Embedded Databases for Embedded Real-Time Systems: A Component-Based Approach[1]

Aleksandra Tešanović

Jörgen Hansson

Linköping University
Department of Computer Science
Linköping, Sweden
{alete,jorha}@ida.liu.se

Dag Nyström

Christer Norström

Mälardalen University
Department of Computer Engineering
Västerås, Sweden
{dag.nystrom,christer.norstrom}@mdh.se

January 18, 2002

[1]This work is supported by ARTES, a network for real-time and graduate education in Sweden, and CENIIT.

Abstract

In the last years the deployment of embedded real-time systems has increased dramatically. At the same time, the amount of data that needs to be managed by embedded real-time systems is increasing, thus requiring an efficient and structured data management. Hence, database functionality is needed to provide support for storage and manipulation of data in embedded real-time systems. However, a database that can be used in an embedded real-time system must fulfill requirements both from an embedded system and from a real-time system, i.e., at the same time the database needs to be an embedded and a real-time database. The real-time database must handle transactions with temporal constraints, as well as maintain consistency as in a conventional database. The main objectives for an embedded database are low memory usage, i.e., small memory footprint, portability to different operating system platforms, efficient resource management, e.g., minimization of the CPU usage, ability to run for long periods of time without administration, and ability to be tailored for different applications. In addition, development costs must be kept as low as possible, with short time-to-market and a reliable software. In this report we survey embedded and real-time database platforms developed in industrial and research environments. This survey represents the state-of-the-art in the area of embedded databases for embedded real-time systems. The survey enables us to identify a gap between embedded systems, real-time systems and database systems, i.e., embedded databases suitable for real-time systems are sparse. Furthermore, it is observed that there is a need for a more generic embedded database that can be tailored, such that the application designer can get an optimized database for a specific type of an application. We consider integration of a modern software engineering technique, component-based software engineering, for developing embedded databases for embedded real-time systems. This merge provides means for building an embedded database platform that can be tailored for different applications, such that it has a small memory footprint, minimum of functionality, and is highly integrated with the embedded real-time system.

Contents

Chapter 1

Introduction

Digital systems can be classified in two categories: general purpose systems and application-specific systems [41]. General purpose systems can be programmed to run a variety of different applications, i.e., they are not designed for any special application, as opposed to application-specific systems. Application-specific systems can also be part of a larger host system and perform specific functions within the host system [20], and such systems are usually referred to as *embedded systems*. An embedded system is implemented partly on software and partly on hardware. When standard microprocessors, microcontrollers or DSP processors are used, specialization of an embedded system for a particular application consists primarily on specialization of software. In this report we focus on such systems. An embedded system is required to be operational during the lifetime of the host system, which may range from a few years, e.g., a low end audio component, to decades, e.g., an avionic system. The nature of embedded systems also requires the computer to interact with the external world (environment). They need to monitor sensors and control actuators for a wide variety of real-world devices. These devices interface to the computer via input and output registers and their operational requirements are device and computer dependent.

Most embedded systems are also real-time systems, i.e., the correctness of the system depends both on the logical result of the computation, and the time when the results are produced [101]. We refer to these systems as *embedded real-time systems*[1]. Real-time systems are typically constructed out of concurrent programs, called tasks. The most common type of temporal constraint that a real-time system must satisfy is the completion of task deadlines. Depending on the consequence due to a missed deadline, real-time systems can be classified as hard or soft. In a *hard real-time system* consequences of missing a deadline can be catastrophic, e.g., aircraft control, while in a *soft-real-time system*, missing a deadline does not cause catastrophic damage to the system, but may affect performance negatively. Below follows a list of examples where embedded real-time systems can be found.

- Vehicle systems for automobiles, subways, aircrafts, railways, and ships.
- Traffic control for highways, airspace, railway tracks, and shipping lines.
- Process control for power plants and chemical plants.
- Medical systems for radiation therapy and patient monitoring.
- Military uses such as advanced firing weapons, tracking, and command and control.
- Manufacturing systems with robots.

[1] We distinguish between embedded and real-time systems, since there are some embedded systems that do not enforce real-time behavior, and there are real-time systems that are not embedded.

- Telephone, radio, and satellite communications.
- Multimedia systems that provide text, graphic, audio and video interfaces.
- Household systems for monitoring and controlling appliances.
- Building managers that control such entities as heat, lights, doors, and elevators.

