Mälardalen University Licentiate Thesis
No.114

Information Centric Development of Component-Based Embedded Real-Time Systems

Andreas Hjertström

December 2009

School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden

ISSN 1651-9256
ISBN 978-91-86135-49-2
Printed by Mälardalen University, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

This thesis presents new techniques for data management of run-time data objects in component-based embedded real-time systems. These techniques enable data to be modeled, analyzed and structured to improve data management during development, maintenance and execution.

The evolution of real-time embedded systems has resulted in an increased system complexity beyond what was thought possible just a few years ago. Over the years, new techniques and tools have been developed to manage software and communication complexity. However, as this thesis show, current techniques and tools for data management are not sufficient. Today, development of real-time embedded systems focuses on the function aspects of the system, in most cases disregarding data management.

The lack of proper design-time data management often results in ineffective documentation routines and poor overall system knowledge. Contemporary techniques to manage run-time data do not satisfy demands on flexibility, maintainability and extensibility. Based on an industrial case-study that identifies a number of problems within current data management techniques, both during design-time and run-time, it is clear that data management needs to be incorporated as an integral part of the development of the entire system architecture.

As a remedy to the identified problems, we propose a design-time data entity approach, where the importance of data in the system is elevated to be included in the entire design phase with proper documentation, properties, dependencies and analysis methods to increase the overall system knowledge. Furthermore, to efficiently manage data during run-time, we introduce database proxies to enable the fusion between two existing techniques; Component Based Software Engineering (CBSE) and Real-Time Database Management Systems (RTDBMS). A database proxy allows components to be decoupled from the underlying data management strategy without violating the component encapsulation and communication interface.

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. Kostnaden för att utveckla dessa komplexa system blir allt högre. För att utveckla elektroniksystemet i en modern bil närmar sig kostnaden för mjukvaruutvecklingen idag 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 strukturera bort detaljerad information genom att gruppera funktioner i olika komponenter som kan kommunicera genom ett förutbestämt gränssnitt. Denna teknik kallas för komponentbaserad utveckling. Det finns en mängd olika verktyg och tekniker för att utveckla dessa komponentbaserade system. Dessa tekniker och verktyg fokuserar främst på funktionell strukturering, men är relativt dåliga på att hantera den stora mängd data som utväxlas mellan dessa komponenter både på en designnivå under utvecklingen samt under drift. Här finns ett tydligt glapp.

Denna avhandling introducerar nya koncept för hantering av data både under utveckling, underhåll och drift av inbyggda realtidssystem. Resultaten i denna avhandling baserar sig på en fallstudie som visar att hanteringen av data måste ingå som en integrerad del av utvecklingen av hela systemets arkitektur. För hantering av data på en utvecklingsnivå introducerar vi begreppet "data entity", där vi poängterar vikten av att varje dataelement i systemet ska modelleras och dokumenteras redan i utvecklingsfasen med korrekt dokumentation,

egenskaper och beroenden för att öka den totala kunskapen om systemet. För hantering av data under drift introducerar vi begreppet "database proxy", som syftar till att länka två existerande tekniker, komponentbaserad utveckling och realtidsdatabaser, samman. En databas proxy möjliggör att dessa två tekniker kan samverka utan att bryta mot grundläggande krav inom komponentbaserad utveckling.

To Anna and Felix, you are my everything.

Acknowledgements

To be honest, I did not really know what to expect when I started as a Ph.D student. It felt as I was in need of some expert guidance, and I got it!

The work presented in this thesis would not have been possible without the expert guidance of my supervisors Dr. Dag Nyström and Prof. Mikael Sjödin. Thanks for all the support and fruitful discussions! In addition I'm grateful for the valuable input and guidance from the coauthors Rikard Land and Mikael Åkerholm of the produced papers. Thanks also to Mimer Information Technology for the cooperation and input to the project. A special thanks to Peter Wallin. If you wouldn't have started your Ph.D studies and so warmly recommended it, I would probably have missed this great opportunity.

I would also like to thank Jörgen Lidholm for the good discussions and helping a friend in need. Many people at the department have made this journey more enjoyable, thanks to Stefan Cedergren, Monica Wasell, Fredrik Ekstrand, Lars Asplund, Karl Ingström, Kaj Hänninen and all the other wonderful people.

To the whole Progress gang, Hans Hanson, Tomas Nolte, Ivica Crnkovic, Paul Pettersson, Hüseyin Aysan, Farhang Nemati, Moris Behnam, Mikael Åsberg, Severine Sentilles, Jukka Mäki-Turja, Johan Kraft, Yue Lu, Stefan Bygde, Marcelo Santos, 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 and thesis.

Most important, I thank my loving family, Anna and Felix for all the support and making my life wonderful. I love you. You are my everything!

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

Andreas Hjertström
Västerås, December, 2009

List of Publications

Papers Included in the Licentiate 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, In Proceedings of 13th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'08), IEEE Industrial Electronics Society, Hamburg, Germany. (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 Technology and Factory Automation, Palma de Mallorca, Spain, September, 2009

Paper C: *Database Proxies: A Data Management approach for Component-Based Real-Time Systems*, Andreas Hjertström, Dag Nyström and Mikael Sjödin, Technical report, To be submitted

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

Licentiate Proposal, INCENSE: Information-Centric Development of Component-Based Embedded Real-Time Systems, Andreas Hjertström, Technical report 2008

Contents

I

Thesis

Chapter 1

Introduction

Many of the products we use in our daily life include functionality that are controlled by embedded computers and software. These computer-controlled systems have in the last 30 years become a natural part of our society and account for more than 98% of the total computer systems available on the market today. Furthermore they are in many cases the main way of realizing new and innovative functionality. As an example, vehicular industry are continuously adding new computer-controlled systems, e.g., embedded systems, and replacing existing mechanical parts with electro-mechanical parts to achieve higher safety, less pollution and to add new functionality. In fact, almost 90% of the innovations in a car today is realized by computer software and hardware [1]. Current embedded systems are also evolving from isolated systems to be increasingly dependent on cross-platform communication with other systems. An example of this is Car to Car (C2C) [2] communication. This require flexible handling of data to be shared between various systems.

This evolution is however not without drawbacks. The software and hardware in many systems are becoming increasingly complex. For example, a high-end car can have about 80 Electrical Control Units (ECUs) containing as much as 2000 or more software based functions that communicates trough an excess of 2500 or more signals [1, 3, 4]. I addition, these ECUs are also distributed and communicates via several different kinds of networks.

Demands for short development cycles and time-to-market in combination with the complexity of today's embedded real-time systems require drastically improved development strategies and tool support.

Two strategies that are intended to reduce complexity in embedded systems are Component-Based Software Engineering (CBSE) [5, 6, 7] and Real-Time Database Management Systems (RTDBMS) [8, 9]. Both CBSE and RTDBMS have the common aim to reduce software complexity. However, CBSE target functional complexity whereas RTDBMS target management of system data.

To achieve a higher level of abstraction for software development at design-time, CBSE has been seen as a possible solution in an effort to lower the complexity by dividing software into well defined building blocks. One of the main driving forces within CBSE is to achieve more efficient development by reusing existing components in order to limit the amount of re-implementation and testing, and instead benefit from reusing existing well-tested components. This has been adopted, not only within the vehicular industry [10, 11], but is also widely used in a large range of systems such as home electronics [12]. However, current design-time tools that are used to develop component-based systems are largely focused on the components and does not manage the complexity residing from the large number of data items passed between the components in the system.

Handling large amounts of data is not a unique problem for embedded systems. Several other areas such as banking financial and web based systems, have experienced a similar evolution. A common solution in these cases has been to incorporate a database management system (DBMS) to enable a higher lever of abstraction for data management, similar to what was achieved for software engineering by CBSE.

An RTDBMS target run-time data produced and consumed in real-time systems by providing uniform storage and data access, concurrency-control, temporal consistency, and overload and transaction management [13]. Furthermore, an RTDBMS can offer several additional features compared to traditional data storage using internal data structures. For example, an RTDBMS can allow data to be exploited throughout the system using dynamic run-time access with regular SQL queries, controlling data access and manage coexistence of soft and/or hard real-time data [14].

Even though CBSE and RTDBMS seem to complement each other, combining them is not intuitive since they promote opposing design goals; CBSE promotes encapsulation and decoupling of component internals from the component environment, whilst RTDBMS provide mechanisms for efficient and safe global data sharing.

This thesis investigates how we can adopt an information centric-view for information and data handling, when developing and maintaining component-based embedded real-time systems. The aim has been to develop techniques

to manage data both during design-time and run-time and thereby bridging the gap between component-based software engineering an data management using real-time database systems.

The contribution of this theses includes a case-study that provides valuable information about data management problems that embedded systems developers are facing. Based on these problems we propose a design-time data management approach denoted *data entity*. This approach allow data management to be an integral part of the design environment as an additional architectural view. Furthermore, the approach allow data, based on the system requirements, to be modeled and analyzed in an early phase of the development, even before component implementation. We propose a new technique denoted *database proxies* to enable a fusion between RTDBMS and CBSE without violating the CBSE principles. The usage of a RTDBMS in an component-based framework will in addition introduce a whole new range of possibilities, such as dynamic run-time queries aids in logging, diagnostics and monitoring and controlled access to shard data.

