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

Andreas Hjertström, Dag Nyström, Mikael Sjödin
Mälardalen Real-Time Research Centre, Västerås, Sweden
{andreas.hjertstrom, dag.nystrom, mikael.sjodin}@mdh.se

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.

1 Introduction

To enable a successful integration of a Real-Time DataBase Management System (RTDBMS) [1, 3, 13, 17] into a Component-Based Software Engineering (CBSE) [8, 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 [18], 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 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 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.

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

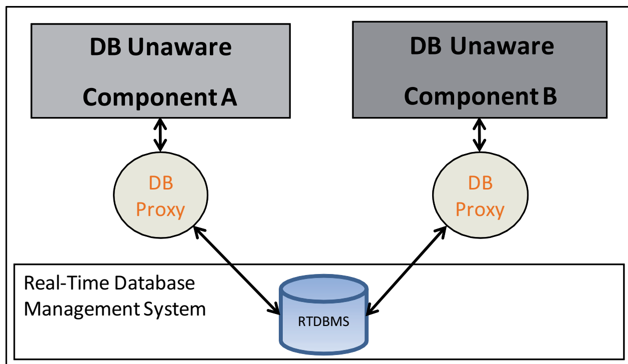

Figure 1. Database Proxies Connecting Components to an RTDBMS

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 [10, 15].
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 2, we present background and motivation for the approach. We also present the specific problems that our approach addresses. In section 3, we present the system model. Section 4 gives a detailed description of the database proxy. Further, in section 5, we illustrate our ideas with an implementation example. Finally, we show a performance and real-time predictability evaluation in section 6 and concluded the paper in section 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) [4] 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 [18]. 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 underlying 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 [19, 8]. 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.

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.

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/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.

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.

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

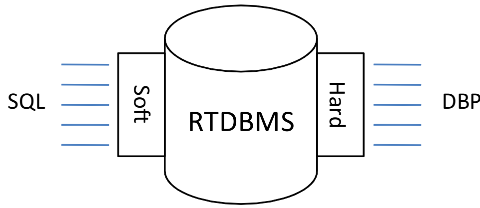

Figure 2. RTDBMS Architectural Overview

separate interfaces. Figure 2 illustrates an RTDBMS which has a soft interface that utilize a regular SQL [7] query interface to enable flexible access from soft real-time tasks. For hard real-time transactions, a database pointer (dbp) [16] 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 [16] is used. 2V-DBP allows hard and soft transactions to share data independent of each otherFigure 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 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 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).

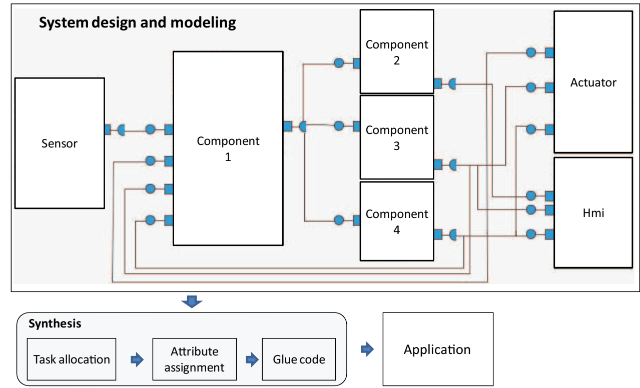

Figure 4. System Design and Modeling

3.2 System Design and Modeling

In application design and modelling 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 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.

3.3 Extended System Design and Modeling

We complement the classical architectural view, presented in section 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 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 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.

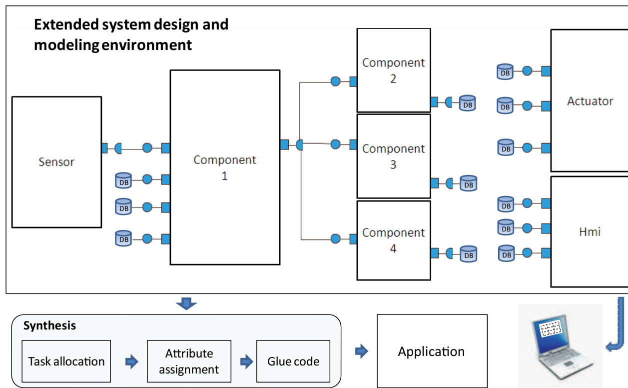

Figure 5. Database View of Application Model

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.

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 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.

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).

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 [16] 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.

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 RT-DBMS 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*.

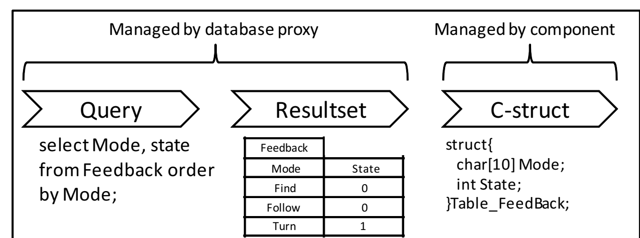

Figure 6. Description of Table Type

A *TABLE* is a realization of a relational table using standard C-types. Figure 6 illustrates the three steps from query to resultset and C-struct. At run-time 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 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.

