Database Proxies for Component-Based Real-Time Systems

Andreas Hjertström
Mälardalen Real-Time Research Centre
Västerås, Sweden
Email: andreas.hjertstrom@mdh.se

Dag Nyström
Mimer Information Technology AB
Uppsala, Sweden
Email: dag.nystrom@mimer.se

Mikael Sjödin
Mälardalen Real-Time Research Centre
Västerås, Sweden
Email: mikael.sjodin@mdh.se

***Abstract*—We introduce the concept of database proxies capable of mitigating the gap between two disjoint productivity-enhancing techniques: Component Based Software Engineering (CBSE) and Real-Time Database Management Systems (RT-DBMS). The coexistence of the two techniques 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 predictable global data sharing. Database proxies decouple components from an underlying database residing in the component framework. This enables components to remain encapsulated and reusable, while providing temporally predictable access to data maintained in a database. We specifically target embedded systems with a subset of functionality with real-time requirements. Our implementation results show that the above benefits do not come at the expense of run-time overheads or less accurate timing predictions.**

***Keywords*-real-time; database; component; proxy; embedded systems;**

I. Introduction

To enable a successful integration of a Real-Time DataBase Management System (RTDBMS) [1], [2], [3], [4] into a Component-Based Software Engineering (CBSE) [5], [6] setting we present a new concept: *database proxies*. A database proxy allows system developers to employ the full potential of both CBSE and RTDBMS, which both aim at reduce complexity and enhance productivity when developing embedded real-time 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 interfaces, providing advanced access control mechanisms. This implies that potentially unsafe software, such as third party software, can be granted access to data in a controlled manner. Furthermore, RTDBMSs significantly cuts time to market by providing high-level query languages, supporting logging, diagnostics, monitoring, and efficient data modeling [7].

Although both CBSE and RTDBMS aim to reduce complexity, 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 built-in assumptions 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, components need to be decoupled from the RTDBMS and the underlying database schema. We present the concept of database proxies to achieve this decoupling of the database by integrating the RTDBMS 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 enable for components to interact with an RTDBMS using their normal interfaces.

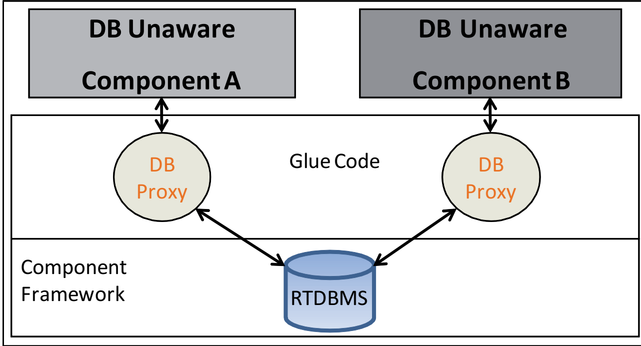

Figure 1. Database Proxies Connecting Components to an RTDBMS

This is possible since the coupling, e.g. database proxy, between the component and the RTDBMS is embedded in the underlying glue code. This glue code is 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.

In this paper, we introduce database proxies that can be automatically generated such that:

1) Components can gain access to an RTDBMS in the component framework with maintained encapsulation and decoupling.
2) Components can have efficient and predictable access to hard real-time data.
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.

The rest of this paper is structured as follows; in section II, we present background and motivation for the approach. We also present the specific problems that our approach addresses. In section III, we present the system model. Section IV gives a detailed description of the database proxy. Further, in section V, we illustrate our ideas with an implementation example. Finally, we show a performance and real-time predictability evaluation in section VI and conclude the paper in section VII.

II. Background and Motivation

The characteristics of today's embedded systems are changing. Many embedded systems are becoming increasingly complex and costly to develop with large amounts of functions and data to manage [8], [9]. Modern techniques within model driven development and component-based software engineering (CBSE) is widely used to, for instance, reduce complexity and increase the understanding and reusability of software functions [10], [11], [12]. However, there is also a need for more flexible, reliable and secure data management techniques to coordinate data [13], [14], such as those provided by an RTDBMS [7].

