Introducing Database-Centric Support in AUTOSAR

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

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

***Keywords*-CBSE; Data Management; RTDBMS; Real-Time; Embedded Systems; AUTOSAR;**

I. Introduction

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

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

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

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

The contribution of this paper presents a solution for how to integrate a real-time database management system, COMET [5], into an AUTOSAR platform and tool suite, such as the Arctic Core [6]. This is achieved without violating the fundamental principles of the component-based approach or, the fundamentals of the AUTOSAR standard itself. The feasibility of our approach is demonstrated with an implementation of an Adaptive Cruise Control (ACC) using database proxies for all component communication on the VFB. Finally, we present an evaluation which shows that the cost for introducing database management support via database proxies in AUTOSAR is negligible. Under typical workload conditions, our concept only introduces a total application CPU overhead, in the order of 4%.

II. Background and Motivation

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

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

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

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

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

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

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

```
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 1. A task that uses a database pointer

III. System Model and Related Techniques

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

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

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

In order to support a predictable mix of both hard and soft real-time transactions, we consider an RTDBMS with two separate interfaces. For hard real-time transactions, a database pointer [21] interface is used to enable the application to access individual data elements in the database with hard real-time performance. For soft real-time transactions, a standardized SQL interface is used.

A. Database Pointers

A database pointer [22] is a hard real-time database access-method which uses an application pointer variable to access individual data in an RTDBMS, see Figure 1. The figure shows an example of a task (thread) that reads a sensor and propagates the sensor value to the database using a database pointer. During the initialization part (lines 2 to 4) the database pointer is created and bound to a data element in the database using the `bind` function. The `bind` function calls the database server which creates a handle directly to the data element.

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

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

B. COMET

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

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

C. Arctic Core

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

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

IV. AUTOSAR Concept Overview

AUTOSAR [4] is a standard component model and middleware platform for the automotive electronic architecture. One of the fundamental concepts of AUTOSAR is to have a clear separation between the underlying infrastructure and the applications which consists of interconnected software components. A simplified explanation is that AUTOSAR consists of the following layers, see Figure 2; the SoftWare Component layer (SWC), Virtual Function Bus (VFB), and the Basic SoftWare layer (BSW).

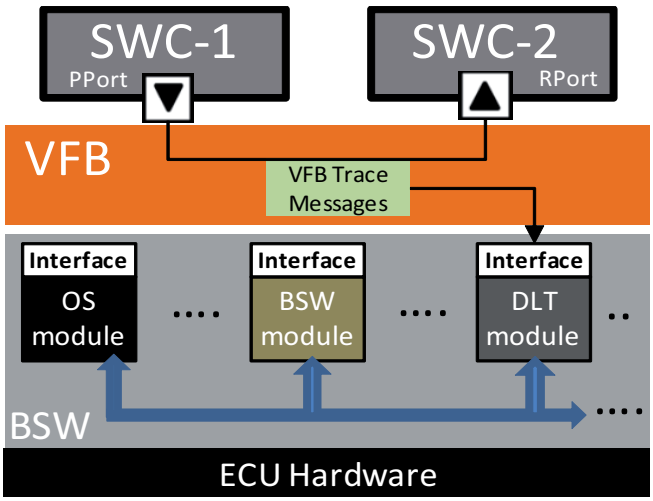

