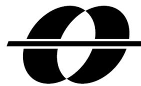
Mälardalen University Doctoral Thesis
No. 125

# Data Management in Component-Based Embedded Real-Time Systems

Andreas Hjertström

June 2012



MÄLARDALEN UNIVERSITY
SWEDEN

School of Innovation, Design and Engineering

Copyright © Andreas Hjertström, 2012
ISSN 1651-4238
ISBN 978-91-7485-064-2
Printed by Mälardalen University, Västerås, Sweden
Distribution: Mälardalen University Press

# Abstract

This thesis presents new data management techniques for run-time data in component-based embedded real-time systems. These techniques enable data to be modeled, analyzed and structured to improve data management during system development, maintenance, and execution. The foundation of our work is a case-study that identifies a number of problems with current state-of-practice in data management for industrial embedded real-time systems.

We introduce two novel concepts: the data entity and the database proxy. The data entity is a design-time concept that allows designers to manage data objects throughout different design and maintenance activities. It includes data-type specification, documentation, specification of timing and quality properties, tracing of dependencies between data objects, and enables analysis and automated validation.

The database proxy is a run-time concept designed to allow the use of state-of-the-art database technologies in contemporary software-component technologies for embedded systems. Database proxies decouple components from an underlying database residing in the component framework. This allows components to remain encapsulated and reusable, while providing temporally predictable access to data maintained in a database, thus enabling the use of database technologies, which has previously excluded, in these systems.

To validate our proposed techniques, we present a tool implementation of the data entity as well as implementations of the database proxy approach, using commercial tools, the AUTOSAR standardized automotive software architecture, and automotive hardware. Our results show that the presented techniques can contribute to the development of future component-based embedded real-time systems, by providing structured and efficient data management.

# Swedish Summary - Svensk Sammanfattning

Inbyggda realtidssystem blir allt vanligare i de produkter och tjänster vi använder. Utvecklingstakten går allt fortare och programvaran blir allt mer komplex. Inbyggda system finns idag i t.ex. mobiltelefoner, bilar, flygplan och robotar, där programvaran kan utgöras av flera miljoner rader kod och tusentals dataelement som är distribuerade över ett stort antal datorer ihopkopplade i nätverk. Utveckling och underhåll av dessa komplexa system medför en allt högre kostnad. För att utveckla elektroniksystemet är kostnaden, i en modern, avancerad bil idag, omkring 40% av den totala utvecklingskostnaden. Inom fordonsindustrin drivs denna utveckling av framför allt hårdare miljökrav, nya funktioner samt krav på bättre aktiv och passiv säkerhet.

För att hantera utvecklingen av dessa system försöker man göra informationen om systemet mer överblickbar genom att gruppera funktioner i olika komponenter som kan kommunicera genom ett förutbestämt gränssnitt. Denna teknik kallas för komponentbaserad utveckling. Komponentbaserade tekniker som används idag fokuserar främst på att hantera funktioner, och saknar bra metoder för att hantera den stora mängd data som utväxlas mellan komponenterna. Nya metoder för att effektivt hantera data har stor potential att göra både utvecklingen och exekveringen av inbyggda system enklare och mer kostnadseffektiv.

Denna avhandling introducerar nya koncept för hantering av data under utveckling, underhåll och exekvering av inbyggda komponentbaserade realtidssystem. Resultaten i denna avhandling baserar sig på en fallstudie som visar på stora problem med att hantera data inom industrin. Dessa resultat visar tydligt att hanteringen av data måste prioriteras mer och ingå som en integrerad del av utvecklingen av hela systemets arkitektur.

För hantering av data under utvecklings- och underhållsfaserna introducerar vi konceptet *data entity*. En *data entity* möjliggör för utvecklare att modellera och dokumentera varje dataelement i systemet korrekt redan i ett tidigt skede av utvecklingsfasen. Därutöver är det också viktigt att på ett enkelt sätt kunna skapa dokumentation och bedöma egenskaper, samt att visualisera dataflöden och beroenden mellan data för att öka den totala kunskapen om systemet. Tekniker för att hantera stora och komplexa datamängder i ett inbyggt system finns tillgängliga i form av databaser. Problemet är att de komponentbaserade teknikerna och databaserna är fundamentalt olika. Här finns ett tydligt glapp, vilket vi försöker överbrygga i denna avhandling. För hantering av data under exekvering introducerar vi konceptet *database proxy*, som möjliggör användandet av en databas utan att bryta mot grundläggande principer inom komponentbaserad utveckling. Syftet med detta är att komplettera den bristande datahanteringen inom komponentbaserad utveckling genom att utnyttja de beprövade tekniker som finns tillgängliga i en databas. Avhandlingen innefattar även ett antal implementationer av verktyg samt evalueringar av de ingående koncepten för hantering av data. Embedded Data Commander (EDC) innehåller en samling verktyg för att integrera och hantera "data entities" i en komponentmodell. Vidare har verktyg för konfigurering samt generering av "database proxies" i komponentmodellen SAVE har implementerats och evaluerats. Slutligen så har "database proxies" implementerats och evaluerats på hårdvara i ett AUTOSAR kontext.

To my Beloved Family

# Acknowledgements

This thesis marks the end of a great journey and at the same time the beginning of something new. To say that this has been an entirely smooth ride would be to lie to myself and others. There have been ups and downs, although the up side has by far exceeded the downside.

Two people have been by my side this entire journey, my supervisors Dr. Dag Nyström and Prof. Mikael Sjödin. Thanks for your excellent support and guidance, both in your role as my supervisors as well as "offline"! Dag, it has been a privilege to work with you. You have always been there for me (probably more than required) and when problems have arisen, your vision and never ending stream of new input and positive thinking carried me forward. Mikael, your ability to concretize and guide me to bring out the essence of my research and paper writing, is amazing.

A special thanks to my friend Peter Wallin. If you would not have started your PhD studies and so warmly recommended it, I would probably have missed this great opportunity. An additional thanks to Mimer Information Technology AB and ArcCore AB for the cooperation and input to the project.

I would also like to thank Jörgen Lidholm for the good discussions and being a great friend. Many people at the department have made this journey more enjoyable, thanks to Fredrik Ekstrand, Karl Ingström, Lars Asplund, Mikael Ekström, Kaj Hänninen, Stefan Cedergren, and all the other wonderful people. In addition, a special thanks to all the administrative people that have helped me with traveling arrangements, paper work, and being great companions.

To the whole Progress gang, Hans Hanson, Tomas Nolte, Ivica Crnkovic, Paul Pettersson, Hüseyin Aysan, Farhang Nemati, Moris Behnam, Mikael Åsberg, Severine Sentilles, Johan Kraft, Yue Lu, Stefan Bygde, Jan Carlsson, Aneta Vulgarakis and all others who have been great traveling companions, friends, and that have provided a lot of input to my work.

Most important, I thank my loving family, Anna, my son Felix, and my daughter Livia for supporting me and making my life wonderful. I love you. You are my everything!

*This work has been supported by the Swedish Foundation for Strategic Research within the PROGRESS Centre for Predictable Embedded Software Systems.*

Andreas Hjertström
Västerås, June, 2012

# List of Publications

## Papers Included in the Thesis

**Paper A:** Design-Time Management of Run-Time Data in Industrial Embedded Real-Time Systems Development, Andreas Hjertström, Dag Nyström, Mikael Nolin and Rikard Land, *13th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Hamburg, Germany, September, 2008*

**Paper B:** A Data-Entity Approach for Component-Based Real-Time Embedded Systems Development, Andreas Hjertström, Dag Nyström and Mikael Sjödin, *14th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Palma de Mallorca, Spain, September, 2009*

**Paper C:** Data Management for Component-Based Embedded Real-Time Systems: the Database Proxy Approach, Andreas Hjertström, Dag Nyström and Mikael Sjödin, *Journal of Systems and Software, vol 85, nr 4, p821-834, Elsevier, April, 2012*

**Paper D:** Introducing Database-Centric Support in AUTOSAR, Andreas Hjertström, Dag Nyström and Mikael Sjödin, *7th IEEE International Symposium on Industrial Embedded Systems (SIES), Karlsruhe, Germany, June, 2012*

**Paper E:** Data Management in AUTOSAR: a Tool Suite Extension Approach, Andreas Hjertström, Dag Nyström and Mikael Sjödin, *MRTC Report, submitted for conference publication*

## Additional Papers by the Author

INCENSE: Information-Centric Run-Time Support for Component-Based Embedded Real-Time Systems, Andreas Hjertström, Dag Nyström, Mikael Åkerholm and Mikael Nolin, *Proceedings of the Work-In-Progress (WIP) Session, 14th IEEE Real-Time and Embedded Technology and Applications Symposium, p 4, Seattle, United States, April, 2007*

Information Centric Development of Component-Based Embedded Real-Time Systems, Andreas Hjertström, *Licentiate Thesis, Mälardalen University Press, December, 2009*

Database Proxies for Component-Based Real-Time Systems, Andreas Hjertström, Dag Nyström, Mikael Sjödin, *22nd Euromicro Conference on Real-Time Systems, p 79 - 89, Brussels, Belgium, July, 2010*

Database Proxies: A Data Management Approach for Component-Based Real-Time Systems, Andreas Hjertström, Dag Nyström and Mikael Sjödin *MRTC, technical report*

# Contents


I Thesis 1

**1 Introduction 3**
- 1.1 Problem Description . . . . . 5
- 1.2 Thesis Outline . . . . . 6

**2 Background and Utilized Techniques 7**
- 2.1 Embedded Systems . . . . . 7
- 2.2 Embedded Real-Time Systems . . . . . 8
- 2.3 System Modeling and Development . . . . . 8
- 2.4 Data Management . . . . . 13

**3 Research Summary 19**
- 3.1 Technical Contributions . . . . . 19
- 3.2 Research Process . . . . . 22
- 3.3 Problem Description, Restated . . . . . 25
- 3.4 Thesis Contributions . . . . . 26

**4 State-of-the-Art 31**
- 4.1 Automotive Systems . . . . . 31
- 4.2 Design-Time Tools for Automotive Data Management . . . . . 34

**5 Conclusions and Contingency 37**
- 5.1 Conclusions . . . . . 37
- 5.2 Contingency . . . . . 39

**II Included Papers 47**

**6 Paper A:**
**Design-Time Management of Run-Time Data in Industrial Embedded Real-Time Systems Development 49**

6.1 Introduction . . . 51
6.2 Research Method . . . 52
6.3 Design-time Data Management . . . 56
6.4 Observations and Problems Areas . . . 60
6.5 Remedies and Vision for Future Directions . . . 65
6.6 Conclusions . . . 67
6.7 Future Work . . . 68

**7 Paper B:**
**A Data-Entity Approach for Component-Based Real-Time Embedded Systems Development 73**

7.1 Introduction . . . 75
7.2 Background and Motivation . . . 77
7.3 The Data Entity . . . 79
7.4 The Data Entity Approach . . . 82
7.5 The ProCom Component Model . . . 84
7.6 Embedded Data Commander Tool-Suite . . . 85
7.7 Use Case . . . 87
7.8 Conclusions . . . 90

**8 Paper C:**
**Data Management for Component-Based Embedded Real-Time Systems: the Database Proxy Approach 95**

8.1 Introduction . . . 97
8.2 Motivation . . . 100
8.3 Background . . . 102
8.4 System Model . . . 105
8.5 Database Proxies . . . 109
8.6 Implementation . . . 119
8.7 Performance Evaluation . . . 123
8.8 Conclusions . . . 129

**9 Paper D:**
**Introducing Database-Centric Support in AUTOSAR 135**
9.1 Introduction . . . 137
9.2 Background and Motivation . . . 138
9.3 System Model and Related Techniques . . . 139
9.4 AUTOSAR Concept Overview . . . 142
9.5 Database Proxies . . . 143
9.6 Integrating Database Proxy Support in AUTOSAR . . . 145
9.7 System Design and Implementation . . . 150
9.8 Evaluation . . . 152
9.9 Conclusions and Future Work . . . 156

**10 Paper E:**
**Data Management in AUTOSAR: a Tool Suite Extension Approach 161**
10.1 Introduction . . . 163
10.2 System Development Roles . . . 167
10.3 Data Management Tool Suite Extension . . . 167
10.4 Conclusions and Future Work . . . 171

# I

# Thesis

# Chapter 1

# Introduction

The evolution of embedded systems affects us all. Embedded systems are nowadays included in almost everything that surrounds us in our daily life. This has mostly to do with our increased demand for new functionalities that cannot be built, or are not practical to build, using traditional mechanics. Mobile phones, medical equipment, kitchen appliances, home entertainment systems, cars, and cameras are examples of such systems. Some of these are highly complex, and huge amounts of software are used to realize the different functionalities. In addition, the current trend is that systems are evolving from closed stand-alone devices to highly dynamic systems interconnected with the surrounding environment.

Developing these kinds of systems is a challenging task. Today, 90% of the innovations in a premium car are related to electronics and software. In addition, as many as 2500 software functions, sometimes dependent on each other, are distributed throughout a highly interconnected architecture with up to 80 Electrical Control Units (ECU) [1]. Furthermore, there is a more frequent mix of critical functionalities, such as breaking assistance, and non-critical functionalities, such as an infotainment system. Developing these kinds of systems is often associated with high cost [2].

From this evolution towards more complex and interconnected systems arises the need for more efficient means to manage data, perform diagnostics, and to provide predictable and dynamic data access. However, this thesis shows that the current state-of-practice of data management is not sufficient to cope with tomorrow's embedded systems. Therefore, the development of new techniques that deal with/can control data management are needed.

> *This thesis contributes to the future of embedded-systems development by identifying problem areas in the current state-of-practice, and by introducing new techniques for the development of component-based embedded real-time systems. These techniques comprise a holistic approach to data management by providing design-time support for modeling, management, documentation and analysis of run-time data, as well as run-time mechanisms for extracting, structuring, and the secure sharing of data.*

**Design-time data management:** A case-study, presented in this thesis, shows that developers are not provided with adequate techniques that enables efficient and up-to-date management of documentation of run-time data. The growing information volume, lack of tool support, poor routines, and the sometimes inadequate documentation, especially concerning internal data in nodes, are becoming an increasing problem. This has for example led to (i) obsolete documentation, (ii) redundant and stale data (data that is not removed due to unknown dependencies), and (iii) companies becoming highly dependent on the undocumented knowledge of individual developers.

The *data-entity approach* is presented in this thesis as a solution to facilitate efficient design-time management of run-time data. This approach has been evaluated and implemented into a tool-suite.

**Run-time data management:** In the development of functions, elevating the abstraction level and providing efficient tool support, is commonly used approaches to manage complexity. One such approach, which is increasingly used by industry today to raise productivity and reduce complexity, is Component-Based Software Engineering (CBSE). Structured development, standardized architectures, and reuse are mentioned as key factors for success. The CBSE focus lies on specifying and developing a component or a set of reusable components with certain functionality. However, CBSE does, so far, not provide structured support for managing data. This has in turn led to highly uncoordinated and ad-hoc management of data in many complex distributed systems [1, 2, 3].

Instead of reinventing data management techniques or developing ad-hoc solutions using internal data structures, the use of existing techniques should be promoted.

Outside the embedded community, a well-proven data management technique that offers standardized interfaces, efficient data management, dynamic access to data, user access control, and effective tool support is available, namely DataBase Management Systems (DBMS). Over the last few years, the use of DBMSs in embedded systems has increased. For example, many control-systems, and virtually all premium mobile phones, such as Apples iPhone and Android-based phones contain databases. However, the use of Real-Time DataBase Management Systems (RTDBMS) within industrial embedded real-time systems is still quite limited, even though there are a few commercial RTDBMSs available [4, 5]. Moreover, this is especially true for component-based systems.

Although both CBSE and RTDBMSs aim to reduce complexity, the co-existence between the techniques is non-trivial since their design goals are fundamentally different (i) within CBSE, decoupling of components from the context in which they are deployed is vital, whereas an RTDBMS provides a blackboard architecture that requires specific database knowledge to be embedded within components in order to access data, (ii) direct access to shared data introduces hidden dependencies between components, thereby violating a fundamental aim of CBSE.

The combined approach, to not only manage the functional complexity of the application and specifying components, but to also utilize the available tools and techniques offered by an RTDBMS, is a research area that is **not well established**.

*Database proxies* are presented in this thesis as a technique to close the gap between CBSE and RTDBMSs. Furthermore, database proxies have been implemented and evaluated as an approach to manage run-time data in the automotive initiative, AUTomotive Open System ARchitecture (AUTOSAR).

The research results presented in this thesis are applicable to many industrial application areas which depend on the efficient development of complex embedded real-time systems with a mix of critical and less critical functions. However, the focus of this thesis is automotive systems from which we borrow the technical background and terminology and apply our results to.

## 1.1 Problem Description

The continuous increase of complexity and new requirements on data management enhances the challenges with respect to performing design-time and run-time data management in a predictable, efficient and structured manner. Developers need new tools and techniques to aid them with the problems of today and tomorrow.

In an effort to understand the problematics concerning data management, this thesis investigates (i) the current problems within industrial embedded systems development, (ii) what tools and techniques could facilitate the development, as well as how and in what contexts they could be deployed.
To be precise, the thesis focuses on the following:

**F1** How is data currently managed in the industry and what are the main problems concerning design-time and run-time data management?

**F2** How can we support design-time data management within CBSE?

**F3** How can we support run-time data management within CBSE by utilizing state-of-the-art RTDBMS technology?

**F4** How can real-time data management techniques be integrated into an industrial development setting?

## 1.2 Thesis Outline

This thesis consists of two main parts. The first part presents an introduction, problem description and background to the scientific work carried out. The second part comprises a collection of published papers, papers A-E.
The remainder of the thesis is structured as follows:

**Chapter 2** presents the background to the research including the techniques and tools that have been used.

**Chapter 3** presents the main technical contributions, the research methodology, the research process, the problem definition, and a summary of the contributions. In addition, a summary of the included papers and my contribution to the results are presented.

**Chapter 4** complements Chapter 2 in that we describe the relevant state-of-the-art, which is related to the work carried out in this thesis.

**Chapter 5** concludes the introductory part of thesis and discusses possible contingency directions.

**Chapter 6-10** correspond to the papers that form the basis of this thesis.

# Chapter 2

# Background and Utilized Techniques

This chapter presents technical information about relevant areas within the scope of this thesis, such as embedded systems, real-time systems, component-based software engineering, and real-time database management systems. In addition, this chapter presents the major tools and techniques that have been used within the scope of this work, e.g. Save CCT, ProCom, AUTOSAR and Arctic Core, COMET, and Mimer SQL Real-Time.

## 2.1 Embedded Systems

An embedded system is a computer system, typically custom-made to perform a certain task or small set of tasks by interacting through sensors and actuators. Nowadays, these embedded systems can be found almost everywhere. They are used in watches, vehicles, robots, airplanes, and even toothbrushes. Their purpose is most often to reduce the number of mechanical parts by replacing them with electronics, in order to add functionality and/or to save costs. Most of these systems that we encounter in our everyday life are static, i.e. the software is never modified. However, there is an increase of devices that are more dynamic and where software can be continuously updated or replaced. An embedded system is characterized by limited hardware resources such as memory size and processor performance. Traditionally, embedded systems are either insolated devices or a part of a larger interconnected system. The current

trend driven by new demands on functionality and features is to change embedded systems from being stand-alone systems to being interconnected with other systems. An example of such a system of systems is Car-to-Car (C2C) communication [6], which allow cars to interact with each other to share information about, for example, a possible nearby hazard or to access the internet for infotainment services. This entails new requirements on how data is accessed and shared. The important aspects include flexibility, dependability and security.

## 2.2 Embedded Real-Time Systems

An embedded real-time system has additional requirements, compared to more general embedded systems, namely not only to perform its task correctly, but also to perform it predictably and within a predefined time interval: not too soon and not too late. Real-time systems are not only about performing a task as fast as possible. I general, real-time embedded systems interact with the environment where external events are perceived by sensors. These events are then analyzed and actuated upon, based on the result of the analysis. A typical example of a real-time system in a vehicle is an air-bag which has to be inflated within a certain time frame if activated by a collision. If the inflation is triggered too soon or too late the air-bag could cause the passengers even more harm than a complete lack of inflation.

Traditionally, real-time systems are divided into two main classes: hard and soft real-time systems. A *hard real-time system* should perform its actions before a defined deadline. A failure in meeting the deadline can have catastrophic consequences if the system is safety-critical. An air-bag is a typical example of such a system.

A *soft real-time system* usually manages less critical applications where a missed deadline can have a negative, but tolerable, effect on the performance of the system. Examples of such systems may concern the display of statistical information, the control of power windows, or to perform diagnostics.

In many applications, combinations of both hard and soft real-time tasks are used.

## 2.3 System Modeling and Development

Developing any type of complex software is most often a difficult and time consuming task. Nowadays, a common solution to manage this problem, is to develop tools and techniques to raise the level of abstraction, build models,

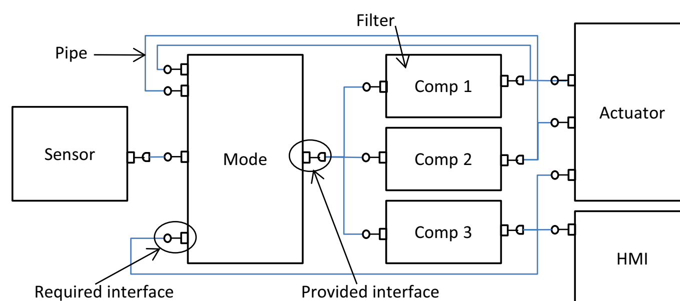



Figure 2.1: CBSE architectural example

generate code, and to reuse as much as possible. A frequently used technique within automotive software development is Component-Based Software Engineering (CBSE).

### 2.3.1 Component-Based Software Engineering

Component-Based Software Engineering aims to achieve a high level of abstraction when designing systems by dividing systems into well-defined and encapsulated building blocks called components. These components have well-defined communication interfaces that make them reusable entities that can be assembled to form entire systems. CBSE introduces a possibility to maintain and improve systems by replacing individual components. In this way, a significant amount of development effort and costs can be saved [7].

Figure 2.1 shows an example of a pipe-and-filter [8] component model where data is passed between components (filters) using connections (pipes). The entry point for the connection to the components is the interface (port). No communication outside of its interface is allowed.

A component can have two types of interfaces: required and provided. The required interface specifies what is needed as input to be able to process (filter) the data and output the result to the provided interface. Furthermore, a component can be either a white-box or a black-box component. A white-box component reveals its internal composition. This can enable developers to directly change the source code if needed. However, a changed behavior of the component, i.e., new versions emerges, can make it difficult to propagate, for instance, bug fixes. A black-box component is typically already compiled and does not reveal any internal details.

There is a great variety of component models which are suitable for different types of systems. COM [9], EJB [10] and .NET [11] are typically used for PC applications since they are not sufficiently taking important embedded systems requirements into account, such as timing properties, safety-criticality and the limited amount of resources available. Examples of component models aimed to at satisfying the requirements of embedded systems are the Rubus component model [12], SaveCCT [13], Koala [14], ProCom [15] and AUTOSAR [16].

In the following sections we describe component technologies which are used in papers B and paper C, namely SaveCCT, ProCom, and AUTOSAR.

### 2.3.2 SaveCCT

The SaveComp Component Technology (SaveCCT) [13] is focused on embedded control software for vehicle systems, with the aim to be predictable and analyzable. The applications are built by connecting input and output ports of components by using their interfaces (see Figure 2.2). Components are then executed using a trigger-based strict "read-execute-write" semantics.

A component is always inactive until triggered. Once triggered it starts to execute by reading data from its input ports to perform the computations. Data is then written to its output ports and outgoing triggering ports are activated. This allows the execution of a component to be functionally independent of any concurrent activity, once it has been triggered. SaveCCT also supports composite components. A composite component is a collection of components that are encapsulated into a single component with the same type of interface and behavior as a primitive component.

Figure 2.2 illustrates an example of a SaveCCT graphical representation of a component. There are two inports into the Engine Controller component, one data port and one trigger port. Data is read by the oilTempIO component from the oilTempSensor inport which is triggered with a frequency of 50Hz. Computations are done and results propagated onto the output port. In this case the output port is a combined trigger and output port.

SaveCCT supports manual design, integrated analysis tools, and automated activities such as task and code generation which transforms the component model into the execution model. In addition, an Integrated Development Environment (IDE) tool is provided, from which developers can develop components and graphically design the system. A number of tools are also available in the IDE for the automated formal analysis of components and architectures.

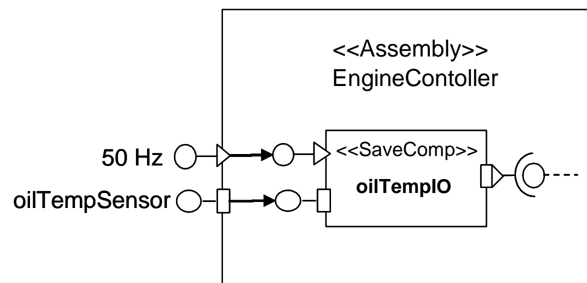



Figure 2.2: Save graphical application design

In the SaveIDE, component development as well as architectural and system modeling, is performed manually while system synthesis, glue-code generation and task allocation are fully automated. Resource usage and timing are resolved statically during the synthesis.

### 2.3.3 ProCom

The ProCom component model [15] extends SaveCCT by addressing key concerns in the development of control-intensive distributed embedded systems. ProCom provides a two-layer component model and distinguishes between a component model used for modeling independent distributed components with complex functionality (called ProSys) and a component model used for modeling smaller parts of control functionality (called ProSave).

In ProSys, a system is modeled as a collection of concurrent, communicating subsystems. Distribution is modeled explicitly, meaning that the physical location of each subsystem is not visible in the model. ProSys is a hierarchical component model where composite subsystems can be built out of other subsystems. This hierarchy ends with the so-called primitive subsystems, which are either subsystems coming from the ProSave layer or non-decomposable units of implementation (such as Commercial-Off-The-Shelf (COTS) or legacy subsystems) with wrappers to enable compositions with other subsystems.

A subsystem is specified by typed input and output message ports, expressing what type of messages the subsystem receives and sends. Message ports are connected through message channels. An example of this is illustrated in Figure 2.3, where a message channel is connected to three subsystems. A message channel is an explicit design entity representing a piece of information

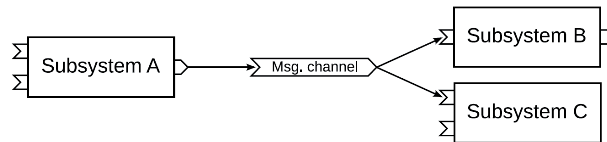



Figure 2.3: ProSys Component Model

that is of interest to one or more subsystems. The message channels make it possible to express that a particular piece of shared data will be required in the system, before any producer or receiver of this data has been defined. This will in addition allow information to remain in the design even if, for example, the producer is replaced by another subsystem.

### 2.3.4 AUTOSAR

The Automotive Open System Architecture (AUTOSAR) [16] is a consortium, where several of the main Original Equipment Manufacturers (OEM), suppliers and software developers within the automotive domain, are members. AUTOSAR defines a standard component model and middleware platform for the automotive electronic architecture. One of the fundamental characteristics of AUTOSAR is the layered architecture that separates the underlying infrastructure from the applications which consist of interconnected software components. Among other things, these abstraction layers enable hardware to be replaced without the need for software updates.

The strategy is to achieve a competitive market for vendors where an OEM can use components and whole applications from "any" supplier. The idea is also that as much as possible can and should be reused to save cost and to reduce time-to-market.

AUTOSAR employs the CBSE approach, where software is encapsulated as components which communicate via well-defined interfaces. The communication between components is managed by a Virtual Function Bus (VFB), which acts as a virtual abstraction of the underlying hardware. This enables early component integration in the development process as they are independent of the ECU hardware. The realization of the VFB when configuring the final target system is the Run-Time Environment (RTE). The RTE represents the concrete implementation of the VFB. The RTE acts as a communication center for both internal Electronic Control Unit (ECU) communication and information exchange between ECUs in the system.

### 2.3.5 ArcCore

Arccore AB [17] is a provider of the open-source Arctic Core embedded AUTOSAR platform developed in Eclipse [18]. The open-source solution, to be used for education and testing, includes Arctic Core and Arctic Studio which is an Integrated Development Environment (IDE). The commercial solution offers a number of licensed professional graphical tools to facilitate development of a complete AUTOSAR system. Arctic Core includes build scripts and services such as, network communication, memory, and operating system. In addition, drivers for different microcontroller architectures are also provided.

Components and their port-based interfaces are developed using the SoftWare Component Builder tool. The Extract Builder tool is used to add selected components to the ECU, connect ports and to validate the extract. The Run-Time Environment Builder models the VFB and generates a run-time implementation of the component communication. The configuration of the target platform is done in the Basic Software Builder tool which also generates the configuration files. Since Arctic Core is provided as open source, it is possible to extend it to also include additional functionalities.

## 2.4 Data Management

Data management is defined by the Data Management Association (DAMA) as:

> "the development, execution and supervision of plans, policies, programs and practices that control, protect, deliver and enhance the value of data and information assets" [19].

All computer systems involve the usage of data in some way. As both the amount of data and its use increase in an area, an increase of complexity is often unavoidable. Routines for the documentation, storage, retrieval and security of data thus become particularly important.

In this thesis we distinguish between two types of data management: design-time data management and run-time data management. This can be exemplified by an embedded system, where design-time data management refers to how run-time data is organized and documented during the design and development phase. Run-time data management refers to how data is organized and accessed in memory during execution of the system.

### 2.4.1 Design-Time Data Management

Design-time data management is the interactive link between a designer and the underlying data management system. Management of data at design-time has been and still is a fundamental part for managing the complexity of large scale and data intensive systems in order to decrease time-to-market, costs, and to increase the quality of the system. A key factor is having up-to-date and correct information about data residing in the system available during the whole development cycle. Proper documentation and structure allow for easy access to information, such as properties that can specify unique naming, type, size and where the data is used. In addition, this usually includes version handling of all design information and providing support for multiple user interactions.

The number of dedicated design-time tools for managing data in embedded real-time systems is quite limited. Most tools focus on the properties of individual data elements and how to create or define new data types. They do to a limited extent present an overview of detailed information on how and where data is used in the system during development [20, 21].

### 2.4.2 Run-Time Data Management

Run-time data management concerns how data is managed during execution of the system. Traditionally, most embedded systems developers handle data ad hoc and/or reinvent new solutions in an effort to meet the requirements of the system. This is often done using internal data-structures. Many of today's systems are developed in a distributed manner, which in turn could lead to many different solutions and strategies residing in the same system. A result of this is that large complex systems become less flexible, difficult, and demanding to maintain and extend.

Outside the embedded community, powerful tools and techniques are well established and have facilitated data management in complex data-intensive systems, such as financial markets, where they have been used for decades.

Similar as the techniques used for modeling a system or for the development of functions with a component-based approach to accomplish a higher level of abstraction, techniques to achieve a more dynamic, structured, and maintainable data management is available [22].

Database Management Systems (DBMS) are used to organize large amounts of data. Figure 2.4 shows a high level picture of a DBMS system. To put it simply, a DBMS is an interface and abstraction layer that manages access to

the physical data stored in memory. A typical application area has so far been large enterprise systems such as libraries, commercial web-sites and financial markets. Examples of enterprise mainstream DBMS are Oracle [23], Microsoft Access [24] and MySQL [25].

One of the main benefits with a DBMS is the ability to access data using a standard language. The Standard Query Language (SQL) [26] is the most common database access language, which in addition is supported by many high-level tools, for uniform data access. In order to access data or manipulate data in the database, a number of operations such as, SELECT, INSERT, and DELETE are used. One or several operations that is executed, as a single logical block of work in the database, is called a transaction. A transaction is either performed completely by ending its block of operations with a COMMIT. If the transaction is aborted before the COMMIT, a rollback to its original state is performed. A successful COMMIT makes the changes permanently stored in the database and must take the database from one consistent state to another.

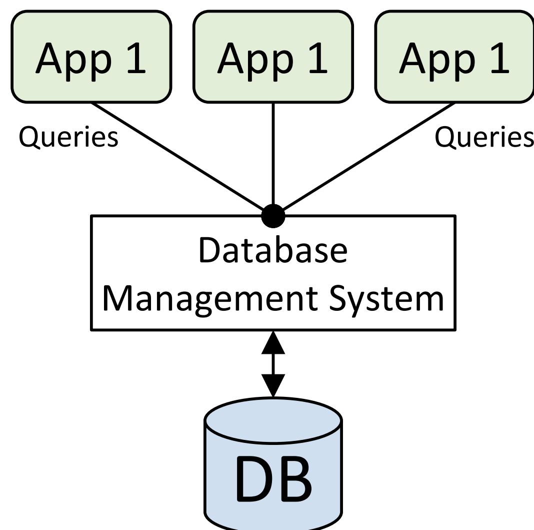



Figure 2.4: DBMS overview

To ensure a correct behavior and safe sharing of data, it is often required that a database transaction should conform to the ACID properties [27]:

- **A**tomicity: either all information in a database transaction is updated or none at all.
- **C**onsistency: after a transaction is completed the database will be in a valid state. If not, the transaction must be rolled back.
- **I**solation: changes that are made to the database will not be revealed to other users until the transaction is committed.
- **D**urability: any change to the database is permanent. The result of a committed transaction cannot be reverted.

