The Journal of Systems and Software 85 (2012) 821–834

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Data management for component-based embedded real-time systems: The database proxy approach

Andreas Hjertström*, Dag Nyström, Mikael Sjödin

Mälardalen Real-Time Research Centre, Västerås, Sweden

ARTICLE INFO

Article history:
Received 13 August 2010
Received in revised form 24 October 2011
Accepted 25 October 2011
Available online 31 October 2011

Keywords:
CBSE
RTDBMS
Real-time
Embedded systems

ABSTRACT

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

1. Introduction

This paper proposes database proxies (Hjertström et al., 2010) as a solution to integrate a real-time database management system (RTDBMS) (Adelberg et al., 1996; Ramamritham et al., 2004) into a component-based software engineering (CBSE) (Buschmann et al., 1996; Crnkovic and Larsson, 2002) setting. Database proxies are automatically generated glue code synthesized from the system architecture that translates data between components ports and an RTDBMS residing in the component framework.

Data management of embedded real-time systems is becoming increasingly important as systems evolve from simple stand-alone devices into becoming complex systems, often interconnected with its surrounding environment. This trend has lead to that developers are confronted with a substantial amount of functions, design-time and run-time data that needs to be managed. In addition, developers are increasingly faced with new requirements such as secure and dynamic data sharing and advanced diagnostics. To reduce the resulting complexity, model driven development (OMG UML, 2011) and CBSE, are increasingly used in industry today. However, these techniques mainly focus on the functional aspects of the software, and rarely target management of data.

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

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

* Corresponding author. Tel.: +46 21 107322; fax: +46 21 103110.
E-mail addresses: andreas.hjertstrom@mdh.se (A. Hjertström), dag.nystrom@mdh.se (D. Nyström), mikael.sjodin@mdh.se (M. Sjödin).

0164-1212/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2011.10.036

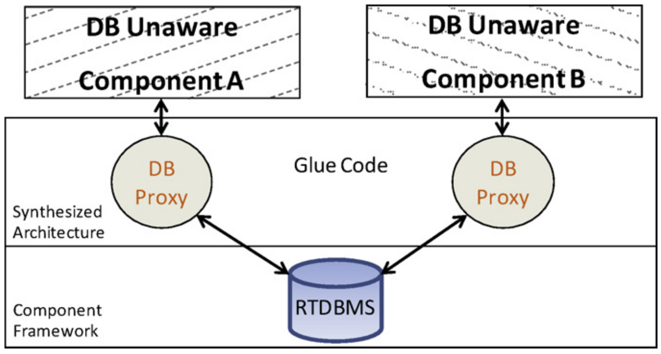


Fig. 1. Database proxies connecting components to an RTDBMS.

query languages, providing advanced access control mechanisms. This implies that potentially unsafe software, such as third party software, can be granted access to data in a controlled manner. Furthermore, RTDBMSs significantly cuts time to market by providing high-level query languages, supporting logging, diagnostics, monitoring, and efficient data modeling (Schulze et al., 2009).

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

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

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

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

1. A component with direct access to the database from within, violates the component's aim to be encapsulated and only communicate through its interface.
2. Direct access to shared data introduces hidden dependencies between components.
3. If an RTDBMS is called from inside the component, the component is dependent on that specific RTDBMS and cannot be used in an alternative setting.

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

As illustrated in Fig. 1, a database proxy is part of the synthesized architecture, thus external to the component. The purpose of the database proxy is to enable for components to interact with an RTDBMS using their normal interfaces. This is possible since the coupling, i.e. the database proxy, between the component and the RTDBMS is embedded in the component framework.

The database proxies are used to bind and integrate components to form the final running system. By decoupling components from the database, and placing the database in the component framework, the decision to use a database or some other data management strategy is removed from the component level and becomes a system design decision.

Database proxy characteristics

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

In our previous work, database proxies was limited to only support native data types e.g. integer, char, float, etc., from one port to another (Hjertström et al., 2010). The work has now been augmented so that components can have efficient and predictable access to complex data structures to/from multiple ports of the same component in hard real-time. In addition, data transfers between components can be extracted to the database, for logging purposes or to share data, without interfering with the regular component communication.

The remainder of this paper is structured as follows: in Section 2, we present motivation for the approach. Section 3 present the specific problems that our approach targets, related work and state of practice. In Section 4, we present the system model. Section 5 gives a detailed description of the database proxy and its constituent parts. Further, in Section 6, we illustrate our ideas with an implementation example. Finally, we show a performance and real-time predictability evaluation in Section 7 and conclude the paper in Section 8.

2. Motivation

The characteristics of today's embedded systems are changing. According to Fürst (2010) and Grimm (2003), 90% of all innovations within the automotive industry stems from software and electronics. In a high end vehicle there can be more than 800 functions, 70 ECUs, and thousands of signals need to be managed (Albert, 2004). This has led to increasingly complex and costly to development of embedded systems.

When developing modern large scale IT systems, the use of standardized platforms as a base for service-oriented architectures, error recovery, etc. is widely used. They provide features such as several abstraction layers, virtualization techniques and scalability. However, in resource constrained embedded systems with limited memory size, limited computing capacity and demand for low energy consumption, this approach is not sufficient since abstraction layers and virtualization techniques add to the amount of resources needed (Liggesmeyer and Trapp, 2009).

Within the embedded community, modern techniques such as model driven development and component-based software engineering are widely used to reduce complexity and increase the understanding and reusability of software functions by elevating the abstraction level (OMG UML, 2011; AUTOSAR, 2011; Åkerholm et al., 2007). However, these techniques do not include methods and tool support for efficient and management of data.

Many of today's systems are developed by different subcontractors in form of whole applications or just individual functions, sometimes each with their own in-house developed solution for how to manage data. In addition, it has been shown that documentation and structured management of internal ECU data is sometimes almost non-existent and dependent on individual developers own solutions (Hjertström et al., 2008).

The increasing need for more structured, flexible, reliable and secure data management techniques to coordinate data both at run-time and at design-time is continuously pointed out has been pointed out as major challenges for the future (Schulze et al., 2009; Nyström et al., 2002; Brooks et al., 2008).

As stated by Pretschner et al. (2007) and Broy (2006), a standardized and overall data model and management system has great potential as a solution to deal with the distributed and uncoordinated data in these complex systems. Furthermore, Schulze et al. (2009) and Saake et al. (2009) point out that the ad hoc and/or reinvented management of data for each ECU with individual solutions using internal data structures, can lead to concurrency and inconsistencies problems. In addition, maintainability, extensibility and flexibility of the system decreases.

Furthermore, sophisticated techniques for diagnostics, error detection, logging and secure data sharing are much needed to improve reliability and system quality. Due to the ineffective diagnostics and error tracing techniques, less than 50% of the replaced ECUs were, in fact, defect (Pretschner et al., 2007). Much of the diagnostics messages and logging that can be retrieved from these systems are statically predefined at design time. An example of this is in the AUTOSAR standard (AUTOSAR, 2011a). In techniques a such as the Program Monitoring and Measuring System (PMMS), it is up to the user to specify pre-conditions and insert code in order to collect data (Delgado et al., 2004). This put high demands on developers to predict future needs of, for instance, service technicians. In difference, the flexible and dynamic behavior of an RTDBMS can provide any single data element or a set of data elements with a single query, providing that the user is granted access.