The above concepts has been implemented in a tool called, the *Embedded Data Commander (EDC)*

1.1 Thesis Outline

The outline of this thesis is divided into two parts:
Part I Presents the background and motivation for the thesis as well as related techniques. In chapter 2, some of the related techniques within component-based development, real-time database management systems and available tools is presented. Chapter 3 presents the research method and contributions. Our conclusions and future research direction is presented in chapter 4.
Part II Describes the technical contribution of the thesis in the form of three papers.

1.2 Paper Overview

Paper A 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 (ETFA'08), IEEE Industrial Electronics Society, Hamburg, Germany. (2008)

In this paper, we present the results of an industrial case-study conducted at five companies in which we have studied the t'current state of practice in data management and documentation in embedded real-time systems. The case-study identify that there is lack of design-time data management which often results in costly development and maintenance. Furthermore, inadequate tools and routines for data management of internal ECU data results in costly development and maintenance and is often entirely dependent of the know-how of single individual experts. Ten specific problems are identified, four key observations and six suggested remedies are presented.

My contributions to the results is the design and realization of the case-study, compilation of the results and being main author of the resulting paper.

Paper B A Data-Entity Approach for Component-Based Real-Time Embedded Systems Development. 14th IEEE International Conference on Emerging Technology and Factory Automation, Palma de Mallorca, Spain, September, 2009

This paper presents our design-time data management approach denoted, the *data entity* approach. The approach allow efficient design-time management of run-time data in component-based real-time embedded systems as an additional architectural view that complements the traditional architectural component inter-connections and development view. The *data entity* approach elevate data to be first level citizens of the architectural design, and allow data to be modeled and analyzed in an early phase of the development. The paper also presents a design-time data management tool suite that have been implemented for our approach called Embedded Data Commander (EDC). EDC provides tools for data modeling, visualization and analysis.

My contributions was to define the data entity approach, implement the tool and being main author of the resulting paper.

Paper C Database Proxies: A Data Management approach for Component-Based Real-Time Systems. Technical report, To be submitted

In our run-time data management approach we present the concept of *database proxies*. Database proxies enable a fusion between RTDBMS and CBSE without violating the CBSE principles such as reusability. The *database proxies* acts as a communication link between the application components input ports and output ports, and the RTDBMS. This enable component implementations to be completely decoupled from the database. As a result, the system can fully

benefit from the advantages of component-based software development combined with the advantages of a real-time database management system since the *database proxies* and RTDBMS is a part of the component framework. Furthermore, the glue-code for the *database proxies* and the connection to the RTDBMS is auto-generated by the framework. We implemented the approach and performed an evaluation which shows an insignificant amount of overhead, with respect to execution time and memory consumption.

My contributions was to define the database proxy concept, implement the tool and being main author of the resulting paper.

Chapter 2

Background and Motivation

This chapter will briefly present some 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. Furthermore, we present the background and some of the main challenges when developing and maintaining a data intensive and complex embedded real-time systems such as a vehicular systems. We will also present some of the existing techniques currently targeting these challenges.

2.1 Embedded Systems

An embedded system differs from regular personal computers in many ways. It is typically designed to perform a certain task or small set of tasks by interacting through sensors and actuators. Nowadays, these systems can be found almost everywhere. They are used in watches, vehicles, robots, airplanes or even toothbrushes. Their purpose is most often to reduce mechanical parts, add functionality or to save cost. Embedded systems are often characterized by limited hardware resources such as memory size and processor performance. Traditionally, embedded systems can been characterized as either insolated stand alone devices or a part of a larger interconnected system. However, current and upcoming demands on new functionality and features are now changing embedded systems from being individual systems to be increasingly dependent on cross-platform communication with other systems. An example of such a system is car to car communication [2], which allow cars to interact with each

other to share information such as a possible nearby hazard as well as connection the car to various internet services. This will introduce new requirements on how data and systems is managed within areas such as flexibility, dependability and security.

2.2 Real-Time Systems

A real-time embedded system, has additional requirements to not only perform its task correctly, it also has to perform tasks within a predefined time interval; not to soon and not to late. Many real-time embedded systems interact with the environment where external events are received by sensors. These events are then analyzed and actuated upon, based on the analysis result. 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 made too soon or too late the air-bag could cause the passengers even more harm then 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 results within before a defined deadline. A failure in meeting the deadline can have catastrophic consequences if the system is safety-critical. However, a hard real-time system can also be an engine controller where a missed deadline leads to poor performance and possibly increased pollution. A typical example of a safety-critical hard real-time system is a vehicle air-bag.

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 are, displaying statistical information, controlling power windows, perform logging or to display information. In many applications, a combination of both hard and soft real-time tasks are used.

2.3 Component-Based Software Engineering

In Component-Based Software Engineering (CBSE), the aim is 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 enables them to be reusable entities that can be put together into entire systems. It also introduces

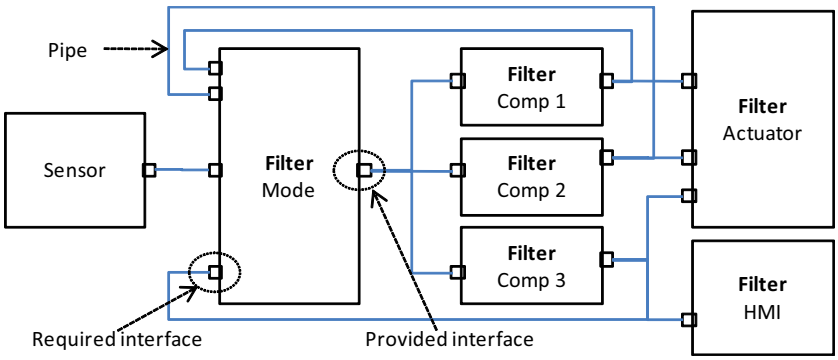

Figure 2.1: CBSE architectural example

a possibility to maintain and improve systems by replacing individual components. In this way a lot of development effort and cost can be saved [15].

Figure 2.1 shows an example of a pipe-and-filter [16] 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 since the interface is treated as a components specification.

A component can have two types of interfaces, required and provided interface. 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 reveal it's internal composition. This enable developers to use the inside functionality and directly change the source code if needed. A black-box component is typically already compiled and does not reveal any internal details.

There is a great verity of component model which are suitable for different types of systems. COM [17], EJB [18] and .NET [19] are typically used for PC applications since they are not sufficiently considering important embedded systems requirements such as timing properties, safety-criticality and the limited amount of resources available. Examples of component models aimed to satisfy the requirements of embedded systems are Rubus [20], SaveCCM [21], Koala [12], ProCom [22] and AUTOSAR [10].

In the following sections we describe SaveCCM and ProCom which are used in paper B and paper C.

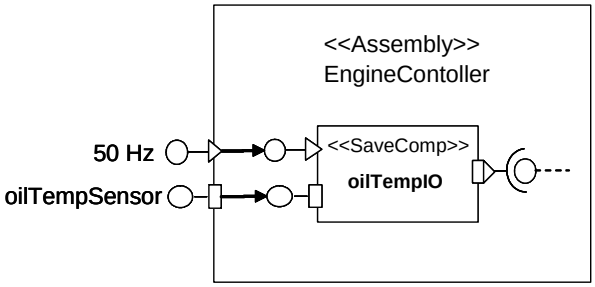

Figure 2.2: Save graphical application design

2.3.1 SaveCCT

The SaveComp Component Technology (SaveCCT) [21] is focused on embedded control software for vehicle systems with an aim to be predictable and analyzable. The applications are built by connecting components input and output ports 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 on input ports to perform its 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 interface and behavior as a primitive component. The difference is that there is only one behavior model or code piece to consider instead of one for each included component.

Figure 2.2 illustrates an example of a SaveCCT graphical representation of a component. There are two inports into the Engine Controller application, one data port and one trigger port. Data is read by the oilTempIO component from the oilTempSensor inport once triggered every 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, automated activities such as task and code generation, integrated analysis tools and an execution model. Developers use an Integrated Development Environment (IDE), a tool supporting graphical composition of components to create applications. A number of tools are also available in the IDE for automated formal analysis of components and

architectures. In SaveIDE, component development, 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.

2.3.2 ProCom

The ProCom component model [22] extends SaveCCT by 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 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 an 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 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.

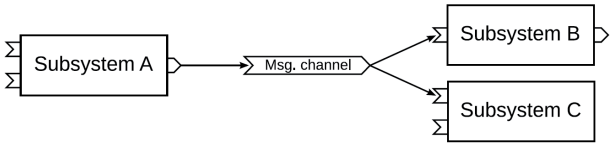

Figure 2.3: ProSys Component Model

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 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.

2.4 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" [23]

All computer systems involve the usage of data in some way. As the amount of data increases as well as the increased usage with different areas, an increase of complexity is often unavoidable. Routines for documentation, storage, retrieval and security of data usually becomes additionally 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 refer to how run-time data is organized during the design and development phase. Run-time data management refers to how data is organized in memory. So far most embedded systems use internal data structures, but database management systems are becoming more and more common in an effort to handle and structure the large amounts of data, data complexity and to provide flexible access.

2.5 Design-Time Data Management

Design-time data management has become increasingly important in order to handle the information complexity in today's system development and maintenance. In addition, the development of systems are often distributed. This often bring on security issues such as, who is allowed to access or alter information. Design-time data management covers many different areas and aims to provide a better overview and understanding of data throughout the whole system life-cycle. This makes proper documentation and project management a crucial part of design-time data management. Proper documentation and structure allows for easy access to information, such as properties that can specify unique naming, type, size and where the data is used. Versioning is another

important aspect in order to have a common view of the data. Many data intensive applications uses a database management system. Database modeling is then an important part of design-time data management. This involves creating a structure that will utilize effective storage, retrieval and proper use of data from the database.