Most DBMSs use concurrency control in order to enforce the ACID properties while handling concurrent operations, in order to avoid transaction conflicts to achieve logical correctness. The most commonly used algorithm is Two-Phase-Locking (2PL) [28] and optimistic concurrency-control [29].

The increasing amount of data and growing data complexity have increased the need for a DBMS also in embedded systems. There are now several commercial embedded DBMSs available that are suited for the specific needs, such as a small footprint, of embedded systems [4, 5, 30].

### 2.4.3 Real-Time Database Management Systems

Embedded real-time systems have different requirements compared to large enterprise systems. CPU usage, footprint and availability are highly important. Embedded Real-Time DataBase Management Systems (RTDBMSs) is developed to support real-time constraints in order to provide a deterministic timing behavior management of data in complex embedded real-time systems. For safety-critical embedded real-time systems, predictable access to data is one of the most important requirements [31].

Compared to the concurrency control algorithms used in a general-purpose DBMS, most RTDBMSs relax the semantics of the ACID properties in order to fulfill the real-time properties. This is sometimes necessary in order to comply with domain-specific requirements [32].

A commonly used concurrency control algorithm that enforce real-time properties is the Two-Phase-Locking, with High Priority abort (2PL-HP) [33] which favors transactions with high priorities, thus aborting lower prioritized transactions, in case of a conflict.

### 2.4.4 COMET RTDBMS

The COMponent-based Embedded real-Time database system [34] (COMET RTDBMS) is the result of a research cooperation between Linköping and Mälardalen University. The focus was on real-time systems in general and vehicle systems in particular. COMET is a real-time database management system is intended to be used as a tightly integrated part of the control-system, providing new techniques and functionalities such as, providing applications with support for a mix of hard and soft real-time requirements.

COMET implements the database pointer interface [35] which is a hard real-time database access-method which uses an application pointer variable to access individual data in an RTDBMS. A key property of the database-pointer concept is that reads and writes through database-pointers have deterministic execution-time with bounded and negligible blocking [36]. They also allow SQL-based soft real-time database transactions to be executed in the background without any predictability loss due to any concurrent database-pointer accesses (i.e. no starvation, conflicts, or restarts of transaction can be caused by database pointers [35]).

To guarantee hard real-time predictability for database accesses while eliminating starvation issues for soft real-time SQL queries, COMET uses the 2V-DBP concurrency-control algorithm [36] that combines versioning and pessimistic concurrency control. 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.

Some of the technologies developed for COMET, including the database pointer concept, has later been adopted by the commercially available real-time database system Mimer SQL Real-Time Edition [4].

### 2.4.5 Mimer SQL Real-Time

Mimer SQL Real-Time (Mimer RT) [4] is a commercial RTDBMS intended for applications such as vehicle systems, process automation and telecommunication systems. Mimer RT supports applications with both hard real-time and non-real-time requirements to safely share data without putting real-time predictability at risk. Hard real-time applications utilize the RTAPI interface to access data using database pointers while non-real-time applications use standard SQL interfaces. Mimer RT combines the standard client/server architecture for SQL queries with an embedded library architecture for real-time access. The client/server architecture allows standard interfaces and tools to be used to access data both locally and remotely.

# Chapter 3

# Research Summary

This thesis presents a number of scientific contributions to facilitate design-time and run-time data management within the area of component-based embedded real-time systems. This chapter presents the technical contributions, presents the research methodology and research process, restates the problem definition, outlines the thesis contributions, and gives a résumé of the included papers.

## 3.1 Technical Contributions

A **Data entity** is a design entity that encapsulates metadata into a compilation of knowledge for run-time data items in the system.

Developers are provided with an additional architectural view, the data architectural view, which complements the traditional component-based design approach. The approach enables run-time data to be acknowledged as design objects during development, as each data item is tightly coupled with proper documentation and where properties such as usage, validity and dependency can be modeled. This enables developers to have an increased knowledge and understanding of the system.

Furthermore, as data entities are defined completely separate from the development of components and functions, data entities persist in the system regardless of any component, function or design changes. Figure 3.1 shows the metadata that is associated with a data entity.

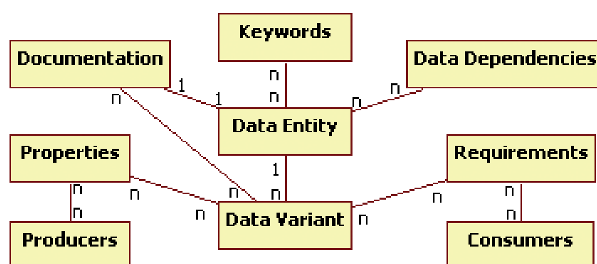



Figure 3.1: Data entity description

Figure 3.2 illustrates how our approach (right-hand-side) complements the traditional component-based design approach represented by dotted lines on the (left-hand side). The component-based approach includes tools for setting up the system architecture, developing components, and to perform analysis. The central database in the middle of the figure acts as the communicating link between the two approaches.

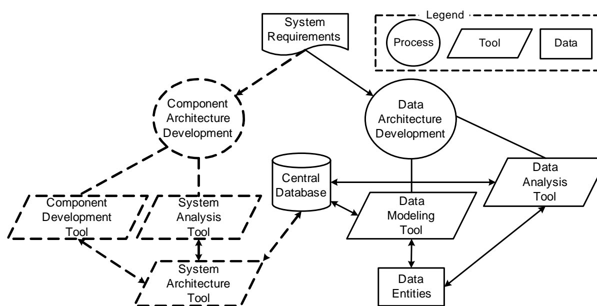



Figure 3.2: The data entity approach

We have developed a tool suite named, the Embedded Data Commander, that provide support for modeling of data entities to keep track of system data, present accurate documentation, and a data analysis tool to perform early analysis on data items. The data entity approach and tool suite serves a direct remedy to some of the problems identified in Paper A where one of the investigated systems suffered from as much as a 15% overhead because of unused and

stale data was being produced. This was due to unknown dependency issues where hardly anything could be removed due to a lack of knowledge.

The goal is to achieve higher software quality, lower development costs, and to provide higher degree of control over the software evolution process.

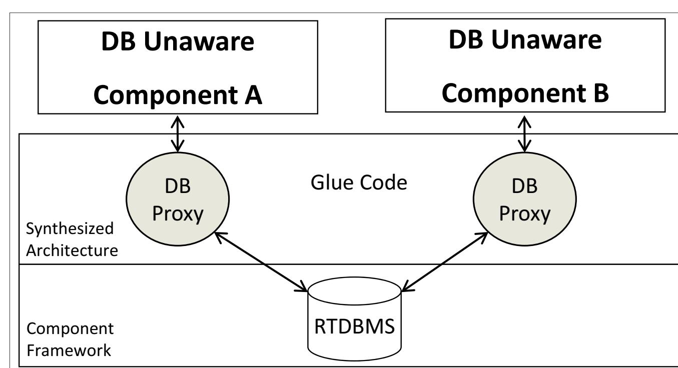



Figure 3.3: Database proxies connecting components to an RTDBMS

The **Database proxy** concept enables a successful integration of an RTDBMS into a component-based system. As illustrated in Figure 3.3, a database proxy is part of the component framework, thus external to the component. The task of the database proxy is to enable for components to interact with an RTDBMS using their normal interfaces. The database proxy is placed between the component and the RTDBMS and includes pieces of code that translates data from a components port to a database call and further on to an RTDBMS residing in the component framework and vice versa. These pieces of code are neither a part of the component nor a part of the RTDBMS; instead database proxies are automatically generated glue-code synthesized from the system architecture.

*Hard proxies* use state-of-the-art database pointers provide predictable access to individual data elements, and *soft proxies* use an SQL interface to provide flexible access to data. A hard real-time database-pointer provides direct access to a data element in memory without calling the database server. In addition, that a hard proxy only translates native data types such as an integer, character or float, implies that no unpredictable type conversions or translation of complex data types that require unbounded iterations are allowed.

```
/*** Original code example ***/
void function(){
DisableAllInterrupts();
Read_Value_Port_1(&Port_1_data_1->value);
EnableAllInterrupts();
}
/*** Database proxy code ***/
void function_DBProxy(){
MimerRTGetInteger(DBP_Actuator, &Port_1_data_1->value);
}
```

Figure 3.4: Differences between regular code and database proxy code

Figure 3.4 illustrated the code differences, using c-code, between an implementation not using, and using hard database proxies. In the original code example, all interrupts are disabled before the call to read the value is made. After the value has been read, interrupts are enabled. When using a database proxy to read the value from the database using a database pointer, the difference to the original code, is that the interrupt disable is not needed within the database proxy, since this is managed by the database.

A soft proxy is typically used for graphical interface components, logging components, and diagnostics components. Therefore, soft proxies emphasize support for more complex data structures by using an SQL interface, towards the RTDBMS.

## 3.2 Research Process

The methodology that has permeated all of the research presented in this thesis is based on the *technology transfer model* presented by Gorschek et al. [37]; see Figure 3.5. However, since this thesis is not a fully integrated industrial project, steps 5 and 7 have not been included and are left for future work. In addition, we have used research approaches: techniques and descriptive models, as well as the validation techniques: implementation, evaluation, and experience techniques described by Shaw [38].

Our research has been guided by the following process (see Fig 3.6), where each item corresponds to specific elements of the *technology transfer model*:

- **Identifying Problems:** A literature study of the state-of-the-art and a case-study conducted at five industrial companies identified that the current status within data management in component-based embedded real-

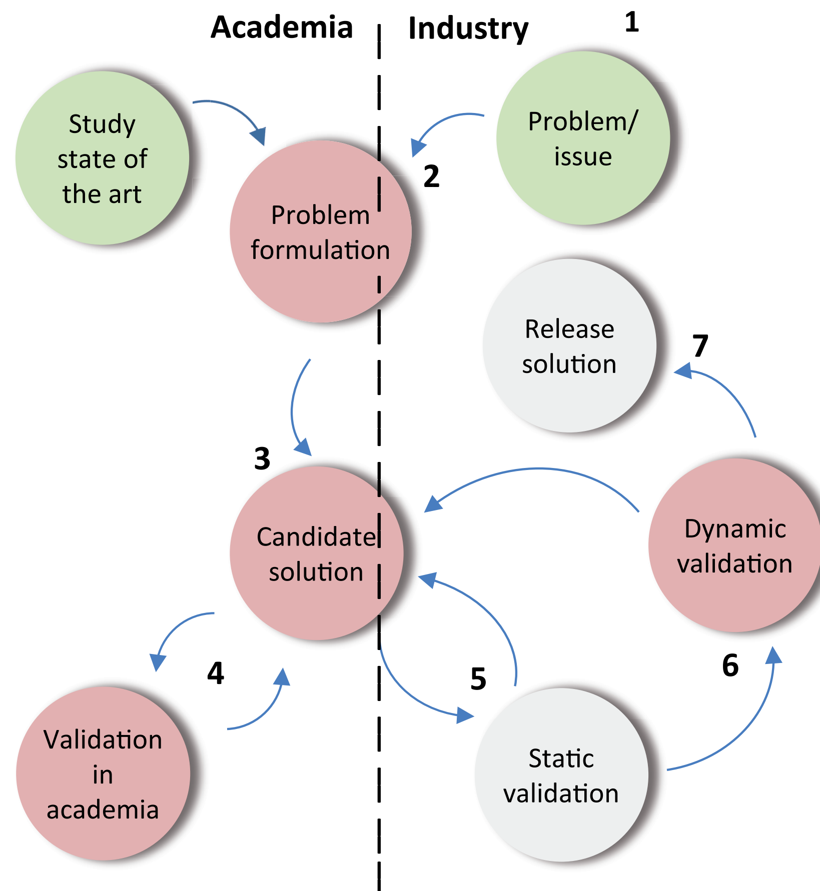



Figure 3.5: The technology transfer model

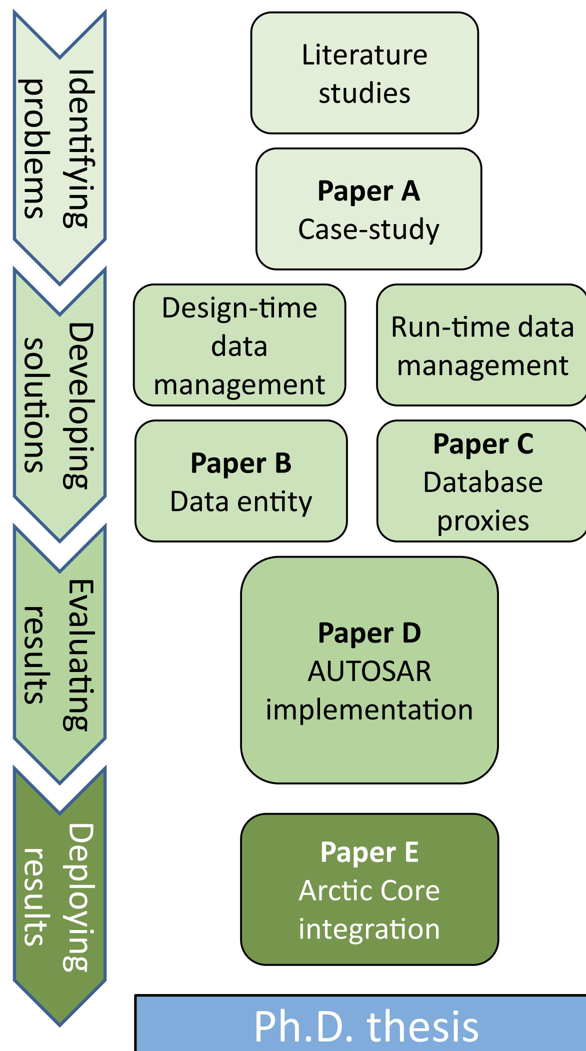



Figure 3.6: Research progression overview

time systems is indeed becoming an increasing challenge for developers and system architects. The case-study, which constitutes **Paper A**, identifies a number of problem areas and possible remedies. These research advances correspond to steps 1 and 2 in Figure 3.5.

- **Developing Solutions:** The continued research, directly targeting the identified problem areas, was sub-divided into two research directions, design-time and run-time data management, which resulted in **papers B** and **C**. **Paper B** presents the data-entity approach that complements design-time tools and techniques with an additional architectural view as well as tools for data management. **Paper C** presents a solution, denoted database proxies, for a successful integration of an RTDBMS into a CBSE setting. Both papers **B** and **C** correspond to steps 3 and 4 in Figure 3.5.
- **Evaluating results:** In the next phase, to validate our approach in an industrial setting, the implementation and evaluation of database proxies in AUTOSAR, a state-of-the-art component-based development architecture, was carried out. An authentic automotive hardware node was used in the evaluation. This resulted in **Paper D**, which corresponds to step 6 in Figure 3.5.
- **Deploying results: Paper E** presents techniques for how to integrate our approach into the commercially available development tool suite, Arctic Core. The use of an RTDBMS in conjunction with database proxies will be included in the meta-model and presented in the graphical user interface, as an additional application design option. This final step corresponds to step 6 in Figure 3.5.

## 3.3 Problem Description, Restated

The continuous increase of complexity and new requirements on data management enhances the challenges with respect to performing design-time and run-time data management in a predictable, efficient and structured manner. Developers need new tools and techniques to aid them with the problems of today and tomorrow.

In an effort to understand the problem concerning data management, (i) this thesis investigates the current issues within industrial embedded systems development, and (ii) what tools and techniques could facilitate that development, i.e. how and in which contexts such systems/tools could be deployed.

To be precise, the thesis focuses on the following:

**F1** How is data currently managed in the industry and what are the main problems concerning design-time and run-time data management?

**F2** How can we support design-time data management within CBSE?

**F3** How can we support run-time data management within CBSE by utilizing state-of-the-art RTDBMS technology?

**F4** How can real-time data management techniques be integrated into an industrial development setting?

## 3.4 Thesis Contributions

The present thesis makes the following major contributions to the area of complex component-based embedded real-time systems:

1. A case-study that emphasizes the importance of data management in order to increase the knowledge and understanding of the system. Ten problem areas within documentation, tool support and routines, as well as remedies, are presented to achieve a more data-centric development strategy. This contribution corresponds to research focus **F1**.

2. The concept of *data entity*, which enables design-time modeling, management, documentation and analysis of run-time data. This contribution corresponds to research focus **F2**.

3. A technique denoted *database proxies*, which enables the integration of an RTDBMS into a component technology. Database proxies are automatically generated glue-code that translates data between component ports and an RTDBMS that resides in the component framework. This contribution corresponds to research focus **F3**.

4. An implementation of tools and techniques for the realization of *data entities* into a component-based development suite named Save CCT. This contribution serves as a validation of contributions 2 and 3.

5. An implementation and evaluation of *database proxies* in AUTOSAR, using industrial tools and hardware. This contribution serves as a possible technology transfer of contribution 3 and corresponds to research focus **F4**.

Part II of the thesis contains five papers, denoted Paper A to Paper E. Each of these papers is summarized below.

**My contribution** to each of the papers has been to define the different concepts, implement the tools, perform the evaluations and be the main author.

### 3.4.1 Paper A

**Paper A:** Design-Time Management of Run-Time Data in Industrial Embedded Real-Time Systems Development, Andreas Hjertström, Dag Nyström, Mikael Nolin and Rikard Land, *13th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Hamburg, Germany, September, 2008*

In this paper, we present the results of an industrial case-study conducted at five companies where we have studied the current state of practice in data management and documentation in embedded real-time systems. The case-study identifies a lack of design-time data management, which often results in costly development and maintenance. It confirms that new processes and techniques for achieving an efficient, up-to-date and satisfactory documentation are needed. Furthermore, inadequate tools and routines for data management of internal ECU data results in costly development and maintenance, which is often entirely dependent on the know-how of single individual experts. Ten specific problems are identified, four key observations and six suggested remedies are presented.

### 3.4.2 Paper B

**Paper B:** A Data-Entity Approach for Component-Based Real-Time Embedded Systems Development, Andreas Hjertström, Dag Nyström and Mikael Sjödin, *14th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Palma de Mallorca, Spain, September, 2009*

This paper presents our design-time data management approach, denoted the *data entity* approach. The motivation for this approach stems from identified problems presented in Paper A.

The approach allows efficient design-time management of run-time data to be included in component-based real-time embedded systems development as an additional architectural view that complements the traditional architectural component inter-connections and development view. The *data entity* approach elevates run-time data to be acknowledged as first class objects of the architectural design, and allows data to be modeled and analyzed in an early phase of the development.

The paper also presents a design-time data management tool suite called Embedded Data Commander (EDC), where data entities can be created, retrieved and modified. Furthermore, they can be associated with design entities such as message channels created from the ProCom component architecture development. In addition, the tool allows documentation to be generated from an ongoing project as well as presenting data dependencies, e.g., who the producers and consumers of a data item are. EDC provides tools for data modeling and analysis.

### 3.4.3 Paper C

**Paper C:** Data Management for Component-Based Embedded Real-Time Systems: the Database Proxy Approach, Andreas Hjertström, Dag Nyström and Mikael Sjödin, *Journal of Systems and Software, vol 85, nr 4, p821-834, Elsevier, April, 2012*

To close the gap between two existing techniques used by the industry to manage the complexity and increase the flexibility, of component-based embedded real-time systems development, we introduce the concept of database proxies. Database proxies are automatically generated glue-code synthesized from the system architecture and used to decouple components from the underlying database in order for components to preserve the components encapsulation and possibility of reuse. A component with direct access to an RTDBMS is dependent on that specific RTDBMS and may not be useable in an alternative environment.

The use of an RTDBMS in the component-based setting provides a new range of possibilities, such as structured data management, as well as flexible and predictable access to both critical and non-critical data. By using database proxies in conjunction with state-of-the-art database pointer techniques, developers can employ the full potential of both CBSE and an RTDBMS. With this approach, developers can focus on application development instead of reinventing data management techniques or develop solutions using internal

data structures. As proof of concept this work has been implemented in the SaveCCT framework where a system can be designed with or without a database. In addition, the database proxy properties are generated to glue-code from its specifications and further to target c-code. Furthermore, an evaluation of the execution time overhead and additional memory overhead is in the order of 1-2%.

### 3.4.4 Paper D

**Paper D:** Introducing Database-Centric Support in AUTOSAR, Andreas Hjertström, Dag Nyström and Mikael Sjödin, *7th IEEE International Symposium on Industrial Embedded Systems (SIES), Karlsruhe, Germany, June, 2012*

In this paper we take the database proxy concept from research-oriented techniques to an industrial setting by showing how a real-time database management system can be integrated into the basic software of AUTOSAR by using database proxies. The aim with the approach is to show how a database-centric strategy can facilitate the development and maintenance of an automotive system by providing the proven capabilities of an RTDBMS. Database proxies are used to manage the communication between components on the AUTOSAR Virtual Function Bus (VFB). The COMET RTDBMS is successfully integrated into the AUTOSAR Basic Software (BSW), and evaluated on an authentic automotive hardware node. The evaluation shows that our approach can be used without components being aware of it, jeopardizing system performance or safety. Moreover, this greatly simplifies the development of soft real-time functions that process large data volumes, e.g., for statistics and logging. Our measurements show that the concept only introduces a CPU overhead in the order of 4% under typical workload conditions.

### 3.4.5 Paper E

**Paper E:** Data Management in AUTOSAR: a Tool Suite Extension Approach, Andreas Hjertström, Dag Nyström and Mikael Sjödin, *MRTC Report, submitted for conference publication*

In this paper, our research is transferred from academia to industry as a proof of concept and to demonstrate the usefulness of our research results. We present how a database proxy can be integrated into the development of automotive systems using industrial tools. Our approach enables a clear separation of

concerns between the system architect, component developer, and the DataBase Administrator (DBA). This separation of concerns allows each part to be managed and reconfigured independent of each other. Furthermore, a plug-in approach, developed for the Arctic Core tool suite and an integration of the Mimer SQL Real-Time [4] database into the basic software of AUTOSAR is presented.

# Chapter 4

# State-of-the-Art

The aim with this chapter is to present relevant background information regarding the development of automotive systems and we introduce some tools and techniques that are important in this respect. This chapter complements Chapter 2 in that we describe the related areas that are mostly orthogonal to the work performed in this thesis.

## 4.1 Automotive Systems

Vehicles have in recent years evolved from mechanical systems to advanced computer-controlled systems where mechanical parts are continuously replaced by computer-controlled functions to achieve higher safety, less pollution, and more comfort. In the early phase of this technological transformation, non-critical tasks such as central locking and parts of the engine-control were handled by small embedded computers. In today's automotive systems, more and more safety-critical functionality is replaced by computers that control breaks, steering, airbags, etc.

In addition, the trend is that automotive systems are evolving from closed stand-alone systems to highly dynamic systems interconnected and communicating with the surrounding environment. There is a lot of research on new technologies such as Car-to-Car (C2C), and Car-to-Infrastructure (C2I) [6] communication. As an example, the system can be used to inform nearby vehicles of possible dangers that have been discovered and even of its own location to avoid a possible collision. In addition, the user demand for integrating third-party applications, such as smart phones and internet connectivity, poses

a range of new challenges concerning areas such as secure data access and handling shared data between safety-related and non-safety-related functionalities.

The amount of software in high-end automotive systems is increasing and is estimated to have reached 1 GB [39]. In addition, an advanced vehicular system can include more than 80 Electrical Control Units (ECUs) which exchange in excess of 2500 signals [40] on several separate bus systems such as CAN, FlexRay, LIN and MOST [41, 42, 43, 44]. To complicate matters further, a high-end system can have more than 2000 functions that often are highly dependent on each other. Consequently, the costs related to software development and electronics have surged and can reach as much as 40% of the total development costs of a car [2]. This has increased the need for flexible platforms that can accommodate entire product lines for several years, in an effort to reduce development costs [45].

The development of functionality in these complex automotive systems requires expertise within the areas of infotainment, engine control, safety-critical applications, etc. As a result of this, Original Equipment Manufacturers (OEM), in-house development is increasingly replaced by Commercial-Off-The-Shelf (COTS) parts from various suppliers with expertise in certain areas [2]. Integrating heterogeneous subsystems from different sources while managing their evolution and maintenance constitutes a great challenge [46]. In addition, as stated by Schulze et al. [39] and Saake et al. [3], the ad-hoc and/or reinvented management of data for each ECU with individual solutions using internal data structures can lead to concurrency and inconsistency problems. A standardized and overall data model and management system has great potential as a solution to deal with the distributed and uncoordinated data in these complex systems [1].

A lot of focus within the automotive industry is the use of a standardized software architecture such as the AUTOSAR [16]; see section 2.3.4. Another approach targeting complexity, cost, time-to-market, etc. when developing automotive systems is Model Driven Engineering (MDE).

### 4.1.1 Model Driven Engineering

Model Driven Engineering (MDE) supplies an abstraction of reality by providing a model of reality that relates to a given aspect of the system. It often does this by only representing a selected part of the system, thus simplifying the overall view. To build a model that represents all aspects of reality or of a system would not only be hard, it would in many respects be impossible to understand [47]. Within computer science, systems are often divided into mod-

els that represent aspects such as requirements, system architecture, validation, etc. MDE has also evolved modeling from being a quite rudimentary form of documentation to serving as formal interchange formats used by tools for precise implementation purposes within computer engineering.

A model must conform to some specified (language and grammar) rules called meta-model in order to be interpretable. There is also the possibility to build hierarchical models, i.e., models of models. An important feature of MDE are to transform one model into another or to generate e.g. code or reports [48].

In the automotive sector, MDE often uses several different modeling languages such as EAST-ADL2 [49], TADL [50], MARTE [51] or SysML [52], which are based on and/or extend concepts from UML.

### 4.1.2 EAST-ADL2

EAST-ADL2 [49] is an automotive architecture description language, developed as a UML 2.0 profile [53] within the ATESST project [54]. EAST-ADL2 aims to support the development of complex automotive software by providing structured system information management at five levels: vehicle level, design level, analysis level, implementation level and operation level.

1. **Vehicle level**: focuses on the features visible to the end users, such as breaks or collision warning. A feature is specified by use cases and requirements to meet, for instance, the configuration of a specific vehicle variant.

2. **Analysis level**: provides analysis support of what the system shall do and describes the functions that enable the available vehicle features. This allows functions to be integrated and validated before the actual software and hardware are developed.

3. **Design level**: includes a behavior description of the functionalities without any implementation constraints in order to meet non-functional constraints such as specific supplier concerns or reusability issues. The focus is on the interaction and behavior of functions.

4. **Implementation level**: a system description of software components, middleware etc.

5. **Operation level**: describes low-level details concerning the deployment on to hardware.

The implementation and operation levels are highly related and complement the AUTOSAR basic software description. In short, AUTOSAR defines the final implementation details and EAST-ADL2 defines the logical and functional architecture aspects.

Neither EAST-ADL2 nor any other of the previously mentioned modeling languages provides specific techniques or support for data management.

## 4.2 Design-Time Tools for Automotive Data Management

Design-time data management is a recognized problem in the automotive industry. Hence, a couple of tools that provide partial solutions to the problem have been developed. To provide data management support at design-time, the dSpace Data Dictionary tool [20] holds information about an ECU for calibration and code generation. It is a central data container for model-independent data management that is used to share information such as interface variables, their scalings, typedefs, etc. throughout an entire project.

A data dictionary is also used for managing AUTOSAR properties, alongside AUTOSAR specification properties at block level in Targetlink [20]. The input to the dSpace data dictionary is templates generated from Simulink [55]. The data dictionary provides access to information such as specifics on C modules, function calls, tasks, variable classes, and data variants. In addition, developers are provided with support to import and export AUTOSAR software-component XML description-files, which can be used by other tools. The information included in the dSpace data dictionary reflects the information included in the software component templates and does not include information about the overall system and what data and signals are included. It is also possible to specify and produce signal lists and spreadsheets with information regarding data. The start of the development process in this tool is to model components and their structure. In contrast to the data entity approach presented in this thesis, the tool does not focus on managing or visualizing the data flow in the system. Neither does it include analysis techniques to make data dependencies visible.

The Automotive Data Dictionary (ADD [21]), is an additional tool that provides a repository solution to centralize data declarations and ensure label and variable uniqueness for companies. ADD has an interface towards MATLAB and Simulink and is used to develop ECUs within the automotive industry. The main goal is to close the gap between software development and

requirements engineering in order to avoid inconsistency throughout the whole development process. It gives the developers an overview of the data specification but does not include any implementation details. Contrary to our data entity approach, ADD mostly focuses on requirements engineering and unique labeling and does not cover information about data flow and data dependencies.

# Chapter 5

# Conclusions and Contingency

In this thesis, we bring together new techniques in order to take a holistic approach to data management in the development of component-based embedded real-time systems.

## 5.1 Conclusions

This research stems from the rapidly growing complexity when it comes to the amount of data and data flow between components in today's embedded real-time systems. This has so far not been addressed by contemporary development techniques. Instead, the focus tends to be on achieving a higher level of abstraction by encapsulating functionality. Current research shows that the state-of-practice for managing data on the system level and internally in individual nodes is not adequate to meet the increasing complexity when building the embedded systems of tomorrow. This was also confirmed by the case-study presented in this thesis.

A starting point was to develop techniques that would enable a Real-Time Database Management System (RTDBMS) to become a native part of the design and to manage the data flow between components. The use of an RTDBMS within data-intensive applications, with high demands on flexibility and structured data management, is not new. It has been a natural next step as systems have evolved and the requirements on data have become increasingly complex. However, the use of an RTDBMS in an embedded setting, and partic-

ular in real-time systems, is still somewhat unconventional, even though today there are several commercial RTDBMSs tailor-made for resource-constrained real-time embedded systems.

To manage data complexity, the use of an RTDBMS in conjunction with Component-Based Software Engineering (CBSE) is an interesting challenge, which we have tackled in this thesis. This approach is not obvious since the design goals of CBSE and RTDBMSs stand in opposition to each other. To overcome these contradictions we have introduced the concept of *database proxies*. In addition, we have shown, through implementations and evaluations using AUTomotive Open System ARchitecture (AUTOSAR) compliant tools and automotive hardware, that this approach offers a number of new possibilities at limited cost with respect to execution time overheads and resource consumption.

Furthermore, this thesis presents a new design-time artifact named *data entity*. The idea behind a data entity is to elevate run-time data to becoming a first-class citizen in the system architectural design and to introduce a data architectural view. Data entities allow data to be documented, modeled and analyzed separately from the actual component implementation. Since data entities are designed in separate, they could be used in other component models, with channels, regular connections or other design approaches then those investigated within this thesis.

In this thesis we have implemented support for data-entities within the SaveComp component technology as well as support for database-proxies with the ProCom and AUTOSAR component technologies. However, the two concepts have not been designed for use with any particular underlying component technology and we believe that results could be generalized to most component technologies that are based on statically configured pipes-and-filter style components. As a run-time backend we have use Mimer Real-Time edition; however any underlying storage-technology could be used with appropriate modification of our glue-code generators. One should note though, that in order to achieve hard real-time support, the storage-technology needs to hard real-time primitives.

It is our firm belief that new, adequate data management techniques are crucial to meet future requirements and to contribute to the evolution of component-based embedded real-time systems development.

## 5.2 Contingency

There is much work to be done in the area of data management in component-based embedded real-rime systems. Existing techniques must incorporate data management as a natural part of the development. In addition, new techniques and tools have to be developed to keep up with the evolution of systems. In this thesis we present techniques that could be of use with respect to some aspects of the development of large complex systems. However, a lot more research and many more solutions are needed in an effort to cover the whole area.

Although we have implemented and evaluated database proxies using commercial tools, this has been done in a "controlled" research environment. A natural next step would be to test the approach in an industrial project to evaluate its usefulness in practice and in the development process. This would be a final step corresponding to steps 5 and 7 in the methodology presented in section 3.2. This would require full-scale integration, in for instance Arctic Core, complete with automatic code generation and validation procedures. Since our implementation is not entirely complete with all functionalities, additional implementation efforts would be necessary.

Some work has been done on the visualization of data entities, as a contingency of the Embedded Data Commander (EDC) tool, which displays data dependency graphs and analytic information. This is an interesting continuation that should be developed and evaluated against other tools and approaches to evaluate its usefulness in an industrial development project.

An additional venue of research, which we have not yet touched upon, is the relation between our proposed techniques and contemporary techniques for model-based development (MBD). While we don't anticipate any conflicts between MBD and our proposals, it remains to be studied, e.g., how and where our data-modeling with data-entities fits into the workflow of an MBD-process.

EDC allows timing requirements to be modeled. However, to validate this type of requirements it would be useful to relate these requirements to timing requirements onto the execution of producers and consumers of data. It would therefore be interesting to map data-timing requirements to Timing Augmented Description Language (TADL) [50] descriptions to allow automated validation through scheduling-analysis.