Secure data sharing is becoming increasingly important when systems are opening up to the surrounding environment using techniques such as CAR2CAR communication (AUTOSAR, 2011) and/or connecting to PDAs, smart phones, GPS, etc. The diversity of these devices have led to in-house proprietary solutions to enable a connection to the infotainment system (Gereon Weiss and Eilers, 2011). A proposed standardized solution to this could be to use a data management system, such as an RTDBMS (Schulze et al., 2009; Saake et al., 2009). An RTDBMS provides both access control to data as well as dynamic data access using a well known standard query language (SQL). In addition, in order to achieve a separation of data management and application logic, a general data management infrastructure is needed (Härder, 2005).

The usage of an RTDBMS when designing and building real-time embedded systems could not only aid developers with standardized tool support for modeling system data at design-time (Shan Chen, 1976), but also provide predictable and efficient routines for managing data at run-time. This could, as an example shorten time-to-market, since developers can manage complex data structures with a single database query instead of using complex programming routines.

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

3. Background

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

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

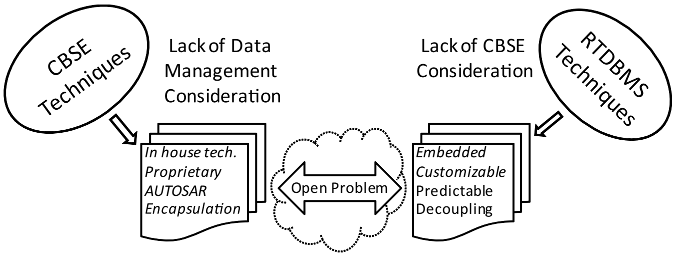


Fig. 2. Combining CBSE and DBMS is an open problem.

The usage of an RTDBMS in a CBSE framework should not introduce any side effects that violate CBSE principles (Crnkovic and Larsson, 2002; Szyperski, 1997).

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

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

3.1. Solution requirements

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

R1 The decision to use an RTDBMS should be made on system level in order to be integrated in existing development models and systems.
R2 The usage of an RTDBMS should not introduce any side effects to the components.
R3 The real-time predictability of the system should not be compromised by using an RTDBMS.

3.2. Related work and state of practice

The research which explicitly aims at combining CBSE and an RTDBMS is novel. Fig. 2 illustrates that there is a gap between CBSE and RTDBMS techniques.

Within the CBSE community there are specialized in-house and proprietary techniques such as Koala (van Ommering, 2000) or global automotive initiatives such as AUTOSAR (2011a). However, these techniques do not prioritize data management. As an example, AUTOSAR provides a uniform way for managing data e.g., save and load data from non volatile memory. However there is no uniform technique to manage run-time data in RAM.

Lau and Taweel (2006) presented a research direction within CBSE on how to manage data by having data flow and data access completely encapsulated within connectors. In this way, components only encapsulate computation. Another approach by Lau and Taweel (2007) is to encapsulate data inside components to achieve encapsulated reusable building blocks, where data is included.

Efforts within the database community, illustrated in the rightmost part in Fig. 2, aim at developing solutions to downsize and optimize RTDBMSs to suite embedded systems. There are solutions for componentizing and/or customizing the database management system to only include features that is actually used in a particular resource constrained system (Saake et al., 2009). In addition, there are also techniques, such as database pointers, that can manage both soft and hard real-time transactions predictably (Nyström et al., 2003). These techniques are available in both research-based and commercial RTDBMSs (Mimer, 2011; Nyström et al., 2004; Polyhedra, 2011).

Both the CBSE and the RTDBMS community have solutions suitable for their respective areas. However, in between CBSE and RTDBMSs, there is a gap with respect to data management, illustrated in Fig. 2. The lack of research within this area has left this gap as an open problem. The aim of this paper is to bridge this gap.

There are mechanisms within the RTDBMS community that aims to simplify the database access by hiding some of the underlying complexity as well as making the access to the RTDBMS more efficient. The standardized interface-language SQL defines the following mechanisms (ISO SQL, 2008):

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

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

1. The database is still accessed from within the component code, not through the component's defined interface. (Violation of R1–R2.)
2. The component is still only partially decoupled from the database since the database name, login details and connection code still need to reside in the component. A component using these mechanisms is therefore no longer generic or reusable. (Violation of R1–R2.)
3. The requirements expressed by the components interface do not reflect the components internal database dependency. (Violation of R2.)
4. These mechanisms are not intended for real-time performance (typically only non-real time DBMS support is available), e.g., the usage of these mechanisms alone would be a violation of R3.

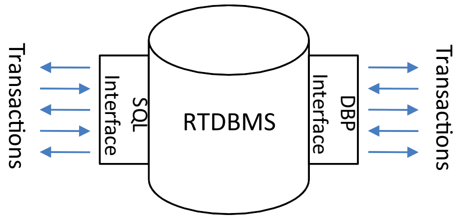


Fig. 3. RTDBMS architectural overview.

4. System model

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

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

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

1. A `native data type` to be a basic data type such as an integer, char or float.
2. A `complex data type` to be a C-struct or an array that consists of a number of native data types.
3. A `fixed-length data` as a data that have a fixed size. An example of this is a single struct containing only native data types.
4. A `variable-length data` as a data that can vary in size. An example of this is an array of structs.

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

Hard real-time tasks, that control critical functionality, use hard real-time transactions (Stankovic and Zhao, 1988) to read and write values from sensors/actuators and execute real-time control loops. Hard real-time tasks communicate with fairly simple data structures such as native data types and more complex but fixed-length data structures such as a C-struct. Variable-length data are not supported since hard real-time tasks require predictable access to data elements.

Soft real-time tasks, that control less timing sensitive functionality. Soft real-time tasks uses soft real-time transactions (Stankovic and Zhao, 1988) to read and write variable-length complex data structures typically to present statistical information, logging, or used as a gateway for service access to the system by technicians in order to perform system updates. Soft real-time tasks could also be used for fault management and perform ad hoc queries at run-time.

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

Fig. 3 illustrates an RTDBMS which has a soft interface that utilizes a regular SQL query interface to enable flexible access from soft real-time tasks. For hard real-time transactions, a database pointer (Nyström et al., 2004) interface is used to enable the application to access individual data elements or a set of data elements in the database with hard real-time performance. Two RTDBMSs that provide these types of interfaces are COMET (Nyström et al., 2004) and Mimer SQL Real-Time Edition (Mimer, 2011).

```
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();
     }
  }
```

Fig. 4. An I/O task that uses a database pointer.

4.1. Database pointers

Database pointers (Nyström et al., 2003) are pointer variables that are used for real-time access to data in a real-time database, see Fig. 4. The figure shows an example of an I/O task that periodically reads a sensor and propagates the sensor value to the database using a database pointer, in this case the data element, oil temperature, in the engine relation. The task consists of two parts, an initialization part (lines 2–4) executed when the system is starting up, and a periodic part (lines 5–8) scanning the sensor in real-time. During the initialization part (lines 2–4) the database pointer is created and bound to a data element in the database.

During the control part of the task in Fig. 4, the `write` function writes the new value `temp` to the database pointer. During this operation, only a few lines of non-blocking code (with a bounded number of instructions) that performs type checking, synchronization with other accesses with the same data element, and writing of the data are executed.