Figure 2. Autosar Overview

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

Data is shared between components by creating connections between Provider Ports (PPort) and Receiver Ports (RPort) of components. Data sets passed between components on an ECU is usually realized through shared RAM areas which have to be protected using semaphores to ensure

data consistency. This makes data internal on the ECU, and visible only to those components which have been specified as receivers when developing the system.

Regulatory requirements and the complexity of automotive systems have increased the need for run-time diagnostics and system monitoring. To meet this need, AUTOSAR has, from version 4.0 included support for system monitoring via the Diagnostic Log and Trace (DLT) module. The DLT is capable of managing multiple types of diagnostic log and trace messages and transmit them to external clients remotely connected over the network. One such trace-type is the VFB trace message which is used to collect component communication in the VFB, see Figure 2.

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

V. Database Proxies

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

Database proxies are automatically generated from the system architecture, and the run-time implementations are synthesized by the system generation tool. Therefore, database proxies are a part of the system architecture, and are realized in the form of glue code. The internal mapping of data in the RTDBMS to the database proxies is made using database pointers for hard real-time components, or pre-compiled SQL statements or stored procedures for soft real-time components. A pre-compiled statement enables a developer to bind a certain database query to a statement which is compiled once during system setup. This has a decoupling effect since the internal database schema is hidden from the component.

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

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. Hard proxies support hard real-time data to be shared between several hard real-time components, or a mix of hard and soft real-time components.

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

A hard real-time database proxy:

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

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

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.

C. Database Proxy Constituents

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

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

VI. Integrating Database Proxy Support in AUTOSAR

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

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

A. Integrating the RTDBMS

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

B. Extending the VFB with database proxies

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

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

During the realization of the VFB these constituents are used to create the proxies as follows (see Figure 3):

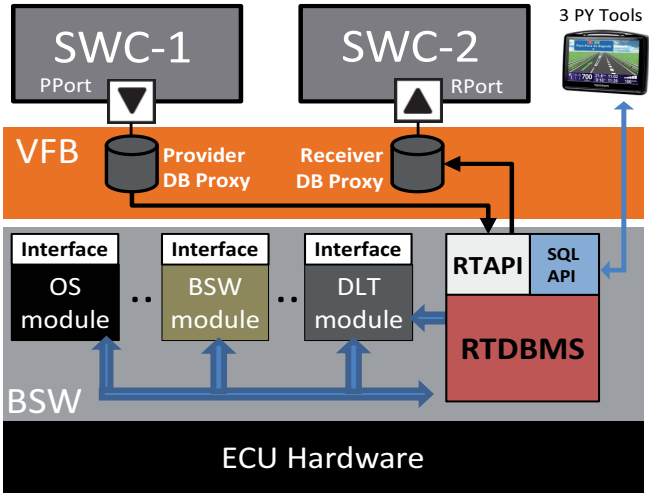

Figure 3. Database proxies support in the VFB

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

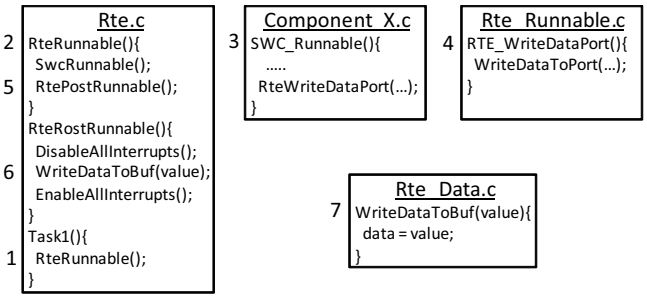

Figure 4. Execution trace for the original AUTOSAR code

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

Figures 4 and 5 presents a simplified illustration of the differences of the execution trace between the original generated implementation and using database proxies for a task that includes a component.

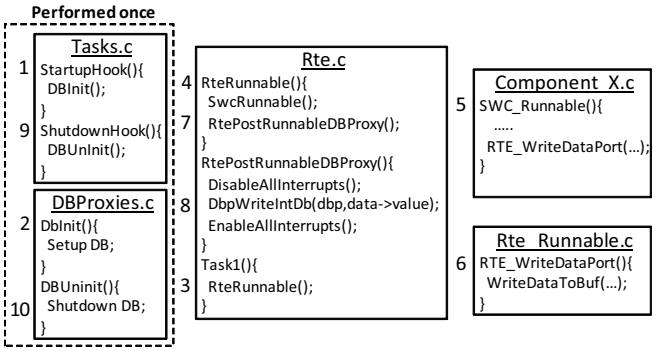

Figure 5. Execution trace using database proxies

Execution trace for the original generated implementation:

1 The task invokes the `RteRunnable`.
2 The software component is invoked by RTE.
3 The component performs its task and write the data to its port.