# Bibliography


[1] Alexander Pretschner, Manfred Broy, Ingolf H. Kruger, and Thomas Stauner. Software Engineering for Automotive Systems: A Roadmap. *Future of Software Engineering*, pages 55–71, 2007.

[2] Manfred Broy. Challenges in Automotive Software Engineering. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 33–42. ACM, 2006.

[3] Gunter Saake, Marko Rosenmuller, Norbert Siegmund, Christian Kästner, and Thomas Leich. Downsizing Data Management for Embedded Systems. *Egyptian Computer Science Journal*, pages 1–13, 2009.

[4] Mimer SQL Real-Time Edition, Mimer Information Technology, Uppsala, Sweden. http://www.mimer.se, 2012.

[5] eXtremeDB in-Memory Database, McObject. Issaquah, WA USA. http://www.mcobject.com/extremedbfamily.shtml.

[6] AUTOSAR Open Systems Architecture. http://www.car-to-car.org.

[7] Ivica Crnkovic. Component-based Software Engineering - New Challenges in Software Development. In *Software Development. Software Focus*, pages 127–133. John Wiley and Sons, 2001.

[8] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley and Sons, 1996.

[9] D. Box. *Essential COM*, chapter Microsoft Component Object Model (COM.

[10] EJB 3.0 Expert Group. Enterprise JavaBeansTM,Version 3.0 EJB Core Contracts and Requirements Version 3.0. *Final Release*, 2006.

[11] .NET Framework. Microsoft Visual Studio Developer Center. http://www.microsoft.com/NET/.

[12] Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, Mats Lindberg, John Lundbäck, and Kurt-Lennart Lundbäck. The Rubus Component Model for Resource Constrained Real-Time Systems. In *3rd IEEE International Symposium on Industrial Embedded Systems*, June 2008.

[13] Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson, and Massimo Tivoli. The Save Approach to Component-Based Development of Vehicular Systems. *Journal of Systems and Software*, 2006.

[14] Rob van Ommering. *The Koala Component Model for Consumer Electronics Software*, volume 33, chapter 3, pages 78 – 85. IEEE Computer Society, Computer archive, 2000.

[15] Tomas Bures, Jan Carlson, Ivica Crnkovic, Severine Sentilles, and Aneta Vulgarakis. ProCom - the Progress Component Model Reference Manual. Technical Report, Mälardalen University, 2008.

[16] AUTOSAR Open Systems Architecture. http://www.autosar.org.

[17] ArcCore. Open Source AUTOSAR Solutions, Göteborg Sweden. http://www.arccore.com.

[18] The Eclipse Foundation, Ottawa, USA. http://www.eclipse.org/.

[19] DAMA International. *The DAMA Guide to the Data Management Body of Knowledge*. Technics Publications, 2009.

[20] dSPACE Data Dictionary, dSPACE Tools. http://www.dspaceinc.com.

[21] Visu-IT, Automotive Data Dictionary. http://www.visu-it.de/ADD/.

[22] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[23] ORACLE, Database Solutions. http://www.oracle.com.

[24] Access - Database Management System, Microsoft. http://www.http://office.microsoft.com/en-us/access/.

[25] MySQL Database, Oracle. http://www.oracle.com/us/products/mysql.

[26] ISO SQL 2008 standard. Defines the SQL language. http://www.iso.org/iso/home.htm, 2009.

[27] Fred R. McFadden, Mary B. Prescott, and Jeffrey A. Hoffer. *Modern Database Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[28] 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.

[29] H. T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6:213–226, 1981.

[30] Polyhedra In-Memory Database. http://www.enea.com, Sept 2011.

[31] Dag Nyström. *Data Management in Vehicle Control-Systems*. PhD thesis, Mälardalen University, October 2005.

[32] Barbara Gallina and Nicolas Guelfi. SPLACID: An SPL-Oriented, ACTA-Based, Language for Reusing (Varying) ACID Properties. *Software Engineering Workshop, Annual IEEE/NASA Goddard*, 0:115–124, 2008.

[33] Robert K. Abbott and Hector Garcia-Molina. Scheduling Real-Time Transactions: a Performance Evaluation. *ACM Trans. Database Syst.*, 17:513–560, September 1992.

[34] 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.

[35] 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, 2003.

[36] 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, 2004.

[37] T. Gorschek, C. Wohlin, P. Carre, and S. Larsson. A Model for Technology Transfer in Practice. *Software, IEEE*, 23(6):88 –95, nov.-dec. 2006.

[38] Mary Shaw. The Coming-of-Age of Software Architecture Research. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 656–, Washington, DC, USA, 2001. IEEE Computer Society.

[39] Sandro Schulze, Mario Pukall, Gunter Saake, Tobias Hoppe, and Jana Dittmann. On the Need of Data Management in Automotive Systems. In Johann Christoph Freytag, Thomas Ruf, Wolfgang Lehner, and Gottfried Vossen, editors, *BTW*, volume 144 of *LNI*, pages 217–226. GI, 2009.

[40] Nicolas Navet. Trends in Automotive Communication Systems. In *Proceedings of the IEEE*, volume 93, pages 1204–1223, June 2005.

[41] Robert Bosch GmbH. *CAN Specification*. Bosch, Postfach 30 02 40 Stuttgart, version 2.0 edition, 1991.

[42] FlexRay Consortium. http://flexray.com.

[43] Local Interconnect Network. http://www.lin-subbus.org.

[44] Media Oriented Systems Transport (MOST). http://www.mostcooperation.com/home/index.html.

[45] Håkan Gustavsson and Jakob Axelsson. Evaluating Flexibility in Embedded Automotive Product Lines Using Real Options. In *SPLC '08: Proceedings of the 2008 12th International Software Product Line Conference*, pages 235–242, Washington, DC, USA, 2008. IEEE Computer Society.

[46] Alexander Pretschner, Manfred Broy, Ingolf H. Kruger, and Thomas Stauner. Software Engineering for Automotive Systems: A Roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pages 55–71, Washington, DC, USA, 2007. IEEE Computer Society.

[47] J. Rothenberg. The Nature of Modeling. pages 75–92. John Wiley & Sons, Inc., New York, NY, USA, 1989.

[48] K. Czarnecki and S. Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Syst. J.*, 45(3):621–645, July 2006.

[49] The ATESST Project, East-ADL Specification. http://www.atesst.org. March, 2012.

[50] The TIMMO Consortium. TADL: Timing Augmented Description Language, Version 2. *TIMMO (TIMing MOdel), Deliverable 6*, October 2009.

[51] MARTE Specifcation Version 1.0 (formal/2009-11-02). http://omgmarte.org. March 2012.

[52] OMG: UML Profile for SysML. http://www.omgsysml.org. March, 2012.

[53] The Object Management Group. Unifed Modeling Language:Superstructure. Version 2.0, OMG document formal/05-07-04, 200.

[54] Advancing Traffic Efficiency and Safety through Software Technology (ATESST). http://www.atesst.org. March, 2012.

[55] The MathWorks. http://www.mathworks.com.

# II

# Included Papers

# Chapter 6

# Paper A: Design-Time Management of Run-Time Data in Industrial Embedded Real-Time Systems Development

Andreas Hjertström, Dag Nyström, Mikael Nolin and Rikard Land
*13th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*

*Hamburg, Germany, September, 2008*

**Abstract**

Efficient design-time management and documentation of run-time data elements are of paramount importance when developing and maintaining modern real-time systems. In this paper, we present the results of an industrial case-study in which we have studied the state of practice in data management and documentation. Representatives from five companies within various business segments have been interviewed and our results show that various aspects of current data management and documentation are problematic and not yet mature. Results show that companies today have a fairly good management of distributed signals, while internal ECU signals and states are, in many cases, not managed at all. This lack of internal data management results in costly development and maintenance and is often entirely dependent of the know-how of single individual experts. Furthermore, it has, in several cases, resulted in unused and excessive data in the systems due to the fact that whether or not a data is used is unknown.

## 6.1 Introduction

Most of today's embedded system developers are experiencing a vast increase of system complexity. The growing amount of data, the increasing number of electrical control units (ECUs) and inadequate documentation are in many cases becoming severe problems. The cost for development of electronics in for instance high-end vehicles, have increased to more than 23% of the total manufacturing cost. These high-end vehicle systems contain more than 70 ECUs and up to 2500 signals [1, 2].

A lot of research has been done in the area of run-time data management for real-time systems. This has lead to the development of both research-oriented data management solutions, such as [3, 4, 5], and commercial real-time data management tools, such as [6, 7, 8]. However, in most cases this research and these tools focus on run-time algorithms and concepts, but do not manage data documentation. In this case-study we investigate state of practice in design-time data management and documentation of run-time data in industrial real-time systems. An earlier case-study on data management in vehicle control-systems has indicated a lack of data management and documentation internally in the ECUs [9]. The case-study in this paper covers a broader scope of companies, and focuses on the development process and documentation of real-time data.

The study includes five companies, four vehicle companies active in different domains and one company producing electrical control systems. The study identifies ten problem areas in the development process and suggests remedies and directions for further studies. Furthermore, we show that the importance of adequate data management is growing along with the increasing complexity of real-time and embedded systems [10].

The main observation from our study is the rudimentary, or in some cases total lack of, data management and data documentation for internal ECU data. This should be compared to distributed network data that, due to adequate tool support, are fairly well managed and documented. We observed that this lack of management, in some cases leads to inadequate development routines when handling data.

Currently, companies developing safety-critical systems are becoming increasingly bound to new regulations, such as the IEC 61508 [11]. These regulations enforce stronger demands on development and documentation. As an example, for data management it is recommended, even on lower safety levels, not to have stale data or data continuously updated without being used. Companies lacking techniques for adequate data management and proper documentation will be faced with a difficult task to meet these demands.

The main contributions of this paper include:

- A case-study investigating state-of-practice in data management for real-time systems.
- Ten identified problem areas in current practice.
- Suggestions of remedies and future research directions.

The outline of the following parts of the paper is as follows. Section 6.2 describes our research method and the five participating companies. Section 6.3 reports the state-of-practice in data management and documentation of industrial embedded real-time systems. In section 6.4 we present four key observations and ten identified problem areas in current practice. In section 6.5 we propose six remedies and future research directions. In section 6.6 and 6.7 we conclude the paper and suggest future work based on the findings in this case-study.

## 6.2 Research Method

This qualitative case-study [12] has been conducted at five companies, mostly in various vehicular business segments, developing systems in various application areas within the embedded real-time domain. The main source of information has been interviews with open-ended questions [13] conducted at the companies. One person at each company with in depth knowledge of their system development, both on high and low level were interviewed. All interviews have, after promises of anonymity, been recorded to be able to have open discussions that could later be evaluated.

The work-flow of these interviews has been as follows; (i) the interviewee was contacted and asked to take part in this interview with a short explanation of the contents. (ii) A short summary explaining our area of interest was sent one week before the interview. (iii) The interview was executed, and set to last for approximately one hour. (iv) After each interview, the recording from the interview was analyzed and the answers written down as a summary question by question. (v) A document with all of the questions and their respective summaries where sent back to the interviewee for possible commenting and approval. In some cases, the document included additional requests for clarification of certain areas.

The interview questions were divided into five parts, with some general questions in the beginning, more detailed in the middle, and open discussions towards the end.

The interviews consisted of the following five parts:
**Part one** of the interview was a series of personal questions to get background information such as the interviewees position at the company, years employed and area of expertise. This was made to ensure that the interviewee had the desired background and knowledge.
**Part two** was a series of short yes/no questions to get some basic understanding about the business domain, product characteristics, and how they manage the system and information today.
**Part three** was the main part which included more exhaustive questions about how data is managed and documented during the development. This section also included questions regarding how and if documentation is continuously updated when changes or corrections occur after release or during maintenance.
**Part four** covered the development process and the organization.
**Part five** consisted of a more open part with a chance for the interviewee to speak more freely about his/her own experiences and observed problems within the area.

### 6.2.1 Case-Study Validity

All of the studied companies are among the world-leaders within their respective domains which indicate a representative selection. Based on this and the fact that the findings are so conclusive among the companies, we believe that the study provides a representative overview of the common practice and can therefore be considered important. However, the purpose of this study is not to claim, based on this population, that the results are statistically confident or valid for all companies in these business segments.

### 6.2.2 Description of Companies

The studied companies have requested to be anonymous and are therefore described in this paper as COMP1-COMP5.

**COMP1** is a producer of heavy vehicular systems. They have a production volume in the range of 50.000-70.000 units per year. Their system are resource-constrained, distributed and with both critical and non-critical parts. In their development they mainly use software components that are developed

in-house. The information is distributed between ECUs via two redundant CAN [14] networks. The system is built on a software platform that is continually evolving.

**COMP2** produces heavy vehicular systems in the range of 60.000-80.000 units per year and they base their systems on a software platform. Distribution of critical data is performed on three CAN networks with different criticality levels where the communication on the most critical bus is cyclic whereas the other two are event triggered.

**COMP3** is another vehicular company with annual volumes in the range of 450.000-550.000 units. Their system can be considered highly safety-critical and resource-constrained. Furthermore, they use several different types of networks to distribute data. Most of the hardware and software are developed by subcontractors. Data is distributed using network protocols, such as CAN, LIN [15] and MOST [16].

**COMP4** is a manufacturer of public transportation systems producing about 1000 units per year. Network communication is made on fieldbuses. They are now shifting to Ethernet communication with their own protocol layers in their latest platform. There are both periodic data and event triggered data on the bus. They have a small amount of software redundancy but are moving towards hardware redundancy. Almost all development is made in-house. Their systems are based on software platforms that are continuously refined during their 30 year product lifetime. Old products are during their lifetime updated with new software platforms.

**COMP5** develops around 10.000 units of large stationary logic control systems that are less resource-constrained than the other systems in the study. Their systems are based on regular software development and where parts of the system are developed separately as components. Their systems are continuously changing and functionality is added throughout the life-time of the system. Network communication is quite limited and based on Ethernet [17]. They have developed their own standard for development based upon the waterfall model [18]. The ECUs in the system contain both critical and non-critical functionality. The system is built using a centralized configuration database where involved nodes collect information such as system parameters and store them locally before usage.

All five companies selected for this study have been active within research and development of distributed embedded real-time systems for many years. This also applies to the interviewees which all could be considered highly competent and have at least five years of company experience. The companies develop products that mainly incorporate both hard and soft real-time properties.

| Company | Vehicular | Product Variance | Sales Volume | Hard real-time | Soft real-time | Resource-constrained | Component-based | Distributed | Platform-oriented |
|---|---|---|---|---|---|---|---|---|---|
| COMP1 | Y | 1 | 2 | Y | Y | 3 | Y | Y | Y |
| COMP2 | Y | 5 | 3 | Y | Y | 4 | N | Y | Y |
| COMP3 | Y | 4 | 5 | Y | Y | 5 | Y | Y | Y |
| COMP4 | Y | 2 | 1 | Y | Y | 2 | N | Y | Y |
| COMP5 | N | 2 | 1 | Y | Y | 1 | Y | Y | N |

Figure 6.1: Company description.
Range: Low=1 and High=5. Yes=Y and No=N

The investigation concerns how their systems are developed and maintained throughout their life-cycle.

Figure 6.1 shows some of the main similarities and differences between the companies. As seen in the figure, four of the involved companies produce vehicular systems and one company, COMP5 develops stationary industrial systems. The column "Product Variance" indicates if a company has large variances between their products. For example at COMP2, less than two products delivered have the same configuration while almost all of COMP1 products are off-the-shelf. Annual sales volume has a range between 1000 delivered products to several hundreds of thousands. The products of all five companies have both soft and hard real-time properties. Furthermore, all of the companies develop resource-constrained systems but systems developed at COMP1-COMP3 are more resource-constrained than the others. The most resource-constrained product developer is in this case COMP3 with high volumes and limited amount of system resources. Finally, the column "Platform-oriented", indicates if the company develops a company-common software platform as a base used in several manufactured products but with different configurations.

## 6.3 Design-time Data Management

In this section we present some state-of-practice issues on how the interviewed companies perform their documentation and process at design-time followed by a number of use cases and scenarios.

### 6.3.1 State of Practice

In the following part we present how the individual companies perform their documentation and what kind tools and processes are used. The main focus is to provide a better understanding of how data is managed throughout development and maintenance. Since there is a lot of information about each company in this section, we have classified each of the companies with a few keywords for readability and overall understanding.

**COMP1** uses Rubus Visual Studio [19], which is a development environment that is tightly integrated with the Rubus operating system. In this tool they have adequate documentation complying with the J1939 and J1587 standard for bus messages. Except from bus messages they only have sparse documentation on data types in the internals of the ECUs. For most of the documentation they are entirely dependent on the person responsible for a specific part of the system. According to the interviewee this has worked quite well previously when their old software platform was used and the projects where smaller. Now they are introducing a new, more advanced, platform and are experiencing a big increase in data flow and system complexity. Current practice, where a small group or a single person alone is responsible for this information, is not sufficient anymore.

*Company classification:* Dependent on individual developers.

**COMP2** Internally within an ECU, documentation and mechanisms such as special control groups evaluating the work are not so extensive. It is more up to the developer to manage data. The documentation and high-level development of internal behavior is made in Enterprise Architect [20] and follows Rational Unified Process (RUP) [21] as their development process.

For network communication they recently moved from text-based specifications to a database built on Vectors CAN db-admin [22], with their own company specific communication layer. An integration group has control of the network documentation and is responsible for how the signals are used. This enables them to have control of the network and its contents. Also, once a month, a more detailed review that works as a filter for detecting errors is

performed. The documentation regarding the network is continuously updated with new information but old data is never removed. A problem for them has been the growing amount of documentation with several hundred pages of text to describe small parts of the system.

They strictly follow a defined process for adding, removing or searching for data or data properties but have also worked out a "speedy" process if you need fast decisions. They also have a routine to once a year go through the system and check if all data on the bus is used and all code really executes.

*Company classification:* Network controlled by an integration group. Little control on internal ECU data management

**COMP3** uses Rational Rose [23] both for documenting internal signals within the ECU and for external, public network signals. From Rational Rose, function, system and software descriptions are generated. All signals are then semi-automatically put in a signal database and also in spread-sheets. From the spread-sheets, a special appendix is generated with specifications on timing requirements, semantics signals etc. The appendix is open for viewing to all involved developers. They struggle with large amount of text, sometimes several thousand pages, needed for describing models etc. Most of the development, both hardware and software is made by subcontractors.

This company builds their systems on different software platforms. Each platform has a leader that has a lot to say about documentation. Except from deciding what should be added or removed in the system, they also look at the entire business case if a change is doable from a technical and economical point of view. If not, they have the power to abort the introduction of new functionality if deemed necessary. The company's knowledge about signals is documented in a signal database on a global level but it is more up to the responsible person for each software component to have internal control of each ECU. This is, according to them, a known problem. For internal ECU changes, there is a standardized document revision on dedicated meetings. Nothing is released to a subcontractor until it is approved due to legal aspects.

They work according to a so called "superset" thinking in their software platform where they have excessive signals to support different versions of the system. A unique configuration file containing specific information for a specific system is then distributed to all nodes in the system to enable or disable desired functionality.

*Company classification:* Good global knowledge on signals. The platform leader controls functionality. Each software developer is responsible for how data is managed internally on the ECU.

**COMP4** uses spread-sheets for both signals and the fieldbus. During development all staff in a project can search and update these spread-sheets until they do a freeze. A first freeze is done before the actual implementation but is changed if faults are discovered. After a freeze, only a special reference group can perform changes in the freezed version. It is a living process until the product is type approved at the customer. All developers can read and reserve signals during development. A company defined process is used to decide when freezes are supposed to be done.

*Company classification:* Reference group controlled. Uses freezed version and spread-sheets for signals.

**COMP5** uses Serena Dimensions [8], an application life-cycle tool where documentation is done together with the code. They also use high-level drawing tools for component development with a specified system interface and c-code generation. Both code and documentation is versioned in Serena Dimensions. The main idea with their system is that data values can be changed in their central database even when the system is up and running. When a change is made in the configuration database and committed, all involved nodes are notified that there are new data in the database. ECUs that use this data, collect a local copy from the database for internal use. Which kind of data a person is able to change in the central database depends on which authorization level the user is assigned. The majority of the data communication is done internally on the ECU and not on the network.

*Company classification:* Central configuration database. Access rights controlled.

### 6.3.2 Use Cases and Scenarios

This section illustrates some of the important use cases that occur during development and maintenance. What are the main differences in how the involved companies handle adding, removing, and searching for data in their system?

**Adding data to the system** This is done differently in all companies. In COMP1, the responsible technician verifies the system architecture, then decides which node to use and how the data should be transported. This is then discussed with the developer in an effort to find flaws in the solution. After that there are no special routines for how this is done. It is up to the developer. This same action is handled completely differently in COMP2 where a developer

has to write a function specification which is approved by the configuration manager. In the change process, applicable on modeling and signaling COMP3 uses a rudimentary web interface to ask for a change. A team then examines the change and physically synchronizes it to see if the change is technically justified. If it is a major change to the system, the business case is also evaluated. In COMP4, adding data is managed within the project but all signals should be added and approved before implementation. If a change is requested after the documentation is freezed, a reference group has to verify and approve the change. COMP5 uses a similar process. If the new data is approved by an authorized person it can be added and used.

**Removal of data** Even if companies have some routine for adding data to the system, routines on how to remove data is usually non-existent. This raises the question if it could be the case that there are signals in the system that are produced but not consumed.

As in the previous section, in COMP1 it is up to the system responsible. Rubus has no support for checking if a produced signal is used or not. In COMP2, COMP3, and COMP5 they do not remove anything at all. COMP2 does a consistency check against a spread-sheet once a year to see if all code in the system actually runs. If a signal on the bus is not to be used anymore, its CAN ID is defined as occupied and is never used again. This is made in order to minimize future mistakes. COMP3 has no technique to automatically do a mapping and see if data is not used and can be removed without affecting the system. It is considered too time consuming to do this mapping. Instead they keep the old data and calculate with a 15% overhead in the system. In an effort to minimize the need for removal of data, COMP4 does a consistency check in the beginning of each project and only include required signals. They also try to remove unnecessary signals during system updates but normally there are buffers for extra signals in a project. This is because they want to avoid changes in the system that can possibly have unknown consequences. If something is removed in COMP5, it is verified in system tests. However they do not really remove the data, instead they hide it so that it cannot be used in the future.

What seems to be unanimous for all of these companies is that removal of signals is problematic. Since there is no good support for this in the tools or routines, it is again up to the developer in COMP1 to take such a decision. In the other companies they either try to eliminate signals when starting a new project, use overhead in the system or do a consistency check and hide unused signals.

**Searching and usage of data** How can a developer or system architect know if a data is already produced or not? COMP1 has a developer responsible for this knowledge and if a signal is needed by another developer, he/she has to ask that person. They have no documentation regarding the contents of the nodes. The network however is better documented. In the other companies it is possible to search for signals in a spread-sheet, signal database or some type of development tool with more or less detailed information. COMP2 is very strict on signals on the bus and developers have to go through an integration group to require information, if a signal exists and can be used. They have less knowledge about the internals of an ECU, what exists and can be used. However a group of people review the system regularly to avoid errors. Both COMP3 and COMP4 uses a spread-sheet where all developers involved can search for a signal. In COMP3 you have to go through the platform group for usage approval.

COMP4 does not have the same control mechanism for the usage of signals. If a signal is broadcasted on the bus it is open for usage, no additional decisions has to be made when using the signal. There is however only one that can write to any given signal. Except from COMP1, it is possible to search for a signal and use after approval by some kind of control group.

## 6.4 Observations and Problems Areas

In this section we have, based on the above use cases and scenarios, formulated four key observations and ten problem areas.

### 6.4.1 Key Observations

**O1.** *Impact of product variability on documentation.* All of the involved companies in this study have different approaches and a variation of techniques for preserving knowledge about their systems. These companies also produce products that vary more or less. It seems that there is a relationship between the quality of the documentation and the product variability. Figure 6.1 showed how variances differ between different companies. COMP1 manufactures off-the-shelf products. COMP2 and COMP3 both have large product variances to support usage in different environment settings or to suit various vehicular equipment alternatives. COMP4 and COMP5 have small variances. In COMP4 the variances mostly concern HMI settings.

With this information in mind, we can clearly see that this is reflected in their system documentation. COMP1 that produces off-the-shelf products has the least amount of documentation on their system. COMP2 and COMP3 has large variances and both have a more rigorous documentation process. One of the reasons for this could be that large product variances in COMP2-COMP3 are one of the reasons that have forced them to have a more developed preservation of system knowledge.

**O2.** *Inclusion of Excessive Signals.* All of the involved companies have excessive signals in the system as well as functionality that is turned on an off. COMP2 always has excessive signals included in the system to support several vehicle variations. Each system is then configured to suit the individual vehicle configuration. An example of this is to have signals that support both automatic and manual gearboxes. In this way they turn on and off required functionality to suit their needs. The reasons for having excessive signals in their systems vary. In most cases excessive signals are included, either to support product variations or because there is a desire to keep them in the system since a change can have unknown effects to the system.

One reason for having excessive signals and functions in the system is to minimize modifications to the system. If proper tools and documentation techniques were available, it would be possible to build the system more optimized, without unused signals and functionality to save system resources and reduce cost.

**O3.** *Prioritization of selected parts of the system.* As a result of ineffective and inadequate tools for documentation, parts of systems are prioritized. Although COMP3 uses several different techniques to manage and document their system, it is a known problem that they prioritize more critical parts of the system as engine control, compared to the more soft infotainment functionality which is lagging behind.

**O4.** *Awareness that common practice is not enough.* To get a flavor of how companies and interviewees consider their documentation and development process they where asked to rank themselves and how they compare to their competitors at the end of each interview.

When ranking themselves on a scale from one to ten where one is the lowest, a majority of the companies ranked themselves below average. One company ranked itself high with the motivation that as long as they don't have to extend their system with new signals and interfaces, current practice is suffi-

cient. This indicates that it is hard to expand, change or add new functionality to their system, which could be a direct result of poor system documentation. The fact that most companies rank themselves below average regarding their documentation and development process indicates that there is much to be done within this area.

When they ranked themselves compared to their competitors, the ranking follows the same pattern with a below average score. This is interesting since these companies use a variation of documentation, from person dependent to extensive signal databases and processes to handle signals, although mostly for distributed signals.

In order to successfully manage these advanced systems, new techniques for how to handle data has to be introduced. As stated earlier one single person having extensive knowledge about the internals of an ECU is not ideal and could be considered as a possible single point of failure.

The overall statement here is that this is how documentation is believed to be handled within their application area. A question that arises here is why companies that produce highly safety-critical applications in their own opinion have below average control of their system, documentation and process.

### 6.4.2 Identified Problem Areas

There are several important aspects to consider regarding how these companies treat and documents data internally on ECUs or on the communication network. We have from the above use cases and scenarios identified ten problems, divided into three areas:

**Documentation volume and structure**

**P1.** *Growing information volume.* A major problem that was repeatedly raised during the interviews was the growing volume of information [10]. As an example, model descriptions are today made in different tools and sometimes in plain text. This is a major problem since there sometimes can be several thousand pages of text. In most cases everything is backward compatible and nothing is ever removed. This continuously adds to the complexity of the documentation and the amount of text. It is not efficient to supply a system-responsible person with several hundred of pages of information with some small changes. This seems to be a neglected problem that is becoming an overwhelming issue for developers and system architects.

**P2.** *Obsolete documentation.* Documentation is perceived as hard to maintain, requiring a lot of effort and time. As a direct consequence of this, correct and up-to-date documentation is lagging behind. One individual person or a group of persons can be responsible for updating reported changes in documentation. However it is hard to do this in parallel with development and this often introduces a delay until the change is reflected in the documentation.

If a company has documentation, it is versioned and there is also some kind of template specifying the how this should be done. However in all cases, how this is done in practice is highly dependent on the individual person managing the documentation. This has in one company lead to a special template used as a simple speedy possibility to go around their own rules. One way companies do this is to let everybody change according to their needs and freeze a version of the documentation regularly. COMP3 does not have this problem since a developer has to request a change beforehand.

**P3.** *Stale data.* Poor preservation of knowledge and inadequate documentation techniques often lead to stale signals in systems that the companies are or are not aware of. An issue with this is that these stale data items, except from adding to memory, bandwidth and CPU usage, may cause failures or unwanted system behavior. Unknown effects such as these are addressed in new, more stringent regulations such as IEC61508.

**P4.** *Inadequate ECU data documentation.* One thing correspond for all of the involved companies. There is a difference in how they treat data and signals on the network compared to internal data on ECUs. The network is documented using various tools and techniques whereas internal ECU data in most cases are not. The lack of efficient tools and techniques have made individual developers responsible for much of the knowledge about data items and functionality inside an ECU.

**P5.** *Dependency on individual developers.* Internal knowledge of an ECU is in several of the involved companies left to a single individual or a group of developers. This is an important issue since companies could lose valuable information due to poor, or non-existent, documentation. As an example, an individual developer in COMP1 can have all information about a certain part of the system or functionality. When other developers need a signal or information regarding that system or function, they have to ask the developer for it. When asked how this would influence the company if a staff member would leave the company, they say that it would not be a disaster but it would mean a lot of effort for someone else to get up to date.

The systems that COMP1 are developing have so far been quite small since large parts of the product have been mechanically controlled. The current trend is to introduce more and more computer-controlled parts, thus rapidly increasing the system complexity. The small size and amount of data in the system made it possible for persons to keep track of most things. This worked up until now.

New platforms are being released with more computer controlled systems that are too complex for a single developer to handle. The new systems are redundant, safety-critical, contain more diagnostics, more signals, human-machine interface (HMI), and other functionalities.

**Tool support**

**P6.** *Lack of efficient tool support.* More efficient documentation, tools and processes are needed and could in the end reduce development costs. Companies themselves indicate that within a few years they will need to use a small set of tools or one single flexible tool to limit the amount of text describing models today.

Since systems and functions require a lot of effort and are costly to develop, companies reuse as much of the system as possible. This puts high demands on documentation in order for developers to be able to understand how a function will work if it is reused in another setting with other dependencies. This is especially true if it is a safety-critical function which often is rigorously verified and tested.

**P7.** *Lack of visualization.* As systems are getting more heterogeneous and more complex, in the sense of more signals, increasing number of ECUs and more distributed data items, developers have raised the question of a need for a graphical view of the whole development chain to aid developers and system architects.

Important aspects to visualize are;

- how functions are connected
- how data is shared between functions
- how ECUs are connected
- where the nodes are physically placed

**Routines**

**P8.** *Poor support for adding data.* Routines for adding data to the system differ a lot between the companies. A problem here is that there is a lot of manual work done by individual developers, or just open discussions to verify how additional data affects the system and there is no effective tool support for this matter.

**P9.** *Difficult to search for data.* As long as a data item is distributed on the network, it is in most cases possible to search for a data item. However the possibility to search for an internal ECU data item is in most cases limited.

**P10.** *No support for removal of data.* Despite the fact that some of these systems are resource-constrained and available resources are sparse, a lot of unnecessary data items remain unused in the system. In an effort to reduce the number of unused data items, some of the companies try to remove old data when starting a new project but they are careful about doing so because they lack knowledge about system dependencies such as, who are producing and who are consuming this data. Instead they either try to hide data, leave it as it is, or mark them as occupied so there will be no new users. It seems that the overall problem here is a lack of feedback from the development tools. There is no way to automatically see dependencies for internal data.

## 6.5 Remedies and Vision for Future Directions

In this section we elaborate, based on the problems (P1-P10), observations (O1-O4) from the study, and future standards and regulations, on possible improvements in data management tools and processes for embedded real-time system's development.

To improve data management we propose to lift data to a higher level during development. A more data centric development is needed, where data is considered early in the development phase and seen as its own entity. To substantially elevate existing data management and documentation towards a more data centric development, we propose six remedies;

**R1.** *A unified development environment.* To successfully be able to manage the problems stated in P1, P2, P5 and P6, scattered information needs to be gathered in one development environment. As seen in the study, some companies successfully use a signal database for bus messages. By extending this to also

include internal signal and state data, an integrated data management environment supporting the entire development chain including requirements, modeling, design, implementation, and testing is achieved. This data management environment could aid developers by filtering out only the relevant documentation for each development activity. Correctly implemented this environment should provide an easy interface for developers from different sub-systems can share to update and manage documentation.

**R2.** *Global data warehousing and data-flow graphs.* Data warehousing is an effective technique, providing means to store, analyze and retrieve data. By introducing global data warehousing, and data-flow graphs to the development environment a company-common documentation base that development projects of different sub-system can access and share is provided. It also gives developers the possibility to identify and visualize data providers and subscribers and thereby aiding designers when adding, managing and removing data. This gives developers the means to solve problems identified as P3, P4, P8-P10.

**R3.** *Automated tools and techniques.* To additionally aid developers solving P2, P4-P5 and maximize the impact of a unified development environment, automated tools and techniques must be introduced to link design-time documentation against run-time mechanisms.

**R4.** *Physical visualization.* By introducing physical visualization, showing the physical layout and data streams of the system, identified as P7, we solve a problem that was explicitly pointed out by some interviewees in the study.