Database pointers can be bound to either individual single data elements, or to sets of data. Depending on the organization of the data in a set, different types of database pointers are used (Mimer, 2011):

Single database pointer: A single database pointer can only be bound to an individual data element, e.g., an integer, string or a float. This type of pointer is useful for storing sensor and actuator values. The data element is the atomic unit, i.e., a single database pointer provides atomic reads and writes of a single value. This implies that a single database pointer does not have any transactional properties such that atomic commits of multiple single value database pointers.
Multicolumn database pointer: A multicolumn database pointer is bound to a set of attributes (columns) of a single database record (row in a table). When a multicolumn database pointer is read or written, all data in the set are read or written atomically. This provides a simple transactional behavior, in the sense that a snapshot of data can be kept consistent.
Multirow database pointer: A multirow database pointer is bound to a certain attribute but spans a set of database records in a table. When a multirow database pointer is bound, the pointer is set to point at the first element in the set. Writes to the database pointer are performed on a row by row basis, and after each write, the pointer is set to point to the next element in the set. When all elements in the set have been written to, the pointer is reset to point to the first element again.
Multicolumn–multirow database pointer: A multicolumn–multirow database pointer combines the functionality of a multicolumn database pointer and a multirow database pointer thus being able to bind a matrix of data elements. These pointers are especially suited for event logging where for example log event information, event data and a timestamp can be logged in a database for future analysis.

The cost and predictability of database pointer execution is, as shown in the performance evaluation in Section 7, comparable to the performance of a shared variable that is protected by a semaphore.

Since database pointers can co-exist with relational (SQL) query management, data can be shared between hard and soft real-time tasks. However, in order to maintain real-time predictability in a concurrent system, some form of concurrency-control is needed. The 2-version database pointer concurrency algorithm (2V-DBP) (Nyström et al., 2004) uses a 2-version versioning algorithm that guarantees that database pointers will never be aborted or subjected to unpredictable blocking. 2V-DBP allow soft real-time transactions to concurrently access the data without experiencing blocking or aborts due to operations through database pointers.

4.2. System architecture and modeling

In the application design and modeling we employ a pipe-and-filter (Buschmann et al., 1996) 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). Fig. 5 shows an example of a component-based system design and modeling architecture.

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

5. Database proxies

A database proxy consists of pieces of code that translates data from a components port to a database call and further on to an RTDBMS residing in the component framework and vice versa. These pieces of code are neither a part of the component nor a part of the RTDBMS, instead database proxies are automatically generated glue code synthesized from the system architecture e.g., the structure and behavior of the system, see Fig. 1. A database proxy contains the following parts:

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

Fig. 6 gives an overview of the different proxy parts which will be presented more in detail in the remainder of this section.

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

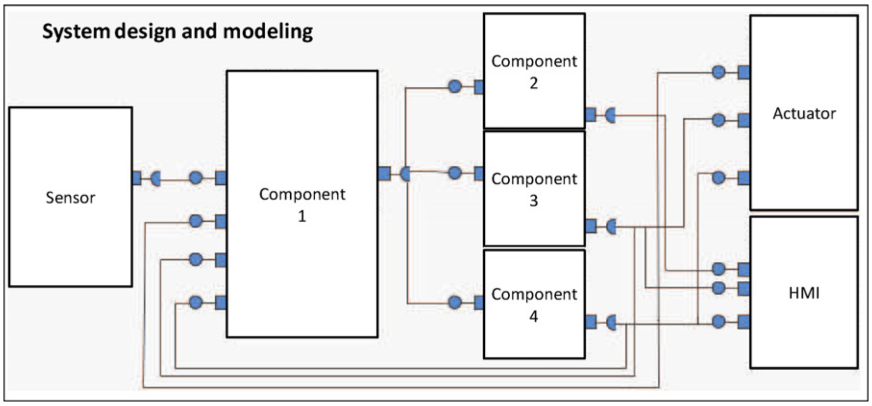


Fig. 5. System design and modeling.

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

As a result, database proxies target requirements **R1**–**R3** presented in Section 3.1, since database proxies are:

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

To support the different requirements of hard and soft real-time tasks (see Section 4), we distinguish between two proxy types, *hard real-time database proxies* (hard proxies) that is used by hard real-time tasks and *soft real-time database proxies* (soft proxies) that is used by soft real-time tasks.

Database proxies can have one of the following configurations:

1. A *read proxy* is used to retrieve data from the database and output the data to a component. This is illustrated by proxies connected to components `Consumer_1-3` to the right in Fig. 7 which each is provided with data from a read proxy.
2. A *write proxy* is used either to update or to insert data that is provided by a component to the database. An update is used to refresh data that is already in the database. An insert is used to add a new data e.g., a new row, in a dynamic database table. An example of a write proxy connected to component `Producer` is illustrated to the left in Fig. 7.
3. A *proxy through* is connected as a communication link between two components. The proxy through is used to listen in on a regular component connection and propagate data to the database as a write proxy. The result is normal communication between the components. However, a copy of the value is stored in the database for example to; perform logging, usage by other components or to be accessed in cross platform communication and telematic services. An example is illustrated in Fig. 8.

5.1. Proxy ports

The entry point of the database proxy is through a port or a set of ports. A port is an interface entity for receiving/sending different data elements to its connected components. A graphical example of proxies with a single port or multiple ports is illustrated in Fig. 7. The ports of a read proxy or a write proxy are always connected to a single component with a one to one mapping between the number of proxy ports and component ports. However, a proxy through is always connected to a pair of components and there is a one to one mapping between the number and types of input and output ports.

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

A complex data type will however be transformed for further processing by the proxy into a set of native data types and vice versa. An example is illustrated in Fig. 6 where the complex data type in **Port_2** is transformed into data elements (b–d) by the proxy and put into an ordered data set. In a similar manner the transformation can be performed in the other direction, elements in the ordered set to the complex data type.

5.2. Proxy data sets

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

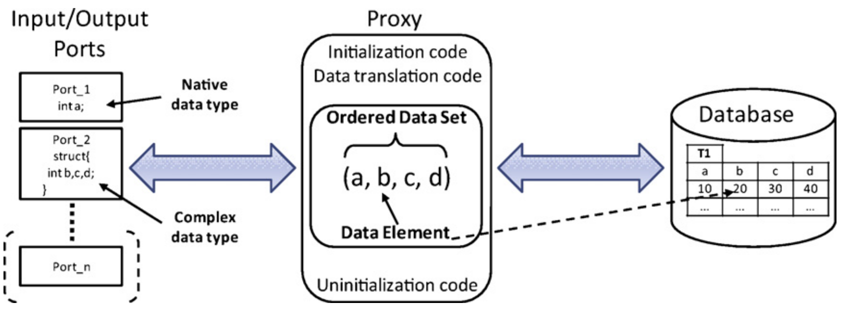


Fig. 6. Database proxy overview.

