems Symposium (RTSS'94)*, pages 259–263, San Juan, Puerto Rico, December 1994. IEEE Computer Society.

[23] OSEK/VDX. Open Systems and the Corresponding Interfaces for Automotive Electronics. http://www.osek-vdx.org/.

[24] OSEK/VDX-Communication. Version 3.0.3, July 2004. http://-www.osek-vdx.org/mirror/OSEKCOM303.pdf.

[25] AUTOSAR. Homepage of Automotive Open System Architecture (AUTOSAR). http://www.autosar.org/.

[26] Stephen Cannan and Gerhard Otten. *SQL - The Standard Handbook*. MacGraw-Hill International, 1993.

[27] ASAP Standard. CCP - Can Calibration Protocol, Version 2.1, February 1999.

[28] AVL LIST GMBH. Cameo for Vehicle Use. http://www.avl.com.

[29] Vector-CANtech, Inc. CANape Graph. http://www.vector-cantech.com.

[30] Tei-Wei Kuo and Aloysius K. Mok. SSP: a Semantics-Based Protocol for Real-Time Data Access. In *Proceedings of 14th IEEE Real-Time Systems Symposium*, pages 76–86. IEEE Computer Society, December 1993.

[31] Thomas Nolte, Hans Hansson, and Lucia Lo Bello. Automotive Communications - Past, Current and Future. In *Proceedings of the* 10^{th} *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'05)*, Catania, Italy, September 2005.

[32] LIN Consortium. LIN - Local Interconnect Network. http://www.lin-subbus.org/.

[33] FlexRay Communications System - Protocol Specification. Version 2.0, June 2004.