The number of dedicated design-time tools for managing data in embedded systems is quite limited. Most tools focus on the properties for individual data elements and how to create or define new data types. They do however not present an overview or detailed information of how data is used in the system during development. The rest of this section briefly presents two such tools.

2.5.1 dSpace Data Dictionary

dSpace Data Dictionary [24] is a central data container for model-independent data management and holds information about an ECU application for calibration and code generation. The tool can be used to share information to an entire project. An example could be interface variables, their scalings, typedefs, etc., which should be stored globally to remain consistent for all users.

The data dictionary is also used for managing AUTOSAR properties, alongside AUTOSAR specification properties at block level in Targetlink [24]. The input to the dSpace data dictionary is templates generated from Simulink [25]. The data dictionary provides access to information such as specifics on C modules, function calls, tasks, variable classes and data variants. dSpace data dictionary also gives user the opportunity to import and export AUTOSAR SWC XML description files which can be used by other tools. The information included in 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 that are included. It is also possible to specify and produce signal lists and spreadsheets with information regarding data. The development process in this tool is to start modeling components and their structure.

This tool does not focus on managing or visualizing the data flow in the system. Neither does it include analysis techniques to see data dependencies.

2.5.2 Visu-IT Automotive Data Dictionary

Automotive Data Dictionary (ADD), is a repository solution to centralize data declarations and ensure label/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 to avoid inconsistency throughout the whole development process. It gives the developers a view of the data specification but does not include any implementation details [26].

ADD mostly focuses on requirements engineering and unique labeling and does nor cover information about data flow and data dependencies.

2.6 Run-Time Data Management

Run-time data management concerns how data is managed during execution of the system. So far, most embedded systems handle data in an ad hoc, traditionally using internal data structures. A more high-level approach for run-time data is to use a database management system.

2.7 Data Management System

A Database Management Systems (DBMS) is used to organize large amounts of data. Figure 2.4 shows an high level picture of a DBMS system. The DBMS is an interface to the physical data stored in memory. A typical application has so far been large enterprise systems such as libraries, commercial websites and banking. Examples of enterprise mainstream DBMS are Oracle [27], Microsoft Access [28] and MySQL [29].

The main purpose of a DBMS is to provide a number of software programs to organize data. Standard Query Language (SQL) [30] is one of the most common language for uniform data access. SQL enables high level tools to request desired information from large amount of data. A DBMS have several important parts which includes a query language, optimized data structures and mechanisms for various transactions.

To ensure a correct behavior and safe sharing of data, the database should conform to the ACID properties [31]:

- Atomicity, either all information in a database transaction is updated or none at all.
- Consistency, after a transaction is completed the database will be in a valid state. If not, the transaction must be rolled back.
- Isolation, changes that are made to the database will not be revealed to other users until the transaction is committed.
- Durability, any change to the database is permanent. The result of a committed transaction can not be reverted.

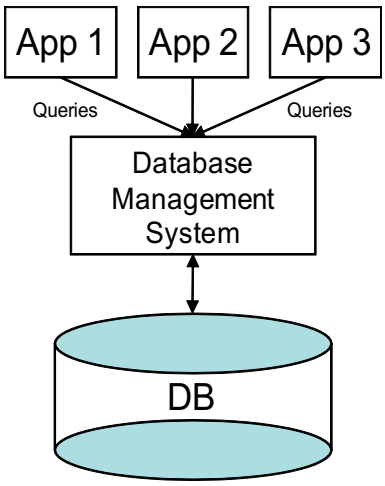

Figure 2.4: DBMS overview

Most DBMS's use concurrency control to handle concurrent operations, avoid transaction conflicts to achieve logical correctness. The most commonly used algorithm is Two-Phase-Locking (2PL) [32].

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 DataBase Management Systems (DBMS) available that have been developed to suite the specific needs of embedded systems, such as small footprint in mind [8, 9, 33].

2.8 Real-Time Database Management Systems

DBMS has evolved to also support real-time embedded systems using Real-Time DataBase Management System (RTDBMS). Embedded real-time systems have different requirements compared to large enterprise systems. CPU usage, footprint and availability are highly important. For safety-critical embedded real-time systems, predictable access to data is one of the most important features required of the database [34]. Compared to the concurrency control algorithms used in a DBMS, a RTDBMS most not only enforce seri-

alization, but also apply to the real-time timing constraints such as deadline. Because of this other types of concurrency algorithms such as 2V-DBP [14] are used.

Below we present some of the commercial and research RTDBMS that are available.

2.8.1 Mimer Real-Time Edition

Mimer Real-Time Edition (Mimer RT) is a commercial real-time database management system (RTDBMS) intended for applications such as vehicle systems, process automation and telecommunication systems. Mimer RT supports applications with both hard and soft real-time requirements without jeopardizing database consistency using the 2V-DBP concurrency algorithm [14] for hard transactions. The algorithm allows soft and hard transactions to share data independent of each other. This is achieved by using two different user interfaces to make soft and hard transactions coexist without compromising real-time properties of the hard transactions. A query from soft transaction uses a 2PL-HP protocol [35] and can be done at any time with regular SQL query. This differs from the hard real-time database pointers since the pointer is bounded to a specific data element during the initialization of the system.

2.8.2 DeeDS

DeeDS [36] is a distributed main-memory Real-Time database developed at Skövde University, Sweden. DeeDS is built for the Enea OSE real-time operating system [37] and supports real-time database systems with soft and hard deadlines. To support soft and hard deadlines, DeeDS uses a dedicated service processor to execute hard transactions separate from the application functions.

In a distributed setting, each node are locally consistent. However, the system view at several nodes might be inconsistent. This implies that critical data has to be stored on a local node, whereas only less critical data can be distributed.

2.8.3 ARTS-RTDB

Carnegie Mellon University, Pittsburgh, has developed a distributed relational database, that supports both hard and soft real-time tasks, for the ARTS real-time operating system [38], the ARTS-Real-Time DataBase (RTDB) [39].

ARTS-RTDB have chosen to optimize the most commonly used data access operations, SELECT, INSERT, UPDATE and DELETE. To avoid costly roll-back operations, two phase locking with high priority abort (2PL-HP) [35] is used. To manage the distribution, a file is used as a shared resource between the different nodes. ARTS-RTDB also utilizes worker threads to periodically do backups in main memory.

Chapter 3

Research Method and Contributions

The aim of this research project is to improve current data management for industrial companies, during development and maintenance of embedded real-time systems.

3.1 Research Method

Current research has pointed out that the design-time and run-time data complexness in today's industrial and vehicular embedded systems as well as in future embedded systems is reaching a point where current tools and techniques are no longer sufficient [1, 4, 40, 41, 42, 43]. CBSE is increasingly used within embedded systems development and seen as one solution. However, CBSE does not target design-time and run-time data management. The focus is rather towards encapsulating functionality and to achieve a higher level of abstraction.

Studies has shown that an RTDBMS can be incorporated to manage run-time data in complex embedded systems [34, 40]. However, the usage of a RTDBMS in a component-based development setting in order to achieve more structured data management of the data flow between components is not covered.

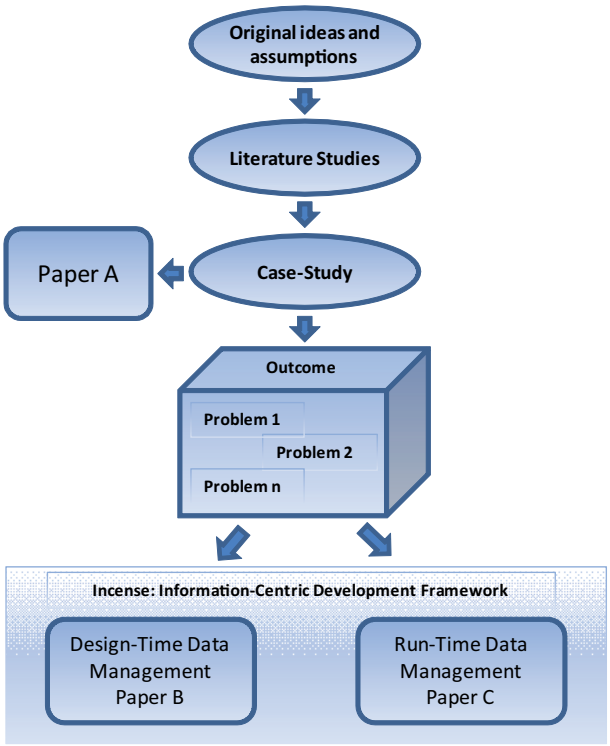

Figure 3.1: Research Overview

Figure 3.1 shows an overview of our research flow. From the initial ideas stated above, we continued the research with literature studies. These studies confirmed that the current status within data management in these systems indeed is becoming an increasing challenge for developers and system architects. To get additional support for our research, we conducted a case-study with five different companies within the industrial and vehicular domain of real-time embedded systems. This study identifies a number problems regarding design-time data management.

The result of this case-study is published in paper A. The continued research was divided in two parts to form papers B and C, as seen in figure 3.1, to form the *Incense: Information-Centric Development Framework*.