4 The RTE writes to a port variable.
5 Since it is a provider port, the RTE writes the data after the component invocation.
6 The `RtePostRunnable` function disables all interrupts and calls function to write the data.
7 The data is written to the buffer. When finished, all interrupts are enabled.

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

1 The AUTOSAR StartupHook calls functions to setup and initialize the database.
2 The database is initialized during startup.

Performed during component communication:

3-6 Corresponds to steps **(1-4)** in the previous example without database proxies.
7 `RtePostRunnableDBProxy` is called.
8 The data is written to the database.

Performed once during system shutdown:

9 The AUTOSAR ShutdownHook is called during system shutdown.
10 The database is uninitialized.

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

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

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

VII. System Design and Implementation

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

A. Application System Design

The system has been designed according to the proposed approach in section VI with the RTDBMS residing in the BSW and database proxies that manages all component communication. As illustrated in Figure 6, the nine components communicate via the RTE. The database proxies in the RTE manage the communication between PPorts and RPorts via the RTDBMS. However, the figure is simplified with a focus on a few connections to clearly illustrate the approach.

As seen in Figure 7, the application design consists of nine components distributed over five hard real-time tasks, T1-T5 and a soft real-time task, T6. The internal implementation of the components varies from simple Proportional Integral Derivative (PID) controllers to more complex controller logic [25].

The nine components assignments are as follows:

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

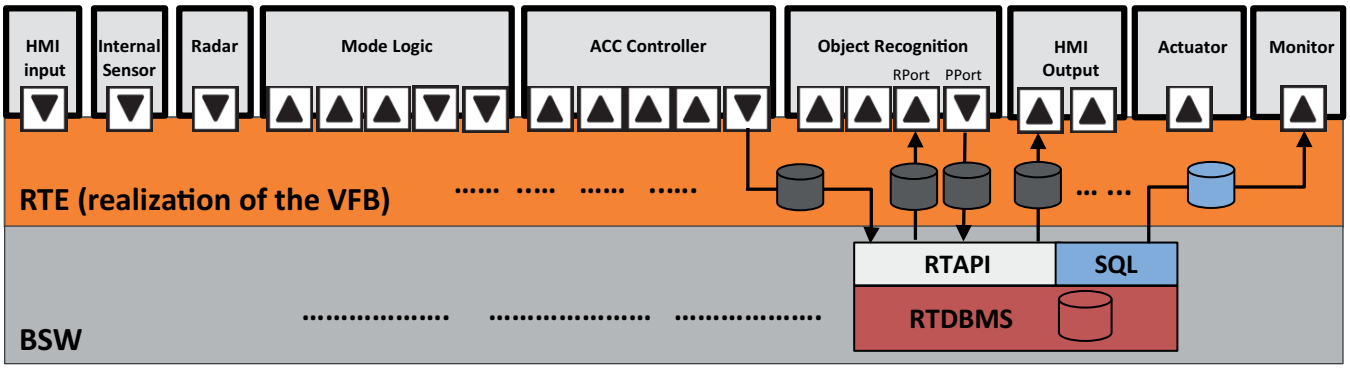

Figure 6. Illustration of the ACC system implementation in AUTOSAR

- **Actuator**, control throttle and speed of the vehicle. (Task 5)
- **Run-Time Monitor**, Monitors the output from system sensors. (Task 6)

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

B. Predictability of the Implementation

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

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

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

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

VIII. Evaluation

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

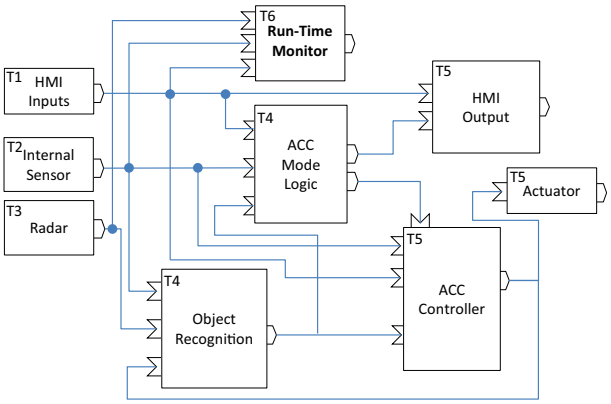

Figure 7. ACC application design

A. Benchmark Setup

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

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

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

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

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

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

B. Test 1: Communication Performance

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

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

Table I
RESULT OF TEST 1

Table I, shows the results of the test. The numbers are shown in average time in nanoseconds (ns) and in the two rightmost columns, difference in percentage and number of CPU clock cycles is shown. From the table it can be seen that a data read (where data are read from the shared variable or RTDBMS and propagated to a component) using database proxies introduces an overhead of 151ns (18 % or 10 CPU cycles) for test case **B**, and 303ns (36 % or 22 CPU cycles) for test case **C**, compared to not using database proxies. For the data write case (where data is propagated to the RTDBMS from a component) the introduced overhead is 79ns (10 % or 6 CPU cycles) for test case **B**, and 191ns (23 % or 14 CPU cycles) for test case **C**, compared to not using database proxies.