In the last years the deployment of embedded real-time systems has increased dramatically. As can be seen from the examples, these systems are now virtually embedded in every aspect of our lives. At the same time the amount of data that needs to be managed is growing, e.g., embedded real-time systems that are used to control a vehicle, such as a modern car, must keep track of several hundreds sensor values. As the amount of information managed by embedded real-time systems increases, it becomes increasingly important that data is managed and stored efficiently and in a uniform manner by the system. Current techniques adopted for storing and manipulating data objects in embedded and real-time systems are ad hoc, since they normally manipulate data objects as internal data structures. That is, in embedded real-time systems data management is traditionally built as a part of the overall system. This is a costly development process with respect to design, implementation, and verification of the system. In addition, such techniques do not provide mechanisms that support porting of data to other embedded systems or large central databases.

Database functionality is needed to provide support for storage and manipulation of data. Embedding databases into embedded systems have significant gains: (i) reduction of development costs due to the reuse of database systems; (ii) improvement of quality in the design of embedded systems since the database provides support for consistent and safe manipulation of data, which makes the task of the programmer simpler; and (iv) increased maintainability as the software evolves. Consequently, this improves the overall reliability of the system. Furthermore, embedded databases provide mechanisms that support porting of data to other embedded systems or large central databases.

However, embedded real-time systems put demands on such embedded database that originate from requirements on embedded and real-time systems.

Most embedded systems need to be able to run without human presence, which means that a database in such a system must be able to recover from the failure without external intervention [80]. Also, the resource load the database imposes on the embedded system should be carefully balanced, in particular, memory footprint. For example, in embedded systems used to control a vehicle minimization of the hardware cost is of utmost importance. This usually implies that memory capacity must be kept as low as possible, i.e., databases used in such systems must have small memory footprint. Embedded systems can be implemented in different hardware environments supporting different operating system platforms, which requires the embedded database to be portable to different operating system platforms.

On the other hand, real-time systems put different set of demands on a database system. The data in the database used in real-time systems must be logically consistent, as well as temporally consistent [50]. Temporal consistency of data is needed in order to maintain consistency between the actual state of the environment that is being controlled by the real-time system, and the state reflected by the content of the database. Temporal consistency has two components:

- *Absolute consistency*, between the state of the environment and its reflection in the database.
- *Relative consistency*, among the data used to derive other data.

We use the notation introduced by Ramamritham [50] to give a formal definition of temporal consistency.

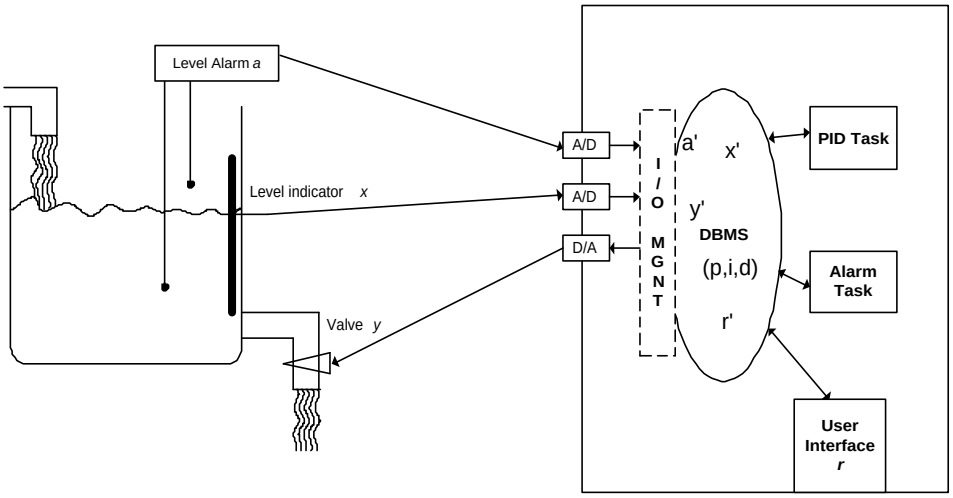

Figure 1.1: *An example system. On the left side a water tank is controlled by the system on the right side. The PID-task controls the flow out by the valve (y) so that the level in tank (x) is the same as the level set by the user (r). An alarm task shuts the system down if the level alarm (a) is activated.*

A data element, denoted d, which is temporally constrained, is defined by three attributes:

- value d_{value}, i.e., the current state of data d in the database,
- time-stamp d_{ts}, i.e., the time when the observation relating to d was made, and
- absolute validity interval d_{avi}, i.e., the length of the time interval following d_{ts} during which d is considered to be absolute consistent.

A set of data items used to derive a new data item forms a relative consistency set, denoted R, and each such set is associated with a relative validity interval, R_{rvi}. Data in the database, such that $d \in R$, has a correct state if and only if [50]

1. d_{value} is logically consistent, and
2. d is temporally consistent, both
   - absolute $(t - d_{ts}) \leq d_{avi}$, and
   - relative $\forall d' \in R, |d_{ts} - d'_{ts}| \leq R_{rvi}$.