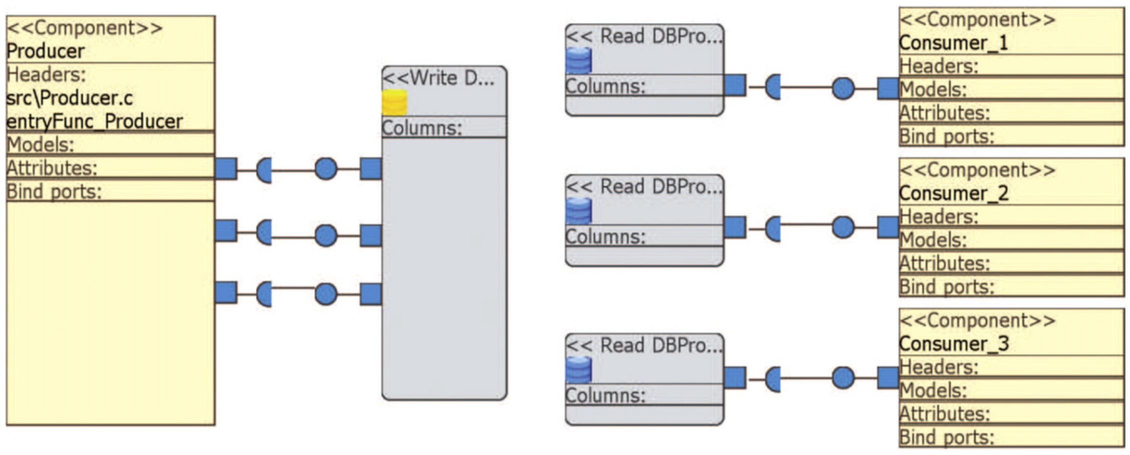


Fig. 7. Single-port and multi-port proxies.

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

An example is illustrated in Fig. 6 where **Port_1** has a native data type a and **Port_2** has a complex data type that includes three the native data types. a is directly put in the ordered data set whereas the complex data type is transformed into elements b, c and d and put in the ordered data set for further transformation via a database query to the database.

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

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

5.3. Hard real-time database proxies

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

A proxy with an ordered data set that consists of a single data element utilizes a single database pointer for predictable data access. However, a proxy with an ordered data set which consists of two or more data elements, utilizes a multi-column database pointer, as stated in Section 4.1, to perform an atomic update or read of a defined set of columns on a single row in the database.

In order to be predictable, a hard proxy only translates native data types and fixed-length complex data types. This implies that no unpredictable type conversions or translations that require unbounded iterations are allowed. A complex data type, such as a C-struct or an array must be of fixed-length. A hard proxy ordered data set is therefore always directly related to a fixed number of columns on a single row in the database.

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

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

A hard real-time database proxy:

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

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

5.4. Soft real-time database proxies

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

A soft real-time database proxy;

- Communicates with the database through a relational interface, thereby providing a flexible data access.

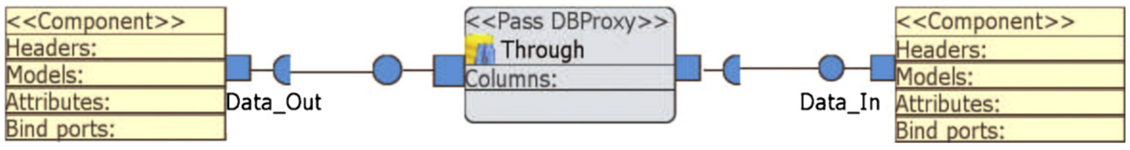


Fig. 8. Proxy through.

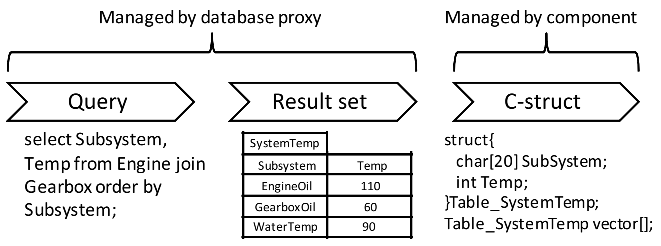


Fig. 9. Description of `TABLE` type.

- Translates complex data types, thereby providing means for components to access complex data.

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

Consider the following example (see Fig. 9):

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

Introducing a `TABLE` data template does not make components database aware since components still can communicate using a `TABLE` instance in absence of a database. Table 1 presents an overview of the similarities and differences between hard proxies and soft proxies presented in Sections 5.3 and 5.4.

5.5. *Extended system design and modeling*

We complement the classical architectural view, presented in Section 4.2, with a new additional design view, the *CBSE database-centric view*. This new view identifies which component ports are connected, via different types of database proxies, to data elements in an RTDBMS. An example of this is illustrated in Fig. 10. The notation simplifies the view of the system by removing the actual connection between the producing and consuming component, thus replacing it with a database symbol.

To enable traceability, this view can be transformed at any time to reveal the data flow through the connections such as shown in Fig. 5. This is similar to an *off-page connector* that is used when designing electrical schemas which involve a large number of components and connections. A connection ends in a symbol or an identification name that is displayed at each producer and consumer. Displaying all connections in a complex schematic diagram would make the electrical schema impossible to read. This approach is also being used by CBSE-tools such as Rubus Integrated Component Environment (Rubus ICE) (Arcticus Systems, 2011).

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

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

Table 1
Hard and soft proxy support overview.

Proxy support	Hard proxy	Soft proxy
Database interface	Database pointer	SQL
Predictable data access	X	–
Flexible data access	–	X
Allowed for hard task	X	–
Allowed for soft task	X	X
Multi value support	X	X
Multi port support	X	X
Database read	X	X
Database update	X	X
Database insert	–	X
Support complex data type	X	X
Support fixed-length data	X	–
Support variable-length data	–	X

5.6. *Database proxy example*

Fig. 11, which has been simplified for readability, shows a simple example of how the glue-code generated from the database proxy specification for hard and soft database proxies of different types are implemented. In the upper left of the figure, the architecture of two tasks is displayed. **Task_1** is a hard real-time task that consists of components **C1** and **C2**. **Task_2** is a soft real-time task that consists of component **C3**.

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

Task_2 shows an example of a soft database proxy implementation where component **C3** reads a type `Table_Mode *` which include the two values updated by the proxy connected to two ports in **Task_1**. The flow pointed out by the arrows in Fig. 11, for the hard real-time task, **Task_1.c**, is also valid for the flow in the soft real-time task, **Task_2.c**.

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

Phase 1: Initialize

1. *application.c* is the main application file. Before the task/tasks containing a database proxy/proxies are called, the database is initialized by calling the `DBInit()` function declared in the separate *DBProxy.c* file.
2. Each task's individual, initialization function, `initDB_Task_1()` and `initDB_Task_2()` respectively, is called to bind hard proxy real-time database pointers and to setup soft proxy real-time statements.

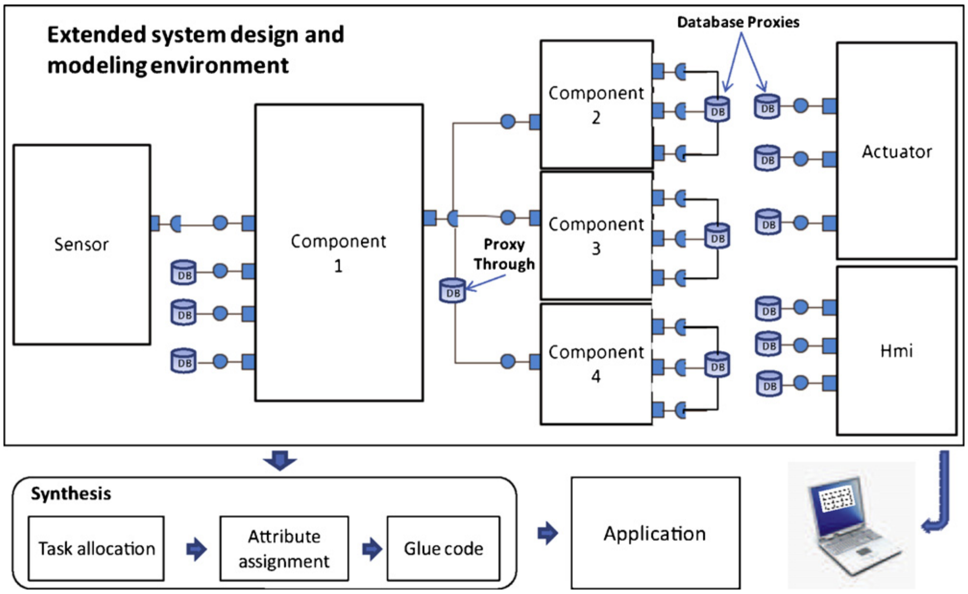


Fig. 10. Database view of application model.