A desirable step would be to integrate an RTDBMS into a CBSE-technique, to improve data management by providing controlled and flexible access to shared system-data. However, to achieve a successful integration of an RTDBMS into a component-based system, a number of CBSE requirements have to be fulfilled.

In CBSE, a component encapsulates functionality and only reveals its 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 database and its structure, or if a database is present or not. Furthermore, a database unaware component does not introduce any side effects such as database communication outside the component's specified interface, thus retaining the reusability of the component.

The usage of an RTDBMS in a CBSE framework should not introduce any side effects that violate basic CBSE principles [6], [15]. For the purpose of this paper, we define a component to be side effect free, with respect to the introduction of an RTDBMS, if it is:

- **Reusable:** A component can still be used in another setting, with or without an RTDBMS.
- **Substitutable:** A component should be substitutable by a component implementing the same interface; regardless if a RTDBMS is used or not.
- **Without implicit dependencies:** The usage of an RTDBMS 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.

A. Solution Requirements

This section identifies a number of requirements, R1-R3, that 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 requirements of the system should not be compromised by using a RTDBMS.

B. Related Work and DBMS Mechanisms

We have not been able to identify any research which explicitly aims at combining CBSE and database management systems (DBMS). There is however, research on how to manage data within CBSE. One direction has been to have data flow and data access, completely encapsulated within connectors. In this way, components only encapsulate computation [16]. Another approach is to encapsulate data inside components in order to achieve encapsulated reusable building blocks, where data is included [17]. However, these approaches are neither developed nor optimized for embedded real-time systems nor do they decouple components from the underlying data management.

There exist mechanisms within the DBMS community that could be used in order to fulfill some of the requirements stated in Section II-A, by decoupling the DBMS from the components. These mechanisms aim to simplify the database access by hiding some of the underlying complexity as well as making the access to the DBMS more efficient. The standardized interface-language SQL defines the following mechanisms [18]:

- **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 reside in the component. A component using these mechanisms is therefore no longer generic or reusable. (Violation of R1-R2.)
3) The requirements expressed by the components interface does not reflect the components internal database dependency. (Violation of R2.)
4) These mechanisms are not intended for real-time performance (typically only non-real time DBMS support is available), e.g., the usage of these mechanisms alone would be a violation of R3.

III. 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).

We consider a system where functionality is divided into the following classes of tasks:
Hard real-time tasks, typically executed at high frequency. Hard real-time tasks uses hard transactions to read and write single values from sensors/actuators and execute real-time control loops. Hard real-time tasks cannot manage complex data structures. This limitation however, is fairly small in practice, since hard real-time components often are static, communicating with fairly simple data structures. When a database is used, hard real-time tasks require predictable access to data elements.
Soft real-time tasks, often running at a lower frequency controlling less critical functionality. Soft real-time tasks uses soft transactions to read and write dynamic and complex data structures typically to present statistical information, logging or used as a gateway for service access to the system by technicians in order to perform system 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. Figure 2 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 [19] interface is used to enable the application to access individual data elements in the database with hard real-time performance.

A. Database Pointers

Database pointers are application pointer variables that are used for real-time access to individual data elements in a real-time database, see Figure 3. The figure shows an example of an I/O task that periodically reads a sensor and

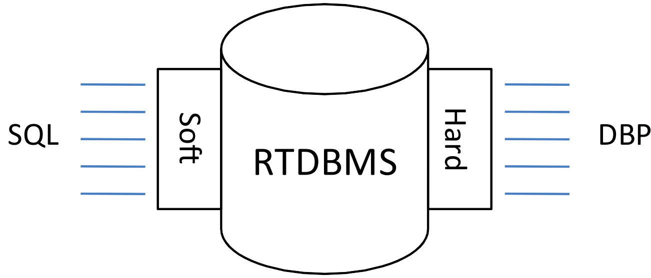

Figure 2. RTDBMS Architectural Overview

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 in real-time.

During the initialization part (lines 2 to 4) the database pointer is created and bound to a data element in the database. This is performed by the `bind` function which calls the database server that executes the query. However, instead of returning the actual value of the query, in this case the oil temperature, the server returns a real-time handle which can be used by the real-time application for predictable direct data access.