Figure 8, presents the execution time for test cases A-C. As the graph shows, the time for reading a value is constant in the three test cases, whereas the write operations have some fluctuations. Since the tasks are periodically executed, these fluctuations could be the result of a probe effect from the time measurement routines, cache misses or

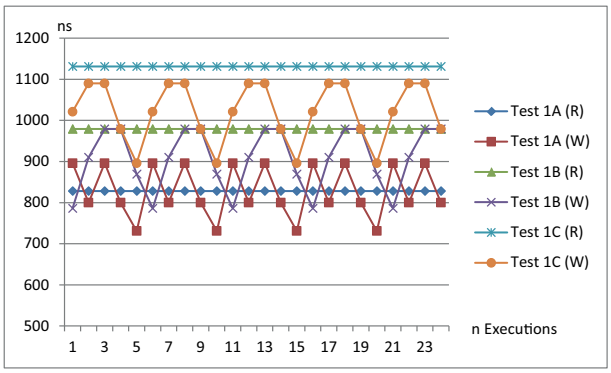

Figure 8. Execution times for read and write operations

a combination of both. In any case, this is not the result of introducing database proxies since identical fluctuations are also present in the original generated code.

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

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

Table II
RESULT OF SOFT READ BY THE RUN-TIME MONITOR

Table II, display the time, in nanoseconds, for reading 1-3 shared data elements using the SQL interface. As seen to the right in the table, 22011ns is the time required to read the output from components 1-3, as seen in Figure 7. Worth noting is that the predictability of the hard real-time tasks reading and writing the shared data is not compromised. No fluctuations or increased execution times were observed.

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

C. Test 2: System Performance

In this test, the execution time for each individual task including the component logic as well as the component communication is measured. Test 2A in Table III is the benchmark reference to which Tests 2B and 2C are evaluated. The soft task (T6) is not included in this table since it cannot be evaluated against the reference implementation which does not include the Run-Time Monitor component. However, in this performance test, T6 is included and

monitors the sensor input values throughout the evaluation for test 2B and 2C in order to show that it will not negatively affect hard proxies while providing enlarged support for sharing, tracing, monitoring and logging.

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

Table III
RESULT OF TEST 2

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

Table III, shows the results of the average execution times. The measurements shows that the overhead of using database proxies under typical workload conditions introduces an overhead of only between 1-10% in test case **B** and slightly higher in test case **C**, with the exception of task T1. In this case the increased overhead of 10% or 17% for case **B-C** can be explained by the fact that the component executed within task T1 is the smallest, therefore introducing data type and access right checking in the communication accounts for a larger relative overhead than in the other tasks. It is worth noting that the introduced overhead in test case **B** and **C** corresponds to as little as 10 and 16 CPU cycles respectively.

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

Table IV
TOTAL APPLICATION CPU OVERHEAD

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

IX. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a technique to integrate predictable database management support in the AUTOSAR basic software layer and the virtual function bus without violating the fundamental principles of the component-based approach of the AUTOSAR standard. To achieve this, the database proxy concept in conjunction with database pointer techniques, adopted by COMET, is used as the communication link on the VFB. This database-centric approach provides predictable timing guarantees, dynamic access to data, maintained component encapsulation and independence from the data-management strategy. Developers and maintenance personal can now exploit the full potential of using a real-time database and extract any trace information from the component interactions in contrast to the static predefined approach that exists today. Furthermore, the approach provides means for any BSW module to act as a database client using a standard API.