**R5.** *Meta-data information.* To aid in solving P2 and P6, a natural coupling between system requirements and data properties meta-data information such as resolution, real-time properties, priorities, criticality, etc. needs to be included into the development environment.

**R6.** *Integrated data modeling tool.* During our previous case-study [9] it became obvious that using internal data structures for internal data storage lead to difficulties to keep track of data and to perform memory optimization. A integrated data modeling tool can provide developers with means to organize and structure all system data, thereby aiding in solving P5. Within the database community several data modeling techniques, such as entity-relationship modeling, [24] exist.

| Problems | R1 | R2 | R3 | R4 | R5 | R6 |
|---|---|---|---|---|---|---|
| P1 | X | | | | | |
| P2 | X | X | X | | X | |
| P3 | | X | | | | |
| P4 | | X | X | | | |
| P5 | X | | X | | | X |
| P6 | X | | | | X | |
| P7 | | | | X | | |
| P8 | | X | | | | |
| P9 | | X | | | | |
| P10 | | X | | | | |

Figure 6.2: Problem areas with associated remedy or remedies.

Introducing these remedies and forming a uniform development environment give developers the prerequisite needed for effectively managing their system development and maintenance. Figure 6.2 illustrates how the problems are linked with the proposed remedies.

## 6.6 Conclusions

In this paper, we show that due to the increasing system complexity, current state of practice in data management is not adequate. There are many important issues observed in this case-study. From these, we have identified ten problem areas and formulated four key observations, based on current practice and future needs. These problem areas and observations set the path for future research and improvement.

It is confirmed by all involved companies that new processes and techniques for achieving a satisfactory documentation on a software system are needed to be able to handle the needs of today and tomorrow. This is something that could be required to meet the upcoming safety regulations, eg. as specified by IEC 61508, and will be a complex and difficult transition for these companies.

The study shows that there is much to be done within the area, especially documentation of data internally on ECUs. Inefficient, or lack of, routines for adding, removing or searching for data or data properties has in some cases made companies completely dependent on individual experts instead of thorough documentation. As the systems grow, this approach is no longer feasible.

Another more unwanted effect of inadequate data management is that there are also data included which no one knows exists. These stale signals is an important safety issue since they could have unknown consequences to the system. An important fact is that these systems are in many cases resource-constrained and stale data waste resources. This could be a major cost factor for mass producing companies with high demands of cost-efficiency.

Currently, adequate tools to manage distributed data exist, resulting in a much better data management for distributed data compared to internal ECU data. In this paper, suggestions for improved tool-support for internal data, as well as overall system data management is presented. It is our belief that a novel tool that incorporate adequate data documentation, management and design views, both for design and run-time would significantly improve current data management practices.

## 6.7 Future Work

From this case-study we could also see an emerging need for more flexible and efficient run-time data management. Several interviewees indicated that there is an increasing need to manage both hard and soft real-time requirements within their systems. There are also indications that a more secure handling of data is needed since there is a desire to connect to the system at run-time for maintenance, upgrades and infotainment purposes. This issue is seems especially important when using telematics to access these safety critical systems. Another issue is the coming standards and regulation which will put higher demand on data management. These are some of the important issues still to investigate based on the outcome from this case-study.

# Bibliography


[1] A. Albert. Comparison of Event-Triggered and Time-Triggered Concepts with Regard to Distributed Control Systems. *Proc. Embedded World*, pages 235–252, 2004.

[2] Leen Gabriel and Heffernan Donal. Expanding Automotive Electronic Systems. *Computer*, 35(1):88–93, Jan 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, 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*. Springer, 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] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano - a Revolution in On-Board Communications. Technical report, Volvo Technology Report, 1998.

[7] Mimer SQL Real-Time Edition, Mimer Information Technology, Uppsala, Sweden. http://www.mimer.se, 2012.

[8] Serena Dimensions. http://www.serena.com/products/.

[9] 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.

[10] Kaj Hänninen, Jukka Mäki-Turja, and Mikael Nolin. Present and Future Requirements in Developing Industrial Embedded Real-Time Systems - Interviews with Designers in the Vehicle Domain. In *13th Annual IEEE Int. Conf, Engineering of Computer Based Systems (ECBS)*, Germany, 2006.

[11] International Electrotechnical Commission IEC. Standard: IEC61508, Functional Safety of Electrical/Electronic Programmable Safety Related Systems. Technical report.

[12] Robert K.Yin. *Case Study Research Design and Methods*. Sage Publications, Inc, third edition edition, 2003.

[13] Carolyn B. Seaman. Qualitative Methods in Empirical Studies of Software Engineering. *Software Engineering*, 25(4):557–572, 1999.

[14] Robert Bosch GmbH. *CAN Specification*. Bosch, Postfach 30 02 40 Stuttgart, version 2.0 edition, 1991.

[15] Local Interconnect Network. http://www.lin-subbus.org.

[16] Media Oriented Systems Transport (MOST). http://www.mostcooperation.com/home/index.html.

[17] Dave Dolezilek. IEC 61850: What You Need to Know About Functionality and Practical Implementation. *Power Systems Conference: Advanced Metering, Protection, Control, Communication, and Distributed Resources*, March 2006.

[18] W.W. Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In *ICSE: Proceedings of the 9th international conference on Software Engineering*, pages 328–338. IEEE Computer Society Press, 1987.

[19] Arcticus Systems. http://www.arcticus-systems.com.

[20] Sparx Systems Ltd. http://www.sparxsystems.eu/.

[21] Ahmad Shuja Jochen Krebs. *IBM Rational Unified Process Reference and Certification Guide : Solutions Designer (RUP).* IBM Press, December 2007.

[22] Vector Informatics, CANdb Admin. http://www.vector-worldwide.com.

[23] IBM Rational Software. New York, USA. http://www-306.ibm.com/software/rational.

[24] Peter Pin-Shan Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Trans. Database Syst.*, 1(1), 1976.

# Chapter 7

# Paper B: A Data-Entity Approach for Component-Based Real-Time Embedded Systems Development

Andreas Hjertström, Dag Nyström and Mikael Sjödin
*14th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*

*Palma de Mallorca, Spain, September, 2009*

**Abstract**

In this paper the *data-entity approach* for efficient design-time management of run-time data in component-based real-time embedded systems is presented. The approach formalizes the concept of a *data entity* which enable design-time modeling, management, documentation and analysis of run-time data items. Previous studies on data management for embedded real-time systems show that current data management techniques are not adequate, and therefore impose unnecessary costs and quality problems during system development. It is our conclusion that data management needs to be incorporated as an integral part of the development of the entire system architecture. Therefore, we propose an approach where run-time data is acknowledged as first class objects during development with proper documentation and where properties such as usage, validity and dependency can be modeled. In this way we can increase the knowledge and understanding of the system. The approach also allows analysis of data dependencies, type matching, and redundancy early in the development phase as well as in existing systems.

## 7.1 Introduction

We present the *data-entity approach* for efficient design-time management of run-time data in embedded real-time systems. We propose methods, techniques and tools that allow modeling of data into a design entity in the overall software architecture. This enables developers to keep track of system data, retrieve accurate documentation and perform early analysis on data items. The goal is to achieve higher software quality, lower development costs, and to provide higher degree of control over the software evolution process. We show how our approach can be coupled to a development environment for component-based software engineering (CBSE), thus bridging the gap between CBSE and traditional data management. For example, it bridges the encapsulation paradigm of CBSE and the blackboard paradigm of traditional data-management techniques.

Our approach primarily targets data intensive and complex embedded real-time systems with a large degree of control functions, such as vehicular, industrial and robotic control-systems. These domains, and also software intensive embedded systems in general, has in recent years become increasingly complex; up to the point that system development, evolution and maintenance is becoming hard to handle, with corresponding decreases in quality and increases of costs [1].

For instance, the cost for development of electronics in for instance high-end vehicles, have increased to more than 40% of the total development cost and systems contain more than 70 electronic control-units (ECUs) and up to 2500 signals [2, 3, 4].

In an effort to handle the increasing complexity of embedded real-time systems, various tools and techniques, such as component-based software engineering [5, 6], real-time data management [7, 8], and network bus management [9], has previously been introduced. While these techniques and tools have the common aim to reduce software complexity, they target different areas of system complexity. CBSE, for example, targets encapsulation of functionality into software components that are reusable entities. Components can be mounted together as building blocks, with a possibility to maintain and improve systems by replacing individual components [5]. On the other hand, real-time data management, e.g. database technologies, target data produced and consumed by functions by providing uniform storage and data access, concurrency-control, temporal consistency, and overload and transaction management. Network management in turn, aim to handle the increasing amount of data that is distributed throughout ECUs in the system and, e.g. manage the temporal behavior of distributed data.

However, despite their common aim of reducing complexity, these techniques in some cases have contradicting means of achieving their goals. For example, the requirement of information hiding and component data interfacing in component-based systems might conflict with the common blackboard data storage architecture using real-time databases. To overcome these contradictions, it is our belief that data management must be made to be an integral part of the design environment as an architectural view. It is also becoming additionally important to consider data freshness in embedded real-time systems as have been done within the real-time database community [10].

The main contributions of this paper include:

- We introduce the concept of a *data entity* to encapsulate all metadata, such as documentation, type, dependencies, and real-time properties concerning a run-time data item in a system. The data-entity approach provides designers with an additional architectural view which allows for data searching, dependency visualization, and documentation extraction.
- The data-entity approach provide techniques which are tightly coupled with component-based software engineering.
- Our approach allows properties of data to be analyzed any time during the development process. A model of data entities can be constructed before development commences, thus giving the possibility to provide early feedback to designers about consistency and type compatibility. Alternatively, a model can be extracted from existing designs, allowing analysis of redundancies and providing a base for system evolution.
- The data-entity architectural view complements other architectural views, such as component-based architectural views, without violating paradigms such as information-hiding, encapsulation and reuse.
- Finally, we have realized this approach by implementing it into a tool-suite, using the existing component model ProCom [11] that also offers a development environment. The tool includes data entity editors as well as a number of analysis tools.

The rest of this paper is structured as follows; in section 7.2, we present background and motivation for the approach. We also present four specific problems that our approach addresses. In section 7.3, a definition of the data entity is presented and the data entity approach is discussed in section 7.4. Further, in section 7.5, we describe the ProCom component-model which is

used in our data entity tool-suite, presented in section 8.6.3. An example of how the data entity tool-suite can be used is presented in section 7.7. Finally, the paper is concluded in section 7.8.

## 7.2 Background and Motivation

The aim of our approach is to bridge the current gap between component-based software engineering and data management by extending the architectural views with a data-centric view that allow run-time data to be modeled, viewed and analyzed. Current system design techniques emphasize the design of components and functions, while often neglecting modeling of flow and dependencies of run-time data. A recent study of data management at a number of companies producing industrial and vehicular embedded real-time systems clearly showed that this gap is becoming increasingly important to bridge, and that current design-techniques are not adequate [1].

The study showed that documentation and structured management of internal ECU data is currently almost non-existent, and most often dependent on single individual persons. Traditionally, the complexity of an ECU has been low enough so that it has been possible for a single expert to have a fairly good knowledge of the entire architecture. However recently, companies are experiencing that even internal ECU data complexity is growing too large for a single person to manage. This has led to a need for a structured data management with adequate tool support for system data. A similar development took place within the vehicular domain in the late 1990s, when the industry took a technological leap with the introduction of bus-management tools, such as the Volcano tool [9]. By that time, the distributed vehicular systems had grown so complex that it was no longer feasible to manage bus packet allocation and network data-flow without proper tool support. It is pointed out in the study, that there is a clear need for a similar technological leap for overall system data management.

### 7.2.1 Problem Formulation

The case study [1] identifies a number of problems related to poor data management in practice today. In this paper, four of these problems are specifically addressed.

**Addressed problems:**

- **P1** ECU signals and states are, in many cases, not managed and documented at all, companies often are entirely dependent of the know-how of single individual experts
- **P2** The lack of structured management and documentation has in several cases led to poor routines for adding, deleting and managing data. Often a "hands-off" approach is used where currently functioning subsystems are left untouched when adding additional functionality, since reuse of existing data is considered too risky due to lack of knowledge of their current usage.
- **P3** Some companies calculate with up to 15% overhead for unused and stale data being produced. It is considered too difficult to establish if and how these stale data are being consumed elsewhere in the system.
- **P4** A lack of adequate tool support to model, visualize and analyze system data.

To further complicate matters, companies developing safety-critical systems are becoming increasingly bound to new regulations, such as the IEC 61508 [12]. These regulations enforce stronger demands on development and documentation. As an example, for data management it is recommended, even on lower safety levels, not to have stale data or data continuously updated without being used. Companies lacking techniques for adequate data management and proper documentation will be faced with a difficult task to meet these demands.

### 7.2.2 Related Work

Several tools within code analysis and code visualization have been developed to be able to explore data-flow and how functions are connected [13, 14]. These are however mainly built to interpret existing code and not focusing on high level data management during development. The increase in complexity and the amount of signals used within ECU development has also been addressed within the data modeling area. A number of data dictionary tools such as dSpace Data Dictionary, SimuQuest UniPhi and Visu-IT Automotive Data Dictionary [15, 16, 17], have been developed in an effort to get an overall view of the systems signals as well as structured labeling and project management. These tools are tightly coupled with MATLAB/Simulink and MATLAB/Targetlink [18] which is line with current state-of-practice. However

none of these tools specifically target CBSE. dSpace Data Dictionary does however additionally provide techniques for managing AUTOSAR [6] property specifications. Furthermore, non of these tools target high-level data management where data can be modeled, analyzed and visualized in an early phase of development.

To confront the intricacy of embedded system development of today and tomorrow, CBSE will play a more central part. However, supporting tools, specifically targeting component-based systems, needs to be developed to support the technological leap needed within data management.

Common for both CBSE and the data entity approach is that they aim to assemble and design systems at a higher level by encapsulating information and functionality into components or entities. The aim with our approach is to add to the functionality of the above stated tools. In our approach, data is seen as a component/entity in the development strategy. This allows data to be modeled separately with a possibility to perform early analysis such as relative validity. It also allows system architects and developers to graphically view data dependencies, similar as components can viewed during system development. This information can then be connected to the data flow in the component model and used as input to the system architect when developing the system.

## 7.3 The Data Entity

In this section we will first introduce the concept of *data entity*. Secondly we present how data entities can be used and analyzed, both in an early phase of development and for already developed systems.

### 7.3.1 Data Entity Definition

The concept of a *data entity* that encapsulates all metadata is the basis of our approach. A data entity is a compilation of knowledge for each data item in the system. A data entity can be defined completely separate from the development of components and functions. This enable developers to set up a system with data entities based on application requirements and perform early analysis even before the producers or consumers of the data are developed. The information collected by data entities are also valuable for developers and system architects when redesigning or maintaining systems. Another important feature is that since a data entity is completely separated from its producers and the consumers, it persists in the system regardless of any component, function or design changes.

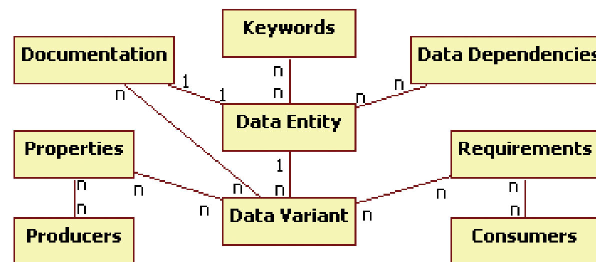



Figure 7.1: Data Entity Description Model

A data entity consists of the following metadata (illustrated in Figure 7.1):

- **Data Entity**, is a top level container describing the overall information of a set of data variants. Data entities are created to aid developers with problem P1 and P2, by elevating the importance level of data during development and maintenance. Required information is associated with the data entity and its data variants, enabling it to persist on its own during all the phases of a system life-cycle. As an example, a data entity could be *vehicleSpeed*. The data entity also includes a top level description to facilitate for a developer in need of high level information.

- **Data Variant**, is the entity that developers will come in closer contact with and consist of properties, requirements, dependencies and documentation. A data variant can be of any type, size or have any resolution. To continue our example from above where the top level data entity is *vehicleSpeed*, we can add a number of variants. For example *vehicleSpeedSensorValue, vehicleSpeedInt, vehicleSpeedDouble, vehicleSpeedMph* and *vehicleSpeedKmh*. Each of these variants with their own properties, requirements, dependencies and documentation. A data variant could for instance be specified with consumer requirements, but without any existing producer properties. The requirements then can later be used as input when searching for an existing producer or when creating a new producer.

- **Data Producers**, the set of components producing the given data variant.

- **Data Consumers**, the set of components consuming the given data variant.

- **Data Variant Properties**, is either an existing producer's property or a set of properties that is based on the consumer requirements. If there is no existing producer, these properties can be used as requirements by the system architect. Examples of properties are: name, type, size, initial value, minimal value, maximal value and frequency.
- **Data Variant Requirements**, are directly related to the requirements of a consumer. These requirements can be matched against producer properties or be the source for the producer properties. Example requirements are: frequency, accuracy and timing consistency parameters.
- **Data Variant Dependencies**, enables a possibility to see which data entities that is dependent on each other regarding for instance temporal consistency and precedence relations.
- **Data Variant Documentation**, gives the developer an opportunity to describe and document the specifics of each data variant.
- **Keywords**, Data entities and data variants can be tagged with keywords to facilitate a better overview and give developers additional benefits where a data entity or a data variant with related information can be searched for using keywords. Since companies can have their own unique naming ontologies, keywords can be adapted to suite a specific need. As an example, if a developer is interested a data entity regarding the vehicle speed with a certain type and resolution. He/she can then search using the keyword, for instance "speed", to receive all speed related signals. From there, find the appropriate data entity and its different data variants.

### 7.3.2 Data Entity Analysis

Using the information contained in the data entities, data-entity analysis is possible during the entire development process, even in the cases where producers or consumers are yet undefined. The approach open up for a number of possible analysis methods such as:

- **Data Flow Analysis.** This analysis show producers and consumers of a specific data entity variant. It is able to detect unproduced as well as unconsumed data, and is thereby directly addressing problem P2, P3 and P4. The output of this analysis can the be forwarded to system architecture tools to expose which components that would be affected by a change to a data entity.

- **Data Dependency Analysis.** Data dependency analysis can facilitate for developers and aid with problem P3, by providing information about which producers and consumers that based on their properties and requirements are dependent on each other regarding temporal behavior and precedence relations.
- **Type Check Analysis.** Data types from the producer properties and the requirements of the consumer is analyzed to make sure that there is a match.
- **Resolution/Domain Analysis.** Matches the data resolution and possible data domains to the connected producers and consumers.
- **Absolute Validity Analysis.** Absolute validity is a measurement of data freshness [19]. An absolute validity interval can be specified for a data entity variant, which specifies the maximum age a data can have before being considered stale. The importance of knowing the end-to-end path delay i.e. data freshness, in an execution chain, especially within the automotive systems domain, have been identified in previous work, such as [20]. Properties from producers are analyzed to see if the requirements of the consumers are achieved.
- **Relative Validity Analysis.** Relative validity is a measurement of how closely two interacting data entity variants have been produced [21]. Even though both data might be absolute consistent, they might be relative inconsistent, which indicate that any derived data would be considered inconsistent. Additional research on methods, tools and techniques for how to find the relative data dependency between several execution chains and their end-to-end deadline is needed in order to guarantee the relative data freshness demanded by a consuming component. To achieve this, we propose an extension of [20], with a formal framework for relative dependency. Similar to absolute validity, properties from producers are analyzed to see if the requirements of the consumers are achieved.

## 7.4 The Data Entity Approach

The data entity approach provides designers with an additional architectural view, the data architectural view. This view allows data to be modeled and analyzed from a data management perspective during development and maintenance of component-based embedded systems.

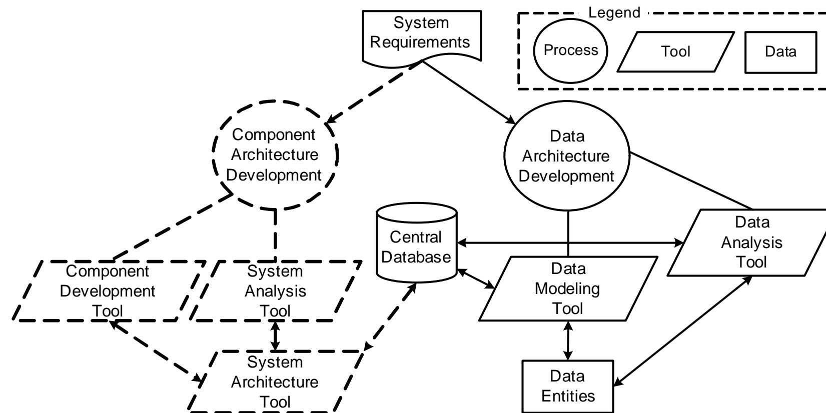



Figure 7.2: The data entity approach

Figure 7.2 show our proposed data entity approach (right-hand side). The figure illustrates how our approach complements the traditional component-based design approach (left-hand side). The *central database* in the middle of the figure acts as the communicating link between the two approaches as well as the main storage for information.

In the *data modeling tool*, data entities can be created, retrieved and modified. Furthermore, they can be associated with design entities such as *message channels* created from the ProCom component architecture development [11]. The *data analysis tool* extracts data and data producer properties based on the requirements placed upon the data from the data consumers. These properties could then be propagated to a system architecture tool as component requirements on the components producing the data. It can also be used as input to system synthesis and scheduling tools. Furthermore, the data analysis tool could provide graphical visualization of all data dependencies, both with respect to data producers and consumers for a certain data, but also visualize dependencies between different data, such as relative consistency and precedence relations.

System design using the data entity approach can start from data architecture design, from system architecture design, or from a combination of both. If for example, in the early stages of an iterative design process, a set of components that provide a given function is designed, it is often the case that the

input signals to this function is not yet defined, and therefore left unconnected in a functional design. If these unconnected data signals are modeled using a data entity, the data analysis tool can be used to derive required properties of this data, which can later be sent as input to a system architect tool as component requirements of the component that is later connected as producer of this data. On the other hand, consider that a commercial, off-the-shelf (COTS), component that provides certain functionality is integrated in a system architecture tool, and that component produces a set of signals of which a subset is currently not needed, these data can still be modeled, and made searchable for future needs. In this case, the data analysis tool can be used to derive the properties of this data.

Also in management and extension of existing systems, the data modeling tool can be used to search for existing data that might be used as producers for the new functionality. The requirements for the new functionality can then be matched towards the existing properties and requirements of the other consumers of the data, to determine whether or not the data can be used for this functionality. This solves the "hands off" problem presented in problem P2.

## 7.5 The ProCom Component Model

The ProCom component model aims at addressing key concerns in the development of control-intensive distributed embedded systems. ProCom provides a two-layer component model, and distinguishes a component model used for modeling independent distributed components with complex functionality (called ProSys) and a component model used for modeling smaller parts of control functionality (called ProSave). In this paper we only focus on the more large scale ProSys. The complete specification of ProCom is available in [11].

In ProSys, a system is modeled as a collection of concurrent, communicating subsystems. Distribution is modeled explicitly; meaning that the physical location of each subsystem is not visible in the model. Composite subsystems can be built out of other subsystems, ProSys is an hierarchical component model. This hierarchy ends with the so-called primitive subsystems, which are either subsystems coming from the ProSave layer or non-decomposable units of implementation (such as COTS or legacy subsystems) with wrappers to enable compositions with other subsystems. From a CBSE perspective, subsystems are the components of the ProSys layer, i.e., they are design or implementation units that can be developed independently, stored in a repository and reused in multiple applications.

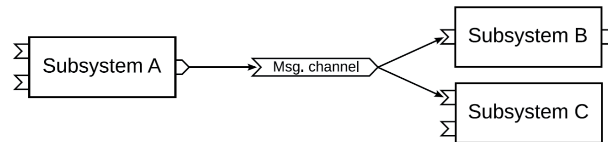



Figure 7.3: ProSys Component Model

For data-management purposes, the communication between subsystems is the most interesting issue. The communication is based on asynchronous message passing, allowing location transparency in communication. A subsystem is specified by typed input and output message ports, expressing what type of messages the subsystem receives and sends. The specification also includes attributes and models related to functionality, reliability, timing and resource usage, to be used in analysis and verification throughout the development process. The list of models and attributes used is not fixed and can be extended.

Message ports are connected though message channels. A message channel is an explicit design entity representing a piece of information that is of interest to one or more subsystems. Figure 7.3 shows an example with three subsystems connected via one message channel. The message channels make it possible to express that a particular piece of shared data will be required in the system, before any producer or receiver of this data has been defined. Also, information about shared data such as precision, format, etc. can be associated with the message channel instead of with the message port where it is produced or consumed. That way, this information can remain in the design even if, for example, the producer is replaced by another subsystem.

## 7.6 Embedded Data Commander Tool-Suite

The *embedded data commander* (EDC) is a tool-suite that implements the data entity approach for the ProSys component-model. The tool-suite, which so far is implemented in the Eclipse framework [22] as a stand alone application that provides a tight integration between Data Entities and ProSys message channels.
The tool-suite, consists of four main parts:

- The Data Collection Center (DCC), which is the common database that holds all information regarding data entities, channels, requirements and subsystems.

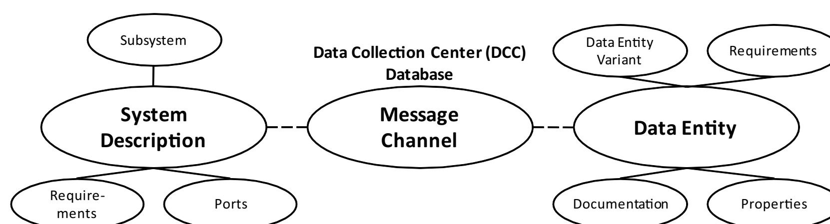



Figure 7.4: DCC data model description

- The Data Entity Navigator (DEN), which is the main application and modeling tool where data entities are administrated.
- The Data Analysis Tool (DAT), which performs data analysis on data variants and subsystems.
- The Channel Connection Tool (CCT), which is the interface tool towards the ProCom tool-suite.

**The Data Collection Center, DCC** is the central database that all EDC tools communicates through. A commercial relational SQL database is used to implement the DCC [7], allowing multiple tools to concurrently access the DCC enabling use of the tool-suite in large development projects.

The DCC consists of three main storage objects (Figure 7.4), the data entity-, the message channel- and the system description-object.

**The Data Entity Navigator, DEN** is the main application of EDC. In DEN developers can create, retrieve or modify data entities. It is also in DEN developers can manage data entity properties, requirements, dependencies, description and documentation.

An important feature in DEN is that developers can view to which other channels and component a data variant is connected, thereby providing valuable information regarding dependencies and an opportunity to navigate between data entities and related subsystems to access information.

The information available in the DEN can also be filtered and divided into sections to facilitate for developers to find the appropriate information. It would also be possible to extend the tool to produce custom-tailored reports containing only the necessary information for its specific purpose.

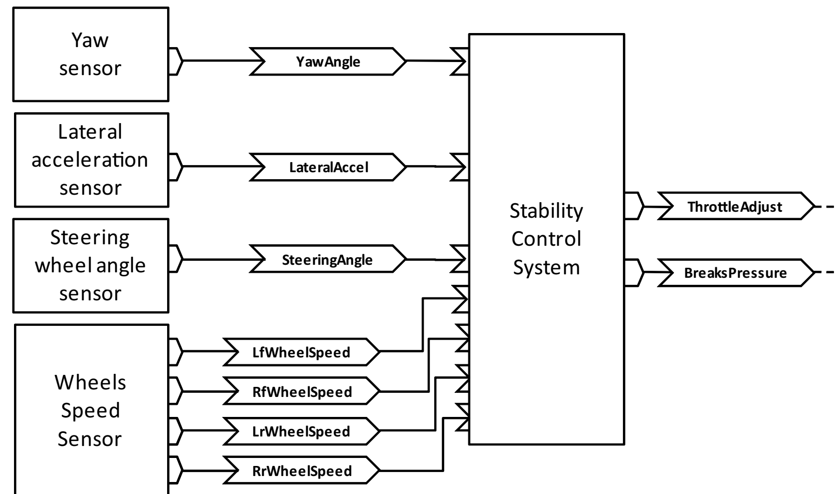



Figure 7.5: Existing ProSys Stability Control System application

**The Data Analysis Tool, DAT**, handles all analysis regarding data entities variants. The current version of the tool support analysis on data flow, type check, resolution and domain analysis but will be extended to support absolute-, and relative-validity analysis.

**The Channel Connection Tool, CCT**, is the connection point between the data entity and the ProCom tool-suite. Since the ProCom tools has been separately developed, a tool to extract architectural information and message channel information was needed.

## 7.7 Use Case

To illustrate our ideas, this section will describe two simple scenarios. The first is, expanding an existing system and the second, verification of the consistency between data-producers and consumer in connection with system validation. This example is illustrated using the concept of data entities and the EDC tool together with ProCom.

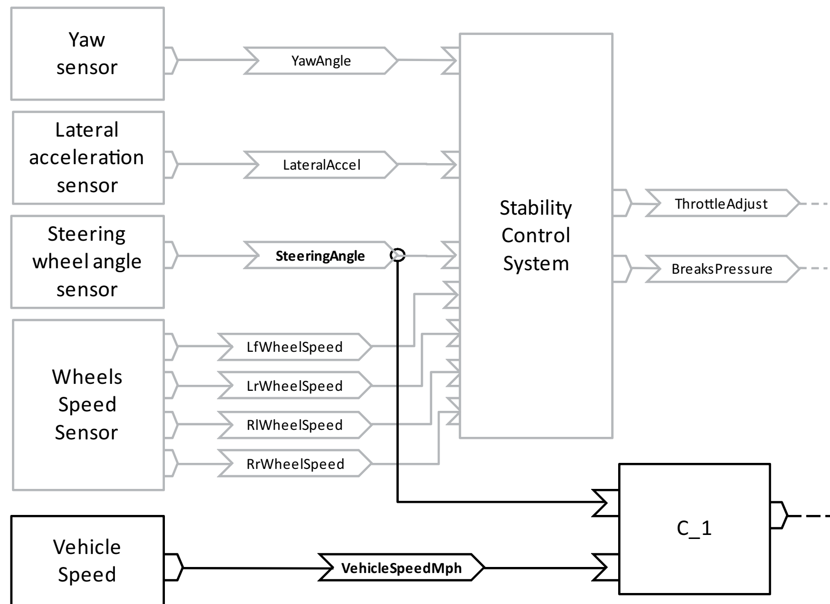



Figure 7.6: Extended ProSys example

### 7.7.1 Expanding an Existing System

This example starts with an existing vehicle application which has already been developed. A part of this system [23] is illustrated in Figure 7.5. We now face a situation where we should add additional functionality. The new functionality demand an additional component, called C_1, to be added that requires two signals as input. To make it easier for the developer when adding these signals and additional functionality the EDC tool can be used to facilitate reuse of existing signals (if suitable signals exists) to avoid redundancy and also to gain knowledge about possible dependencies between data entities.

The new component C_1 is added to the application with a number of requirements. For simplicity we only consider those that are interesting for this example. The signals required are *vehicle steering wheel angle* and *vehicle speed*, with the following requirements:

| Data variant | Type | Size | Unit | P(ms) |
|---|---|---|---|---|
| SteeringAngle | Int | 16 | Arcsec | 20 |
| VehicleSpeedMph | Int | 16 | MPH | 10 |

Figure 7.7: Producer properties.

| SteeringAngle: | | VehicleSpeedMph: | |
|---|---|---|---|
| Type: | Integer | Type: | Integer |
| Size: | 16 bit | Size: | 16 bit |
| Unit: | Arcsec | Unit: | MPH |
| Absolute validity interval: | 20 ms | Absolute validity interval: | 20 ms |

To be able to locate an existing data entity, a search in the existing application can be performed using relevant keywords. A keyword search for "steering" generated a possible candidate variant *SteeringAngle*, that is already used in the system and can be seen in the center of Figure 7.6. If the properties of the proposed data variant satisfy the requirements, it can be used, and no additional producer have to be added or implemented. A appropriate data variant using the keyword "speed" results in several possible variants but none that matches the requirements of "vehicle speed". A new data variant *VehicleSpeedMph* in the lower center of Figure 7.6, is created and associated to a message channel, with properties such as type, size and unit according to the requirements of C_1. These properties will then be the requirements of the producer component.

### 7.7.2 Validation

When the system modifications are completed, a validation of the whole system should be performed. However in this example we only focus on the newly introduced component C_1. A series of analysis can be performed to validate that the requirements of C_1 is fulfilled.

In this example we will focus on three types of analysis, type, size and absolute validity analysis. The producer properties is stated in Figure 7.7.

- **Type check analysis** is performed by comparing the properties assigned to *SteeringAngle* and *VehicleSpeedMph* and to make sure that they correspond to the requirements of C_1. In this case requirements to receive an integer is fulfilled.

- **Size analysis** is a similar analysis as type check where properties of *SteeringAngle* and *VehicleSpeedMph* are compared with the requirements of C_1. Requirements are fulfilled.