3.2 Contributions

In this section we present the main scientific contributions of this thesis.
Case-Study

1. We indicate that current tools and methods for data management during design-time and run-time are not adequate.
2. We conclude that the importance of data management needs to be substantially elevated in order to increase the knowledge and understanding of the system.
3. We identify ten problems within documentation, tool support and routines.
4. We propose six remedies to address these problems.

Data Entity

1. We present the concept of the *data entity* that enables design-time modeling, management, documentation and analysis of run-time data.
2. We propose that run-time data should be acknowledged as first class objects that can be modeled, analyzed, and where data dependencies can viewed during the whole development phase.
3. We present a proof of concept implementation data management tool, the Data Entity Navigator (DEN).

Database Proxy

1. We present a technique to enable a fusion between component-based software engineering and a real-time database management system.
2. We introduce the concept of *database proxies* to decouple components from the underlying database.
3. We have implemented a framework as proof of concept where a system can be designed with or without a database, where database proxy properties are generated from its specifications to glue code and further to executable C-code.
4. We evaluate the approach which indicate that the execution time overhead and additional memory overhead is in order of 1-2%.

Chapter 4

Conclusions and Future Research Directions

4.1 Conclusions

This research steams from the rapidly growing complexity with respect to the amount of data and data flow between components in today's embedded real-time systems. This is not addressed by contemporary development techniques, since they are mostly focusing on achieving a higher level of abstraction by encapsulating functionality.

Current tools and techniques for managing data are mostly focusing on distributed data and creating libraries to define and manage new data types. However, our research has shown that current state of practice for managing internal ECU data is not adequate. There is an increasing need for tools and techniques that manage data at both design-time and run-time.

The result of this thesis is a set of new tools and techniques to enhance current and future data management-strategies during design-time and run-time by adopting an information-centric approach.

We have introduced a new design-time approach, the *data entity* approach, that elevates run-time data to become a first class citizen in the system architectural design as a data architectural view. The approach allows data to be documented, modeled and analyzed separately from the actual component implementation.

Similar to what has been adopted by several other areas that are data intensive, with high demands on flexibility and structured data management, we propose to use a database management system. However, the usage of a real-time database management system, in conjunction with component-based development is not obvious since the design goals of component-based software engineering and real-time database management systems are contradicting. To overcome these contradictions we have introduced the concept of *database proxies* which enable a successful fusion between real-time database management system and component-based software engineering. We have furthermore showed that this fusion introduces a number of new possibilities for components-based development at a minimum cost with respect to executions time and memory overhead.

From our point of view, the introduction of new data management tools and techniques is inevitable in order to meet the needs of component-based real-time embedded systems development of today and tomorrow.

4.2 Future Research Directions

Based on the results presented in this thesis, a number of new research directions are opened.

Paper B presents the *data entity* concept. The data-entity approach provides designers with an additional architectural view which allows for graphical modeling of data, visualization of dependencies, properties, documentation etc. However, the graphical visualization implementation which enables developers to get an overview of the data architecture, similar as the architectural overview of interconnected components, has not been completed. Furthermore, we aim to extend the analysis capabilities of our tool to include formal end-to-end and relative timing validity analysis for producing and consuming components [44]. In addition to this, we would like to perform an industrial evaluation to validate our approach.

Paper C presents the concept of database proxies that enable a fusion between Component-Based Software Engineering and Real-Time Database Management Systems. The technique does however not consider composite components. Information that a component within a composite component utilizes a database is not reviled in the interface of the composite component. Additional research on how to transfer knowledge about the existence of a database in-

side a composite component to its interface as well as possible usage without a database is needed.

We would also like to extend *soft database proxies* to support additional SQL data manipulation such as INSERT operations. Further evaluation and analysis on an industrial application would also be interesting.

Additional research directions To reach a more extensive usage of our *database proxy* approach further research on inter-process communication [45] and distributed queries [46] is necessary in order to, for instance maintenance and service tools to access different parts of the system.

An additional aim is to perform a case-study with several people developing a system, with or without our approach, to further evaluate the impact in an industrial setting.

Bibliography

[1] M. Broy. Automotive Software and Systems Engineering. In *MEMOCODE '05: Proceedings of the 2nd ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, pages 143–149, Washington, DC, USA, 2005. IEEE Computer Society.

[2] CAR 2 CAR Communication Consortium. http://www.car-to-car.org/.

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

[4] Stefan Voget. Future Trends in Software Architectures for Automotive Systems. *Advanced Microsystems for Automotive Applications*, 2003.

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

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

[7] George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together (ACM Press)*. Addison-Wesley Professional, June 2001.

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

[9] eXtremeDB, McObject. Issaquah, WA USA. http://www.mcobject.com/.

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

[11] Arcticus Systems. http://www.arcticus.se.

[12] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics Software. *IEEE Computer Society*, 33(3):78–85, Mar 2000.

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

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

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

[16] 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 & Sons, 1996.

[17] Dale Rogerson. Inside com. *Microsoft Press*, 1997.

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

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

[20] 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.

[21] 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.

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

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

[24] dSPACE Tools. http://www.dspaceinc.com.

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

[26] Visu-IT. http://www.visu-it.de/ADD/.

[27] ORACLE. http://www.oracle.com.

[28] Access, Microsoft. http://www.microsoft.com/.

[29] MySQL, Sun Microsystems. http://www.mysql.com.

[30] ISO SQL 2008 standard. *Defines the SQL language*, 2009.

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

[32] 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.

[33] Enea Data, Polyhedra. http://www.enea.com/polyhedra.

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

[35] R.K Abbott and H. Garcia-Molina. Scheduling Real-time Transactions: A Performance Evaluation. *ACM Transactions on Database Systems*, 17, September 1992.

[36] 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.

[37] ENEA Data. OSE Real-Time System. http://www.enea.se.

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

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

[40] Sandro Schulze and Mario Pukall and Gunter Saake and 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.

[41] Manfred Broy. Challenges in Automotive Software Engineering. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 33–42, New York, NY, USA, 2006. ACM.

[42] Alexander Pretschner, Christian Salzmann, and Thomas Stauner. 2nd intl. icse workshop on software engineering for automotive systems. *SIGSOFT Softw. Eng. Notes*, 30(4):1–2, 2005.

[43] 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.

[44] 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.

[45] Mentor Graphics. http://www.mentor.com/products/vnd/.

[46] Thomas Nolte and Dag Nyström. Introducing Substitution-Queries in Distributed Real-Time Database Management Systems. In *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*. IEEE Computer Society Press, September 2005.

II

Included Papers

Chapter 5

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
In Proceedings of 13th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'08), IEEE Industrial Electronics Society, Hamburg, Germany. (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 casestudy 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.

5.1 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 2 describes our research method and the five participating companies. Section 3 reports the state-of-practice in data management and documentation of industrial embedded real-time systems. In section 4 we present four key observations and ten identified problem areas in current practice. In section 5 we propose six remedies and future research directions. In section 6 and 7 we conclude the paper and suggest future work based on the findings in this case-study.

5.2 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.

5.2.1 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.

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

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

Figure 5.1 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.

5.3 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.

5.3.1 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.

5.3.2 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.

5.4 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.

5.4.1 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 5.1 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 sufficient. 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.

5.4.2 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.

5.5 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.

Introducing these remedies and forming a uniform development environment give developers the prerequisite needed for effectively managing their system development and maintenance. Figure 5.2 illustrates how the problems are linked with the proposed remedies.

	Remedies					
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 5.2: Problem areas with associated remedy or remedies.

5.6 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.

5.7 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. 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.

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

[9] Dag Nyström, Aleksandra Tesanovic, Christer Norström, Jörgen Hansson, and Nils-Erik Bånkestad. Data management issues in vehicle control

systems: a case study. In *Euromicro Real-Time Conference 2002*, 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. 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.se.

[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 6

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 Technology and Factory Automation, 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.

6.1 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 6.3, a definition of the data

entity is presented and the data entity approach is discussed in section 6.4. Further, in section 6.5, we describe the ProCom component-model which is used in our data entity tool-suite, presented in section 7.5.3. An example of how the data entity tool-suite can be used is presented in section 6.7. Finally, the paper is concluded in section 6.8.

6.2 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.

6.2.1 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.

6.2.2 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.

6.3 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.

6.3.1 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.

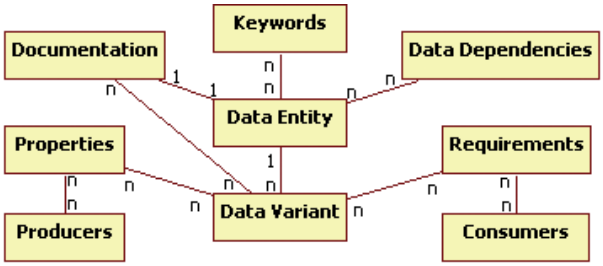

Figure 6.1: Data Entity Description Model

A data entity consists of the following metadata (illustrated in Figure 6.1):

- **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.

6.3.2 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.

6.4 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.

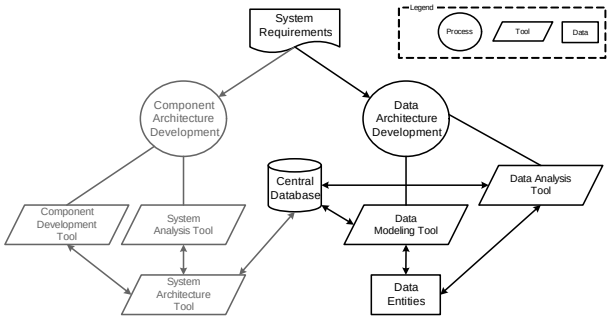

Figure 6.2: The data entity approach

Figure 6.2 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.