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

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

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

ACKNOWLEDGMENT

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

REFERENCES

[1] A. Hjertström, D. Nyström, and M. Sjödin, "Data Management for Component-Based Embedded Real-Time Systems: the Database Proxy Approach," *Journal of Systems and Software*, vol. 85, pp. 821–834, April 2012.

[2] 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.

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

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

[5] D. Nyström, A. Tešanović, M. Nolin, C. Norström, and J. Hansson, "COMET: A Component-Based Real-Time Database for Automotive Systems," in *Proceedings of the Workshop on Software Engineering for Automotive Systems*. The IEE, June 2004, pp. 1–8.

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

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

[8] A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner, "Software Engineering for Automotive Systems: A Roadmap," *Future of Software Engineering*, pp. 55–71, 2007.

[9] 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*, vol. 80, no. 5, pp. 655–667, May 2007.

[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, Germany*, September 2008.

[11] 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.

[12] R. R. Brooks, S. Sander, J. Deng, and J. Taiber, "Automotive System Security: Challenges and State-Of-The-Art," in *Proceedings of the 4th Annual Workshop on Cyber Security and Information Intelligence Research: Developing Strategies to Meet the Cyber Security and Information Intelligence Challenges Ahead*. ACM, 2008.

[13] D. Nyström, A. Tešanović, C. Norström, J. Hansson, and N.-E. Bånkestad, "Data Management Issues in Vehicle Control Systems: a Case Study," in *Proceedings of the 14th Euromicro Conference on Real-Time Systems*. IEEE Computer Society, June 2002, pp. 249–256.

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

[15] G. Saake, M. Rosenmüller, N. Siegmund, C. Kästner, and T. Leich, "Downsizing Data Management for Embedded Systems," *Egyptian Computer Science Journal*, pp. 1–13, 2009.

[16] N. Delgado, A. Q. Gates, and S. Roach, "A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools," *IEEE Trans. Softw. Eng.*, vol. 30, pp. 859–872, December 2004.

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

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

[19] R. K. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions: a Performance Evaluation," *ACM Trans. Database Syst.*, vol. 17, pp. 513–560, September 1992.

[20] J. Mellin, J. Hansson, and S. Andler, Eds., *Real-Time Database Systems: Issues and Applications*. Kluwer Academic Publishers, 1997, ch. Refining Timing Constraints of Application in DeeDS.

[21] 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, 2004, pp. 261–270.

[22] D. Nyström, A. Tešanović, C. Norström, and J. Hansson, "Database Pointers: a Predictable Way of Manipulating Hot Data in Hard Real-Time Systems," in *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications*, 2003, pp. 623–634.

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

[24] P. P.-S. Chen, "The Entity-Relationship Model - Toward a Unified View of Data," *ACM Trans. Database Syst.*, vol. 1, no. 1, 1976.

[25] K. Åström, "The Future of PID Control," *Control Engineering Practice*, vol. 9, no. 11, pp. 1163–1175, Nov. 2001.

[26] ArcCore VK-Board, "Open Source AUTOSAR Solutions, Göteborg Sweden," http://arccore.com/wiki/VK-Board.

[27] 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*, Sept 2009.