- **Absolute validity** is achieved if both *SteeringAngle* and *VehicleSpeed-Mph* is updated within 20 ms. Requirements are fulfilled.

This example illustrates developers can use the data entity approach when adding functionality to an existing application and how to locate and use existing signals in the system. It also shows how a new data variant can be created and defined according to requirements and how data entity analysis can be used to validate the system or to how to use requirements as input to a system architect tool and scheduler. In this example we perform the analysis on one level in the system. A next step could be to be to support timing and dependency analysis through several steps in the chain, from sensor trough a chain on consumers and producers. The DEA tool is still in an early stage of development and additional research is needed to be able to deal with these more complex issues.

## 7.8 Conclusions

We have presented our new *data entity approach* towards development of real-time embedded systems. The data entity approach gives system designers a new architectural view, the data architecture, which complements traditional architectural views for e.g. component inter-connections and deployment. Using the data architecture view, run-time data entities becomes first level citizens of the architectural design, and data can be modeled and analyzed for consistency irrespectively any other implementation concerns, e.g. even before implementation begins.

The motivation for our approach stems from observations industrial practices and needs [1]. Related to the four key problems that we stated in section 7.2.1 the approach provides:

**P1** A uniform way to document external signals and internal state data in ECUs.

**P2** A unified view of data in a whole system and their interdependencies. Thus, providing the basis for safe modifications, updates and removal of data entities.

**P3** Tracking of data dependencies and dependencies to producers and consumers of data. Thereby enabling removal of stale and obsolete data without jeopardizing system integrity, allowing system resource to be re-claimed when data entities are no longer needed.

**P4** The foundation to build tools for automated analysis and visualization of data in a system.

We have implemented support for our approach in a tool suite called Embedded Data Commander (EDC). EDC provides tools for data modeling, visualization and analysis.

EDC also provides integration with the ProCom component-model and allows automated mapping between data entities and ProCom's *message channels*. While our data entity approach is independent of any target platforms, the integration with an implementation environment (ProCom in this case) gives significant benefits since the transformation from the data-model to the implementation model can be automated. Our implementation also supports the possibility to generate a data-model from an existing component assembly; hence allowing developers to re-gain control of their data in an existing legacy system. To better understand how the data entity approach and the EDC tool-suite could be used, a use case example is also presented.

In the future we will extend the analysis capabilities of the EDC to include end-to-end and relative validity by extending [20], introduce graphical data modeling, implement EDC as an integrated part of ProCom development environment and evaluate the tool-suite in real software development projects. We also plan to release the EDC as open source, to enable other researchers to provide integrations to other implementation environments. Specifically, it would be interesting to study how the data entity approach would be mapped to the AUTOSAR [6] component technology.

# Bibliography


[1] Andreas Hjertström, Dag Nyström, Mikael Nolin, and Rikard Land. Design-Time Management of Run-Time Data in Industrial Embedded Real-Time Systems Development. In *Proceedings of 13th IEEE International Conference on Emerging Technologies and Factory Automation, Germany*, September 2008.

[2] A. Albert. Comparison of Event-Triggered and Time-Triggered Concepts with Regard to Distributed Control Systems. *Proc. Embedded World*, pages 235–252, 2004.

[3] Leen Gabriel and Heffernan Donal. Expanding Automotive Electronic Systems. *Computer*, 35(1):88–93, Jan 2002.

[4] Klaus Grimm. Software Technology in an Automotive Company - Major Challenges. *Software Engineering, International Conference on Software Engineering*, page 498, 2003.

[5] Ivica Crnkovic. Component-based Software Engineering - New Challenges in Software Development. In *Software Development. Software Focus*, pages 127–133. John Wiley and Sons, 2001.

[6] Harald Heinecke, Klaus-Peter Schnelle, and Helmut Fennel et al. AUTomotive Open System ARchitecture - An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E-Architectures. Technical report, 2004.

[7] Mimer SQL Real-Time Edition, Mimer Information Technology, Uppsala, Sweden. http://www.mimer.se, 2012.

[8] 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.

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

[10] Yuan Wei, Sang H. Son, and John A. Stankovic. Maintaining Data Freshness in Distributed Real-Time Databases. In *ECRTS '04: Proceedings of the 16th Euromicro Conference on Real-Time Systems*, 2004.

[11] Tomas Bures, Jan Carlson, Ivica Crnkovic, Severine Sentilles, and Aneta Vulgarakis. ProCom - the Progress Component Model Reference Manual. Technical Report, Mälardalen University, 2008.

[12] International Electrotechnical Commission IEC. Standard: IEC61508, Functional Safety of Electrical/Electronic Programmable Safety Related Systems. Technical report.

[13] Johan Andersson, Joel Huselius, Christer Norström, and Anders Wall. Extracting Simulation Models from Complex Embedded Real-Time Systems. In *Proceedings of the 2006 International Conference on Software Engineering Advances, ICSEA'06*. IEEE, October 2006.

[14] Understand, Analysis Tool by Scientific Toolworks. http://www.scitools.com/products/understand/.

[15] dSPACE Data Dictionary, dSPACE Tools. http://www.dspaceinc.com.

[16] SimuQuest. http://www.simuquest.com/.

[17] Visu-IT, Automotive Data Dictionary. http://www.visu-it.de/ADD/.

[18] The MathWorks. http://www.mathworks.com.

[19] Xiaohui Song. *Data Temporal Consistency in Hard Real-Time Systems*. PhD thesis, Champaign, IL, USA, 1992.

[20] Nico Feiertag and Kai Richter et.al. A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems Under Different Path Semantics. In *EEE Real-Time System Symposium (RTSS), (CRTS'08) : Barcelona, Spain*. IEEE, 2008.

[21] P. Raja, L. Ruiz, and J.D. Decotignie. Modeling and Scheduling Real-Time Control Systems with Relative Consistency Constraints. *Real-Time Systems, 1994. Proceedings., Sixth Euromicro Workshop on*, pages 46–52, Jun 1994.

[22] The Eclipse Foundation, Ottawa, USA. http://www.eclipse.org/.

[23] Severine Sentilles, Aneta Vulgarakis, Tomas Bures, Jan Carlson, and Ivica Crnkovic. A Component Model for Control-Intensive Distributed Embedded Systems. In *Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE2008)*. Springer Berlin, October 2008.

# Chapter 8

# Paper C: Data Management for Component-Based Embedded Real-Time Systems: the Database Proxy Approach

Andreas Hjertström, Dag Nyström and Mikael Sjödin
*Journal of Systems and Software, vol 85, nr 4, p821-834, Elsevier, April, 2012*

**Abstract**

We introduce the concept of database proxies intended to mitigate the gap between two disjoint productivity-enhancing techniques: Component Based Software Engineering (CBSE) and Real-Time Database Management Systems (RTDBMS). The two techniques promote opposing design goals and their co-existence is neither obvious nor intuitive. CBSE promotes encapsulation and decoupling of component internals from the component environment, whilst an RTDBMS provide mechanisms for efficient and predictable global data sharing. A component with direct access to an RTDBMS is dependent on that specific RTDBMS and may not be useable in an alternative environment. For components to remain encapsulated and reusable, database proxies decouple components from an underlying database residing in the component framework, while providing temporally predictable access to data maintained in a database. Our approach provide access to features such as extensive data modeling tools, predictable access to hard real-time data, dynamic access to soft real-time data using standardized queries and controlled data sharing; thus allowing developers to employ the full potential of both CBSE and an RTDBMS. Our approach primarily targets embedded systems with a subset of functionality with real-time requirements. The implementation results show that the benefits of using proxies do not come at the expense of significant run-time overheads or less accurate timing predictions.

## 8.1 Introduction

This paper propose database proxies [1] as a solution to integrate a Real-Time DataBase Management System (RTDBMS) [2, 3] into a Component-Based Software Engineering (CBSE) [4, 5] setting. Database proxies are automatically generated glue code synthesized from the system architecture that translates data between components ports and an RTDBMS residing in the component framework.

Data management of embedded real-time systems is becoming increasingly important as systems evolve from simple stand-alone devices into becoming complex systems, often interconnected with its surrounding environment. This trend has lead to that developers are confronted with a substantial amount of functions, design-time and run-time data that needs to be managed. In addition, developers are increasingly faced with new requirements such as secure and dynamic data sharing and advanced diagnostics. To reduce the resulting complexity, model driven development [6] and CBSE, are increasingly used in industry today. However, these techniques mainly focus on the functional aspects of the software, and rarely target management of data.

The introduction of database proxies enable a clear separation of system functionalities and data management, thereby letting developers focus more on the functional behavior of the system rather than developing in-house specialized solutions for managing data. Predictable access to hard real-time data, dynamic run-time data access, secure data sharing and data modeling tools are just some of the benefits that the usage of database proxies in conjunction with an RTDBMS can provide. Both CBSE and RTDBMS, aims to reduce complexity and enhance productivity when developing these systems. CBSE promotes encapsulation of functionality into reusable software entities that communicate through well defined interfaces and that can be assembled as building blocks. This enables a more efficient and structured development where, for instance, available components can be reused or COTS (Commercial Of The Shelf) components effectively can be integrated in the system to save cost and increase quality.

An RTDBMS provides a blackboard storage architecture to share global data predictably and efficiently by providing concurrency-control, temporal consistency, overload management and transaction management. The usage of an RTDBMS allows real-time systems to be built around a data layer, supporting safe sharing of data between applications, both proprietary as well as third party software. Access to data is made through standardized query languages, providing advanced access control mechanisms. This implies that potentially

unsafe software, such as third party software, can be granted access to data in a controlled manner. Furthermore, RTDBMSs significantly cuts time to market by providing high-level query languages, supporting logging, diagnostics, monitoring, and efficient data modeling [7].

However, the coexistence between the techniques is non-trivial since their design goals are contradicting.

The techniques offered by an RTDBMS allow the internal representation and management of data to be decoupled from the data usage. However, RTDBMSs promotes the use of shared data with potentially hidden dependencies amongst data-users.

CBSE, on the other hand, strives to decouple components from the context in which they are deployed. One aspect of this is that a component should not have should not have hidden dependencies on the existence of certain data-elements. This decoupling is achieved by encapsulating component-functionality and making visible only a component-interface describing a component's provided and required services.

Using an RTDBMS in existing component-based systems would require RTDBMS specific code to be used from within a component. This introduces negative side effects that violate several basic principles of CBSE, for instance:

1. A component with direct access to the database from within, violates the component's aim to be encapsulated and only communicate through its interface.

2. Direct access to shared data introduces hidden dependencies between components.

3. If an RTDBMS is called from inside the component, the component is dependent on that specific RTDBMS and cannot be used in an alternative setting.

In order to succeed with the integration of an RTDBMS into a component framework, we present the concept of database proxies.

As illustrated in Figure 8.1, a database proxy is part of the synthesized architecture, thus external to the component. The purpose of the database proxy is to enable for components to interact with an RTDBMS using their normal interfaces. This is possible since the coupling, i.e. the database proxy, between the component and the RTDBMS is embedded in the component framework.

The database proxies are used to bind and integrate components to form the final running system. By decoupling components from the database, and

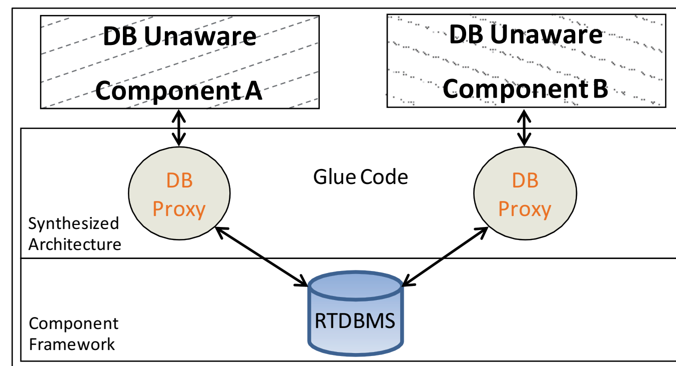



Figure 8.1: Database Proxies Connecting Components to an RTDBMS

placing the database in the component framework, the decision to use a database or some other data management strategy is removed from the component level and becomes a system design decision.

**Database proxy characteristics**

1. Database proxies are automatically generated as glue code in the synthesized architecture, leaving the component code unchanged.
2. Components can gain access to an RTDBMS in the component framework with maintained encapsulation and decoupling.
3. Components with soft real-time requirements can access multiple data items using dynamic run-time queries without blocking hard real-time data accesses.
4. Components can be reused regardless of the existence of a database in the component framework.

In our previous work, database proxies was limited to only support native data types e.g. integer, char, float etc., from one port to another [1]. The work has now been augmented so that components can have efficient and predictable access to complex data structures to/from multiple ports of the same component in hard real-time. In addition, data transfers between components can be extracted to the database, for logging purposes or to share data, without interfering with the regular component communication.

The remainder of this paper is structured as follows; in section 8.2, we present motivation for the approach. Section 8.3 present the specific problems that our approach targets, related work and state of practice. In section 9.3, we present the system model. Section 8.5 gives a detailed description of the database proxy and its constituent parts. Further, in section 8.6, we illustrate our ideas with an implementation example. Finally, we show a performance and real-time predictability evaluation in section 8.7 and conclude the paper in section 8.8.

## 8.2 Motivation

The characteristics of today's embedded systems are changing. According to Fürst [8] and Grimm [9], 90% of all innovations within the automotive industry stems from software and electronics. In a high end vehicle there can be more than 800 functions, 70 ECUs, and thousands of signals needs to be managed [10]. This has led to increasingly complex and costly to development of embedded systems.

When developing modern large scale IT systems, the use of standardized platforms as a base for service-oriented architectures, error recovery etc. is widely used. They provide features such as several abstraction layers, virtualization techniques and scalability. However, in resource constrained embedded systems with limited memory size, limited computing capacity and demand for low energy consumption, this approach is not sufficient since abstraction layers and virtualization techniques add to the amount of resources needed. [11].

Within the embedded community, modern techniques such as model driven development and component-based software engineering are widely used to reduce complexity and increase the understanding and reusability of software functions by elevating the abstraction level [6, 12, 13]. However, these techniques do not include methods and tool support for efficient and management of data.

Many of today's systems are developed by different subcontractors in form of whole applications or just individual functions, sometimes each with their own in-house developed solution for how to manage data. In addition, it has been shown that documentation and structured management of internal ECU data is sometimes almost non-existent and dependent on individual developers own solutions [14].

The increasing need for more structured, flexible, reliable and secure data management techniques to coordinate data both at run-time and at design-time is continuously pointed out has been pointed out as major challenges for the future [7, 15, 16]. As stated by Pretschner et al. [17] and Broy [18], a stan-

dardized and overall data model and management system has great potential as a solution to deal with the distributed and uncoordinated data in these complex systems. Furthermore, Schulze et al. [7] and Saake et al. [19] points out that the ad-hoc and/or reinvented management of data for each ECU with individual solutions using internal data structures, can lead to concurrency and inconsistencies problems. In addition, maintainability, extensibility and flexibility of the system decreases.

Furthermore, sophisticated techniques for diagnostics, error detection, logging and secure data sharing are much needed to improve reliability and system quality. Due to the ineffective diagnostics and error tracing techniques, less than 50% of the replaced ECUs were, in fact, defect [17]. Much of the diagnostics messages and logging that can be retrieved from these systems are statically predefined at design time. An example of this is in the AUTOSAR standard [12]. In techniques a such as the Program Monitoring and Measuring System (PMMS), it is up to the user to specify pre-conditions and insert code in order to collect data [20]. This put high demands on developers to predict future needs of, for instance, service technicians. In difference, the flexible and dynamic behavior of an RTDBMS can provide any single data element or a set of data elements with a single query, providing that the user is granted access.

Secure data sharing is becoming increasingly important when systems are opening up to the surrounding environment using techniques such as CAR2-CAR communication [21] and/or connecting to PDAs, smart phones, GPS, etc. The diversity of these devices have led to in-house proprietary solutions to enable a connection to the infotainment system [22]. A proposed standardized solution to this could be to use a data management system, such as an RT-DBMS [7, 19]. An RTDBMS provides both access control to data as well as dynamic data access using a well known standard query language (SQL). In addition, in order to achieve a separation of data management and application logic, a general data management infrastructure is needed [23].

The usage of an RTDBMS when designing and building real-time embedded systems could not only aid developers with standardized tool support for modeling system data at design-time [24], but also provide predictable and efficient routines for managing data at run-time. This could, as an example shorten time-to-market, since developers can manage complex data structures with a single database query instead of using complex programming routines.

It is thereby well established that CBSE and RTDBMS are two important technologies for future development of embedded real-time systems. An integration of these two technologies is not trivial and requires new methods that can bridge the gap between their contradictive design goals.

## 8.3 Background

In CBSE, a component encapsulates functionality and only reveals an interface of provided and required services. A component which communicates with a database outside its revealed interface, i.e., directly from within the component-code, introduces a number of unwanted side effects such as hidden dependencies and limited reusability. We define such a component to be **database aware**.

To utilize the benefits of CBSE, a component must be fully decoupled from the database. From a components perspective, it should not matter if the consumed or produced data originates in data structures or in a database. We define a component to be **database unaware** if it has no notion of an underlying data storage. Furthermore, a database unaware component does not introduce any side effects such as database communication outside the component's specified interface, thus retaining the reusability of the component.

The usage of an RTDBMS in a CBSE framework should not introduce any side effects that violate CBSE principles [5, 25].

For the purpose of this paper, we define a component to be side effect free, with respect to the introduction of an RTDBMS, if it is:

- **Reusable:** A component can still be used in another setting, with or without an RTDBMS.
- **Substitutable:** A component should be substitutable by a component implementing the same interface; regardless if a RTDBMS is used or not.
- **Without implicit dependencies:** A component should not introduce implicit dependencies such as database access from within a component.
- **Using only interface communication:** A component may only communicate through its interface. Our approach does not consider management of the internal state in a component.

### 8.3.1 Solution Requirements

This section identifies a number of requirements, R1-R3, which needs to be fulfilled in order to enable the introduction of an RTDBMS into a CBSE-setting.

**R1** The decision to use an RTDBMS should be made on system level in order to be integrated in existing development models and systems.

**R2** The usage of an RTDBMS should not introduce any side effects to the components.

**R3** The real-time predictability of the system should not be compromised by using a RTDBMS.

### 8.3.2 Related Work and State of Practise

The research which explicitly aims at combining CBSE and an RTDBMS is novel. Figure 8.2 illustrates that there is a gap between CBSE and RTDBMS techniques.

Within the CBSE community there are specialized in-house and proprietary techniques such as Koala [26] or global automotive initiatives such as AUTOSAR [12]. However, these techniques do not prioritize data management. As an example, AUTOSAR provides a uniform way for managing data e.g., save and load data from non volatile memory. However there is no uniform technique to manage run-time data in RAM.

Lau et al. [27] presented a research direction within CBSE on how to manage data by having data flow and data access completely encapsulated within connectors. In this way, components only encapsulate computation. Another approach by Lau et al. [28] is to encapsulate data inside components to achieve encapsulated reusable building blocks, where data is included.

Efforts within the database community, illustrated in the rightmost part in Figure 8.2, aim at developing solutions to downsize and optimize RTDBMSs to suite embedded systems. There are solutions for componentizing and/or customizing the database management system to only include features that is actually used in a particular resource constrained system [19]. In addition, there are also techniques, such as database pointers, that can manage both soft and hard real-time transactions predictably [29]. These techniques are available in both research-based and commercial RTDBMSs [30, 31, 32].

Both the CBSE and the RTDBMS community have solutions suitable for their respective areas. However, in between CBSE & RTDBMSs, there is a gap with respect to data management, illustrated in Figure 8.2. The lack of research within this area has left this gap as an open problem. The aim of this paper is to bridge this gap.

There are mechanisms within the RTDBMS community that aims to simplify the database access by hiding some of the underlying complexity as well as making the access to the RTDBMS more efficient.

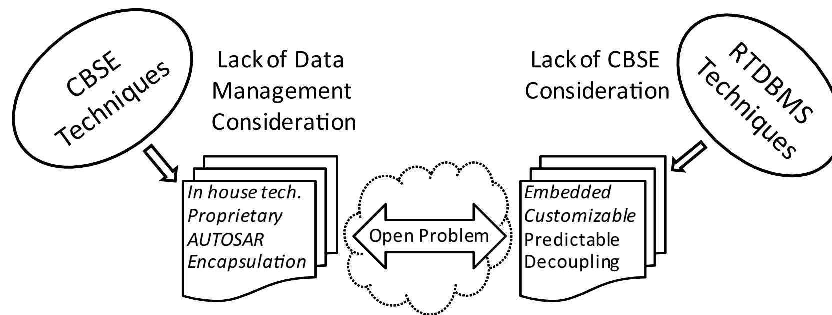



Figure 8.2: Combining CBSE and DBMS is an open problem.

The standardized interface-language SQL defines the following mechanisms [33]:

- **Pre-compiled statements**, enable a developer to bind a certain database query to a statement at design-time. The statement is compiled once during the setup phase, instead of compiling the statement for each use during run-time. This has a decoupling effect since the internal database schema is hidden. Each statement is bound to a specific name that is used to access the data.
- **Views**, are virtual tables that represent the result of stored queries. A database view has a similar decoupling effect as pre-compiled statements since schema changes can be masked to users by enabling a user to receive information from several tables perceived as a single table.
- **Stored procedures**, enable developers to decouple logical functions from the application and move them into the database. A stored procedure is a program used when several SQL statements need to be executed within the database in order to achieve the result. This is achieved with a single call to the procedure.
- **Functions**, are programs within the database, similar to a stored procedure. A function performs a desired task and must return a single value.

These mechanisms provide partial decoupling of a component from the DBMS. However none of them are completely sufficient to use in a component-based setting, since:

1. The database is still accessed from within the component code, not through the component's defined interface. (Violation of R1-R2.)

2. The component is still only partially decoupled from the database since the database name, login details and connection code still need to reside in the component. A component using these mechanisms is therefore no longer generic or reusable. (Violation of R1-R2.)

3. The requirements expressed by the components interface does not reflect the components internal database dependency. (Violation of R2.)

4. These mechanisms are not intended for real-time performance (typically only non-real time DBMS support is available), e.g., the usage of these mechanisms alone would be a violation of R3.

## 8.4 System Model

The tools and techniques in this paper primarily target data intensive, and complex, component-based embedded real-time systems with a large degree of control functions, such as vehicular, industrial and robotic control-systems. These applications involve both hard real-time functionality that include safety-critical control-functions, as well as soft real-time functionality.

Our techniques are equally applicable to distributed and centralized systems (however current implementations as described in latter sections, are for single node systems).

To clarify some terms that will be used throughout this paper, we define;

1. A `native data type` to be a basic data type such as an integer, char or float.

2. A `complex data type` to be a C-struct or an array that consists of a number of native data types.

3. A `fixed-length data` as a data that have a fixed size. An example of this is a single struct containing only native data types.

4. A `variable-length data` as a data that can vary in size. An example of this is an array of structs.

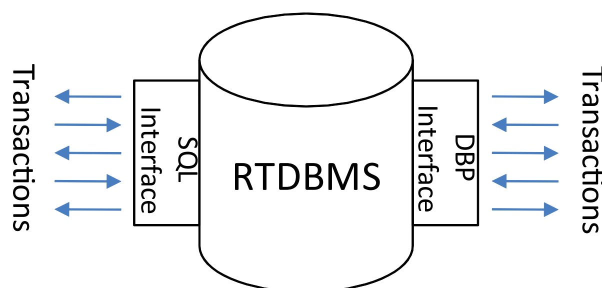



Figure 8.3: RTDBMS Architectural Overview

We consider a system where functionality is divided into the following classes of tasks:

**Hard real-time tasks**, that control critical functionality, uses hard real-time transactions [34] to read and write values from sensors/actuators and execute real-time control loops. Hard real-time tasks communicate with fairly simple data structures such as native data types and more complex but fixed-length data structures such as a C-struct. Variable-length data are not supported since hard real-time tasks require predictable access to data elements.

**Soft real-time tasks**, that control less timing sensitive functionality. Soft real-time tasks uses soft real-time transactions [34] to read and write variable-length complex data structures typically to present statistical information, logging, or used as a gateway for service access to the system by technicians in order to perform system updates. Soft real-time tasks could also be used for fault management and perform ad-hoc queries at run-time.

In order to support a predictable mix of both hard and soft real-time transactions, we consider an RTDBMS with two separate interfaces where hard real-time predictability is not compromised by soft transactions. Note that we allow both hard, and soft tasks to access any data element, thus we do not separate between hard and soft data elements.

Figure 8.3 illustrates an RTDBMS which has a soft interface that utilizes a regular SQL query interface to enable flexible access from soft real-time tasks. For hard real-time transactions, a database pointer [35] interface is used to enable the application to access individual data elements or a set of data elements in the database with hard real-time performance. Two RTDBMSs that provide these types of interfaces are COMET [31] and Mimer SQL Real-Time Edition [30].

### 8.4.1 Database Pointers

Database pointers [29] are pointer variables that are used for real-time access to data in a real-time database, see Figure 9.1. The figure 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 data element, oil temperature, in the engine relation. The task consists of two parts, an initialization part (lines 2 to 4) executed when the system is starting up, and a periodic part (lines 5 to 8) scanning the sensor in real-time. During the initialization part (lines 2 to 4) the database pointer is created and bound to a data element in the database.

```
1 TASK oilTemp(void){
     //Initialization part
2   int temp;
3   DBPointer *dbp;
4   bind(&dbp,"Select TEMP from ENGINE
                 where SUBSYSTEM='oil'");
     //Control part
5    while(1){
6      temp=readOilTempSensor();
7      write(dbp,temp);
8      waitForNextPeriod();
     }
  }
```

Figure 8.4: An I/O Task That Uses a Database Pointer

During the control part of the task in Figure 9.1, the `write` function writes the new value `temp` to the database pointer. During this operation, only a few lines of non-blocking code (with a bounded number of instructions) that performs type checking, synchronization with other accesses with the same data element, and writing of the data are executed.

Database pointers can be bound to either individual single data elements, or to sets of data.

Depending on the organization of the data in a set, different types of database pointers are used [30]:

**Single database pointer:** A single database pointer can only be bound to an individual data element, e.g., an integer, string or a float. This type of pointer is useful for storing sensor and actuator values. The data element is the atomic unit, i.e., a single database pointer provides atomic reads

and writes of a single value. This implies that a single database pointer does not have any transactional properties such that atomic commits of multiple single value database pointers.

**Multicolumn database pointer:** A multicolumn database pointer is bound to a set of attributes (columns) of a single database record (row in a table). When a multicolumn database pointer is read or written, all data in the set are read or written atomically. This provides a simple transactional behavior, in the sense that a snapshot of data can be kept consistent.

**Multirow database pointer:** A multirow database pointer is bound to a certain attribute but spans a set of database records in a table. When a multirow database pointer is bound, the pointer is set to point at the first element in the set. Writes to the database pointer are performed on a row by row basis, and after each write, the pointer is set to point to the next element in the set. When all elements in the set have been written to, the pointer is reset to point to the first element again.

**Multicolumn-Multirow database pointer:** A multicolumn-multirow database pointer combines the functionality of a multicolumn database pointer and a multirow database pointer thus being able to bind a matrix of data elements. These pointers are especially suited for event logging where for example log event information, event data and a timestamp can be logged in a database for future analysis.

The cost and predictability of database pointer execution is, as shown in the performance evaluation in section 8.7, comparable to the performance of a shared variable that is protected by a semaphore.

Since database pointers can co-exist with relational (SQL) query management, data can be shared between hard and soft real-time tasks. However, in order to maintain real-time predictability in a concurrent system, some form of concurrency-control is needed. The 2-version database pointer concurrency algorithm (2V-DBP) [35] uses a 2-version versioning algorithm that guarantees that database pointers will never be aborted or subjected to unpredictable blocking.

2V-DBP allow soft real-time transactions to concurrently access the data without experiencing blocking or aborts due to operations through database pointers.

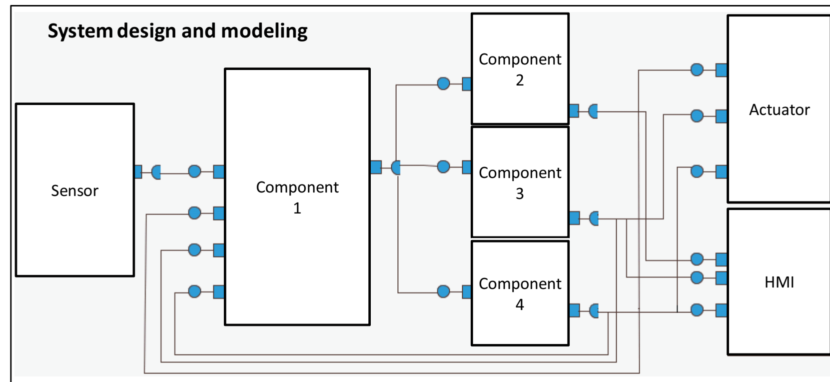



Figure 8.5: System Design and Modeling

### 8.4.2 System Architecture and Modeling

In the application design and modeling we employ a pipe-and-filter [4] component model where data is passed between components (filters) using connections (pipes). The entry point for the connection to the components is the interface (port). Figure 8.5 shows an example of a component-based system design and modeling architecture.

The communication between components in the system is made by connecting output-ports, where a component provides data, to input-ports where components receives data. An output-port can be connected to one or many input-ports.

## 8.5 Database Proxies

A database proxy consists of pieces of code that translates data from a components port to a database call and further on to an RTDBMS residing in the component framework and vice versa. These pieces of code are neither a part of the component nor a part of the RTDBMS, instead database proxies are automatically generated glue code synthesized from the system architecture e.g., the structure and behavior of the system, see Figure 8.1.

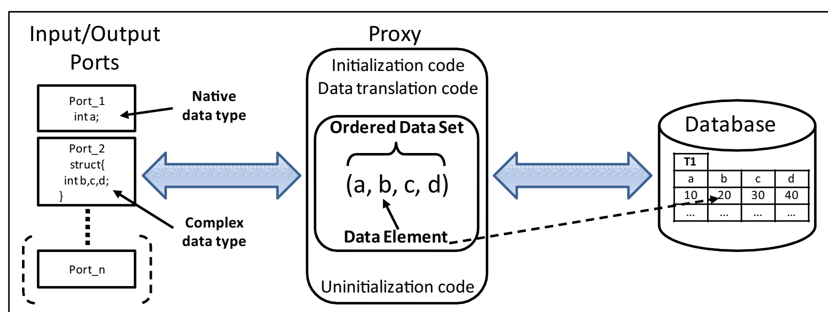



Figure 8.6: Database Proxy Overview

A database proxy contains the following parts:

- **Input/output ports** that connects to one or many ports of a component or a pair of connected components.
- **Initialization code** that connects to the RTDBMS and initiate database accesses.
- **Data translation code** that performs database accesses (database reads and write) and translates the result to a data set that match the component ports.
- **Uninitialization code** that closes database accesses and disconnects from the RTDBMS.

Figure 8.6 gives an overview of the different proxy parts which will be presented more in detail in the remainder of this section.

A database proxy achieves decoupling between the components and the RTDBMS by enabling components to remain encapsulated and reusable. From a component perspective, communication to the RTDBMS is transparently performed through the regular in- and out ports in the component interface.

From an RTDBMS perspective, decoupling is achieved by encapsulating the underlying database schema from the components, only allowing data access to database proxies through pre-compiled statements, views, stored procedures or database pointers.

As a result, database proxies target requirements **R1-R3** presented in Section 8.3.1, since database proxies are:

- Automatically generated from the system architecture. The decision to use an RTDBMS has been moved from component level to system level. (Targets R1)
- Implemented as glue code, leaving the component code unchanged, and all communication is still performed through the components interface. No side-effects are introduced. (Targets R2)
- Uses database pointers, that provides hard real-time guarantees. (Targets R3)

To support the different requirements of hard and soft real-time tasks (see Section 8.4), we distinguish between two proxy types, **hard real-time database proxies** (hard proxies) that is used by hard real-time tasks and **soft real-time database proxies** (soft proxies) that is used by soft real-time tasks.

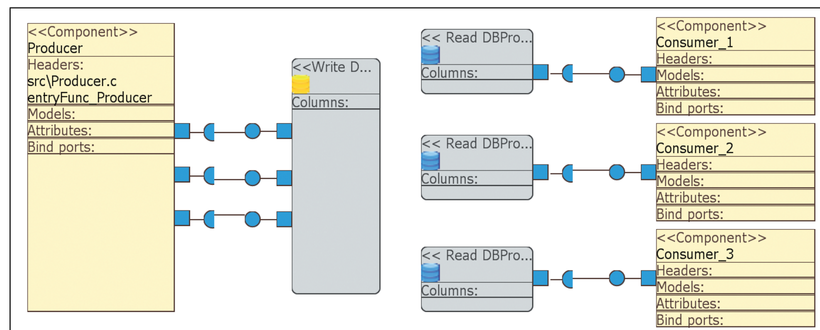



Figure 8.7: Single-Port and Multi-Port Proxies.