6.5 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.

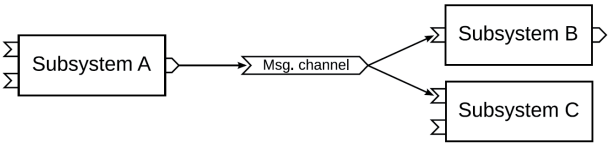

Figure 6.3: ProSys Component Model

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 6.3 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.

6.6 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.

Figure 6.4: DCC data model description

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.
- 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 6.4), 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.

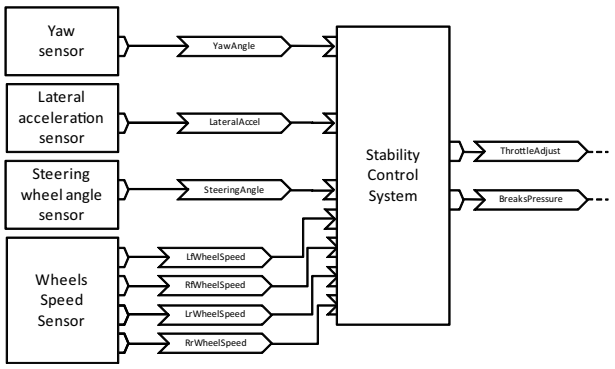

Figure 6.5: Existing ProSys Stability Control System application

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.

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.

6.7 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.

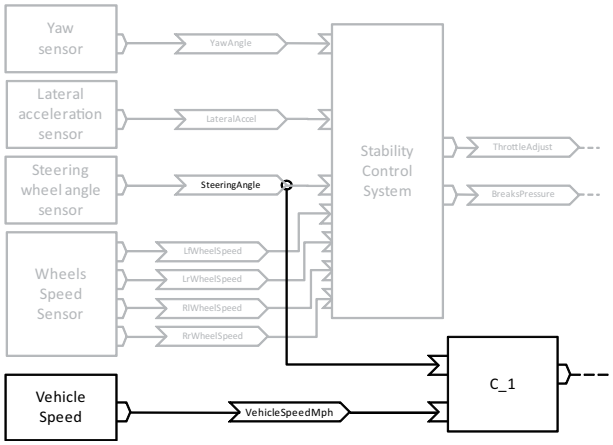

Figure 6.6: Extended ProSys example

6.7.1 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 6.5. 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:

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

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

Figure 6.7: Producer properties.

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 6.6. 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 6.6, 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.

6.7.2 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 6.7.

- **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 *VehicleSpeedMph* 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.

6.8 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 6.2.1 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 (ETFA'08), IEEE Industrial Electronics Society, Hamburg, Germany*, September 2008.

[2] A. Albert. Comparison of Event-Triggered and Time-Triggered Concepts with Regard to Distributed Control Systems. 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 *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.

[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, Séverine 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 Tools. http://www.dspaceinc.com.

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

[17] Visu-IT. 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] Séverine 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 7

Paper C: Database Proxies: A Data Management approach for Component-Based Real-Time Systems

Andreas Hjertström, Dag Nyström and Mikael Sjödin
Technical report

Abstract

We present a novel concept, database proxies, which enable the fusion of two disjoint productivity-enhancement techniques; Component Based Software Engineering (CBSE) and Real-Time Database Management Systems (RTDBMS). This fusion is neither obvious nor intuitive since CBSE and RTDBMS promotes opposing design goals; CBSE promotes encapsulation and decoupling of component internals from the component environment, whilst RTDBMS provide mechanisms for efficient and safe global data sharing. Database proxies decouple components from an underlying database thus retaining encapsulation and component reuse, while providing temporally predictable access to data maintained in database. We specifically target embedded systems with a subset of functionality with real-time requirements, and the results from our implementations shows that the run-time overhead from introducing database proxies is negligible and that timing predictability does not suffer from the introduction of an RTDBMS in a component framework.

7.1 Introduction

To enable a successful integration of a Real-Time DataBase Management System (RTDBMS) [1, 2, 3, 4] into a Component-Based Software Engineering (CBSE) [5, 6] framework we present a new concept, *database proxies*. A database proxy allows system developers to employ the full potential of both CBSE and RTDBMS, which aim both to reduce complexity and enhance productivity when developing embedded real-time systems.

Although both CBSE and RTDBMS aims to reduce complexity, a fusion between them is not trivial since their design goals are contradicting. RTDBMS promotes techniques, such as a common blackboard storage architecture to share global data safely and efficiently by providing concurrency-control, temporal consistency, and overload and transaction management. Furthermore the typical interface provided by the RTDBMS opens up for a whole new range of possibilities, much needed by industry, such as dynamic run-time queries which could be a welcome contribution to aid in logging, diagnostics, monitoring [7], compared to the pre defined static data access often used by developers today.

CBSE promotes encapsulation of functionality into reusable software entities that communicates through well defined interfaces and that can be mounted together as building blocks. This enable a more efficient and structured development where available components can be reused or COTS components effectively can be integrated in the system to save cost and increase quality.

The techniques offered by an RTDBMS allow the internal structure of the RTDBMS to be decoupled from its users. However, using these techniques in a component-based system implies calling the database from within the component code, thereby introducing unwanted side-effects such as awareness that the database exist, severely influencing component reusability. A component with direct access to the database from within, introduces side-effects thereby violating the components aim to be encapsulated. Furthermore, if the RTDBMS is used inside the component, the component cannot be used without a database.

To overcome these problems we propose the concept of database proxies which decouple the database from the component, and instead uses the database as a part of the component framework.

As illustrated in figure 7.1, a database proxy is part of the component framework, thus external to the component. The task of the database proxy is to manage access to the RTDBMS to make it possible for components to interact with a database through its interface as the coupling is embedded in the

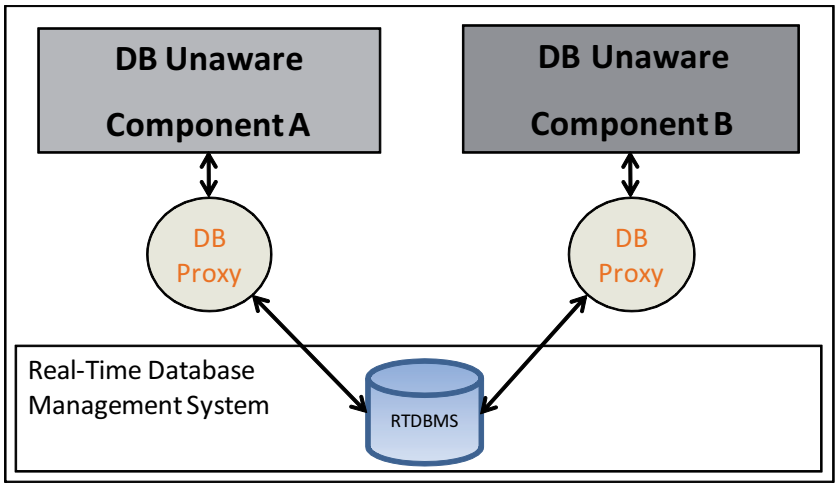

Figure 7.1: Database Proxies Connecting Components to an RTDBMS

underlying glue code. By decoupling components from the database, and placing the database in the component framework, the decision to use a database or not is moved from the component design to the system design.

The tools and techniques in this paper 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 [8, 9].
The results and main contributions of this paper include:

1. A framework where components and data is reliably managed and structured to enable flexibility and reusability.

2. A system where soft and hard real-time tasks can execute and keep isolation properties.

3. A system that can handle critical transactions and at the same time enable openness to run-time queries.

4. A system where new functionality can be added or removed without side effects to the system.

The rest of this paper is structured as follows; in section 7.2, we present background and motivation for the approach. We also present the specific problems that our approach addresses. In section 7.3, we present the system model. Section 7.4 gives a detailed description of the database proxy. Further, in section 7.5, we illustrate our ideas with an implementation example. Finally, we show a performance and real-time predictability evaluation in section 7.6 and concluded the paper in section 7.7.

7.2 Background and Motivation

The characteristics of today's embedded systems are changing. Embedded systems are in many cases not isolated to a single system, or a small set of systems. Many embedded systems are increasingly dependent on cross-platform communication with other systems. An example of this could be so called Car-to-Car (C2C) [10] communication. This increases the need for flexible, reliable and secure data management since nodes in different systems will interact with each other by accessing and updating various data items. This implies that a database with a well proven standardized interface such as Open Database Connectivity (ODBC) would provide an attractive solution [7]. In addition, this will require a clear separation between safety-critical and non-critical access to data to preserve safety requirements.

A strong trend in embedded-system's development is CBSE. However, to achieve a successful integration of RTDBMS into the CBSE framework, a number of CBSE requirement has to be fulfilled. In CBSE, a component encapsulates a function or a set of functions and only reveals its interfaces to specify the services that it will provide or require.

A component which communicates with a database outside its revealed interface, i.e directly from within the component-code, introduce a number of unwanted properties such as hiden dependencies and limited reusability. We define such a component to be *database aware*.