Phase 2: Task execution

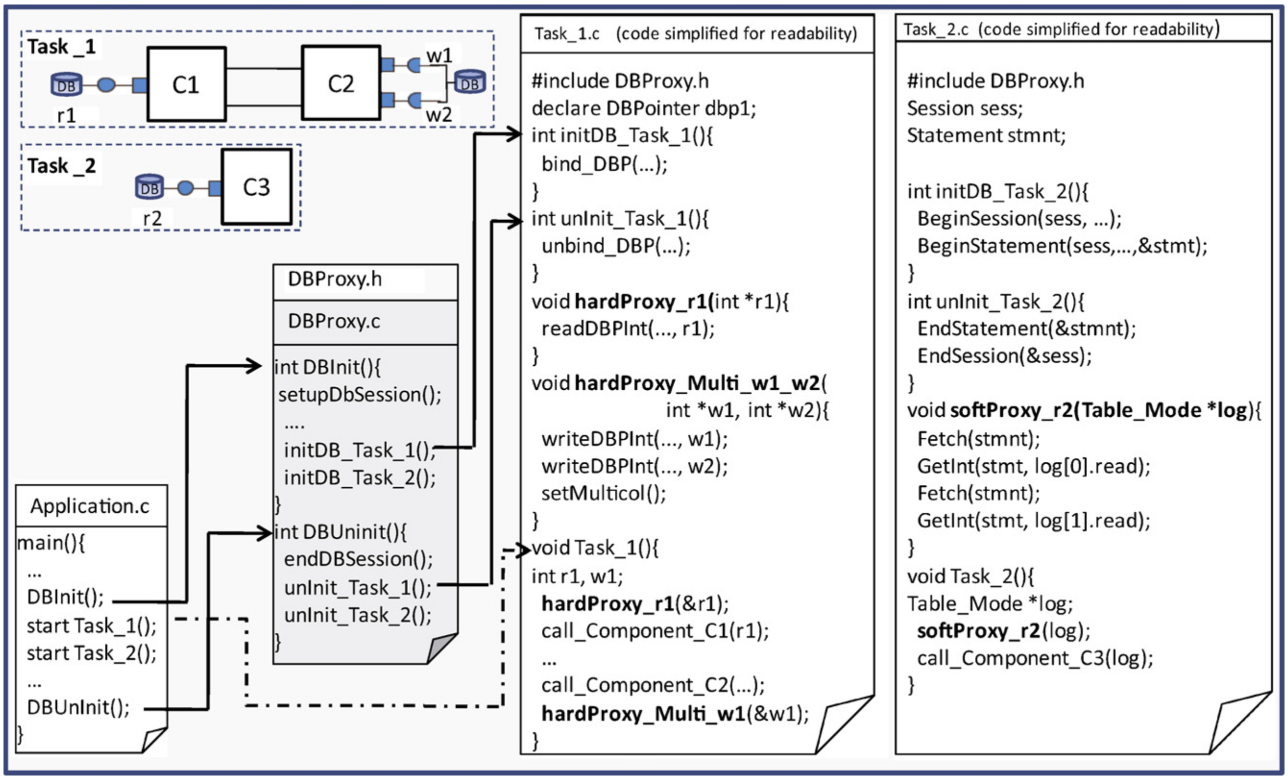
1. The database proxies are included in the task files, **Task_1.c** and **Task_2.c**.
2. The database proxies are declared as separate functions which are called before the component call if it is connected to an input port in order to read the required value/values.
3. If the database proxy is connected to an output port the call to the database proxy is made after the component's call to write/update the database.

Phase 3: Un-initialize

1. When the task has completed its execution, `DBUninit()` is called.
2. `DBUninit()` un-initializes the database connections in all tasks.

6. Implementation

To demonstrate the practicability of database proxies and as a proof of concept, we have implemented our approach. Three existing tools and technologies, namely the SaveComp Component



DBProxy.h

DBProxy.c

```
int DBInit(){
 setupDbSession();
  ....
  initDB_Task_1();
  initDB_Task_2();
}
int DBUninit(){
  endDBSession();
  unInit_Task_1();
  unInit_Task_2();
}
```

Application.c

```
main(){
 ...
 DBInit();
 start Task_1();
 start Task_2();
 ...
 DBUnInit();
}
```

Task_1.c (code simplified for readability)

```
#include DBProxy.h
declare DBPointer dbp1;
int initDB_Task_1(){
  bind_DBP(...);
}
int unInit_Task_1(){
  unbind_DBP(...);
}
void hardProxy_r1(int *r1){
  readDBPInt(..., r1);
}
void hardProxy_Multi_w1_w2(
                int *w1, int *w2){
  writeDBPInt(..., w1);
  writeDBPInt(..., w2);
  setMulticol();
}
void Task_1(){
int r1, w1;
  hardProxy_r1(&r1);
  call_Component_C1(r1);
  ...
  call_Component_C2(...);
  hardProxy_Multi_w1(&w1);
}
```

Task_2.c (code simplified for readability)

```
#include DBProxy.h
Session sess;
Statement stmnt;

int initDB_Task_2(){
  BeginSession(sess, ...);
  BeginStatement(sess,...,&stmt);
}
int unInit_Task_2(){
  EndStatement(&stmnt);
  EndSession(&sess);
}
void softProxy_r2(Table_Mode *log){
  Fetch(stmnt);
  GetInt(stmt, log[0].read);
  Fetch(stmnt);
  GetInt(stmt, log[1].read);
}
void Task_2(){
Table_Mode *log;
  softProxy_r2(log);
  call_Component_C3(log);
}
```

Fig. 11. Hard and soft proxy glue-code generation example.

Technology (SaveCCT) (Åkerholm et al., 2007), Mimer Real-Time edition (Mimer RT) (Mimer, 2011) and the embedded data commander (EDC) (Hjertström et al., 2009), have been used to manage the different parts of the development. A brief introduction and the role of these tools and technologies are presented in the following three parts of this section. In the last two parts, presents our development framework and discuss the predictability of our implementation.

6.1. SaveCCT component technology

The SaveComp Component Technology (SaveCCT) (Åkerholm et al., 2007) distinguishes between manual design, automated activities, and execution. The developer can create his/her application in the graphical tool Save Integrated Development Environment (Save-IDE). Automated synthesis activities generate code used to glue components together and group them into tasks. The tasks can then be executed on a real-time operating system. SaveCCT is intended for applications with both hard and soft real-time requirements.

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

6.2. Mimer SQL real-time edition

The Mimer SQL Real-Time Edition (Mimer RT) (Mimer, 2011) is a real-time database management system intended for applications with a mix of hard and soft real-time requirements. Mimer RT implements the database pointer interface to access real-time data in an efficient and deterministic manner. All hard real-time data access is performed in main-memory using predictable real-time algorithms. Mimer RT supports single, multirow and multicolumn database pointers, which can be flushed to persistent storage without interrupting real-time predictability of read and write operations.