During the control part of the task in Figure 3, the `write` function writes the new value `temp` to the database pointer. During this operation, only a few lines of sequential code that performs type checking, synchronization with other accesses with the same data element, and writing of the data are executed. The cost and predictability of this execution is, as shown in the performance evaluation in section VI, 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) [19] uses a 2-version versioning algorithm that guarantees that database pointers will never be aborted or subjected to unpredictable blocking, while allowing soft real-time transactions to concurrently access the data.

B. System Architecture and Modeling

In the application design and modeling we assume a pipe-and-filter [5] 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 component-based system design and modeling architecture. A set of components are connected through ports and connections to form the system. From the modeled system, glue code is automatically generated by mapping components onto tasks to form the application.

```
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 I/O task that uses a database pointer

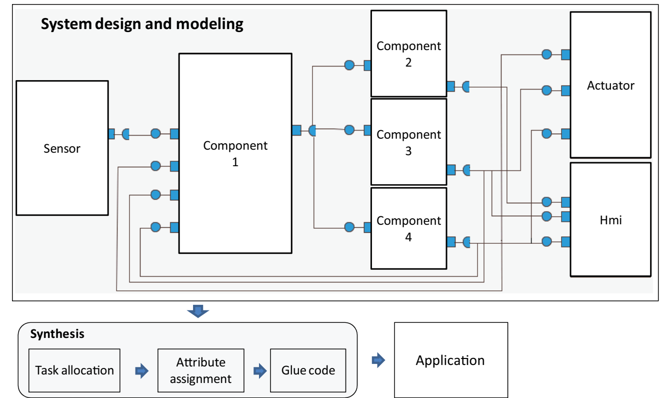

Figure 4. System Design and Modeling

IV. 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, see Figure 1. A database proxy contains the following constituent parts:

- **Initialization code** that connects to the RTDBMS and initiate database accesses (database pointers as well as SQL queries).
- **Data translation code** that performs database accesses (database reads and updates) and translates the result to match the component ports.
- **Uninitialization code** that closes database accesses and disconnects from the RTDBMS.

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

From an RTDBMS perspective, decoupling is achieved by encapsulating the underlying database schema from the components, only allowing data access to database proxies through pre-compiled statements, views, stored procedures or database pointers. As a result, database proxies target requirements **R1 & R2** presented in Section II-A, 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 additional side-effects are introduced. (**Targets R2**)

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

A. Hard Real-Time Database Proxies