To fully utilize the benefits of CBSE, a component should be able to interact with a database without any knowledge of the database schema, i.e., the structure of the data in the database. We define such a component to be *database unaware*. A database unaware component has no notion of the un-

derlying database and its structure, neither if a database is used or not. Furthermore, a database unaware component introduces no side-effects such as database communication outside the components specified interface, thus retaining components reusability.

The usage of an RTDBMS in a CBSE framework may not introduce any side-effects that violates key CBSE architectural principles [11, 5]. For the purpose of this paper, we define a component to be side-effect free if it is:

- **Reusable:** A component has to be reusable without any direct dependencies to the surrounding environment.
- **Substitutable:** A component should be substitutable with another component if the new component meets the original interface requirements.
- **Without implicit dependencies:** A component may not have any data dependencies other then the dependencies expressed in its interface.
- **Using only interface communication:** A component may only communicate trough its interface.

7.2.1 RTDBMS Access Mechanisms

There are several existing RTDBMS mechanisms that could be used in order to reach some of above stated principles by decoupling the internal structure of the RTDBMS from its users. However, these mechanisms are placed directly in the component code. This implies that there exists a dependency between the component and the RTDBMS.
Available RTDBMS mechanisms [12];

- **Pre-compiled statements**, enables a developer to bind a certain database query to a statement at design-time. The statement is compiled once instead of a sending it to the database and compiling it for each use. 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**, holds a stored predefined query that can represent a subset of the data contained in one or several tables which can be accessed as a virtual table. Views has similar decoupling effect as pre-compiled statements, but in this case the name represent a simplified view of several tables.

- **Stored Procedures**, decouples logical functions that is moved into the database, hidden from the user. Several SQL statements can be executed within the database using declared variables and loop trough trough tables using, IF or WHILE statements etc. declared in the SQL language. However only result sets can be returned for additional processing.
- **Functions**, is a subprogram, similar to a stored procedure, the logical functions, the internal database schema is decoupled from the user. Functions perform a desired task and returns a single value.

These mechanisms seemingly provide means of decoupling a component from the RTDBMS, however none of the above stated RTDBMS mechanisms are sufficient to use in a component-based setting, since;

1. The database access is side-effect not visible in the component's interface.
2. The component are bound to always use a database.
3. The component is not fully decoupled from the database. The database name, specific login details and connection information etc. resides in the component. The component is therefore no longer generic nor reusable.
4. The requirements expressed by the components interface does not reflect the database dependency, the component are therefore no longer substitutable.

An additional implication of using the existing RTDBMS mechanisms is that since the component will be dependent on a database, the component will determine whether a database should be used or not. As a result of this, the decision to use a database or not is made on the component-design level, instead of the system-design level. Usage of these mechanisms will also introduce hidden dependencies since the communication is outside the components interface. Individual components should not introduce side-effects or dictate the overall system architecture.

7.2.2 System Requirements

In this section we list a number of requirements that has to be fulfilled to enable the introduction of an RTDBMS into a component-based application without violating the fundamental aim of CBSE. The usage of an RTDBMS should be

seen as an additional design feature for systems where data management using internal data structures are not sufficient.

R1 The decision to use an RTDBMS or not should be made on system, application or product level.
R2 The usage of an RTDBMS should not introduce any side-effects to the components or system.
R3 A component should be possible to use both with or without an RTDBMS.
R4 The real-time properties should not be compromised.

7.3 System Model

The tools and techniques in this paper 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 applications involve both hard 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).

We consider a system where functionality are divided into the following classes of tasks:
Hard real-time tasks, typically executed at high frequency to read or write values from sensors or component output ports to memory or database. When a database is used, hard real-time tasks require a predictable access to data elements.
Soft real-time tasks, often running at a lower frequency controlling less critical functions such as presenting statistical information, logging or used as a gateway for service access to the system by technicians to perform system updates, fault management or if the system permits, perform ad-hoc queries at run-time. These tasks puts high demands on system flexibility and standard interfaces.

7.3.1 Real-Time Database Architecture

In order to support a predictable mix of both hard and soft real-time transactions, we consider a database with two separate interfaces. Figure 7.2 illustrates an RTDBMS which has a soft interface that utilize a regular SQL [13] query interface to enable flexible access from soft real-time tasks. For hard real-time

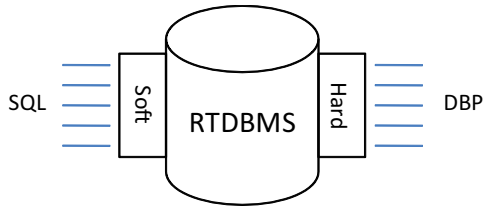

Figure 7.2: RTDBMS Architectural Overview

transactions, a database pointer (dbp) [14] interface is used to enable the application to access individual data elements in the database similar as a shared variable. This approach enable us to share data between hard and soft real-time tasks. To achieve database consistency without jeopardizing the real-time requirements the 2V-DBP concurrency control algorithm [14] is used. 2V-DBP allows hard and soft transactions to share data independent of each other.