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

6.3. Embedded data commander tool-suite

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

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

A database proxy definition is represented in XML. Fig. 12 shows an example of a hard proxy description using Mimer RT. The XML code is disposed as follows. (1) The id of the signal and which component it resides in. (2) The definition of type and pointer declaration. (3) The function used to bind the database pointer with a pre-compiled statement. In this example however represented by an SQL query to enhance readability. (4) The type of call to use, in this case an update call since it is a write proxy. (5) End of proxy definition.

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

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

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

Fig. 12. Hard proxy representation.

6.4. The database proxy development framework

In our implementation of the database proxy development framework (see Fig. 13), SaveCCT is used to manage the development chain from system design to target code generation. EDC is used to model and generate database definition files and database proxy description files. Mimer RT manages all database activities at run-time.

In our framework, the system architect can utilize a database as an additional design feature. If a database is included in the design, the generated *system description file* is extracted from SaveCCT to the EDC in order to perform the data modeling and generate a *database proxy descriptions file*. These files are then weaved together using the *code generator* in SaveCCT to form the C-code for the target system. A *database definition file* is also generated from the EDC to setup Mimer RT.

6.5. Predictability of implementation

For hard proxies, the generated code contains no unbounded behavior and WCET and memory usage can easily be statically

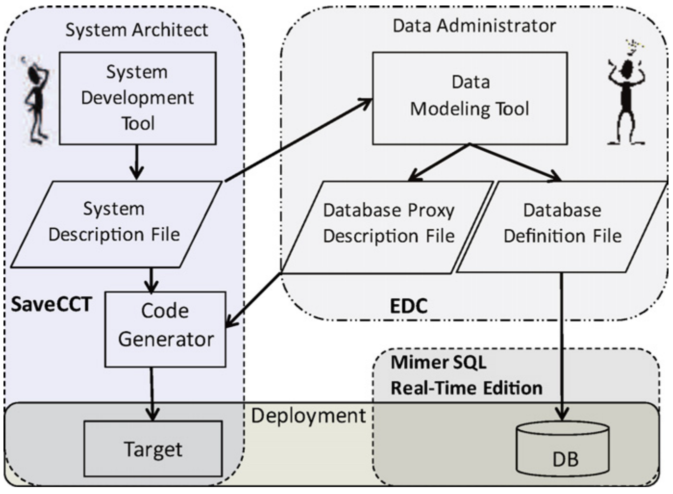


Fig. 13. Database proxy development framework.

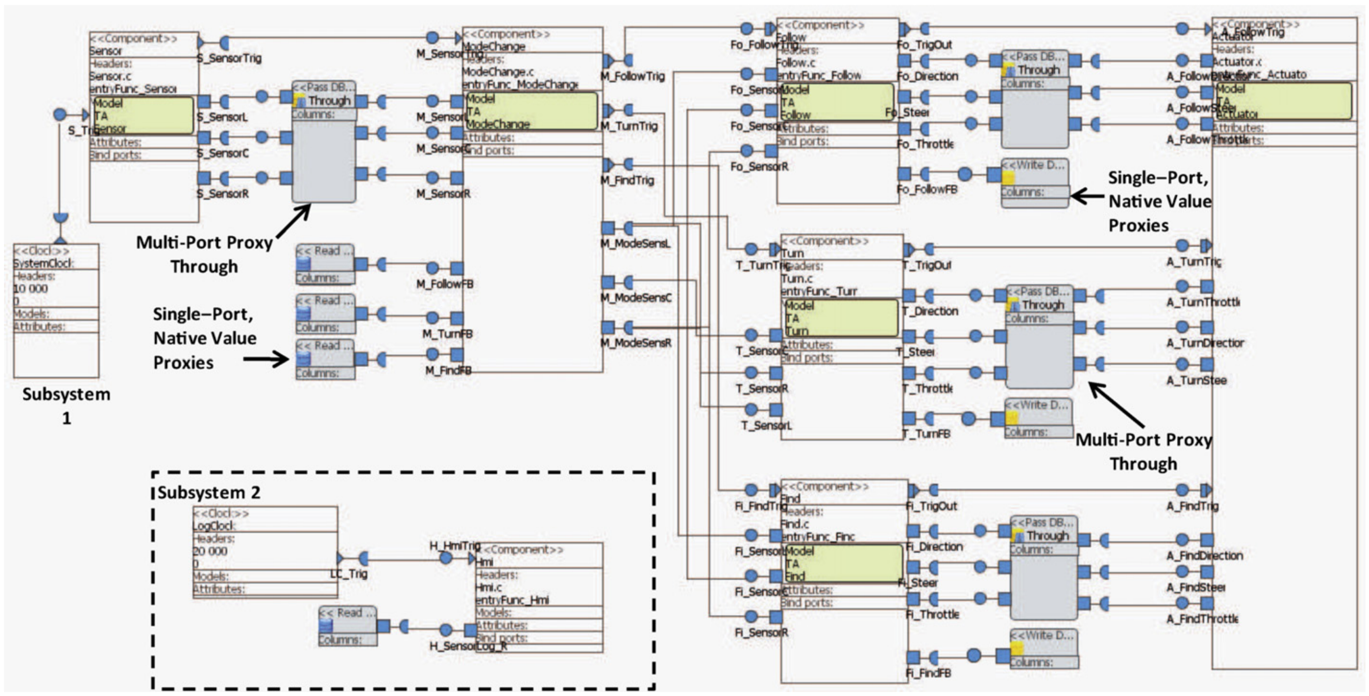


Fig. 14. Graphical representation of implementation.

bounded (although such analysis is beyond the scope of this paper). Also, the database-pointer interface of Mimer RT provides the same functions that has been proven temporally and spatially predictable within the COMET project (Nyström et al., 2004). Thus, our implementation is suitable for use in hard-real time systems.

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

7. Performance evaluation

This section presents the results of a performance evaluation where we have implemented an embedded control system and measured execution times and memory overheads. The aim of the evaluation is to measure if the database proxies will have an impact on the observed worst- or average-case execution time and how it will affect memory consumption of the system compared to using internal data structures. Two separate implementation configurations are evaluated. (1) Using only single-port, native values database proxies and (2) a more complex configuration which includes logging and a mix of single-port, native value database proxies and multi-port proxy through.

7.1. The application

To evaluate our approach, an application that includes two subsystems and two configurations has been implemented using the Save-IDE. The implementation is done according to Fig. 14, which utilizes a mix of both internal data structures and an RTDBMS. The application consists of seven components and simulates a truck that first follows a line. At the end of the line, the truck turns for a certain amount of time until it finds the line and starts following it again (see Fig. 15).

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

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

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

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

In order to measure the impact of introducing database proxies in a CBSE system under typical workload conditions, a standard worst case execution time benchmark code called *ndes* has been used as workload in the control components. The workload is a part of a collection of benchmark codes used by different research groups and tool vendors around the world to mimic the behavior of a typical embedded system (The Worst-Case, 2011).