Hard proxies are intended for hard real-time components, which need efficient and deterministic access to individual 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. Further usages include hard real-time data that is accessed using external applications, such as via service tools or third party applications.

Since hard real-time components manage hard real-time data, hard proxies emphasize predictable and efficient data access. Hard proxies are therefore implemented using database pointers. By using database pointers, that provide hard real-time guaranties, to access individual 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.
- translates native data types only, thereby providing predictable data translation.

That a hard proxy only translates native data types implies that no unpredictable type conversions or translation of complex data types that require unbounded iterations are allowed.

B. Soft Real-Time Database Proxies

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

A soft real-time database proxy;

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

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

Consider the following example (see Figure 5):

- 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, represented by a C-struct containing the members `SubSystem` and `Temp`.
- The type of the port in the component is then set to a `(Table_SystemTemp *)`.
- The database proxy is then implemented using a query (in the form of a pre-compiled statement) that matches the data in the `TABLE`. In Figure 5, a regular SQL query is used for readability.
- The translation glue code then iterates through the result set from the database and fills the vector with the correct data.

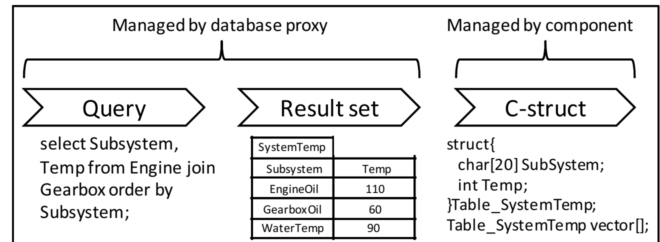

Figure 5. Description of TABLE Type

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.

C. Extended System Design and Modeling

We complement the classical architectural view, presented in section III-B, with a new additional design view, the *CBSE database-centric view*. This new view identifies which component ports are connected to data elements in an RTDBMS, illustrated in Figure 6. 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.

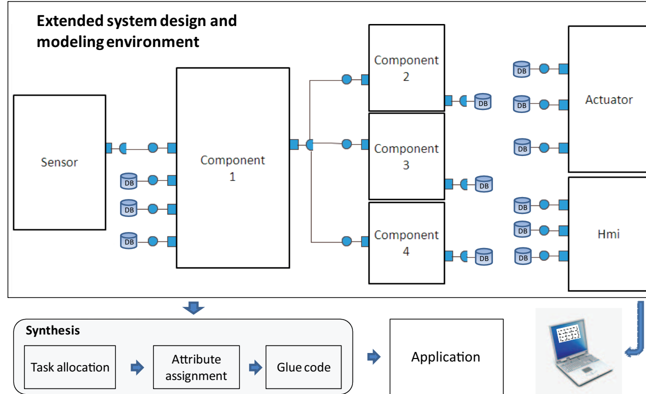

Figure 6. Database View of Application Model

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) [20].

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.

D. Database Proxy Example

Figure 7, 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 are implemented. In the lower left of the figure, two example applications are displayed.