Database proxies can have one of the following configurations:

1. A **read proxy**, is used to retrieve data from the database and output the data to a component. This is illustrated by proxies connected to components `Consumer_1-3` to the right in Figure 8.7 which each is provided with data from a read proxy.

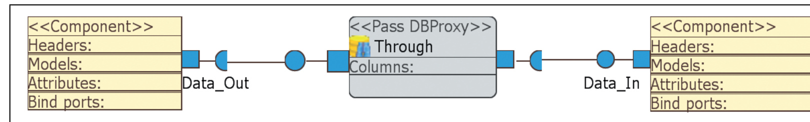



Figure 8.8: Proxy Through

2. A **write proxy**, is used either to update or to insert data that is provided by a component to the database. An update is used to refresh data that is already in the database. An insert is used to add a new data e.g., a new row, in a dynamic database table. An example of a write proxy connected to component `Producer` is illustrated to the left in Figure 8.7.

3. A **proxy through**, is connected as a communication link between two components. The proxy through is used to listen in on a regular component connection and propagate data to the database as a write proxy. The result is normal communication between the components. However, a copy of the value is stored in the database for example to; perform logging, usage by other components or to be accessed in cross platform communication and telematic services. An example is illustrated in Figure 8.8.

### 8.5.1 Proxy Ports

The entry point of the database proxy is through a port or a set of ports. A port is an interface entity for receiving/sending different data elements to its connected components. A graphical example of proxies with a single port or multiple ports is illustrated in Figure 8.7. The ports of a read proxy or a write proxy are always connected to a single component with a one to one mapping between the number of proxy ports and component ports. However, a proxy through is always connected to a pair of components and there is a one to one mapping between the number and types of input and output ports.

A proxy port receives or sends data which can be of two types, either a native data type or a complex data type.

A complex data type will however be transformed for further processing by the proxy into a set of native data types and vice versa. An example is illustrated in Figure 8.6 where the complex data type in Port_2 is transformed into data elements (b, c, d) by the proxy and put into an ordered data set. In a similar manner the transformation can be performed in the other direction, elements in the ordered set to the complex data type.

### 8.5.2 Proxy Data Sets

Database proxy's supports either read or write operations. The data flow and data transformation from a proxy port to the database is thereby made in two directions.

The data type of each port connected to a write proxy is transformed into an ordered data set. An ordered data set consists of a number of data elements which are all native data types. A complex data type will therefore be transformed into a set of native data types and included as data elements in the ordered data set. This ordered data set is then directly mapped to a database query where each data element in the ordered set corresponds to a specific row and column in the database.

An example is illustrated in Figure 8.6 where **Port_1** has a native data type `a` and **Port_2** has a complex data type that includes three the native data types. `a` is directly put in the ordered data set whereas the complex data type is transformed into elements `b, c and d` and put in the ordered data set for further transformation via a database query to the database.

The flow of a read proxy is the opposite. A database query provides the proxy with data elements that match the ordered data set. Each data element is then transformed into a type that matches the type of the individual output port/ports.

Since both native and complex data types are known during system development, the transformation routines are created during the glue-code generation.

### 8.5.3 Hard Real-Time Database Proxies

Hard proxies are intended for hard real-time components, which need efficient and deterministic access to individual data elements or a predefined set of data elements. Typical usages of hard proxies are for hard real-time data that is shared between several hard real-time components, or a mix of hard and soft real-time components. Hard proxies are implemented using database pointers.

A proxy with an ordered data set that consists of a single data element utilizes a single database pointer for predictable data access. However, a proxy with an ordered data set which consists of two or more data elements, utilizes a multi-column database pointer, as stated in section 8.4.1, to perform an atomic update or read of a defined set of columns on a single row in the database.

In order to be predictable, a hard proxy only translates native data types and fixed-length complex data types. This implies that no unpredictable type conversions or translations that require unbounded iterations are allowed. A

complex data type, such as a C-struct or an array must be of fixed-length. A hard proxy ordered data set is therefore always directly related to a fixed number of columns on a single row in the database.

Regardless of the number of ports, the database pointer interface will always ensure an atomic update/read of all data elements in the ordered data set.

By using database pointers, that provide hard real-time guaranties, to access individual/multiple data items in a database, our requirement **R3** is satisfied.
A hard real-time database proxy:

- Communicates with the database through a database pointer, thereby providing predictable data access.
- Reads or updates multiple data items atomically.
- Translates native and complex data types between components and database, using predictable data translation mechanisms.

Hard real-time database proxies can also be used to perform efficient and predictable logging. In this case, a table is defined with a fixed number of rows which are updated sequentially as a circular buffer using a multirow database pointer that automatically moves to the next row at the end of each transaction.

### 8.5.4 Soft Real-Time Database Proxies

Soft proxies are intended for soft real-time components, which usually have a more dynamic behavior and thus might have a need for more complex variable-length data. Typical usages for soft proxies include graphical interface components, logging components, and diagnostics components. Therefore, soft proxies emphasize support for more complex data structures by using a relational interface provided by SQL, towards the RTDBMS.
A soft real-time database proxy;

- Communicates with the database through a relational interface, thereby providing a flexible data access.
- Translates complex data types, thereby providing means for components to access complex data.

Since the relational interface is capable of accessing complex data, more elaborate data translation is needed in order for the components to remain

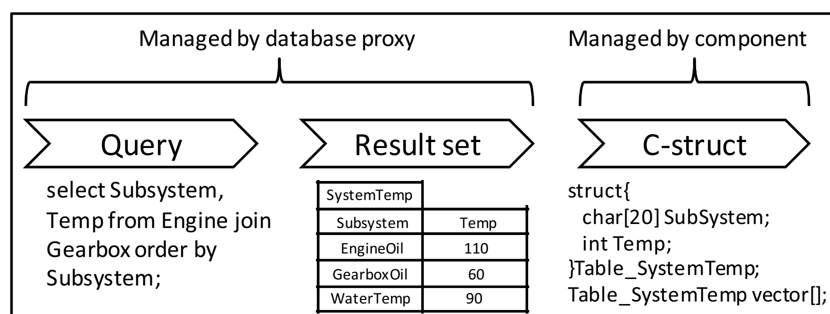



Figure 8.9: Description of TABLE Type

database unaware. To solve this, a special data template denoted `TABLE` is introduced. A `TABLE` is automatically instantiated as a C language representation of a record (row) in a relational table, and the database proxy produces (or receives) a vector of these instantiations. The component model is then augmented to allow components to communicate using ports with data types matching the instance of the `TABLE`.

Consider the following example (see Figure 8.9):

- A component used to log temperatures in a vehicle needs information about all temperature variables that exist in the system and their current value.
- An instance of a `TABLE` called `Table_SystemTemp` is created in the generation of glue-code, represented by a C-struct containing the members `SubSystem` and `Temp`.
- The type of the port in the component is then set by the component developer to a `(Table_SystemTemp *)`.
- The database proxy is then implemented using a query that matches the members in the `TABLE`.
- The translation glue code iterates through the result set from the database and fills the vector with data from the result set.

| Proxy Support | Hard Proxy | Soft Proxy |
|---|---|---|
| Database Interface | Database Pointer | SQL |
| Predictable Data Access | X | - |
| Flexible Data Access | - | X |
| Allowed for Hard Task | X | - |
| Allowed for Soft Task | X | X |
| Multi Value Support | X | X |
| Multi Port Support | X | X |
| Database Read | X | X |
| Database Update | X | X |
| Database Insert | - | X |
| Support Complex Data Type | X | X |
| Support Fixed-Length Data | X | - |
| Support Variable-Length Data | - | X |

Table 8.1: Hard and Soft Proxy Support Overview

Introducing a `TABLE` data template does not make components database aware since components still can communicate using a `TABLE` instance in absence of a database. Table 8.1 presents an overview of the similarities and differences between hard proxies and soft proxies presented in Sections 8.5.3 and 8.5.4.

### 8.5.5 Extended System Design and Modeling

We complement the classical architectural view, presented in section 8.4.2, with a new additional design view, the **CBSE database-centric view**. This new view identifies which component ports are connected, via different types of database proxies, to data elements in an RTDBMS. An example of this is illustrated in Figure 8.10. The notation simplifies the view of the system by removing the actual connection between the producing and consuming component, thus replacing it with a database symbol.

To enable traceability, this view can be transformed at any time to reveal the data flow through the connections such as shown in Figure 8.5. This is similar to an **off-page connector** that is used when designing electrical schemas which involve a large number of components and connections. A connection ends in a symbol or an identification name that is displayed at each producer and

consumer. Displaying all connections in a complex schematic diagram would make the electrical schema impossible to read. This approach is also being used by CBSE-tools such as Rubus Integrated Component Environment (Rubus ICE) [36].

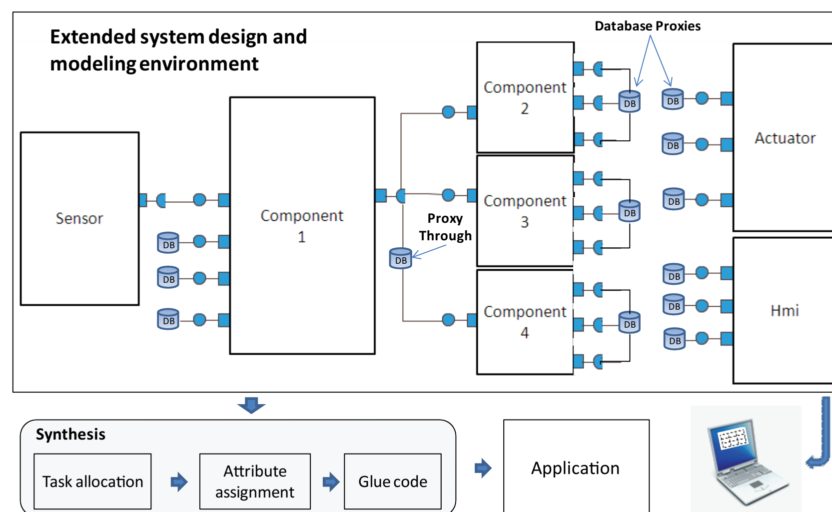



Figure 8.10: Database View of Application Model

During system design, an architect or developer can utilize both traditional data passing through connections or via an RTDBMS providing a blackboard data management architecture. An RTDBMS can be used as the single source of memory management or it is possible to utilize a mix of both connections and an RTDBMS when additional data management is needed to meet the system requirements.

As an example, the usage of an RTDBMS could be considered useful when several components and tasks share data and/or there is a need to perform logging, diagnostics or to display information on an HMI. However, if two components share a single data item that is of no additional interest, it is probably not necessary to map that item to the RTDBMS.

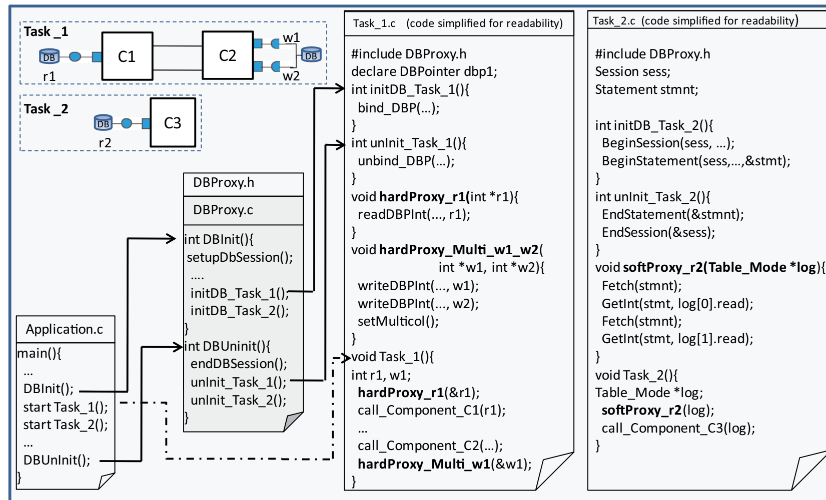



Figure 8.11: Hard and Soft Proxy Glue-Code Generation Example

### 8.5.6 Database Proxy Example

Figure 8.11, which has been simplified for readability, shows a simple example of how the glue-code generated from the database proxy specification for hard and soft database proxies of different types are implemented. In the upper left of the figure, the architecture of two tasks is displayed. Task_1 is a hard real-time task that consists of components **C1** & **C2**. Task_2 is a soft real-time task that consists of component **C3**.

Task_1 implements two hard database proxies. Component **C1** uses a database proxy to read a native data type from the database, filters it and outputs the result to component **C2**. **C2** writes its output to the database using a single database proxy to achieve an atomic write of data from two ports. This is illustrated by `hardProxy_Multi_w1_w2()`, where the two input values are sequentially written by the call to function `writeDBPInt()` and atomically committed by `setMulticol()`. These data items can then be used by any other component in the system using a database proxy.

Task_2 shows an example of a soft database proxy implementation where component **C3** reads a type `Table_Mode *` which include the two values

updated by the proxy connected to two ports in Task_1. The flow pointed out by the arrows in Figure 8.11, for the hard real-time task, **Task_1.c**, is also valid for the flow in the soft real-time task, **Task_2.c**.

The flow of the execution can be divided in three phases, initialize, running task and un-initialize.

**Phase 1: Initialize**

1. **application.c** is the main application file. Before the task/tasks containing a database proxy/proxies are called, the database is initialized by calling the `DBInit()` function declared in the separate **DBProxy.c** file.

2. Each task's individual, initialization function, `initDB_Task_1()` and `initDB_Task_2()` respectively, is called to bind hard proxy real-time database pointers and to setup soft proxy real-time statements.

**Phase 2: Task execution**

1. The database proxies are included in the task files, **Task_1.c** and **Task_2.c**.

2. The database proxies are declared as separate functions which are called before the component call if it is connected to an input port in order to read the required value/values.

3. If the database proxy is connected to an output port the call to the database proxy is made after the component's call to write/update the database.

**Phase 3: Un-initialize**

1. When the task has completed its execution, `DBUninit()` is called.

2. `DBUninit()` un-initializes the database connections in all tasks.

## 8.6 Implementation

To demonstrate the practicability of database proxies and as a proof of concept, we have implemented our approach. Three existing tools and technologies, namely the SaveComp Component Technology (SaveCCT) [13], Mimer Real-Time edition (Mimer RT) [30] and the Embedded Data Commander (EDC) [37], have been used to manage the different parts of the development. A brief introduction and the role of these tools and technologies are presented in the following three parts of this section. In the last two parts, presents our development framework and discuss the predictability of our implementation.

### 8.6.1 SaveCCT Component Technology

The SaveComp Component Technology (SaveCCT) [13] distinguishes between manual design, automated activities, and execution. The developer can create his/her application in the graphical tool Save Integrated Development Environment (Save-IDE). Automated synthesis activities generate code used to glue components together and group them into tasks. The tasks can then be executed on a real-time operating system. SaveCCT is intended for applications with both hard and soft real-time requirements.

In our implementation, the SaveCCT synthesis has been extended to also support database proxies.

### 8.6.2 Mimer SQL Real-Time Edition

The Mimer SQL Real-Time Edition (Mimer RT) [30] is a real-time database management system intended for applications with a mix of hard and soft real-time requirements. Mimer RT implements the database pointer interface to access real-time data in an efficient and deterministic manner. All hard real-time data access is performed in main-memory using predictable real-time algorithms. Mimer RT supports single, multirow and multicolumn database pointers, which can be flushed to persistent storage without interrupting real-time predictability of read and write operations.

For soft real-time database access SQL queries are used. To enable both flexibility and predictability, Mimer RT combines the traditional client/server architecture with a shared memory approach in which all real-time clients access the real-time data directly through shared memory areas. This enable efficient and predictable access to real-time data without introducing sources of unpredictability otherwise found in most traditional database managers. Examples of such sources are; context-switches between client and server, query management, index lookups, disc I/O, and data searches. Synchronization between concurrent database pointers and soft real-time SQL-queries are performed using optimized and predictable real-time locks with bounded blocking times.

### 8.6.3 Embedded Data Commander Tool-Suite

The **Embedded Data Commander** (EDC) is a tool-suite intended for high-level data management of run-time data. The tool suite has been extended with new functionality to support SaveCCT.

```
1.<SIGNAL id="P_FindFB_W" component="Find">
2.<SNIPPETDEF type="int Fi_FindFB;"
  pointerdefine="MimerRTDbp dbp_P_FindFB_W;"/>

3.<SNIPPETINIT bindquery="MimerRTBindDbp(
  &hrtsess,&dbp_P_FindFB_W,DBP_DEFAULT,
  L"SELECT state FROM Mode WHERE
  Subsystem="find");"/>

4.<UPDATECALL call="MimerRTPutInt(&
  dbp_P_FindFB_W,Fi_FindFB);"/>
5.</SIGNAL>
```

Figure 8.12: Hard Proxy Representation

Save-IDE generated description files are used by EDC in order to model the database and generate a database definition file. A database proxy description file is also generated using the Save-IDE description files and the database model. The database proxy description file is then used by Save-IDE to generate the glue code.

A database proxy definition is represented in XML. Figure 8.12 shows an example of a hard proxy description using Mimer RT. The XML code is disposed as follows. (1) The id of the signal and which component it resides in. (2) The definition of type and pointer declaration. (3) The function used to bind the database pointer with a pre-compiled statement. In this example however represented by an SQL query to enhance readability. (4) The type of call to use, in this case an update call since it is a write proxy. (5) End of proxy definition.

### 8.6.4 The Database Proxy Development Framework

In our implementation of the database proxy development framework (see Figure 8.13), SaveCCT is used to manage the development chain from system design to target code generation. EDC is used to model and generate database definition files and database proxy description files. Mimer RT manages all database activities at run-time.

In our framework, the system architect can utilize a database as an additional design feature. If a database is included in the design, the generated **System Description File** is extracted from SaveCCT to the EDC in order to perform the data modeling and generate a **Database Proxy Descriptions File**.

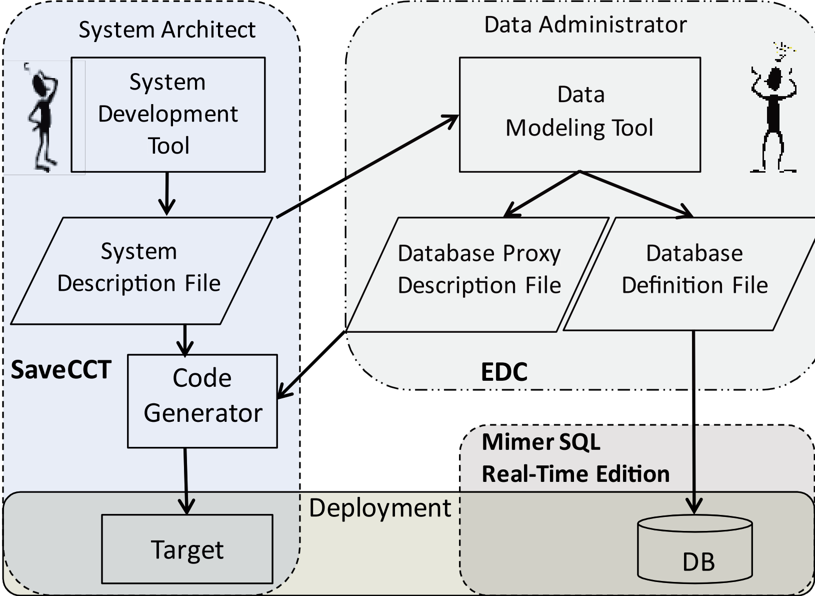



Figure 8.13: Database Proxy Development Framework

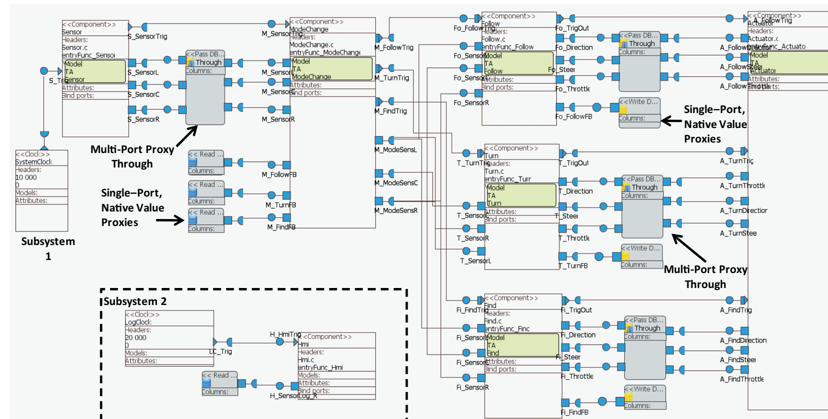



Figure 8.14: Graphical Representation of Implementation

These files are then weaved together using the **Code Generator** in SaveCCT to form the C-code for the target system. A **Database Definition File** is also generated from the EDC to setup Mimer RT.

### 8.6.5 Predictability of Implementation

For hard proxies, the generated code contains no unbounded behavior and WCET and memory usage can easily be statically bounded (although such analysis is beyond the scope of this paper). Also, the database-pointer interface of Mimer RT provides the same functions that has been proven temporally and spatially predictable within the COMET project [35]. Thus, our implementation is suitable for use in hard-real time systems.

Soft proxies do not affect the predictability properties of the system.

## 8.7 Performance Evaluation

This section presents the results of a performance evaluation where we have implemented an embedded control system and measured execution times and memory overheads. The aim of the evaluation is to measure if the database proxies will have an impact on the observed worst- or average-case execution time and how it will affect memory consumption of the system compared to

using internal data structures. Two separate implementation configurations are evaluated. (1) Using only single-port, native values database proxies and (2) A more complex configuration which includes logging and a mix of single-port, native value database proxies and multi-port proxy through.

### 8.7.1 The Application

To evaluate our approach, an application that includes two subsystems and two configurations has been implemented using the Save-IDE. The implementation is done according to Figure 8.14, which utilizes a mix of both internal data structures and an RTDBMS. The application consists of seven components and simulates a truck that first follows a line. At the end of the line, the truck turns for a certain amount of time until it finds the line and starts following it again (see Figure 8.15).

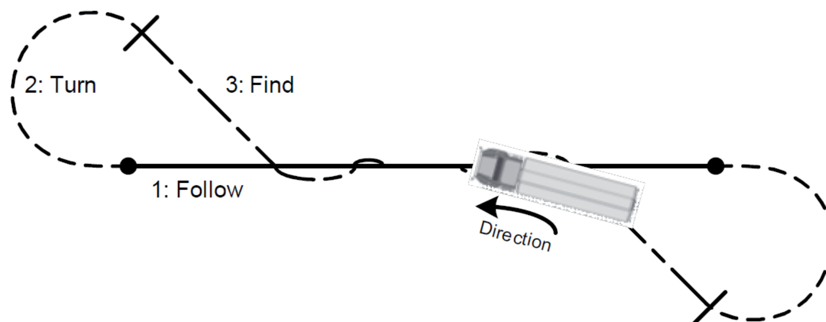



Figure 8.15: Truck Application

The first subsystem consists of a hard real-time control loop including six components that are periodically executed every 10ms.

In **configuration (1)**, six single-port, native value proxies are used in a feedback loop. The feedback values read by the three proxies are then used as input to the Modechange component in the same subsystem. In addition, these values are also used as input via a proxy to the second soft real-time HMI subsystem.

In **configuration (2)**, four proxy through is added to the six proxies in configuration (1). This more complex configuration is used to monitor and log the output from components Sensor, Follow, Turn and Find. For each component there is a log that consists of a 1000 rows that is updated as a circular buffer.

The second subsystem consists of a soft real-time HMI component that is periodically executed every 20ms. Common for the two subsystems is that they share data that needs to be protected. However, when a database proxy is used, this is managed automatically by the database.

In order to measure the impact of introducing database proxies in a CBSE system under typical workload conditions, a standard worst case execution time benchmark code called *ndes* has been used as workload in the control components. The workload is a part of a collection of benchmark codes used by different research groups and tool vendors around the world to mimic the behavior of a typical embedded system [38].

### 8.7.2 Benchmarking Setup

We have conducted a performance evaluation with four different implementations variants, and the above stated two configurations of the truck application. Each implementation is evaluated using both configurations, except for the usage of regular SQL in configuration 2. In this evaluation, the result from the usage of SQL in configuration 1 covers our interest by showing that it is not a predictable solution.

The tests have been performed on a Hitachi SH-4 series processor [39] with VxWorks v6 [40] as real-time operating system. Furthermore, Mimer RT, SaveCCT and EDC have been used throughout the implementation.

The descriptions of the four implementations (shown in Figure 8.16), are as follows:

**Test 1** A baseline implementation using internal data structures without any database connection. All component glue-code is generated by Save-IDE. Protection of shared data is hand coded using semaphores.

**Test 2** An implementation using database unaware components that is generated by Save-IDE. The hard real-time subsystem utilizes hard real-time database proxies to manage access to the database. The soft real-time subsystem utilizes a soft real-time database proxy to manage access to the database. The RTDBMS manages protection of all shared data.

**Test 3** An implementation using database aware components. The access to the database is made from within the components using database pointers. The components are generated by Save-IDE. However, the code to access the RTDBMS has been hand coded. The RTDBMS manages all protection of shared data.

**Test 4** An implementation using database aware components with access to a non real-time database from within the component using regular SQL queries without hard real-time performance. The components are generated by Save-IDE. Thus, the SQL queries inside the components have been hand coded. The DBMS manages protection of shared data.

### 8.7.3 Proxy Real-Time Performance Results

Figure 8.16 shows the result of the execution times for 1800 executions of the hard real-time control applications for the four test-cases explained in section 8.7.2 and the two configurations explained in section 8.7.1.

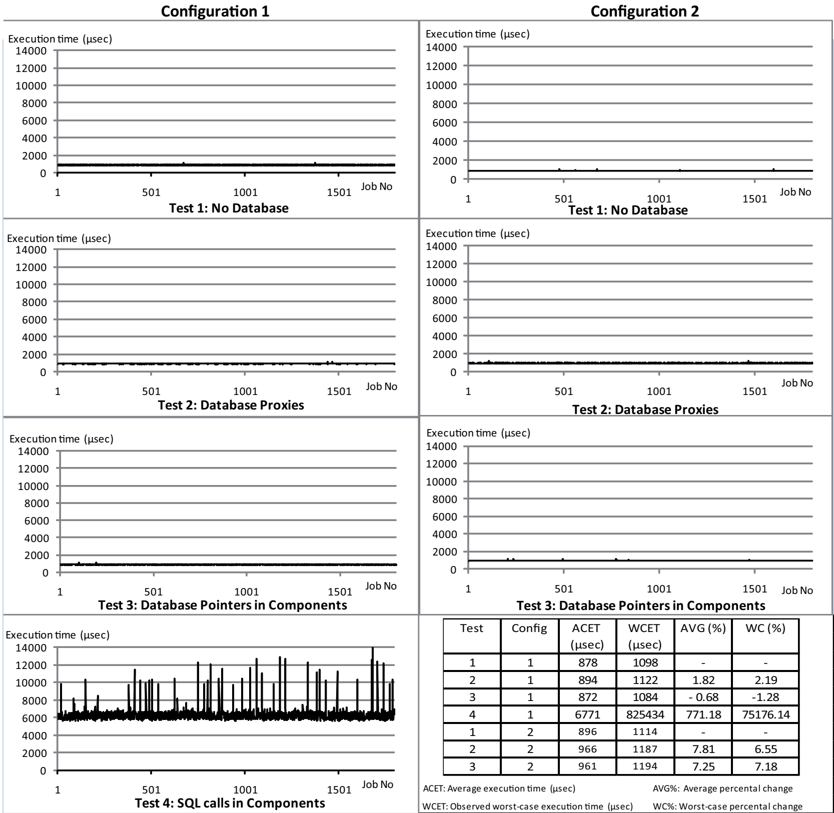
The graphs clearly illustrate that the introduction of a real-time database using database pointers, either directly in the component-code or through database proxies, does not affect the real-time predictability and adds little extra execution time overhead. On the other hand, using SQL queries directly in the component-code severely affects both predictability and performance negatively.

The table in the lower right hand corner of Figure 8.16 shows the evaluation results for each test and configuration. The change of the Average Case Execution Time (ACET) and observed Worst Case Execution Time (WCET) in the two rightmost columns shows the change in percent, compared to our baseline, **Test 1**.

For the first three tests, the ACET and WCET values do not differ from one test to another with more than a few percent. Test four, configuration 1 does, as could be expected, not perform anywhere near the other tests.

As this evaluation aims to measure and evaluate the performance of database proxies, **Test 2** is most interesting. Configuration 1 shows that the ACET is increased by only 1.82% and the WCET by 2.19%. For configuration 2, which includes more complex operations the result shows that the ACET is increased by 7.81% and the WCET by 6.55%.

Figure 8.17 shows more detailed information of the first 100 executions for both configurations of **Test 1** and **Test 2**. The evenness of the results clearly illustrates that the usage of database proxies in combination with an RTDBMS such as Mimer RT is predictable, and the amount of overhead in average and worst-case execution time is limited. Furthermore, these results confirm our predictability of implementation discussion in Section 8.6.5.




| Test | Config | ACET (µsec) | WCET (µsec) | AVG (%) | WC (%) |
|---|---|---|---|---|---|
| 1 | 1 | 878 | 1098 | - | - |
| 2 | 1 | 894 | 1122 | 1.82 | 2.19 |
| 3 | 1 | 872 | 1084 | - 0.68 | -1.28 |
| 4 | 1 | 6771 | 825434 | 771.18 | 75176.14 |
| 1 | 2 | 896 | 1114 | - | - |
| 2 | 2 | 966 | 1187 | 7.81 | 6.55 |
| 3 | 2 | 961 | 1194 | 7.25 | 7.18 |

ACET: Average execution time (µsec)
AVG%: Average percental change
WCET: Observed worst-case execution time (µsec)
WC%: Worst-case percental change

Figure 8.16: Evaluation Results From Configuration 1 and 2

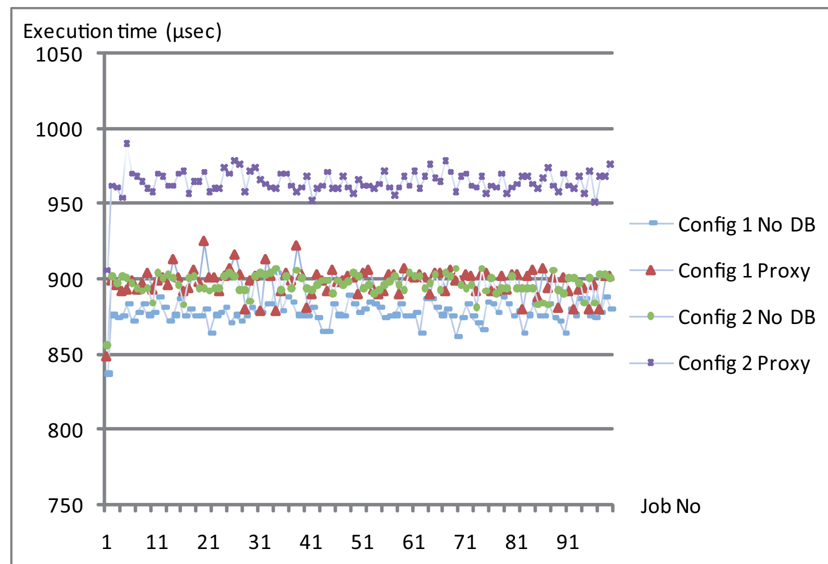



Figure 8.17: Execution Time Evaluation Results for 100 Executions

Our conclusion is that the slight decrease in ACET and WCET for **Test 3** is the result of optimized synchronization primitives used by Mimer RT compared to the regular POSIX semaphore routines used in **Test 1**.

| **Access Method** | **Code Size** | **Change (%)** |
|---|---|---|
| No Database | 653 512 bytes | - |
| Database Pointers | 666 564 bytes | 1.99 |
| Database Proxies | 666 988 bytes | 2.06 |

Table 8.2: Application Code Size

### 8.7.4 Memory Consumption Results

Table 8.2 shows how the client code size changes when using different data management methods. As can be seen in the table, integrating a real-time database client with the calls hand coded in the component code introduces 1.99% extra code size. By using database proxies that have been automatically generated, the code size grows with as little as 2.06%.

Introducing a real-time database server in the system of course also introduces extra memory consumption, but embedded database servers are becoming smaller and smaller. The Mimer SQL database family that is used in this evaluation has a code footprint ranging from 273kb for the Mimer SQL Nano database server, up to 3.2Mb for the Mimer SQL Engine for enterprise systems.

The RAM usage for Mimer SQL Nano is as low as 24kb. The limited increase of client code size, as well as the small size of modern embedded database servers makes the memory overhead for database proxies in conjunction with a real-time database affordable for many of today's real-time embedded systems. This added code size and memory overhead should also be considered in balance with the added value of the techniques.