7.2. Benchmarking setup

We have conducted a performance evaluation with four different implementations variants, and the above stated two configurations of the truck application. Each implementation is

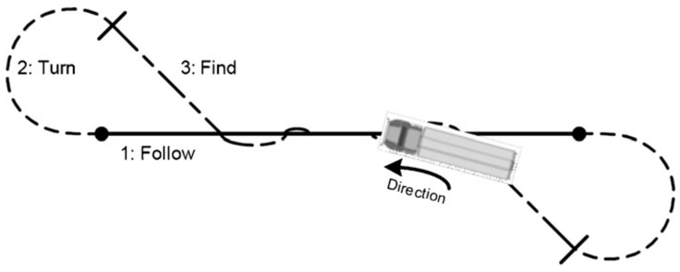
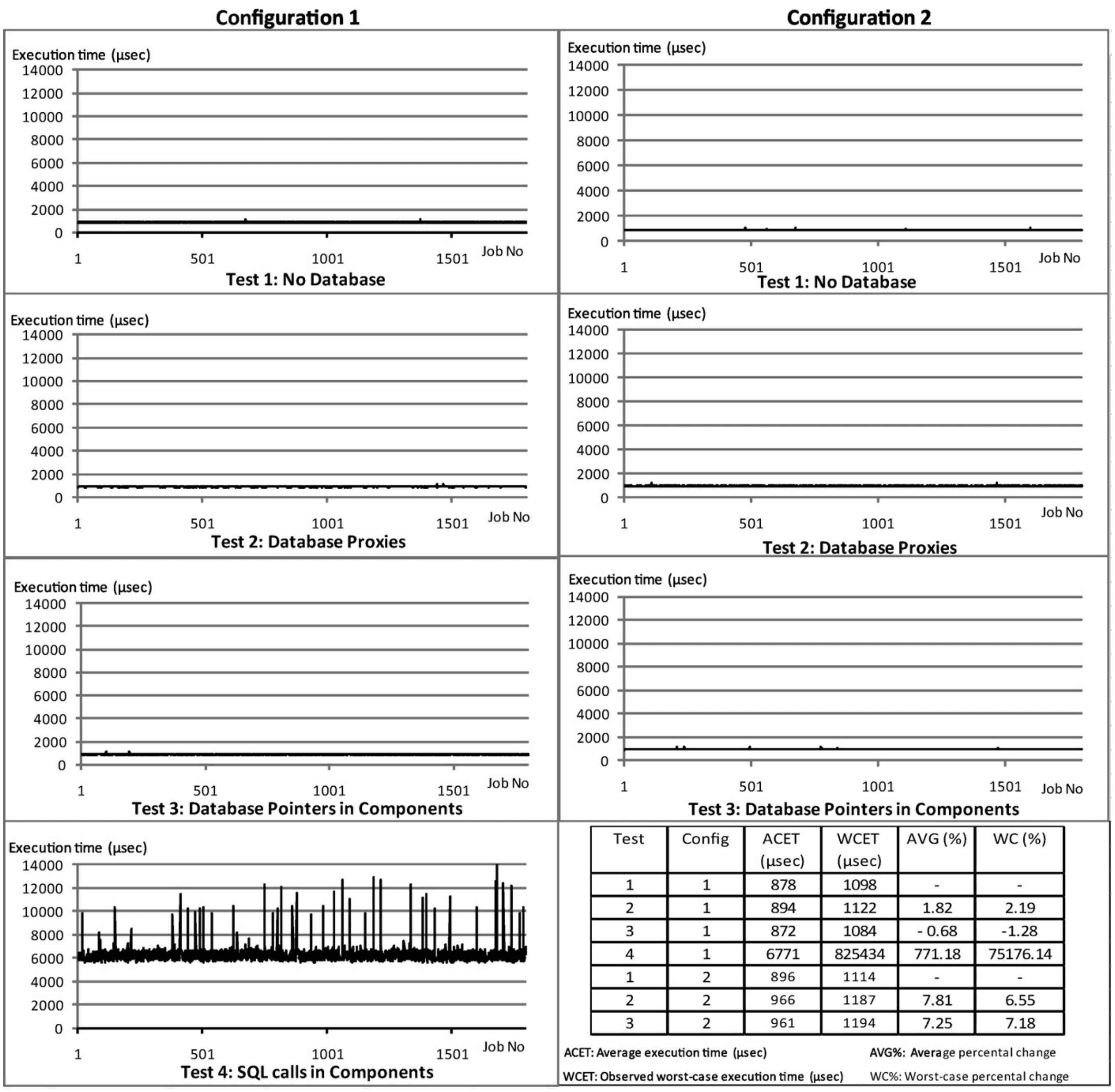


Fig. 15. Truck application.



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

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

Fig. 16. Evaluation results from configurations 1 and 2.

evaluated using both configurations, except for the usage of regular SQL in configuration 2. In this evaluation, the result from the usage of SQL in configuration 1 covers our interest by showing that it is not a predictable solution.

The tests have been performed on a Hitachi SH-4 series processor (Hitachi, 2011) with VxWorks v6 (VxWorks, 2011) as real-time operating system. Furthermore, Mimer RT, SaveCCT and EDC have been used throughout the implementation. The descriptions of the four implementations (shown in Fig. 16), are as follows:

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

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

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

Test 4 An implementation using database aware components with access to a non real-time database from within the component using regular SQL queries without hard real-time

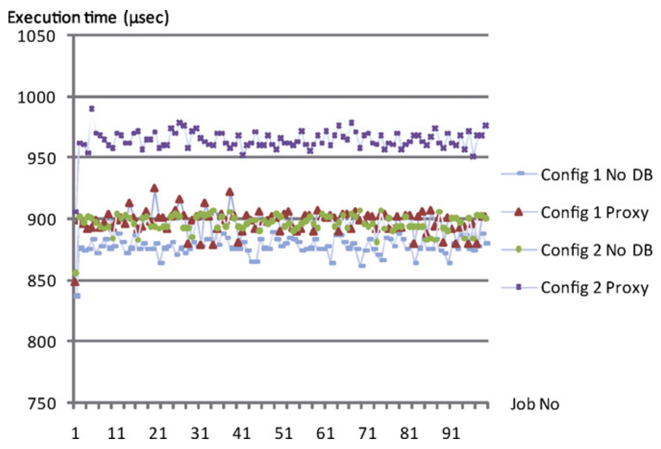


Fig. 17. Execution time evaluation results for 100 executions.

performance. The components are generated by Save-IDE. Thus, the SQL queries inside the components have been hand coded. The DBMS manages protection of shared data.

7.3. Proxy real-time performance results

Fig. 16 shows the result of the execution times for 1800 executions of the hard real-time control applications for the four test-cases explained in Section 7.2 and the two configurations explained in Section 7.1.

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

The table in the lower right hand corner of Fig. 16 shows the evaluation results for each test and configuration. The change of the Average Case Execution Time (ACET) and observed Worst Case Execution Time (WCET) in the two rightmost columns shows the change in percent, compared to our baseline, **Test 1**.

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

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

Fig. 17 shows more detailed information of the first 100 executions for both configurations of **Test 1** and **Test 2**. The evenness of the results clearly illustrates that the usage of database proxies in combination with an RTDBMS such as Mimer RT is predictable, and the amount of overhead in average and worst-case execution time is limited. Furthermore, these results confirm our predictability of implementation discussion in Section 6.5.

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

7.4. Memory consumption results

Table 2 shows how the client code size changes when using different data management methods. As can be seen in the table, integrating a real-time database client with the calls hand coded in the component code introduces 1.99% extra code size. By using database proxies that have been automatically generated, the code size grows with as little as 2.06%.

Table 2
Application code size.