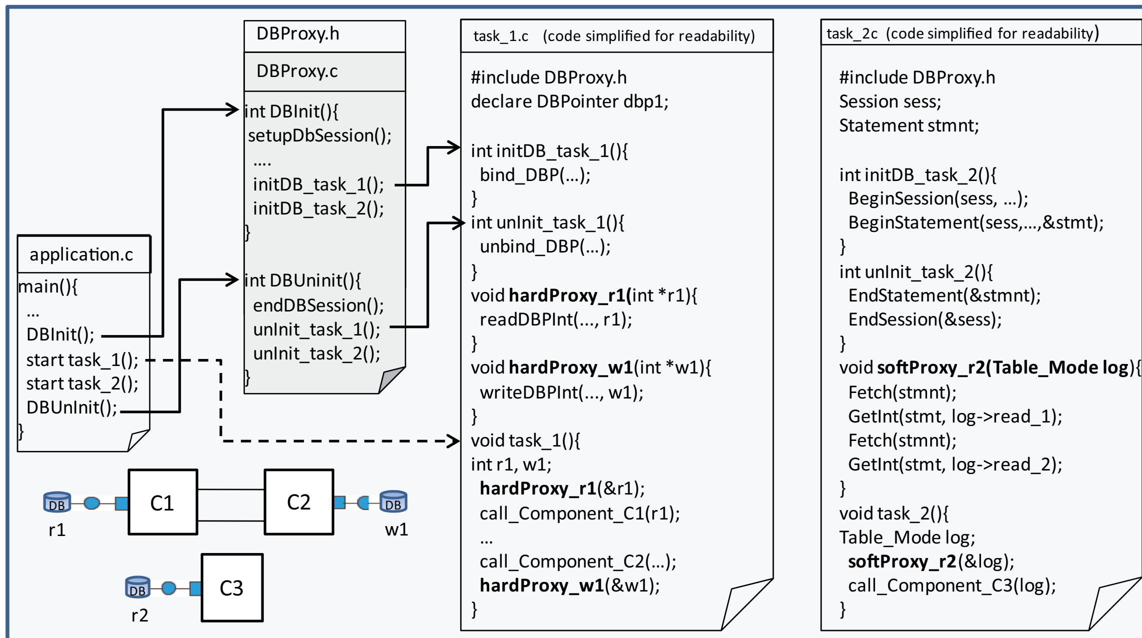

Figure 7. Hard and Soft Proxy Glue-Code Generation Example

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.

4.3 Proxy Implementation Description

Figure 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 has been simplified for readability. The flow pointed out by the arrows in figure 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.

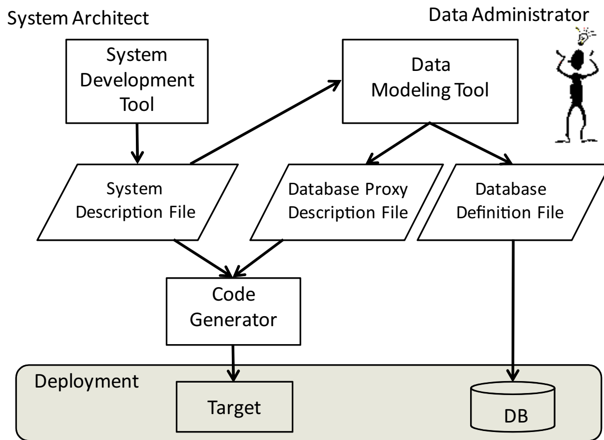

Figure 8. The Approach

5 Implementation

In our approach, we have extended the CBSE system development framework to include database proxies and data modeling, see figure 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 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 8. Save-IDE [2], Mimer Real-Time edition (MimerRT) [14] and the Data Entity Navigator (DEN) [11].

5.1 Mimer Real-Time Edition

The Mimer SQL Real-Time Edition (Mimer RT) [14] 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 [16] to access individual data elements in an efficient and deterministic manner. For soft real-time database management, standard SQL [7] queries are used. To achieve database consistency without jeopardizing the real-time requirements the 2V-DBP concurrency control algorithm presented in section 3.1

5.2 SaveCCT Real-Time Component Technology

The SaveComp Component Technology (SaveCCT) [2] 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.