## 8.8 Conclusions

This paper presents the database proxy approach which enables an integration of real-time database management systems into a component-based software engineering framework. While maintaining strict component encapsulation, we achieve benefits such as the possibility to access data via standard SQL interfaces, concurrency-control, temporal consistency, and transaction manage-

ment. In addition, a new possibility to use dynamic run-time queries to aid in logging, diagnostics and monitoring is introduced.
The motivation for our approach stems from observations of industrial practices and documented needs for a standardized and overall data model and management system to deal with the distributed and uncoordinated data in these complex systems. Furthermore, it is clearly stated that the adhoc/reinvented management of data as well as individual solutions using internal data structures, can lead to concurrency- and inconsistency problems and decreases maintainability, extensibility and flexibility of the system [7, 14, 17, 19].

To evaluate our approach, an implementation that covers the whole development chain has been performed, using both research oriented and commercial tools and techniques. The system architecture is implemented in Save-IDE. The architectural information is then generated and exported to a data management tool, where the database proxies and interface to the database is created. The data management tool then generates the database proxy information back to Save-IDE for further generation of glue-code and tasks for the entire system.

To validate our approach further, a series of execution time tests has been performed on the generated C-code for a research application. These tests show that our approach using native value communication, only increases the average and the worst-case execution time with approximately 2%. In addition, complex database proxies connected to several ports of a component which performs atomic updates of circular logs, each consisting of 1000 rows, only increases the average execution time with approximately 7.8% and the worst-case execution time with approximately 6.5%. Furthermore, the memory overhead, also about 2%, introduced by database proxies can be affordable for many classes of embedded systems.

We conclude that the database proxy approach enables an RTDBMS to be successfully integrated into a component-based software engineering framework. This enables developers to utilize the benefits from an RTDBMS which offers a range of valuable features that can solve current and future issues when developing, maintaining and evolving real-time embedded systems at a minimal cost with respect to resource consumption.

## Acknowledgment

This work is supported by the Swedish Foundation for Strategic Research within the PROGRESS Centre for Predictable Embedded Software Systems.

# Bibliography


[1] Andreas Hjertström, Dag Nyström, and Mikael Sjödin. Database Proxies for Component-Based Real-Time Systems. In *22st Euromicro Conference on Real-Time Systems*, July 2010.

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

[3] 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.

[4] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley and Sons, 1996.

[5] Ivica Crnkovic and Magnus Larsson. *Building Reliable Component-Based Software Systems*. Artech House, 2002.

[6] OMG UML. The Unified Modeling Language UML. http://www.uml.org/.

[7] Sandro Schulze, Mario Pukall, Gunter Saake, Tobias Hoppe, and Jana Dittmann. On the Need of Data Management in Automotive Systems. In Johann Christoph Freytag, Thomas Ruf, Wolfgang Lehner, and Gottfried Vossen, editors, *BTW*, volume 144 of *LNI*, pages 217–226. GI, 2009.

[8] Simon Fürst. Challenges in the Design of Automotive Software. In *Proceedings of the Conference on Design, Automation and Test in Europe*,

DATE 10, pages 256–258, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[9] Klaus Grimm. Software Technology in an Automotive Company - Major Challenges. *Software Engineering, International Conference on Software Engineering*, page 498, 2003.

[10] A. Albert. Comparison of Event-Triggered and Time-Triggered Concepts with Regard to Distributed Control Systems. *Proc. Embedded World*, pages 235–252, 2004.

[11] Peter Liggesmeyer and Mario Trapp. Trends in embedded software engineering. *IEEE Softw.*, 26:19–25, May 2009.

[12] AUTOSAR Open Systems Architecture. http://www.autosar.org.

[13] Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson, and Massimo Tivoli. The Save Approach to Component-Based Development of Vehicular Systems. *Journal of Systems and Software*, 80(5):655–667, May 2007.

[14] Andreas Hjertström, Dag Nyström, Mikael Nolin, and Rikard Land. Design-Time Management of Run-Time Data in Industrial Embedded Real-Time Systems Development. In *Proceedings of 13th IEEE International Conference on Emerging Technologies and Factory Automation, Germany*, September 2008.

[15] 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.

[16] R. R. Brooks, S. Sander, J. Deng, and J. Taiber. Automotive System Security: Challenges and State-of-the-Art. In *Proceedings of the 4th annual workshop on Cyber security and information intelligence research: developing strategies to meet the cyber security and information intelligence challenges ahead*. ACM, 2008.

[17] Alexander Pretschner, Manfred Broy, Ingolf H. Kruger, and Thomas Stauner. Software Engineering for Automotive Systems: A Roadmap. *Future of Software Engineering*, pages 55–71, 2007.

[18] Manfred Broy. Challenges in Automotive Software Engineering. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 33–42. ACM, 2006.

[19] Gunter Saake, Marko Rosenmuller, Norbert Siegmund, Christian Kästner, and Thomas Leich. Downsizing Data Management for Embedded Systems. *Egyptian Computer Science Journal*, pages 1–13, 2009.

[20] Nelly Delgado, Ann Quiroz Gates, and Steve Roach. A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools. *IEEE Trans. Softw. Eng.*, 30:859–872, December 2004.

[21] AUTOSAR Open Systems Architecture. http://www.car-to-car.org.

[22] Marc Zeller Gereon Weiss and Dirk Eilers. *Towards Automotive Embedded Systems with Self-X Properties*. InTech, 2011.

[23] Theo Härder. DBMS Architecture – Still an Open Problem. In *PROC. DATENBANKSYSTEME IN BUSINESS, TECHNOLOGIE UND WEB (BTW 2005)*, pages 2–28. Springer, 2005.

[24] Peter Pin-Shan Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Trans. Database Syst.*, 1(1), 1976.

[25] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, December 1997.

[26] Rob van Ommering. *The Koala Component Model for Consumer Electronics Software*, chapter The Koala Component Model, pages 78–85. Computer, IEEE Computer Society, March 2000.

[27] Kung-Kiu Lau and Faris M. Taweel. Towards Encapsulating Data in Component-Based Software Systems. In *CBSE*, pages 376–384, 2006.

[28] K.-K. Lau and F. Taweel. Data Encapsulation in Software Components. In *Proc. 10th Int. Symp. on Component-based Software Engineering, LNCS 4608*, pages 1–16. Springer-Verlag, 2007.

[29] 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, 2003.

[30] Mimer SQL Real-Time Edition, Mimer Information Technology, Uppsala, Sweden. http://www.mimer.se, 2012.

[31] 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.

[32] Polyhedra In-Memory Database. http://www.enea.com, Sept 2011.

[33] ISO SQL 2008 standard. Defines the SQL language. http://www.iso.org/iso/home.htm, 2009.

[34] John A. Stankovic and Wei Zhao. On Real-Time Transactions. *SIGMOD Rec.*, 17:4–18, March 1988.

[35] 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, 2004.

[36] Arcticus Systems. Rubus ICE. www.arcticus-systems.com/html/prod-rubus-ice.html.

[37] Andreas Hjertström, Dag Nyström, and Mikael Sjödin. A Data-Entity Approach for Component-Based Real-Time Embedded Systems Development. In *14th IEEE International Conference on Emerging Technology and Factory Automation*, September 2009.

[38] The Worst-Case Execution Time (WCET) analysis project. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[39] Hitachi SH-4 32-bit RISC CPU Core Family. http://www.hitachi.com/.

[40] VxWorks Real-Time Operating System, by Wind River. http://www.windriver.com/.

# Chapter 9

# Paper D: Introducing Database-Centric Support in AUTOSAR

Andreas Hjertström, Dag Nyström and Mikael Sjödin
*7th IEEE International Symposium on Industrial Embedded Systems (SIES)*

*Karlsruhe, Germany, June, 2012*

**Abstract**

We propose to integrate a real-time database management system into the basic software of the AUTOSAR component model. This integration can be performed without violating the fundamental principles of the component-based approach of AUTOSAR. Our database-centric approach allows developers to focus on application development instead of reinventing data management techniques or develop solutions using internal data structures. We use state-of-the-art database pointer techniques to achieve predictable timing, and database proxies to maintain component encapsulation and independence of data-management strategies. The paper illustrates the feasibility of our proposal when database proxies are used to manage the data communication between components and to perform run-time monitoring on the virtual function bus. Our implementation results show that the above benefits do not come at the expense of less accurate timing predictions while only introducing a total application CPU overhead, in the order of 4%.

## 9.1 Introduction

By integrating an RTDBMS into AUTOSAR, developers gain access to well established and powerful tools and techniques that have facilitated data management in complex, data intensive systems within other areas such as financial markets for decades. Techniques to achieve more dynamic and structured data management which include data extraction, management of user access rights and dynamic or static runtime monitoring would thereby be made available even for such critical domains as automotive systems. In addition, this approach allows information to be shared, traced, logged and viewed using diagnostics tools or third party tools.

Database proxies [1] has been presented as a successful technique that enables the integration of a Real-Time DataBase Management System (RT-DBMS) [2] into a component technology [3]. Database proxies are automatically generated glue code, synthesized from the system architecture, that translates data between the ports of the components and an RTDBMS residing in the component framework.

For component-based automotive systems such as AUTomotive Open System ARchitecture (AUTOSAR) [4], integrating an RTDBMS is not trivial since the component-based approach of AUTOSAR favor encapsulation and reuse, while the database centric approach favor an open blackboard data architecture.

Component-Based Software Engineering (CBSE), strives to decouple components from the context in which they are deployed. One aspect of this is that a component should not have built-in assumptions about external data-elements. This decoupling is achieved by encapsulating component-functionality and making visible only a component-interface describing the provided and required services. Using an RTDBMS in existing component-based systems would require database calls from within the component thereby making the component unusable in an alternative setting. In order to succeed with the integration of an RTDBMS into AUTOSAR, components need to be decoupled from the RTDBMS and the underlying database schema.

**The contribution of this paper** presents a solution for how to integrate a real-time database management system, COMET [5], into an AUTOSAR platform and tool suite, such as the Arctic Core [6]. This is achieved without violating the fundamental principles of the component-based approach or, the fundamentals of the AUTOSAR standard itself. The feasibility of our approach is demonstrated with an implementation of an Adaptive Cruise Control (ACC) using database proxies for all component communication on the VFB.

Finally, we present an evaluation which shows that the cost for introducing database management support via database proxies in AUTOSAR is negligible. Under typical workload conditions, our concept only introduces a total application CPU overhead, in the order of 4%.

## 9.2 Background and Motivation

Since the computerization of cars, automotive software systems have evolved from in-house monolithic control-systems to integrated component-based software-systems. Initially, automotive software controlled only fundamental functions as fuel and ignition control; today automotive software has become fully interconnected with the surrounding environment through entertainment software, internet access and advanced diagnostics systems. This evolvement has led to an increasing complexity resulting in costly development and maintenance [7, 8].

To reduce this complexity, model driven development and component-based software engineering, e.g. using AUTOSAR, are widely used in industry today [9]. However, these techniques mainly focus on the functional aspects of the software, and rarely target management of data. In addition, the lack of run-time data management was pointed out as a significant problem for the automotive domain, as well as for transportation and industrial control [10]. Moreover, the increasing need for more structured, flexible, reliable and secure data management techniques to coordinate data both at run-time and at design-time is continuously pointed out as major challenges for the future [11, 12, 13].

As stated by Pretschner et al. [8] and Broy [14], a standardized and overall data model and management system has great potential as a solution to deal with the distributed and uncoordinated data in these complex systems. Furthermore, Schulze et al. [11] and Saake et al. [15] points out that the ad-hoc and/or reinvented management of data for each ECU with individual solutions using internal data structures, can lead to concurrency and inconsistency problems. In addition, maintainability, extensibility and flexibility of the system decrease.

Moreover, sophisticated techniques for diagnostics, error detection, logging and secure data sharing are much needed to improve reliability and system quality. Inefficient diagnostics and error tracing techniques has led to that more than 50% of replaced ECUs are in fact not defect [8]. Much of the diagnostics messages and logging that can be retrieved from these systems are statically predefined at design time. A possibility to perform dynamic run-time monitoring and/or diagnostics of the system could greatly aid developers.

In other techniques such as the Program Monitoring and Measuring System (PMMS), it is up to the user to specify pre-conditions and insert code in order to collect data [16]. This put high demands on developers to predict future needs of what could be of interest to for instance a service technician.

There are well established database techniques that can aid developers with the above stated complexity available, such as Mimer SQL Real-Time Edition [17] and ExtremeDB [18]. These database systems include efficient and predictable concurrency-control, temporal consistency, and overload and transaction management [19, 20, 21]. In addition, there are efficient and well proven tools available from the database community that can aid developers in dealing with the data complexity. In spite of the fact that RTDBMSs are available, they remain unused in automotive embedded systems.

It is thereby well established that the integration of an RTDBMS into AUTOSAR could not only aid developers with standardized tool support for modeling system data at design-time, but also provide predictable and efficient routines for managing data at run-time.

## 9.3 System Model and Related Techniques

AUTOSAR supports hard real-time functionality that include critical control-functions, as well as soft real-time functionality. We therefore consider a system where functionality is divided into the following classes of tasks:

**Hard real-time tasks**, typically have high arrival rates. Hard real-time tasks use hard transactions to read and write simple values from sensors/actuators and execute real-time control loops. Hard real-time tasks cannot manage complex data structures. This limitation however, is fairly small in practice, since hard real-time components often are static, communicating with fairly simple data structures. When a database is used, hard real-time tasks require predictable access to data elements.

**Soft real-time tasks**, often with a lower arrival rate, control less critical functionality. Soft real-time tasks uses soft transactions to read and write dynamic and complex data structures typically to present statistical information, logging or used as a gateway for service access to the system by technicians in order to perform system maintenance. Soft real-time tasks could also be used for fault management and perform ad-hoc queries at run-time.

In order to support a predictable mix of both hard and soft real-time transactions, we consider an RTDBMS with two separate interfaces.

```
1 TASK oilTemp(void){
     //Initialization part
2   int temp;
3   DBPointer *dbp;
4   bind(&dbp,"Select TEMP from ENGINE
                where SUBSYSTEM='oil'");
     //Control part
5    while(1){
6      temp=readOilTempSensor();
7      write(dbp,temp);
8      waitForNextPeriod();
     }
  }
```

Figure 9.1: A task that uses a database pointer

For hard real-time transactions, a database pointer [21] interface is used to enable the application to access individual data elements in the database with hard real-time performance. For soft real-time transactions, a standardized SQL interface is used.

### 9.3.1 Database Pointers

A database pointer [22] is a hard real-time database access-method which uses an application pointer variable to access individual data in an RTDBMS, see Figure 9.1. The figure shows an example of a task (thread) that reads a sensor and propagates the sensor value to the database using a database pointer. During the initialization part (lines 2 to 4) the database pointer is created and bound to a data element in the database using the `bind` function. The `bind` function calls the database server which creates a handle directly to the data element.

During the control part, the `write` function uses this handle to directly write to the data element without calling the database server. The write operation consists of only a few lines of sequential code that performs type checking, synchronization, and writing of the data.

A key property of the database-pointer concept is that reads and writes through database-pointers have deterministic execution-time with bounded and negligible blocking [21]. They also allow SQL-based transactions to be executed in the background without any predictability loss due to any concurrent database-pointer accesses (i.e. no starvation, conflicts, or restarts of transaction can be caused by database pointers [22]).

### 9.3.2 COMET

The COMponent-based Embedded real-Time database system [5] (COMET RTDBMS) is a real-time database management system intended for applications with a mix of hard and soft real-time requirements. The COMET RTDBMS implements the database pointer interface to access individual data elements in an efficient and deterministic manner. For soft real-time database access, SQL queries are used. To guarantee hard real-time predictability for database accesses while eliminating starvation issues for soft real-time SQL queries, COMET RTDBMS uses the 2V-DBP concurrency-control algorithm [21] that combines versioning and pessimistic concurrency-control. 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.

Some technologies developed for COMET RTDBMS, including the database pointer concept, has later been adopted by the commercially available RTDBMS, Mimer SQL Real-Time Edition [17].

### 9.3.3 Arctic Core

Arccore AB [6] is a provider of the open-source Arctic Core embedded AUTOSAR platform developed in Eclipse [23]. The open-source solution, to be used for education and testing, includes Arctic Core and Arctic Studio which is an Integrated Development Environment (IDE). The commercial solution offers a number of licensed professional graphical tools to facilitate development of a complete AUTOSAR system. Arctic Core includes build scripts and services such as, network communication, memory, and operating system. In addition, drivers for different microcontroller architectures are also provided.

Components and their port-based interfaces are developed using the SoftWare Component Builder tool. The Extract Builder tool is used to add selected components to the ECU, connect ports and to validate the extract. The Run-Time Environment Builder models the VFB and generates a run-time implementation of the component communication. The configuration of the target platform is done in the Basic Software Builder tool which also generates the configuration files. Since Arctic Core is provided as open source, it is possible to extend it to also include RTDBMS support.

## 9.4 AUTOSAR Concept Overview

AUTOSAR [4] is a standard component model and middleware platform for the automotive electronic architecture. One of the fundamental concepts of AUTOSAR is to have a clear separation between the underlying infrastructure and the applications which consists of interconnected software components. A simplified explanation is that AUTOSAR consists of the following layers, see Figure 9.2; the SoftWare Component layer (SWC), Virtual Function Bus (VFB), and the Basic SoftWare layer (BSW).

The main focus in this paper is the VFB, which is the central mechanism that manages the connections and data sharing between AUTOSAR components residing in an Electronic Control Unit (ECU) or between ECUs in the system. The purpose of the VFB is that it enables a virtual integration of components early in the development phase. Since the VFB manages all component interactions, there is a clear separation between the software components and the underlying infrastructure. The realization of the VFB is the Run-Time Environment (RTE) which is generated from the specifications and the underlying BSW components. The RTE acts as a communication center for both internal ECU communication and information exchange between ECUs in the system.

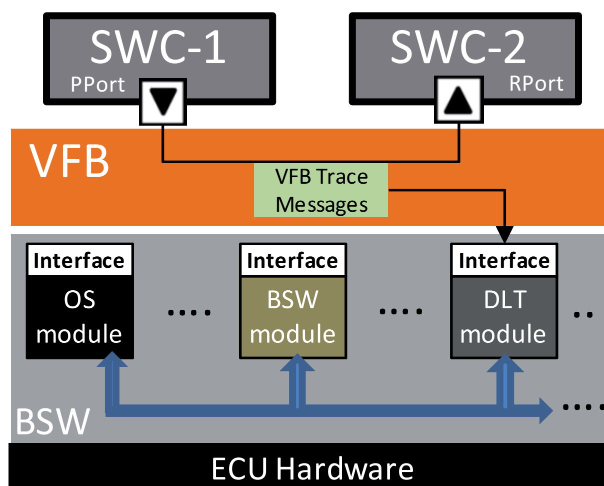



Figure 9.2: Autosar Overview

Data is shared between components by creating connections between Provider Ports (PPort) and Receiver Ports (RPort) of components. Data sets passed between components on an ECU is usually realized through shared RAM areas which have to be protected using semaphores to ensure data consistency. This makes data internal on the ECU, and visible only to those components which have been specified as receivers when developing the system.

Regulatory requirements and the complexity of automotive systems have increased the need for run-time diagnostics and system monitoring. To meet this need, AUTOSAR has, from version 4.0 included support for system monitoring via the Diagnostic Log and Trace (DLT) module. The DLT is capable of managing multiple types of diagnostic log and trace messages and transmit them to external clients remotely connected over the network. One such trace-type is the VFB trace message which is used to collect component communication in the VFB, see Figure 9.2.

Today, all diagnostics messages must be statically defined at design time, thus is it not possible to view or subscribe to VFB communication that has not been tagged for VFB tracing. An external client can initiate a session to the DLT and request that a subscription of a VFB trace should be initiated and periodically published on the network.

## 9.5 Database Proxies

Database proxies [1] are shown to be an efficient and predictable technique that offers a range of valuable features to component-based embedded real-time systems development, maintenance and evolution at a minimal cost with respect to resource consumption. A database proxy translates data between component ports and an RTDBMS that resides in the component framework (i.e., the AUTOSAR BSW) and vice versa. This allows full decoupling of the RTDBMS from the component, i.e., the component and the RTDBMS are unaware of the existence of the other. Database proxies remove the need for database calls within the component, thus preserving component encapsulation and enable component reuse. Furthermore, the schema of the database can be modeled and optimized separately and is independent of the component implementation.

Database proxies are automatically generated from the system architecture, and the run-time implementations are synthesized by the system generation tool. Therefore, database proxies are a part of the system architecture, and are realized in the form of glue code. The internal mapping of data in the

RTDBMS to the database proxies is made using database pointers for hard real-time components, or pre-compiled SQL statements or stored procedures for soft real-time components. A pre-compiled statement enables a developer to bind a certain database query to a statement which is compiled once during system setup. This has a decoupling effect since the internal database schema is hidden from the component.

To support the different requirements of hard and soft real-time tasks, we distinguish between *hard real-time database proxies* (hard proxies) and *soft real-time database proxies* (soft proxies).

### 9.5.1 Hard Real-Time Database Proxies

Hard proxies are intended for hard real-time components, which need efficient and deterministic access to individual data elements. Hard proxies support hard real-time data to be shared between several hard real-time components, or a mix of hard and soft real-time components.

Since hard real-time components manage hard real-time data, hard proxies emphasize predictable and efficient data access. Hard proxies are therefore implemented using database pointers.

A hard real-time database proxy:

- communicates with the database through a database pointer, thereby providing predictable data access.
- translates native data types only, thereby providing predictable data translation.

That a hard proxy only translates native data types such as integer, character, or float implies that no unpredictable type conversions or translation of complex data types that require unbounded iterations are needed.

### 9.5.2 Soft Real-Time Database Proxies

Soft proxies are intended for soft real-time components, which usually have a more dynamic behavior and thus might have a need for more complex data-structures. Typical usages for soft proxies include graphical interface components, logging components, and diagnostics components. Therefore, soft proxies emphasize support for more complex data structures by using a *relational interface* provided by SQL, towards the RTDBMS.

A soft real-time database proxy:

- Communicates with the database through a relational interface, thereby providing a flexible data access.
- Translates complex data types, thereby providing means for components to access complex data.

### 9.5.3 Database Proxy Constituents

The realization of a database proxy contains the following constituent parts:

- **Initialization code** that connects to the RTDBMS and opens a pre-compiled database statement or database pointer. The initialization code is executed at system startup.
- **Data translation code** which is the glue code that access the database and translate the result to/from the components. The data translation code is executed prior to or after every component execution.
- **Uninitialization code** that closes the database statement or database pointer and disconnects from the RTDBMS. The uninitialization code is executed at system shutdown.

## 9.6 Integrating Database Proxy Support in AUTOSAR

To enable efficient and dynamic data management in AUTOSAR, our approach proposes that communication between components over the VFB is handled by the RTDBMS using database proxies instead of internal RAM areas. This has substantial benefits both during design-time and run-time of the system. At design-time, all system data could be explicitly modeled using well established data modeling techniques, such as Entity/Relationship modeling [24], to achieve an efficient and optimized data model. Run-time system management would benefit from the approach since all communication would be stored in the database, thus dynamically enabling monitoring and tracing of any data during run-time. This is especially beneficial for testing and debugging purposes since internal data now can be made available for external access.

To implement a VFB using database proxies, the virtual function bus needs to be extended and the RTDBMS needs to be integrated into the BSW.

### 9.6.1 Integrating the RTDBMS

The RTDBMS is integrated in the system as a BSW module that is responsible for all system data that has been designated to be managed by the RTDBMS, see Figure 9.3. However, if two components share a single data item that is of no additional interest for other components, nor for logging or diagnostics purposes, a mapping to the RTDBMS could be superfluous. All real-time accesses to the database from the VFB are made through the Real-Time database pointer APplication Interface (RTAPI), while internal soft real-time accesses or external tools and 3rd party applications, use an SQL-based interface. These APIs are also utilized internally by other BSW modules such as the DLT module. The DLT module extracts information from the database and uses the BSW diagnostic services to forward the data to an external client.

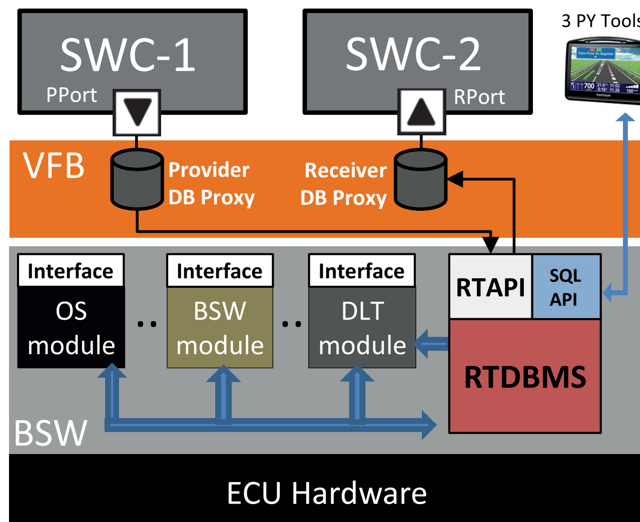



Figure 9.3: Database proxies support in the VFB

### 9.6.2 Extending the VFB with Database Proxies

The current model of the VFB, where connections are using RAM areas to connect one PPort of the providing component to the RPort of the receiving component needs to be extended to contain the following constituents:

- **PPort:** The port that provides the data.
- **RPort:** The port that requires the data.
- **Database statement:** A database statement that uniquely associates the data with a data element in the database.

During the realization of the VFB these constituents are used to create the proxies as follows (see Figure 9.3):

- **Provider DB Proxy:** The provider DB Proxy translates data from the PPort to the database using a database pointer which is bound to the database statement of the connection.
- **Receiver DB Proxy:** The receiver DB proxy translates data from the database to the RPort using a database pointer which is bound to the database statement of the connection.

The initialization code and uninitialization code of the two proxies are generated and placed in the AUTOSAR standard startup and shutdown routines, and the data translation code is placed in the connection, e.g., glue code, itself. This implies that all communication between components will be performed through the RTAPI interface of the RTDBMS, thus removing the need to use RAM areas or inter process communication operations from the operating system.

Figures 9.4 and 9.5 presents a simplified illustration of the differences of the execution trace between the original generated implementation and using database proxies for a task that includes a component.

**Execution trace for the original generated implementation:**

**1** The task invokes the `RteRunnable`.

**2** The software component is invoked by RTE.

**3** The component performs its task and write the data to its port.

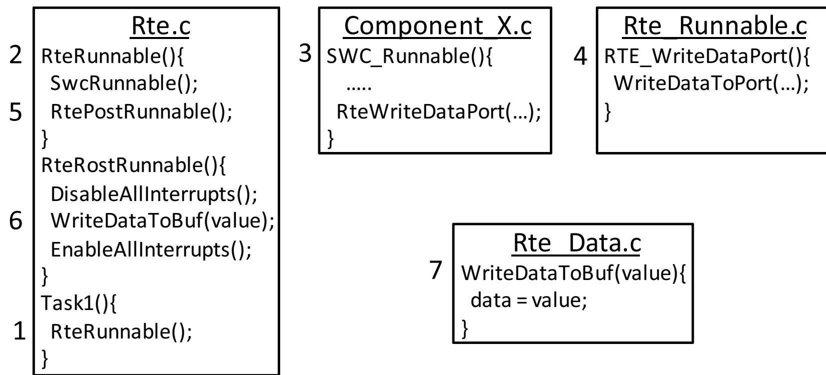



Figure 9.4: Execution trace for the original AUTOSAR code

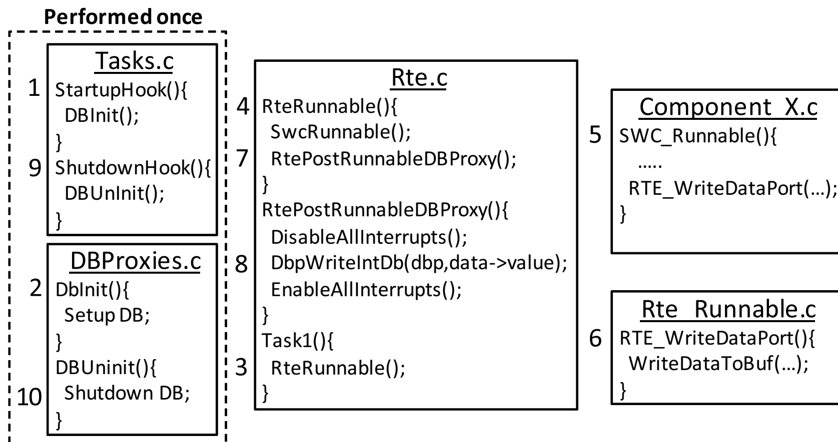



Figure 9.5: Execution trace using database proxies

**4** The RTE writes to a port variable.

**5** Since it is a provider port, the RTE writes the data after the component invocation.

**6** The `RtePostRunnable` function disables all interrupts and calls function to write the data.

**7** The data is written to the buffer. When finished, all interrupts are enabled.

**Execution trace using database proxies:**
Performed once during system startup:

**1** The AUTOSAR StartupHook calls functions to setup and initialize the database.

**2** The database is initialized during startup.

Performed during component communication:

**3-6** Corresponds to steps **(1-4)** in the previous example without database proxies.

**7** `RtePostRunnableDBProxy` is called.

**8** The data is written to the database.

Performed once during system shutdown:

**9** The AUTOSAR ShutdownHook is called during system shutdown.

**10** The database is uninitialized.

For proxies connected to an RPort, the execution flow is similar apart from that the database proxy is executed before the component is called.

The differences in call flow, except from the initialization part of the database, which is executed once during system startup is what happens when data is stored during post write. In the case of not using a database, the task of **Rte_Data.c**, is in some sense a predefined static data manager which provides functions to read and write data for each port.

When using database proxies, the post write is made to the database. Provided that a user have the correct data access rights, any data item from a single component port or data items from several port, even from different components can be queried. In addition, if the outcome from a query is a large volume of data, the data can be filtered to not provide the user with superfluous information.

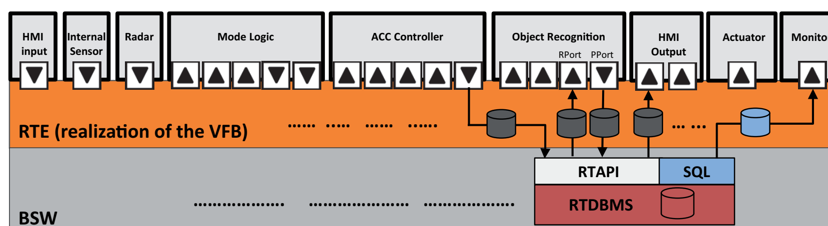



Figure 9.6: Illustration of the ACC system implementation in AUTOSAR

## 9.7 System Design and Implementation

As a proof of concept and to demonstrate and evaluate the usefulness of our approach, we have implemented an application that mimics the behavior of an Adaptive Cruise Control (ACC) system and deployed it on an AUTOSAR hardware node. The software tools and techniques that have been used are ArcCore AUTOSAR open source and professional solutions and the COMET RTDBMS. The design of the ACC application, a brief introduction and the role of included tools and technologies as well a discussion regarding the predictability of our implementation is presented in the remainder of this section.

### 9.7.1 Application System Design

The system has been designed according to the proposed approach in section 9.6 with the RTDBMS residing in the BSW and database proxies that manages all component communication. As illustrated in Figure 9.6, the nine components communicate via the RTE. The database proxies in the RTE manage the communication between PPorts and RPorts via the RTDBMS. However, the figure is simplified with a focus on a few connections to clearly illustrate the approach.

As seen in Figure 9.7, the application design consists of nine components distributed over five hard real-time tasks, T1-T5 and a soft real-time task, T6. The internal implementation of the components varies from simple Proportional Integral Derivative (PID) controllers to more complex controller logic [25].

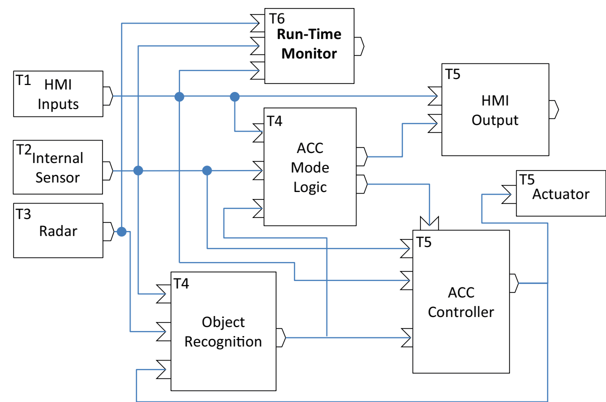



Figure 9.7: ACC application design

The nine components assignments are as follows:

- **HMI Input**, controls if the ACC is active or not as well as the desired speed. (Task 1)
- **Internal Sensor**, handles the throttle level and the actual speed. (Task 2)
- **Radar**, measures and outputs the distance to a vehicle in front. (Task 3)
- **Mode Logic**, handles the logic for different states of the vehicle and sets mode accordingly. (Task 4)
- **Object Recognition**, determine if there is an obstacle in front. If so, calculate the relative velocity and trigger a mode switch to for instance reduce speed. (Task 4)
- **ACC Controller**, manages speed control according to the distance and mode. (Task 5)
- **HMI Output**, outputs information regarding the vehicle state and displays it to the driver. (Task 5)
- **Actuator**, control throttle and speed of the vehicle. (Task 5)
- **Run-Time Monitor**, Monitors the output from system sensors. (Task 6)

The whole application is developed and generated using the different tools in Arctic Studio.