Figure 7.3 shows an example of a database aware I/O task that periodically reads a sensor and propagates the sensor value to the database using a database pointer, in this case the oil temperature in the engine relation. The task consists of two parts, an initialization part (lines 2 to 4) executed when the system is starting up, and a periodic part (lines 5 to 8) scanning the sensor.

```
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 7.3: A database aware I/O task that uses a database pointer

The initialization of the database pointer is first done by declaring the database pointer (line 3) and then binding it to the data element containing

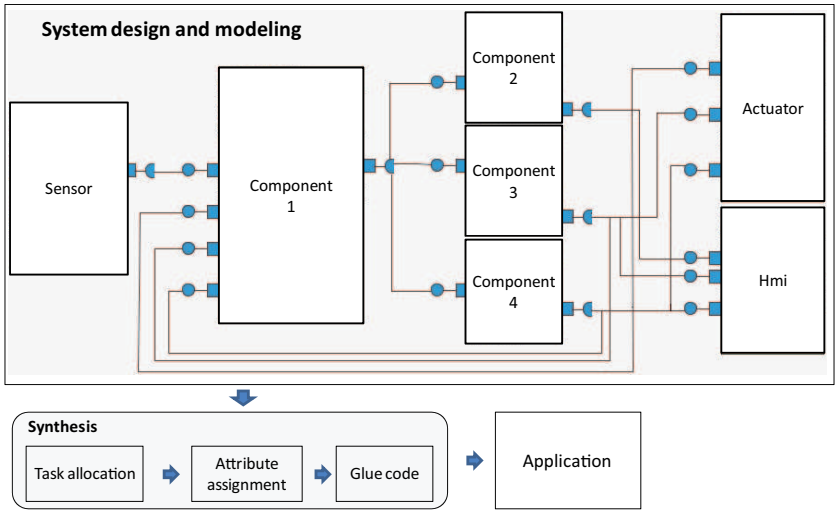

Figure 7.4: System Design and Modeling

the oil temperature in the engine (line 4). When the initialization is completed, the task begins to periodically read the value of the sensor (line 6), then propagates the value to the RTDBMS using the database pointer (line 7), and finally awaits the next invocation of the task (line 8).

7.3.2 System Design and Modeling

In application design and modeling we assume a pipe-and-filter [6] 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 7.4 shows an example of a system design and modeling architecture for CBSE. A set of components are connected trough ports and connections to form the system. From the modeled system, the low level code is generated to tasks, attributes and glue code to the application.

7.3.3 Extended System Design and Modeling

We complement the classical architectural view, presented in section 7.3.2, with a new additional view, the *CBSE database-centric view*. This new view

visualizes the component ports that are connected to data elements in an RTDBMS, illustrated in figure 7.5. 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 however be transformed at any time to reveal the data flow through the connections such as shown in 7.4.

This is similar to an *off-page connector* that is used when designing electrical schemas for embedded systems which could involve a large number of components and connections. A connection ends in a symbol or a an identification name that is displayed at each producer and consumer. To display all connections in a complex schematic diagram would make the electrical schema impossible to read.

During the design of the system the system architect or developer can utilize both traditional data passing through connections or via an RTDBMS providing a black-board data management architecture. An RTDBMS can be used as the single source of memory management or a mix of both internal data structures and an RTDBMS when additional flexibility 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 display information on an HMI. However, if two components share a single data item that are of no additional interest, it is probably not necessary to map that item to the RTDBMS.

7.4 Database Proxy

To succeed in combining CBSE and RTDBMS, we introduce an architectural framework object, the database proxy which acts as a communication link between the application components and the RTDBMS as seen in figure 7.1. The database proxy and communication interface to the database is embedded in the glue code between component calls and connects to a components input or output port. This creates an interface which matches the interface of the component. As a result, the system can fully benefit from the advantages of component-based software development combined with the advantages of a real-time database management system since the components are decoupled from a specific database engine or database schema. The database proxy interface descriptions should be automatically generated, based on the system design description and the appropriate data model.

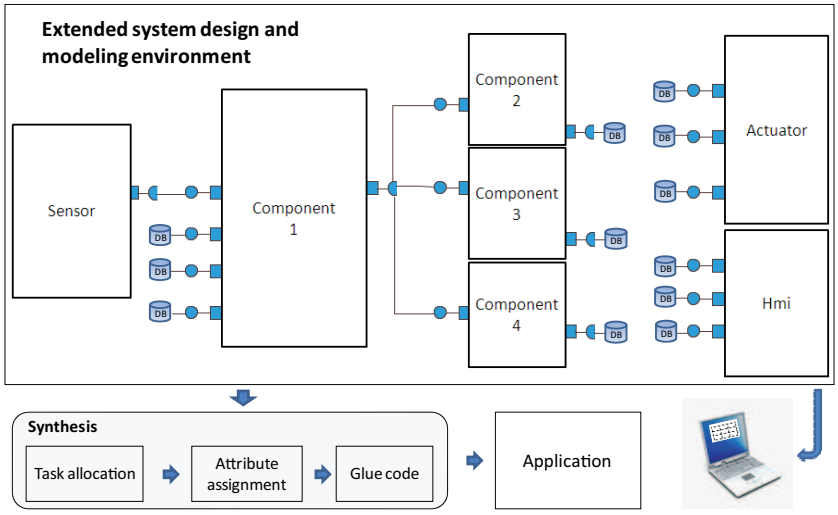

Figure 7.5: Database View of Application Model

The architectural framework introduced in this paper distinguish between two types of database proxies, namely *hard real-time database proxies* (hard proxies) and *soft real-time database proxies* (soft proxies).

7.4.1 Hard Real-Time Database Proxy

Hard proxies are intended for hard real-time components, which need efficient and deterministic access to individual data elements.
A hard real-time database proxy;

- is connected to a component's in- or out-port, thereby acting as a communication link to the database.
- is realized with a database pointer to enable predictable data access to individual data elements.
- contains all information to set up a database pointer, which will be constructed in the component framework as glue code between component calls.

- uses a predictable concurrency control algorithm such as 2V-DBP [14] that provides constant response-time for database pointers.
- can provide a data element of any type.
- can be used with any existing components since the database is fully transparent to the component.

7.4.2 Soft Real-Time Database Proxies

Soft proxies are intended for soft real-time components, which might need more complex data-structures. Consider a component monitoring the overall status of a subsystem, e.g., all the temperatures in an engine, or logging of errors etc.

In order for a component to be decoupled from the RTDBMS and use a soft proxy, it utilizes a *relational interface*, which means that the components interface has the notion of a relational table. Therefore a new type is introduced, *TABLE*.

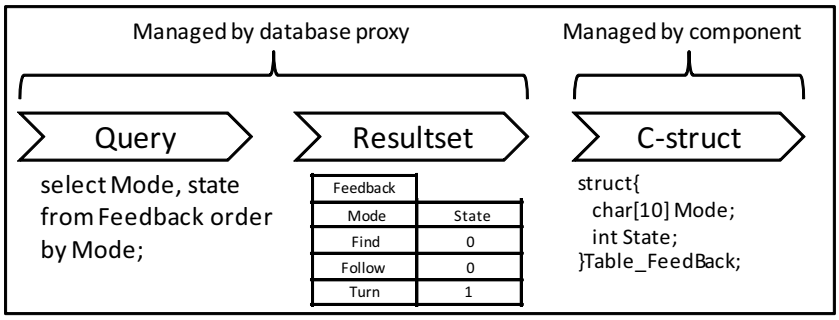

Figure 7.6: Description of Table Type

A *TABLE* is a realization of a relational table using standard C-types. Figure 7.6 illustrates the three steps from query to resultset and C-struct. At runtime the database query returns a resultset that is converted by the soft proxy into the defined *TABLE* to match the interface of the consuming component. The specified type *TABLE*, is generated into the component code.

This approach enables a component to be database unaware as the database proxy does not introduce any side-effects. Since a component can receive a

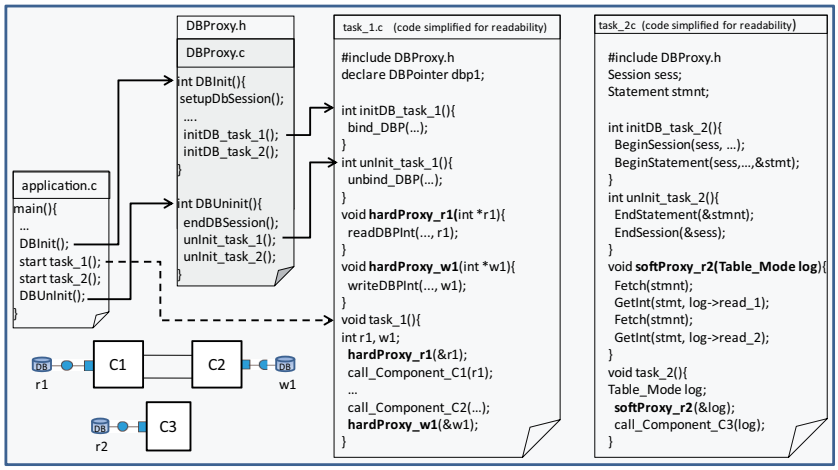

Figure 7.7: Hard and Soft Proxy Glue-Code Generation Example

type *TABLE* without the usage of a database proxy, the component is still reusable without a database. This is possible since the data transformation between the database and the *TABLE* is encapsulated in the proxy.
A soft real-time database proxy;

- is connected to a component's in- or out-port, thereby acting as a communication link to the database.
- uses standard SQL query language.
- converts the resultset from the database query into the format of the *TABLE* which is realized by a standard c-type.
- hides the database query in the glue-code associated with the proxy.

7.4.3 Proxy Implementation Description

Figure 7.7, shows a simple example of how the glue-code generated from the proxy specification for hard and soft database proxies are implemented. In the lower left of the figure, two example applications are displayed. One application includes component *C1*, that reads a value from the database, filters it

and outputs to component *C2*. *C2*, writes a value to the database. The other application shows an example of a soft database proxy implementation where component *C3* reads a type *Table_Mode*. Figure 7.7 has been simplified for readability. The flow pointed out by the arrows in figure 7.7, 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 implementation can be divided in three phases, initialize, running task and un-initialize.

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. This is done for both hard and soft proxies.

2. Each tasks individual, initialization function, *initDB_task_1()* and *initDB_task_2()* is called to bind hard proxy real-time database database pointers and to setup soft proxy real-time statements.

Task execution

1. The database proxies are included in the task files, *task_1.c* and *task_2.c*.

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

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

Un-initialize

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

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

7.5 Implementation

In our approach, we have extended the CBSE system development framework to include database proxies and data modeling, see figure 7.8. In this framework the system architect can utilize the usage of a database as an additional design feature. If a database is included in the design, the generated *System*

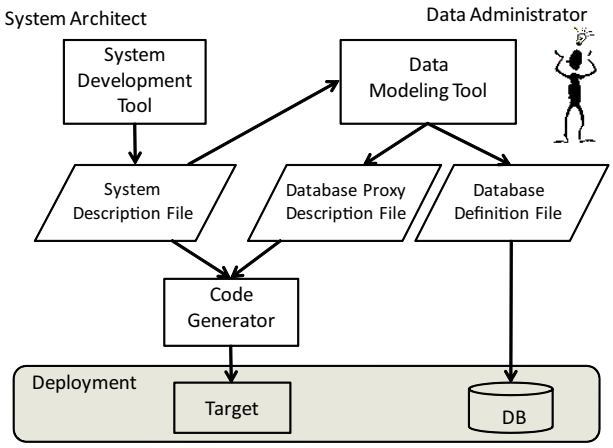

Figure 7.8: The Approach

Description File, is extracted from the *System Development Tool* in order to perform the data modeling and generate a *Database Proxy Descriptions File*. These files are then weaved together with the *Code Generator* to form the run-time C-code. A *Database Definition File* is also generated from the data modeling to enable the database setup. Three existing tools and technologies have been used in our proof of concept implementation of the approach in figure 7.8. Save-IDE [21], Mimer Real-Time edition (MimerRT) [15] and the Data Entity Navigator (DEN) [16].

7.5.1 Mimer Real-Time Edition

The Mimer SQL Real-Time Edition (Mimer RT) [15] is a real-time database management system intended for applications with a mix of hard and soft real-time requirements. Mimer RT uses the concept of database pointers [14] to access individual data elements in an efficient and deterministic manner. For soft real-time database management, standard SQL [13] queries are used. To achieve database consistency without jeopardizing the real-time requirements the 2V-DBP concurrency control algorithm presented in section 7.3.1

7.5.2 SaveCCT Real-Time Component Technology

The SaveComp Component Technology (SaveCCT) [21] is described by distinguishing manual design, automated activities, and execution. The entry point for a developer is the Save Integrated Development Environment (Save-IDE), a tool supporting graphical composition of components, where the application is created. Developers can utilize a number of available analysis tools with automated connectivity to the design tool. SaveCCT is based on a textual XML syntax which allows components and applications to be specified. Automated synthesis activities generate code used to glue components together and allocate them to tasks. SaveCCT is, as Mimer RT, intended for applications with both hard and soft real-time requirements.

SaveCCT applications are built by connecting components input and output ports using well defined interfaces. Components are then executed using trigger based strict "read-execute-write" semantics. A component is always inactive until triggered. Once triggered it starts to execute by reading data on input ports to perform its computations. Data is then written to its output ports and outgoing triggering ports are activated. Except from regular connections, SaveCCT also provide a flexible connection concept denoted complex connections. This is the entrance point in the component model for the database proxies. The database proxy configuration is defined in the model of the complex connection.

7.5.3 Embedded Data Commander Tool-Suite

The *Embedded Data Commander* (EDC) is a tool-suite that implements the data entity approach [16] for the ProSys component-model [17]. A data entity is a compilation of knowledge for each data item in the system and can be defined completely separate from the development of components and functions. This enable developers to crate a system with data entities based on application requirements and perform early analysis even before the producers or consumers of the data are developed.

This tool suite has in our continued research on database proxies been extended with new functionality that supports SaveCCT, real-time component technology [21].
The tool-suite has been extended with:

- The System Signal Manager (SSM), manages the SaveCCT signals, proxy and component information.