Access method	Code size	Change (%)
No database	653 512 bytes	–
Database pointers	666 564 bytes	1.99
Database proxies	666 988 bytes	2.06

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

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

8. Conclusions

This paper presents the database proxy approach which enables an integration of real-time database management systems into a component-based software engineering framework. While maintaining strict component encapsulation, we achieve benefits such as the possibility to access data via standard SQL interfaces, concurrency-control, temporal consistency, and transaction management. In addition, a new possibility to use dynamic run-time queries to aid in logging, diagnostics and monitoring is introduced.

The motivation for our approach stems from observations of industrial practices and documented needs for a standardized and overall data model and management system to deal with the distributed and uncoordinated data in these complex systems. Furthermore, it is clearly stated that the ad hoc/reinvented management of data as well as individual solutions using internal data structures, can lead to concurrency- and inconsistency problems and decreases maintainability, extensibility and flexibility of the system (Schulze et al., 2009; Hjertström et al., 2008; Pretschner et al., 2007; Saake et al., 2009).

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

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

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

Acknowledgement

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

References

Åkerholm, M., Carlson, J., Fredriksson, J., Hansson, H., Håkansson, J., Möller, A., Pettersson, P., Tivoli, M., 2007. The save approach to component-based development of vehicular systems. Journal of Systems and Software 80 (5), 655–667.

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

Albert, A., 2004. Comparison of Event-Triggered and Time-Triggered Concepts with Regard to Distributed Control Systems, 235–252.

Arcticus Systems, Rubus ICE, www.arcticus-systems.com/html/prod-rubus-ice.html.

AUTOSAR Open Systems Architecture, 2011. http://www.autosar.org.

Brooks, R.R., Sander, S., Deng, J., Taiber, J., 2008. Automotive system security: challenges and state-of-the-art. In: Proceedings of the 4th Annual Workshop on Cyber Security and Information Intelligence Research: Developing Strategies to Meet the Cyber Security and Information Intelligence Challenges Ahead, ACM, pp. 26:1–26:3.

Broy, M., 2006. Challenges in automotive software engineering. In: ICSE '06: Proceedings of the 28th International Conference on Software Engineering, ACM, pp. 33–42.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., Sommerlad, P., Stal, M., 1996. Pattern-Oriented Software Architecture: A System of Patterns, vol. 1. John Wiley and Sons.

Car 2 Car communication consortium, 2011. http://www.car-to-car.org.

Crnkovic, I., Larsson, M., 2002. Building Reliable Component-Based Software Systems. Artech House.

Delgado, N., Gates, A.Q., Roach, S., 2004. A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Transactions on Software Engineering 30, 859–872.

Fürst, S., 2010. Challenges in the design of automotive software. In: Proceedings of the Conference on Design, Automation and Test in Europe, DATE 10, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, pp. 256–258.

Gereon Weiss, M.Z., Eilers, D., 2011. Towards Automotive Embedded Systems with Self-X Properties. InTech.

Grimm, K., 2003. Software technology in an automotive company – major challenges. In: Software Engineering, International Conference on Software Engineering, p. 498.

Härder, T., 2005. DBMS architecture – still an open problem. In: Proc. Datenbanksysteme in Business, Technologie Und Web (Btw 2005). Springer, pp. 2–28.

Hitachi SH-4 32-bit RISC CPU Core Family, 2011. http://www.hitachi.com/.

Hjertström, A., Nyström, D., Sjödin, M., 2010. Database proxies for component-based real-time systems. In: 22nd Euromicro Conference on Real-Time Systems. IEEE Computer Society, pp. 79–89, doi:http://doi.ieeecomputersociety.org/10.1109/ECRTS.2010.26.

Hjertström, A., Nyström, D., Nolin, M., Land, R., 2008. Design-time management of run-time data in industrial embedded real-time systems development. In: Proceedings of 13th IEEE International Conference on Emerging Technologies and Factory Automation, Germany, pp. 1285–1293.

Hjertström, A., Nyström, D., Sjödin, M., 2009. A data-entity approach for component-based real-time embedded systems development. In: 14th IEEE International Conference on Emerging Technology and Factory Automation, IEEE Press, pp. 170–177.

ISO SQL 2008 Standard, 2009. Defines the SQL Language, http://www.iso.org/iso/home.htm.

Lau, K.-K., Taweel, F.M., 2006. Towards encapsulating data in component-based software systems. In: CBSE, pp. 376–384.

Lau, K.-K., Taweel, F.,2007. Data encapsulation in software components. In: Proc. 10th Int. Symp. on Component-based Software Engineering, LNCS, vol. 4608. Springer-Verlag, pp. 1–16.

Liggesmeyer, P., Trapp, M., 2009. Trends in embedded software engineering. IEEE Software 26, 19–25.

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

Nyström, D., Tešanović, A., Norström, C., Hansson, J., Bånkestad, N.-E., 2002. Data management issues in vehicle control systems: a case study. In: Proceedings of the 14th Euromicro Conference on Real-Time Systems, IEEE Computer Society, pp. 249–256.

Nyström, D., Tešanović, A., Norström, C., Hansson, J., 2003. Database pointers: a predictable way of manipulating hot data in hard real-time systems. In: Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications, pp. 623–634.

Nyström, D., Tešanović, A., Nolin, M., Norström, C., Hansson, J., 2004. COMET: a component-based real-time database for automotive systems. In: Proceedings of the Workshop on Software Engineering for Automotive Systems, IEEE Computer Society, pp. 1–8.

Nyström, D., Nolin, M., Tešanović, A., Norström, C., Hansson, J., 2004. Pessimistic concurrency control and versioning to support database pointers in real-time databases. In: Proceedings of the 16th Euromicro Conference on Real-Time Systems, IEEE Computer Society, pp. 261–270.

OMG UML, 2011. The Unified Modeling Language UML, http://www.uml.org/.

Polyhedra In-Memory Database, 2011, September. http://www.enea.com.

Pretschner, A., Broy, M., Kruger, I.H., Stauner, T., 2007. Software engineering for automotive systems: a roadmap. Future of Software Engineering, 55–71.

Ramamritham, K., Son, S.H., Dipippo, L.C., 2004. Real-time databases and data services. Journal of Real-Time Systems 28 (2/3), 179–215.

Saake, G., Rosenmüller, M., Siegmund, N., Kästner, C., Leich, T., 2009. Downsizing data management for embedded systems. Egyptian Computer Science Journal, 1–13.

Schulze, S., Pukall, M., Saake, G., Hoppe, T., Dittmann, J., 2009. On the need of data management in automotive systems. In: Freytag, J.C., Ruf, T., Lehner, W., Vossen, G. (Eds.), BTW Vol. 144 of LNI GI. , pp. 217–226.

Shan Chen, P.P., 1976. The entity-relationship model: toward a unified view of data. ACM Transactions on Database Systems 1, 9–36.

Stankovic, J.A., Zhao, W., 1988. On real-time transactions. SIGMOD Record 17, 4–18.

Szyperski, C., 1997. Component Software Beyond Object-Oriented Programming. Addison-Wesley Professional.

The Worst-Case Execution Time (WCET) Analysis Project, 2011. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

van Ommering, R., 2000. The Koala component model for consumer electronics software. In: Computer, IEEE Computer Society, Ch. The Koala Component Model, pp. 78–85.

VxWorks Real-Time Operating System, by Wind River, 2011. http://www.windriver.com/.