### 9.7.2 Predictability of the Implementation

The ACC implementation includes a mix of hard and soft database proxies, the code contains no unbounded behavior and WCET and memory usage can be calculated (although such analysis is beyond the scope of this paper). A hard real-time database-pointer provides direct access to a data element in memory without calling the database server.

This implies that from a predictability perspective, database proxies do not introduce any additional context switches, compared to the original implementation. A write operation consist a few lines of sequential code that performs type checking, synchronization, and writing of the data. This is similar approach as using a pointer variable and semaphores in C.

In addition, the database-pointer interface has been proven temporally and spatially predictable within the COMET project [21]. Thus, our implementation is suitable for use in hard-real time systems.

The use of an RTDBMS, which is developed for this purpose and have undergone extensive validation, could be seen as single point of failure. However, this must be compared to the ad hoc and individual solutions, currently used in these complex systems [11, 15].

## 9.8 Evaluation

In order to evaluate the performance of our approach and to validate the practicality of database proxies under realistic workload conditions, a performance evaluation of the ACC application has been conducted. In addition, the evaluation includes a soft run-time monitor component that continuously extracts data, using a soft proxy. The aim of the evaluation is to verify the predictability and measure the CPU overhead introduced by integrating database proxies.

### 9.8.1 Benchmark Setup

The evaluation was performed on a board, named VK-EVB-M3, equipped with a STM32F107 ARM Cortex M3 processor, fitted with 256 kB Flash and 64 kB RAM [26]. The AUTOSAR OS included in Arctic Core was used and the application was compiled using the GNU C compiler (gcc) version 4.3.4. An Olimex ARM-USB-OCD was used as the communication link to the board, and for debugging; the GDB Hardware Debugger.

The reported execution times are the measured execution times based on 5000 task executions. The elapsed time is measured using the OS system tick function and the collected measurements were written to the Arctic Core ramlog, which is a defined RAM area for logging. The data from the measurements was then read separately after the test case was completed.

The two performance tests each contain three test cases, A-C as follows:

**A** AUTOSAR generated code using original Arctic Core mechanisms. In this test case, component communication is performed using shared variables without any RTDBMS support in the BSW. This case does not include the run-time monitor (Task 6), since database support is not included.

**B** AUTOSAR generated code with database proxies. In this test case, the database proxies ensure mutual exclusion and atomic access, but performs no data type or access right checking, i.e., the same level of checking as in test case A is used. This approach is useful for static systems where all component communication is known beforehand and type checking and access rights can be validated at design-time. In addition, the run-time monitor component periodically queries the system for shared data.

**C** AUTOSAR generated code with database proxies. This test also includes data type and access right checking at run-time. This approach is useful in more dynamic environments which could include third party applications and communication with the surrounding environment. In addition, the run-time monitor component periodically queries the system for shared data.

### 9.8.2 Test 1: Communication Performance

In this test, the execution times of the individual read and write operations of a single data element (16-bit integer) are measured using the three test cases A-C. Test 1A in Table 9.1 is the benchmark reference to which Tests 1B and 1C are evaluated.

Table 9.1, shows the results of the test. The numbers are shown in average time in nanoseconds (ns) and in the two rightmost columns, difference in percentage and number of CPU clock cycles is shown. From the table it can be seen that a data read (where data are read from the shared variable or RTDBMS and propagated to a component) using database proxies introduces an overhead of 151ns (18 % or 10 CPU cycles) for test case **B**, and 303ns (36 % or 22 CPU cycles) for test case **C**, compared to not using database proxies. For

| | Test 1 A | Test 1 B | Test 1 C | Diff B (ns) | Diff C (ns) | Diff B % / CPU c | Diff C % / CPU c |
|---|---|---|---|---|---|---|---|
| 1R | 828 | 979 | 1131 | 151 | 303 | 18 / 10 | 36 / 22 |
| 1W | 825 | 904 | 1016 | 79 | 191 | 10 / 6 | 23 / 14 |

Table 9.1: Result of test 1

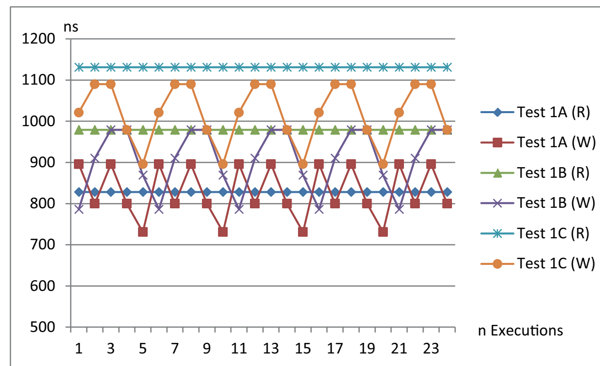



Figure 9.8: Execution times for read and write operations

the data write case (where data is propagated to the RTDBMS from a component) the introduced overhead is 79ns (10 % or 6 CPU cycles) for test case **B**, and 191ns (23 % or 14 CPU cycles) for test case **C**, compared to not using database proxies.

Figure 9.8, presents the execution time for test cases A-C. As the graph shows, the time for reading a value is constant in the three test cases, whereas the write operations have some fluctuations. Since the tasks are periodically executed, these fluctuations could be the result of a probe effect from the time measurement routines, cache misses or a combination of both. In any case, this is not the result of introducing database proxies since identical fluctuations are also present in the original generated code.

The number of executions in the graph is limited for readability. Worth noting is that these numbers are representative for the rest of the executions in the evaluation.

| Monitor func | 1 int (ns) | 2 int (ns) | 3 int (ns) |
|---|---|---|---|
| Soft Read | 20631 | 21431 | 22011 |

Table 9.2: Result of soft read by the run-time monitor

Table 9.2, display the time, in nanoseconds, for reading 1-3 shared data elements using the SQL interface. As seen to the right in the table, 22011ns is the time required to read the output from components 1-3, as seen in Figure 9.7. Worth noting is that the predictability of the hard real-time tasks reading and writing the shared data is not compromised. No fluctuations or increased execution times were observed.

A code analysis showed that the difference in execution time between test case **A** and **B** is mainly caused by that the database proxy pushes two parameters (the value and the database pointer handle) to the stack when performing the read/write compared to in test case **A** where only the value needs to be pushed to the stack. Our analysis show that the extra parameter alone cost 6 of the 10 CPU cycles that differs (see Table 9.1).

### 9.8.3 Test 2: System Performance

In this test, the execution time for each individual task including the component logic as well as the component communication is measured. Test 2A in Table 9.3 is the benchmark reference to which Tests 2B and 2C are evaluated. The soft task (T6) is not included in this table since it cannot be evaluated against the reference implementation which does not include the Run-Time Monitor component. However, in this performance test, T6 is included and monitors the sensor input values throughout the evaluation for test 2B and 2C in order to show that it will not negatively affect hard proxies while providing enlarged support for sharing, tracing, monitoring and logging.

The aim is to measure the impact of introducing database proxies in relation to the execution time of the whole application.

Table 9.3, shows the results of the average execution times. The measurements shows that the overhead of using database proxies under typical workload conditions introduces an overhead of only between 1-10% in test case **B** and slightly higher in test case **C**, with the exception of task T1. In this case the increased overhead of 10% or 17% for case **B-C** can be explained by the fact that the component executed within task T1 is the smallest, therefore in-

| Tasks | Test 2 A | Test 2 B | Test 2 C | Diff B (ns) | Diff C (ns) | Diff B % / CPU c | Diff C % / CPU c |
|---|---|---|---|---|---|---|---|
| T1 | 1271 | 1407 | 1490 | 136 | 219 | 10 / 10 | 17 / 16 |
| T2 | 5671 | 5726 | 5892 | 55 | 221 | 1 / 4 | 4 / 16 |
| T3 | 4719 | 4899 | 5064 | 180 | 345 | 4 / 13 | 7 / 25 |
| T4 | 24191 | 24936 | 26537 | 745 | 2346 | 3 / 53 | 10 / 168 |
| T5 | 16905 | 17760 | 19002 | 855 | 2097 | 5 / 61 | 12 /150 |

Table 9.3: Result of test 2

| Total Execution Time | Test 2 A (ns) | Test 2 B (ns) | Test 2 C (ns) | OH % | OH % |
|---|---|---|---|---|---|
| T1-T5 | 527 57 | 547 28 | 579 85 | 3,74 | 9,91 |

Table 9.4: Total application CPU overhead

troducing data type and access right checking in the communication accounts for a larger relative overhead than in the other tasks. It is worth noting that the introduced overhead in test case **B** and **C** corresponds to as little as 10 and 16 CPU cycles respectively.

So far the focus has been on the overhead for the individual tasks. To get a better overview of the overhead on an application level, the execution time for the whole application is measured and presented in Table 9.4. The table shows that the total application CPU overhead of using database proxies under typical workload conditions in test case **B** is as low as 3.74%, and in test case **C**, 9.91%.

## 9.9 Conclusions and Future Work

In this paper, we propose a technique to integrate predictable database management support in the AUTOSAR basic software layer and the virtual function bus without violating the fundamental principles of the component-based approach of the AUTOSAR standard. To achieve this, the database proxy concept in conjunction with database pointer techniques, adopted by COMET, is used as the communication link on the VFB. This database-centric approach

provides predictable timing guarantees, dynamic access to data, maintained component encapsulation and independence from the data-management strategy. Developers and maintenance personal can now exploit the full potential of using a real-time database and extract any trace information from the component interactions in contrast to the static predefined approach that exists today. Furthermore, the approach provides means for any BSW module to act as a database client using a standard API.

To validate the feasibility of our approach, we have performed a series of execution time tests which shows that the database proxy approach offers a range of added value features for AUTOSAR systems development, maintenance and evolution at a minimal cost with respect to resource consumption.

Our conclusion is that an RTDBMS that implements the concept of database pointers can be successfully integrated into AUTOSAR, without components being aware of it, or jeopardizing system performance. This in turn, greatly simplifies development of soft real-time functions that process large data volumes, e.g., for statistics and logging.

In the future we plan to further extend the support for our approach in the Arctic Core open source tool by integrating modeling and configuration tools. Furthermore, the data-entity approach that provide techniques for visualization of data dependencies and documentation extraction for efficient design-time management of run-time data will be included [27].

# Bibliography

[1] Andreas Hjertström, Dag Nyström, and Mikael Sjödin. Data Management for Component-Based Embedded Real-Time Systems: the Database Proxy Approach. *Journal of Systems and Software*, 85:821–834, April 2012.

[2] 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.

[3] Ivica Crnkovic and Magnus Larsson. *Building Reliable Component-Based Software Systems*. Artech House, 2002.

[4] AUTOSAR Open Systems Architecture. http://www.autosar.org.

[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] ArcCore. Open Source AUTOSAR Solutions, Göteborg Sweden. http://www.arccore.com.

[7] Klaus Grimm. Software Technology in an Automotive Company - Major Challenges. *Software Engineering, International Conference on Software Engineering*, page 498, 2003.

[8] Alexander Pretschner, Manfred Broy, Ingolf H. Kruger, and Thomas Stauner. Software Engineering for Automotive Systems: A Roadmap. *Future of Software Engineering*, pages 55–71, 2007.

[9] Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson, and Massimo Tivoli. The Save Approach to Component-Based Development of Vehicular Systems. *Journal of Systems and Software*, 80(5):655–667, May 2007.

[10] Andreas Hjertström, Dag Nyström, Mikael Nolin, and Rikard Land. Design-Time Management of Run-Time Data in Industrial Embedded Real-Time Systems Development. In *Proceedings of 13th IEEE International Conference on Emerging Technologies and Factory Automation, Germany*, September 2008.

[11] Sandro Schulze, Mario Pukall, Gunter Saake, Tobias Hoppe, and Jana Dittmann. On the Need of Data Management in Automotive Systems. In Johann Christoph Freytag, Thomas Ruf, Wolfgang Lehner, and Gottfried Vossen, editors, *BTW*, volume 144 of *LNI*, pages 217–226. GI, 2009.

[12] R. R. Brooks, S. Sander, J. Deng, and J. Taiber. Automotive System Security: Challenges and State-of-the-Art. In *Proceedings of the 4th annual workshop on Cyber security and information intelligence research: developing strategies to meet the cyber security and information intelligence challenges ahead*. ACM, 2008.

[13] 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.

[14] Manfred Broy. Challenges in Automotive Software Engineering. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 33–42. ACM, 2006.

[15] Gunter Saake, Marko Rosenmuller, Norbert Siegmund, Christian Kästner, and Thomas Leich. Downsizing Data Management for Embedded Systems. *Egyptian Computer Science Journal*, pages 1–13, 2009.

[16] Nelly Delgado, Ann Quiroz Gates, and Steve Roach. A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools. *IEEE Trans. Softw. Eng.*, 30:859–872, December 2004.

[17] Mimer SQL Real-Time Edition, Mimer Information Technology, Uppsala, Sweden. http://www.mimer.se, 2012.

[18] eXtremeDB in-Memory Database, McObject. Issaquah, WA USA. http://www.mcobject.com/extremedbfamily.shtml.

[19] Robert K. Abbott and Hector Garcia-Molina. Scheduling Real-Time Transactions: a Performance Evaluation. *ACM Trans. Database Syst.*, 17:513–560, September 1992.

[20] J. Mellin, J. Hansson, and S. Andler, editors. *Real-Time Database Systems: Issues and Applications*, chapter Refining Timing Constraints of Application in DeeDS. Kluwer Academic Publishers, 1997.

[21] 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, 2004.

[22] 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, 2003.

[23] The Eclipse Foundation, Ottawa, USA. http://www.eclipse.org/.

[24] Peter Pin-Shan Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Trans. Database Syst.*, 1(1), 1976.

[25] K. Åström. The future of PID control. *Control Engineering Practice*, 9(11):1163–1175, November 2001.

[26] ArcCore VK-Board. Open Source AUTOSAR Solutions, Göteborg Sweden. http://arccore.com/wiki/VK-Board.

[27] Andreas Hjertström, Dag Nyström, and Mikael Sjödin. A Data-Entity Approach for Component-Based Real-Time Embedded Systems Development. In *14th IEEE International Conference on Emerging Technology and Factory Automation*, September 2009.

# Chapter 10

# Paper E: Data Management in AUTOSAR: a Tool Suite Extension Approach

Andreas Hjertström, Dag Nyström and Mikael Sjödin
*MRTC report, submitted for conference publication*

**Abstract**

In this paper, we show how a tool for database proxies can be implemented into an industrial AUTOSAR environment. AUTOSAR has been introduced as a remedy for the increasing complexity and rising costs within automotive systems development. However, AUTOSAR does not provide sufficient support for the increased complexity with respect to data management. Database proxies have been presented as a promising solution to provide software component technologies with the capabilities of a state-of-the-art real-time database management system.

## 10.1 Introduction

This paper presents an approach for introducing real-time data management tool support into AUTOSAR development environments. This is made possible due to the usage of database proxies [1] which is a technique to provide run-time data management support to component-based real-time systems using a Real-Time Database Management System (RTDBMS). We also present a tool implementation that extends the commercially available AUTOSAR tool suite Arctic Core [2].

Managing run-time data in complex embedded real-time systems is considered as one of the major challenges for the future development of automotive systems [3, 4]. The automotive industry is continuously evolving and introducing new complex techniques, such as active safety and infotainment systems to improve the competitiveness of their products. This development introduces an increased dependency between vehicle-internal functions, as well as an increased demand for communication with external applications or infrastructure [5]. This increases the demand for open, secure, and flexible access to data. Solutions using a customisable database in an automotive context to manage the security aspects of interconnected systems have been proposed in [6].

It has been reported that the lack of techniques and tools for data management has led to the use of ad hoc techniques and redundant work in reinventing management of data for each Electrical Control Unit (ECU).

Current solutions have reached their limits and are no longer adequate to deal with the future requirements on data, neither at design-time [7] nor at run-time [6, 8].

Database technologies are well established and have proven their usefulness to manage data in complex systems. In our previous work, database proxies have been presented as a promising solution to integrate an RTDBMS into a component-based setting [1]. In addition, the approach has been successfully implemented and evaluated on an AUTOSAR hardware node [9]. Database proxies allow components to communicate through their defined interfaces, remaining unaware of the database, while providing temporally predictable access to data maintained in a database. Database proxies use the state-of-the-art database pointer technique [10], which enables predictable hard real-time access to data along with a flexible SQL-based soft real-time interface. We have evaluated the efficiency of database proxies and shown that it is an affordable technique with very low overheads and large a degree of flexibility [1]; thus we conclude that database proxies can be an attractive technique when trying to improve data management in vehicular software.

In this paper, we present how a database proxy can be integrated into the development of automotive systems using industrial tools. Our approach enables a clear separation of concerns between the system architect, component developer, and the DataBase Administrator (DBA). This separation of concerns allows each part to be managed and reconfigured independent of each other. Furthermore, a plug-in approach, developed for the Arctic Core tool suite and an integration of the Mimer SQL Real-Time [11] database into the basic software of AUTOSAR is presented.

In the remainder of this section, the main tools and technologies used are briefly presented.

### 10.1.1 AUTOSAR

The Automotive Open System Architecture (AUTOSAR) [12] defines a standard component model and middleware platform for automotive electronic architectures. AUTOSAR defines a set of layers to separate the underlying infrastructure from the interconnected software components. AUTOSAR employs the Component-Based Software Engineering (CBSE) approach, where software is encapsulated as components which communicate through well defined interfaces. The communication between system wide components is managed by a Virtual Function Bus (VFB), which acts as a virtual abstraction of the underlying hardware. The realisation of the VFB to a concrete implementation of the final target system is the Run-Time Environment (RTE).

### 10.1.2 ArcCore - Development Environment

ArcCore AB [2] is a provider of the Arctic Core open source AUTOSAR platform developed in Eclipse [13]. Arctic Studio is an Integrated Development Environment (IDE) for Arctic Core, which offers a number of professional graphical tools to facilitate development. Figure 10.1 shows the different tools provided by the Arctic Core tool suite. (1) Components and their interfaces are developed, configured, and generated using the SoftWare Component (SWC) builder tool. (2) The Extract Builder tool is used to add components to an ECU, connect ports and to validate the extract. (3) The Run-Time Environment Builder models the VFB and generates a run-time implementation of the component communication. (4) The configuration of the target platform e.g. operating system, communication, and memory etc., and generation of target c-code is done in the Basic Software Builder tool.

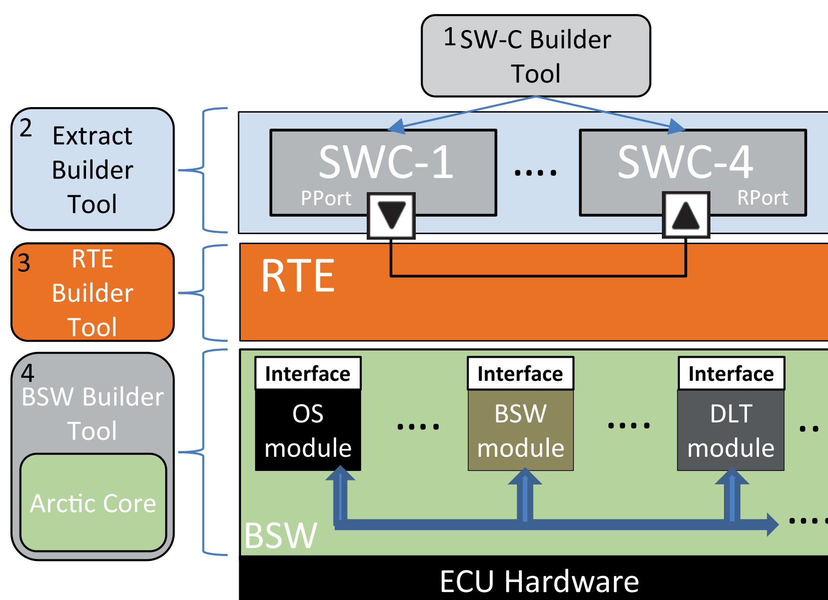



Figure 10.1: Arctic Core tool suite overview

### 10.1.3 Mimer SQL Real-Time

Mimer SQL Real-Time (Mimer RT) [11] is a commercial RTDBMS that allows applications with both hard real-time and non real-time requirements to safely share data without risking real-time predictability. Hard real-time applications use the RTAPI interface to access data using database pointers [10] while non real-time applications use standard SQL interfaces. Mimer RT combines the standard client/server architecture for SQL queries to enable tools and/or applications to access data both locally and remotely, with an embedded library architecture for real-time access.

### 10.1.4 Database Proxies

Database proxies [1] translates data between component ports and an RTDBMS that resides in the component framework (i.e., the AUTOSAR BSW) and vice versa. This allows full decoupling of the RTDBMS from the component, i.e.,

the component and the RTDBMS are unaware of the existence of the other. Database proxies remove the need for database calls within the component, thus preserving component encapsulation and enable component reuse. The schema of the database is modelled and optimised separately and is thereby independent of the component implementation.

The internal mapping of data in the RTDBMS to the database proxies is made using database pointers for hard real-time components, or pre-compiled SQL statements or stored procedures for soft real-time components. A pre-compiled statement enables a developer to compile and bind a certain database query to a statement. This has a decoupling effect since the internal database schema is hidden from the component.

Database proxies are shown to be an efficient and predictable technique that offers a range of valuable features to real-time embedded systems development, maintenance and evolution at a minimal cost with respect to resource consumption [1].

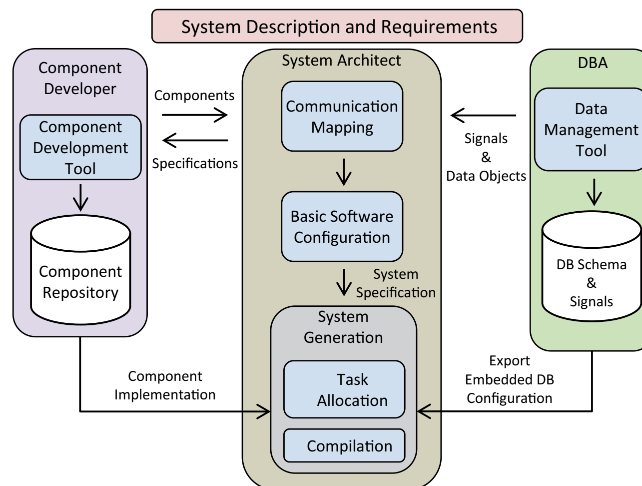



Figure 10.2: System development roles

## 10.2 System Development Roles

To support efficient use of database proxies in a development setting we divide system development into the following three roles, see figure 10.2:

- **The DataBase Administrator (DBA)**, manages the database design, data modelling & optimisation, managing access rights, and generation of the database configuration. Database objects are mapped to signals, which are used by the system architect. With this approach, a complete separation of concerns is achieved, since the database model is separated from the run-time signal. This allows for a reconfiguration of the database without affecting the system configuration. At compile time, an embedded custom made database is synthesised into the final system.
- **The component developer**, provides the system architect with a set of components according to specifications provided by the system architect. The component developer takes no consideration of database connections and calls, since database proxies provide a full separation of concern between these entities. The component implementation details are provided to the system generation and compilation tools.
- **The system architect**, has the overall architectural system view. The system architect assembles the system from components available in the repository or provides the component developer with specifications. The components ports are mapped to signals provided by the DBA. BSW modules such as the operating system, memory, and network communication are configured and components are mapped to tasks. Finally the system is compiled and ready to be deployed.

This approach enables the three roles to be fully decoupled from one another. Thus, the tasks of each role can be performed independently.

## 10.3 Data Management Tool Suite Extension

This section describes the proposed database proxy integration into the Arctic Core tool suite. The tool suite is extended to support the three development roles introduced in section 10.2. Figure 10.3 illustrates the tool suite extended with database management support. In short, a database modelling tool and a data management module plug-in has been introduced in the BSW builder. Since database proxies decouple the RTDBMS from the components, the SWC builder is kept as it is. Finally, the tool suite is extended with a synthesis tool that incorporates the database and database proxies into the run-time system.

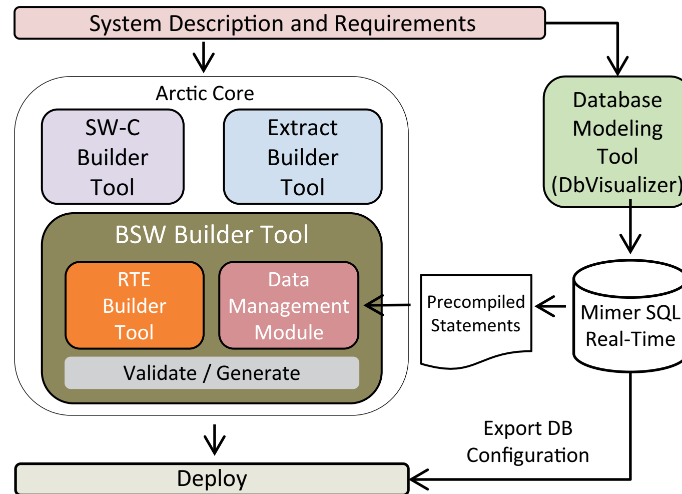



Figure 10.3: Data Management module integration overview

### 10.3.1 The Database Modelling Tool

During the development, the DBA uses a standard database modelling tool (DbVisualizer [14]) to model, optimise and implement the database schema. The DbVisualizer was selected because of its tight integration with the Mimer SQL Real-Time database and its support for precompiled statements. When the database is modelled, the DBA maps database values to signals using precompiled statements which are stored in the database. These signals are later used to map database proxies to objects in the database.

### 10.3.2 The Data Management Module

The Data Management Module, which is implemented as an Arctic Core Eclipse Plug-in, allows the system architect to create database proxies to connect component ports to signals in the database.

The Data management module includes four main configuration settings for each added database proxy, see figure 10.4:

1. A global setting to enable the usage of database proxies and to include the Mimer RT in the BSW.

2. A list of all hard and soft proxies in the system.

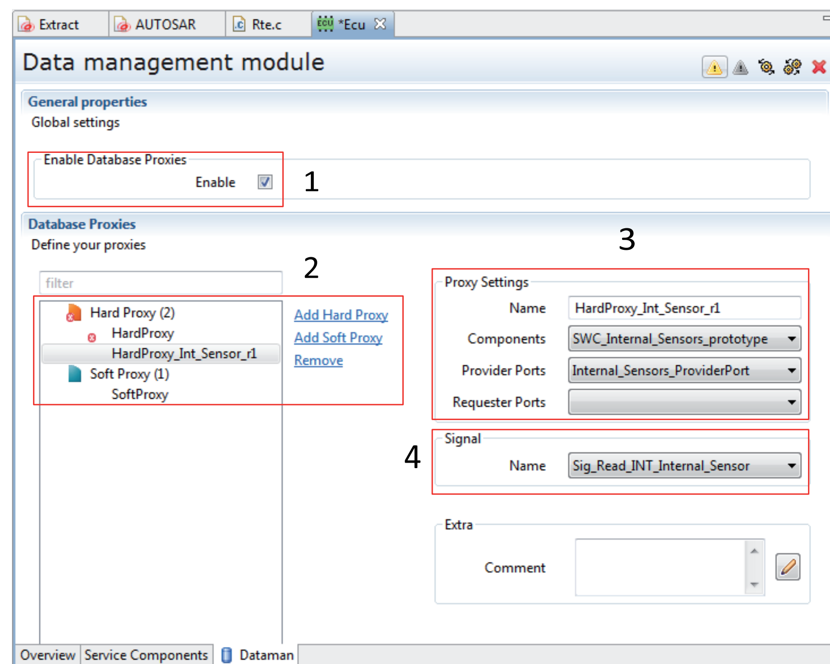



Figure 10.4: The Data Management Module

3. A configuration area where the database proxy is named and assigned to either a provider or requester port of a component.

4. A configuration area where the database proxy is associated with a specific signal in the database. The tool automatically retrieves the available signals names from the database.

### 10.3.3 Synthesis Tool

When the runnable system is generated, a synthesis tool is called that automatically performs the following:

- Extraction of the relevant information from the development database and transformations of this into an embedded database file that is optimised for execution on the target platform.

- Generation of the initialization code for database proxies. This includes the creation of the *proxies.h* and *proxies.c* files which contain the initialization and uninitialisation functions of the database proxies that the RTE invokes during system startup and shutdown [1].

- Substitution of the original ArcCore component communication code with the database proxy component communication code.

Figure 10.5 shows the differences between the original communication code sequence and the code sequence that uses database proxies. In the original code, *Rte_ActuatorRunnable* is called from within the task. Since the component has a requester port, the call to retrieve the data from memory is made before the component itself is called. The *Rte_PRE_ActuatorRunnable* function disables all interrupts, calls an additional function that reads the data from the buffer, and enables all interrupts. After this, the component is called.

```
/*** Original Arctic Core code ***/
void Rte_ActuatorRunnable(){
 Rte_PRE_ActuatorRunnable();
 ActuatorRunnable();
}
void Rte_PRE_ActuatorRunnable(){
DisableAllInterrupts();
Rte_ReadBuffer_Rte_Buf_Actuator_RPort_1_data_1
(&Rte_Inst_Actuator.Act_RPort_1_data_1->value);
EnableAllInterrupts();
}
/*** Database proxy code ***/
void Rte_ActuatorRunnable(){
Rte_PRE_ActuatorRunnable_DBProxy();
 ActuatorRunnable();
}
void Rte_PRE_ActuatorRunnable_DBProxy(){
MimerRTGetInteger(DBP_Actuator, &Rte_Inst_
Actuator.Act_RPort_1_dataElem_1->value);
}
```

Figure 10.5: Example of original code and database proxy code

When using database proxies, *Rte_ActuatorRunnable* calls the *Rte_PRE_ActuatorRunnable_DBProxy* function to read the value from the database using a database pointer. In contrast to the original code, the interrupt disable is not needed within the database proxy, since this is managed by Mimer RT.

Since Mimer RT in addition to the RTAPI provides a standard SQL-based interface, data access is not limited to database proxies. Data can also be made available to external tools and 3rd party applications as well as to other internal BSW modules such as the Data Log and Trace (DLT) module that can utilise the data for diagnostic purposes.

## 10.4 Conclusions and Future Work

This paper has presented an approach for introducing real-time data management in a commercial AUTOSAR development tool suite. This has been made possible with the usage of database proxies which is a technique to provide predictable run-time data management support for component-based real-time embedded systems. Our approach, together with a proposed tool support, enables a clear separation of concerns for the system architect, component developer and the database administrator.

As future work, we plan to complement the tool suite presented in this paper to support automated validation and an augmentation of our existing synthesis tool [1] to fully support the AUTOSAR component model.

# Bibliography

[1] Andreas Hjertström, Dag Nyström, and Mikael Sjödin. Data Management for Component-Based Embedded Real-Time Systems: the Database Proxy Approach. *Journal of Systems and Software*, 85:821–834, April 2012.

[2] ArcCore. Open Source AUTOSAR Solutions, Göteborg Sweden. http://www.arccore.com.

[3] Alexander Pretschner, Manfred Broy, Ingolf H. Kruger, and Thomas Stauner. Software Engineering for Automotive Systems: A Roadmap. *Future of Software Engineering*, pages 55–71, 2007.

[4] Sandro Schulze, Mario Pukall, Gunter Saake, Tobias Hoppe, and Jana Dittmann. On the Need of Data Management in Automotive Systems. In Johann Christoph Freytag, Thomas Ruf, Wolfgang Lehner, and Gottfried Vossen, editors, *BTW*, volume 144 of *LNI*, pages 217–226. GI, 2009.

[5] Marc Zeller Gereon Weiss and Dirk Eilers. *Towards Automotive Embedded Systems with Self-X Properties*. InTech, 2011.

[6] Mario Pukall Gunter Saake Thomas Thüm, Sandro Schulze and Sebastian Günther. Secure and Customizable Data Management for Automotive Systems: A Feasibility Study. *ISRN Software Engineering*, 2012, 2012.

[7] Andreas Hjertström, Dag Nyström, and Mikael Sjödin. A Data-Entity Approach for Component-Based Real-Time Embedded Systems Development. In *14th IEEE International Conference on Emerging Technology and Factory Automation*, September 2009.

[8] M. Broy, I.H. Kruger, A. Pretschner, and C. Salzmann. Engineering Automotive Software. *Proceedings of the IEEE*, 95(2):356 –373, feb. 2007.

[9] Andreas Hjertström, Dag Nyström, and Mikael Sjödin. Introducing Database-Centric Support in AUTOSAR. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES12)*, June 2012.

[10] 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, 2003.

[11] Mimer SQL Real-Time Edition, Mimer Information Technology, Uppsala, Sweden. http://www.mimer.se, 2012.

[12] AUTOSAR Open Systems Architecture. http://www.autosar.org.

[13] The Eclipse Foundation, Ottawa, USA. http://www.eclipse.org/.

[14] DbVisualizer. DbVis Software AB, Stockholm, Sweden. http://www.dbvis.com/.