```
1.<SIGNAL id="P_FindFB_W" component="Find">
2.<SNIPPETDEF type="int Fi_FindFB;"
  pointerdefinition="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 7.9: Hard Proxy Representation

- The DataBase Administrator Tool (DBAT), used to model, setup and generate database schemas, load files and database proxies.

Save-IDE generated description files can be imported and interpreted by the SSM to give the developer a more data centric view and information rather than focusing in components as in Save IDE. From this information the DBAT is used to design the database and generate the appropriate load files for the RTDBMS. This information is then used to generate the database proxy information files that is exported to Save-IDE in order to generate the component glue code.

A database proxy definition is represented in XML. Figure 7.9 shows an example of a generated hard proxy description using MimerRT. 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 to bind the database pointer, including the sql query. 4 the type of call to use, in this case an update call since it is a write proxy. 5 end of proxy definition.

7.6 Performance Evaluation

In this section we describe the results from a performance evaluation where we have implemented an embedded control system and measured execution times and memory overheads.

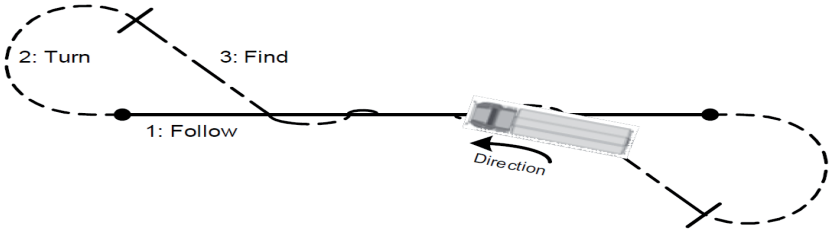

Figure 7.10: Truck Application

7.6.1 The Application

To evaluate or approach, we have implemented an application in Save-IDE. The application consists of seven components that simulates a truck. The application has three modes, follow, turn and find which are connected to an actuator component. Components follow, turn and find are also connected to mode change component via feedback loops. The truck first follow 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 7.10.

The application consists of two tasks, a hard real-time task and a soft real-time task. The hard real-time task is triggered every 10ms and consist of six components. A sensor component that outputs sensor values to the *ModeChange* component that decides which of the three modes *follow*, *turn* and *find* to activate and the actuator component. The architectural design decision of the application is to replace the three interconnected loop-back signals from the three mode components to the *ModeChange* component with hard real-time database proxies. Components *follow*, *turn* and *find* each updates a value in the database that is read by three database proxies connected to component *ModeChange*.

Since the task performed by the included components is quite trivial, we have added a more realistic work load in the system. We have added a complex embedded benchmark code used within the area of worst-case execution time (WCET) analysis [18] to components follow, turn and find. The benchmark code performs a lot of bit manipulation, shifts, array and matrix calculations.

The soft real-time task is triggered every 20ms and consist of one HMI component. The component uses a database proxy to periodically read the three values updated by the hard real-time task.

7.6.2 Benchmarking Setup

To evaluate our approach, we have performed a performance evaluation of four possible implementations of the evaluation application. The aim of the evaluation is to measure if the usage of our approach using database proxies will have an impact on the real-time performance and memory consumption of the system.

The tests have been performed on a Hitachi SH-4 series processor [19] with VxWorks [20] as real-time operating system. The hard real-time tasks are executed 1800 times.

The four evaluated implementations shown in figure 7.11:

Test 1 Baseline implementation using regular memory without any database connection. The feedback loops implemented as shared variables protected using semaphores.

Test 2 Implementation using database unaware components with access to the database using the concept of database proxies.

Test 3 Implementation using database aware components with access to the database from within the components using database pointers.

Test 4 Implementation using database aware components with access to the database from within the component using only regular SQL queries without hard real-time database pointers.

7.6.3 Real-Time Performance Results

Figure 7.11 shows the result of the response-times of the hard real-time control application for the four test-cases. The graphs clearly show that the introduction of a real-time database using database pointers, either directly in the component-code or through proxies does not affect the real-time predictability and adds little extra execution time overhead, while using SQL queries directly in the component-code severely affects both predictability and performance negatively. Table 7.1, shows a table with the evaluation results. The change of the Average Case Execution Time (ACET) and Worst Case Execution Time (WCET) in the two rightmost columns shows the change in percent, with test 1 as a benchmark. The ACET and WCET between the first three tests does not differ more than a few percent. The fourth test does, as could be expected, not perform anywhere near the other tests.

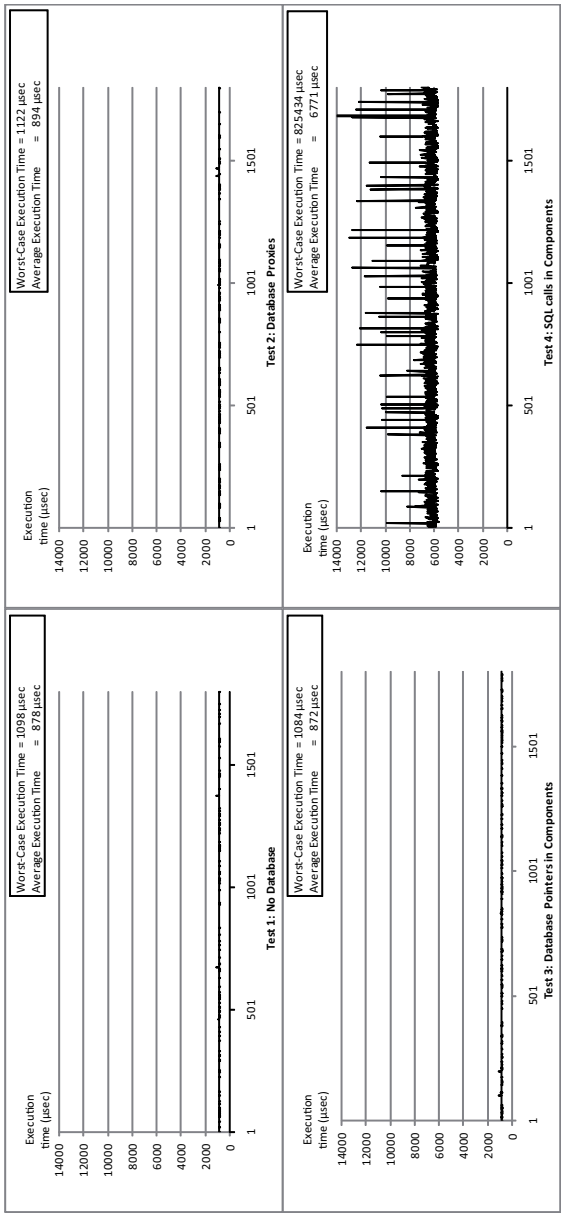

Figure 7.11: Evaluation Results

Test	ACET	WCET	ACET (%)	WCET (%)
1	878	1098	-	-
2	894	1122	1.82	2.19
3	872	1084	-0.68	-1.28
4	6771	825434	771.18	75176.14

Table 7.1: Application Execution Time

Test	Code Size	Change (%)
No Database	653 512 bytes	-
Database Pointers	666 564 bytes	1.99
Database Proxies	666 988 bytes	2.06

Table 7.2: Application Code Size

In these tests we are most interested in test 2, which shows that the ACET is increased by only 1.82% and the WCET by 2.19%. Furthermore, the evenness of the results clearly shows that the usage of database proxies is predictable, and with the amount of overhead in average and worst-case execution time is limited. We interpret the slight decrease in ACET and WCET for test 3 to be a result of optimized synchronization primitives used by MimerRT compared to the regular POSIX routines.

7.6.4 Memory Consumption Results

Table 7.2 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. 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 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 24k. The increase of client code size, as well as the small size of modern embedded database servers makes the memory overhead for database proxies and real-time database affordable for many of today's real-time embedded systems.

7.7 Conclusions

This paper presents the database proxy approach which enable fusion between real-time database management systems (RTDBMSes) and component-based software engineering (CBSE). Our approach allows the introduction of RT-DBMSes, and the associated range of new possibilities, to CBSE; this includes the possibility to access data via standard SQL interfaces, concurrency-control, temporal consistency, and overload and transaction management. I 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 [7, 8].

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 EDC tool, where the database proxies and interface to the database is created. The EDC 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, we has performed a series of execution time tests on the generated C-code for a research application. These tests shows that our approach only increase (both the average and the worst-case) execution time with approximately 2%. 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 offers a range of valuable features that to real-time embedded systems development, maintenance and evolution at a minimal cost with respect resource consumption.

Bibliography

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

[2] 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.

[3] 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.

[4] 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.

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

[6] 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 & Sons, 1996.

[7] Sandro Schulze and Mario Pukall and Gunter Saake and 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] 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 (ETFA'08), IEEE Industrial Electronics Society, Hamburg, Germany*, September 2008.

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

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

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

[12] ISO SQL 2008 standard. *Defines the SQL language*, 2009.

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

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

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

[16] 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.

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

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

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

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