The system is synthesized into two tasks. Task_1 is a hard real-time control task that consists of components *C1* & *C2*. Task_2 is a soft real-time HMI task that consists of component *C3*.

Task_1 implements two hard database proxies. Component *C1* uses a database proxy to read a value from the database, filters it and outputs the result to component *C2*. *C2* writes its output to the database using a database proxy.

Task_2 shows an example of a soft database proxy implementation where component *C3* reads a type *Table_Mode* $*$. 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 execution can be divided in three phases, initialize, running task and un-initialize.

Phase 1: Initialize

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

Phase 2: Task execution

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

Phase 3: Un-initialize

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

V. Implementation

To demonstrate the usefulness of database proxies and as a proof of concept, we have implemented our approach. Three existing tools and technologies, namely the SaveComp Component Technology (SaveCCT) [10], Mimer Real-Time edition (Mimer RT) [21] and the Embedded Data Commander (EDC) [22], 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, we will present our development framework and discuss the predictability of our implementation.

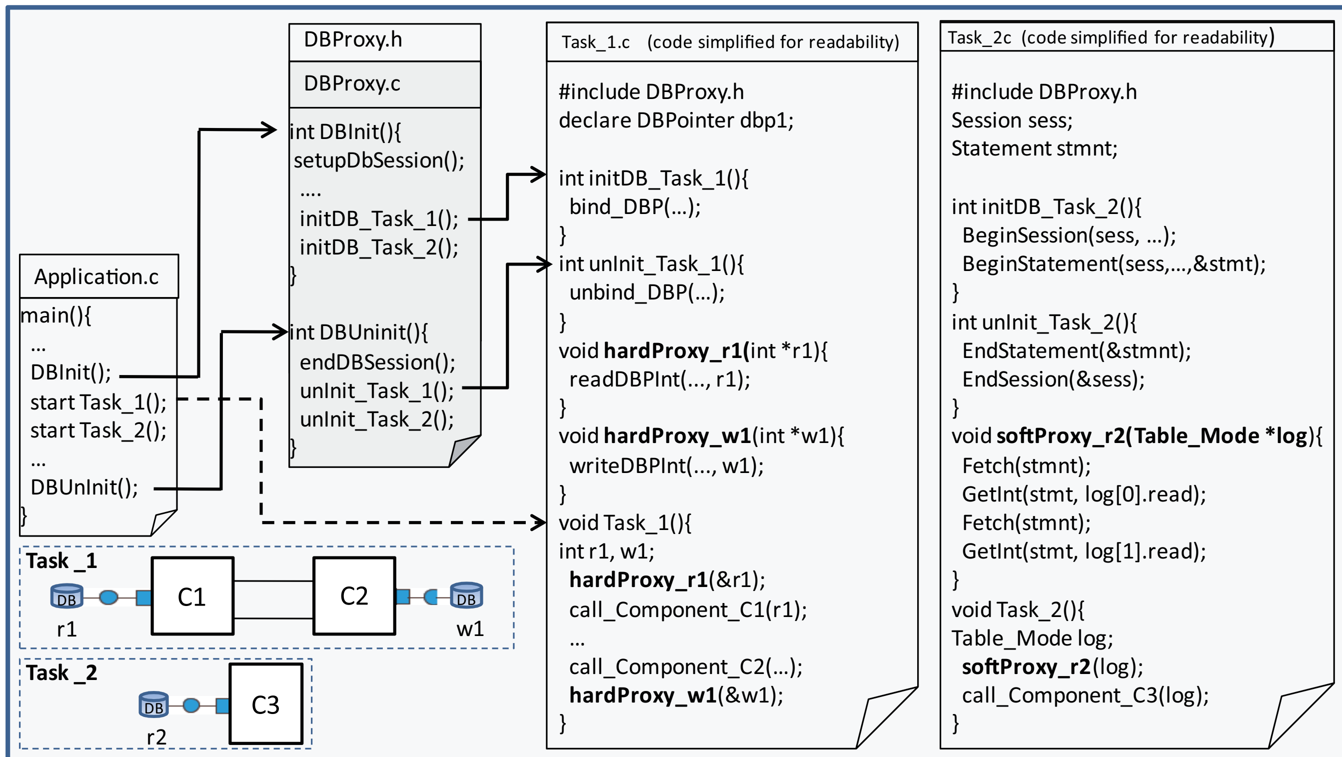

Figure 7. Hard and Soft Proxy Glue-Code Generation Example

A. SaveCCT Component Technology

The SaveComp Component Technology (SaveCCT) 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, we have extended the SaveCCT synthesis to also support database proxies.

B. Mimer SQL Real-Time Edition

The Mimer SQL Real-Time Edition (Mimer RT) 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 individual data elements in an efficient and deterministic manner. 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.

C. 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 recently been extended with new functionality to support SaveCCT real-time component technology.

Save-IDE generated description files are used by the tool in order to model the database and generate a database definition file. A database proxy description file is also generated from the model in order for Save-IDE to generate the glue code.

A database proxy definition is represented in XML. Figure 8 shows an example of a generated 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 precompiled 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>
```