5.3 Embedded Data Commander Tool-Suite

The *Embedded Data Commander* (EDC) is a tool-suite that implements the data entity approach [11] for the ProSys component-model [5]. 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 [2].
The tool-suite has been extended with:

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

```
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 9. Hard Proxy Representation

A database proxy definition is represented in XML. Figure 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.

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.

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 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*.

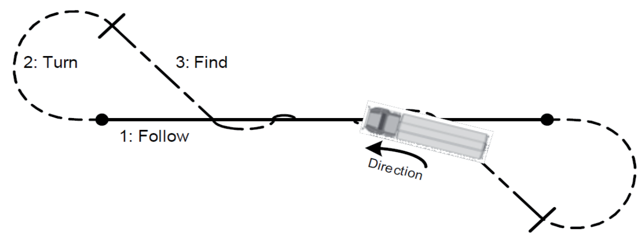

Figure 10. Truck Application

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 [20] 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.

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 [9] with VxWorks [21] as real-time operating system. The hard real-time tasks are executed 1800 times.

The four evaluated implementations shown in figure 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.

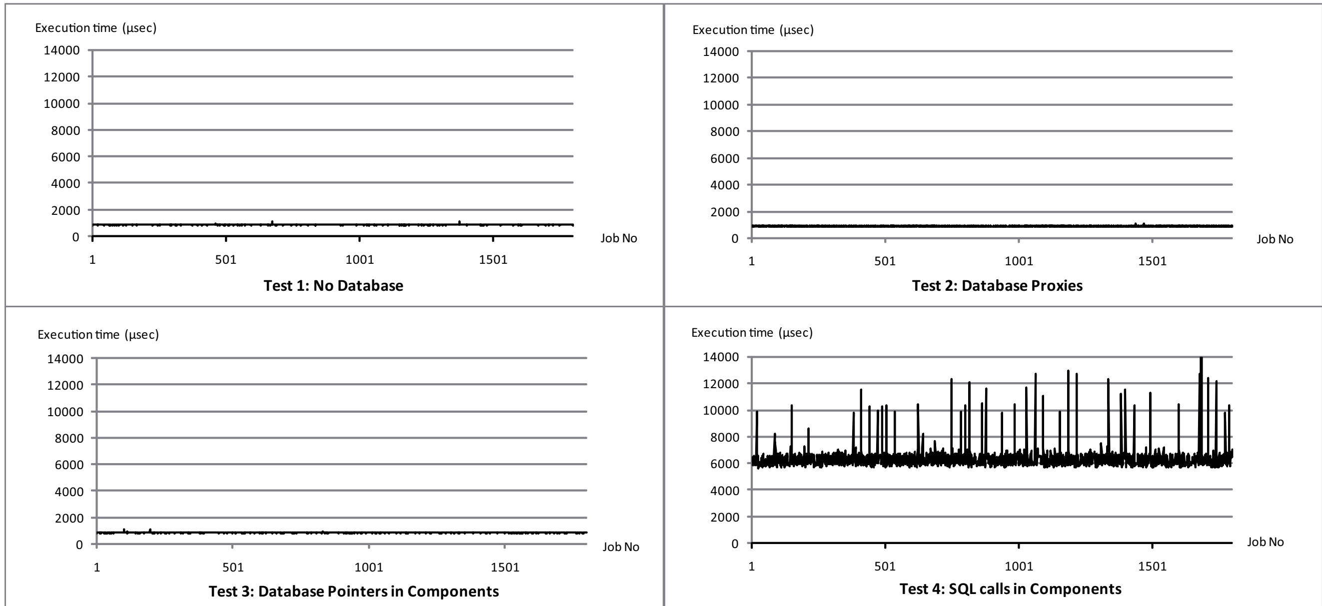

Figure 11. Evaluation Results

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 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 2. Application Code Size

6.3 Real-Time Performance Results

Figure 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 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.

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.

6.4 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 handcoded 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 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 RTDBMSes, 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 [10, 18].

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.

References

[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] M. Åkerholm, J. Carlson, J. Fredriksson, H. Hansson, J. Håkansson, A. Möller, P. Pettersson, and M. Tivoli. The Save Approach to Component-Based Development of Vehicular Systems. *Journal of Systems and Software*, 2006.

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

[5] T. Bures, J. Carlson, I. Crnkovic, S. Sentilles, and A. Vulgarakis. ProCom - the Progress Component Model Reference Manual. Technical Report, Mälardalen University, 2008.

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

[7] S. Cannan and G. Otten. *SQL - The Standard Handbook*. MacGraw-Hill International, 1993.

[8] I. Crnkovic and M. Larsson. Building reliable component-based software systems, 2002.

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

[10] A. Hjertström, D. Nyström, M. Nolin, and R. 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.

[11] A. Hjertström, D. Nyström, and M. 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.

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

[13] 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.

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

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

[16] D. Nyström, M. Nolin, A. Tešanović, C. Norström, and J. 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.

[17] K. Ramamritham, S. H. Son, and L. C. Dipippo. Real-Time Databases and Data Services. *Journal of Real-Time Systems*, 28(2/3):179–215, November/December 2004.

[18] S. Schulze, M. Pukall, G. Saake, T. Hoppe, and J. Dittmann. On the need of data management in automotive systems. In J. C. Freytag, T. Ruf, W. Lehner, and G. Vossen, editors, *BTW*, volume 144 of *LNI*, pages 217–226. GI, 2009.

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

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

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