Figure 8. Hard Proxy Representation

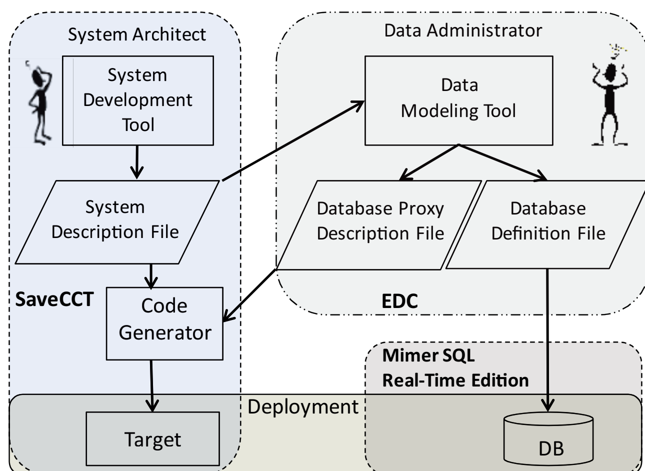

Figure 9. Database Proxy Development Framework

D. The Database Proxy Development Framework

In our implementation of the database proxy development framework (see Figure 9), 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.

E. Predictability of Implementation

For hard proxies, the generated code contains no unbounded behavior and WCET and memory usage can easily be statically bounded (although such analysis is beyond the scope of this paper). Also, the database-pointer interface of

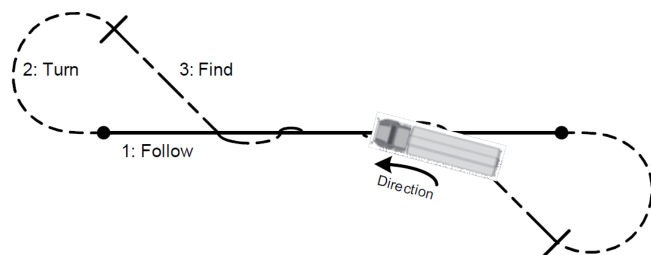

Figure 10. Truck Application

Mimer RT provides the same functions that has been proven temporally and spatially predictable within the COMET project [19]. Thus, our implementation is suitable for use in hard-real time systems.

Soft proxies do not affect the predictability properties of the system. Both soft and hard database proxies reflect the predictability and performance of the underlying database.

VI. Performance Evaluation

In this section we present 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 worst- or average-case execution time and how it will affect memory consumption of the system compared to using internal data structures.

A. The Application

To evaluate our approach, we have implemented an application using the Save-IDE that includes two subsystems. The application consists of seven components and simulates a truck that first follows a line. At the end of the line, the truck turns for a certain amount of time until it finds the line and starts following it again (see Figure 10).

The first subsystem consists of a hard real-time control loop including six components that are periodically executed every 10ms. The second subsystem consists of a soft real-time HMI component that is periodically executed every 20ms. Common for the two subsystems is that they share data that needs to be protected.

Since the task performed by the included components is quite trivial, we have added a more realistic workload in the system. As workload a standard worst case execution time benchmark code, called ndes [23], that mimics a complex embedded control application has been added to components follow, turn and find.

B. Benchmarking Setup

We have conducted a performance evaluation with four different implementations of the truck application. The tests have been performed on a Hitachi SH-4 series processor [24] with VxWorks v6 [25] as real-time operating system.

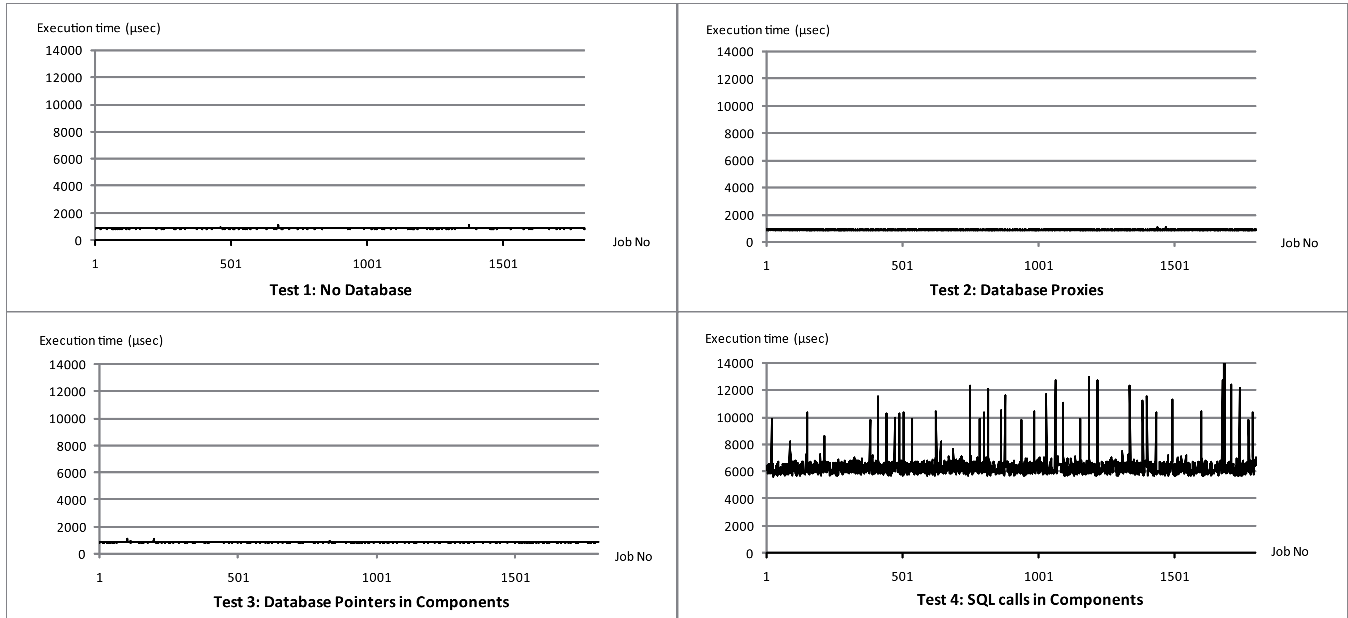

Figure 11. Evaluation Results

The descriptions of the four implementations (shown in Figure 11), are as follows:

Test 1 A baseline implementation using internal data structures without any database connection. All component 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 and has not been modified. The hard real-time subsystem utilizes hard real-time database proxies to manage access to the database. The soft real-time subsystem utilizes a soft real-time database proxy to manage access to the database. The RTDBMS manages protection of all shared data.

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

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

C. Real-Time Performance Results

Figure 11 shows the result of the execution times of the hard real-time control applications for the four test-cases. 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.

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

Legend:

AVGET Average execution time (μs)
WCET Worst-case execution time (μsec)
AVG% Average percental change
W% Worst-case percental change

Table I
APPLICATION EXECUTION TIME

Table I, 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, 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. 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 illustrates that the usage of database proxies in combination with an RTDBMS is predictable, and 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 Mimer RT compared to the regular POSIX semaphore routines used in **Test 1**.

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

Table II
APPLICATION CODE SIZE

D. Memory Consumption Results

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

Introducing a real-time database server in the system of course also introduces extra memory consumption, but embedded database servers are becoming smaller and smaller. The Mimer SQL database family that is used in this evaluation has a code footprint ranging from 273kb for the Mimer SQL Nano database server, up to 3.2Mb for the Mimer SQL Engine for enterprise systems. The RAM usage for Mimer SQL Nano is as low as 24k. 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.

VII. 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 overload 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 [7], [26].

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, we have performed a series of execution time tests on the generated C-code for a research application. These tests show that our approach only increases, 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 to real-time embedded systems development, maintenance and evolution at a minimal cost with respect to resource consumption.

ACKNOWLEDGMENT

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

REFERENCES

[1] B. Adelberg, B. Kao, and H. Garcia-Molina, "Overview of the STanford Real-time Information Processor (STRIP)," *SIGMOD Record*, vol. 25, no. 1, pp. 34–37, 1996.

[2] S. F. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring, "DeeDS Towards a Distributed and Active Real-Time Database System," *ACM SIGMOD Record*, vol. 25, 1996.

[3] J. Lindstrom, T. Niklander, P. Porkka, and K. Raatikainen, "A Distributed Real-Time Main-Memory Database for Telecommunication," in *Proceedings of the Workshop on Databases in Telecommunications*. Springer, 1999.

[4] K. Ramamritham, S. H. Son, and L. C. Dipippo, "Real-Time Databases and Data Services," *Journal of Real-Time Systems*, vol. 28, no. 2/3, pp. 179–215, November/December 2004.

[5] 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.

[6] I. Crnkovic and M. Larsson, *Building Reliable Component-Based Software Systems*. Artech House, 2002.

[7] S. Schulze, M. Pukall, G. Saake, T. Hoppe, and J. Dittmann, "On the Need of Data Management in Automotive Systems," in *BTW*, ser. LNI, J. C. Freytag, T. Ruf, W. Lehner, and G. Vossen, Eds., vol. 144. GI, 2009, pp. 217–226.

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

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

[10] 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.

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

[12] OMG UML, "The Unified Modeling Language UML," http://www.uml.org/.

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

[14] A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner, "Software Engineering for Automotive Systems: A Roadmap," in *FOSE '07: 2007 Future of Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 55–71.

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

[16] K.-K. Lau and F. M. Taweel, "Towards Encapsulating Data in Component-Based Software Systems," in *CBSE*, 2006, pp. 376–384.

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

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

[19] 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*. IEEE Computer Society, June 2004, pp. 261–270.

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

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

[22] 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.

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

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

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

[26] 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, IEEE Industrial Electronics Society, Hamburg, Germany*, September 2008.