A transaction, i.e., a sequence of read and write operations on data items, in conventional databases must satisfy following properties: atomicity, consistency, isolation, and durability, called ACID properties. In addition, transactions that process real-time data must satisfy temporal constraints. Some of the temporal constraints on transactions in a real-time database come from the temporal consistency requirement, and some from requirements imposed on the system reaction time (typically, periodicity requirements) [50]. These constraints require time-cognizant transaction processing so that transactions can be processed to meet their deadlines, both with respect to completion of the transaction as well as satisfying the temporal correctness of the data.

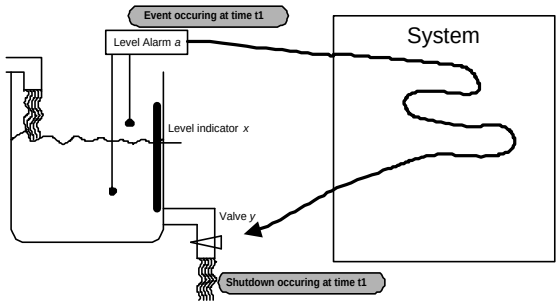

Figure 1.2: *The end-to-end deadline constraint for the alarm system. The emergency shutdown must be completed within a given time* dl_{alarm} *implying that* $t2 - t1 \leq dl_{alarm}$.

An example application

We describe one example of a typical embedded real-time system. The example is typical for a large class industrial process control system that handles large volume of data, where data have temporal constraints. In order to keep the example as illustrative and as simple as possible we limit our example to a specific application scenario, namely the control of the water level in a water tank (see figure 1.1). Through this example we illustrate demands put on data management in such a system. Our example-system contains a real-time operating system, a database, an I/O management subsystem and a user-interface.

A controller task (PID-regulator) controls the level in the water tank according to the desired level set by the user, i.e., the highest allowed water level. The environment consists of the water level, the alarm state, the setting of the valve and the value of the user interface. The environment state is reflected by the content of the database (denoted a', x', y', r' in figure 1.1). Furthermore, PID variables that reflect internal status of the system are also stored in the database. The I/O manager (I/O MGNT) is responsible for data exchange between the environment and the database. Thus, the database stores the parameters from the environment and to the environment, and configuration data. Data in the database needs to be temporally consistent, both absolute and relative. In this case, absolute validity interval can depend on the water flow. That is, if the the tank in figure 1.1 is very big and the flow is small, absolute validity interval can be greater than in the case where the water-flow is big and the data needs to be sampled more frequently. The relative consistency depicts the difference between the oldest data sample and the youngest sample of data. Hence, if the alarm in our system is activated, due to high water level, and *x'* indicates a lower level, these two values are not valid even though they have absolute validity.

For this application scenario, an additional temporal constraint must be satisfied by the database, and that is an end-to-end deadline. This temporal constraint is important because the maximum processing time for the alarm event must be smaller than the end-to-end-deadline. Figure 1.2 shows the whole end-to-end process for an alarm event. When an alarm is detected, an alarm sensor sends the signal to the A/D converter. This signal is read by the I/O manager recording the alarm in the database. The alarm task then analyzes the alarm data and sends a signal back to indicate an emergency shutdown.

In our example, the database can be accessed by the alarm task, the PID task, the user interface, and the I/O manager. Thus, an adequate concurrency control must be ensured in order to serialize transactions coming from these four different database clients.

Let us now assume that the water level in the tank is just below the highest allowed level, i.e., the level when an alarm is triggered. The water flowing into the tank creates ripples on the surface. These ripples could cause the alarm to go on and off with every ripple touching the sensor, and consequently sending bursts of alarms to the system. In this case, one more temporal constraint must be satisfied, a delayed response. The delayed

response is a period of time within which the water level must be higher than the highest allowed level in order to activate the alarm.

As we can see, this simple application scenario puts different requirements on the database. A complete industrial process control system, of which this example is part of, would put a variety of additional requirements on a database, e.g., logging.

Note that requirements placed on the database by the embedded real-time system are to some extent general for all embedded and real-time applications, but at the same time, there are requirements that are specific to an application in question (e.g., delayed response). Thus, an embedded database system must, in a sense, be tailored (customized) for each different application to give an optimized solution. That is, given the resource demands of the embedded real-time system, a database must be tailored to have minimum functionality, i.e., only functionality that a specific application needs.

In recent years, a significant amount of research has focused on how to incorporate database functionality into real-time systems without jeopardizing timeliness, and how to incorporate real-time behavior into embedded systems. However, research for embedded databases used in embedded real-time systems, explicitly addressing the development and design process, and the limited amount of resources in embedded systems is sparse. Hence, the goal of our report is to identify the gap between the following three different systems: real-time systems, embedded systems, and database systems. Further, we investigate how component-based software engineering would provide a feasible approach for bridging this gap by enabling development of a customizable embedded database suitable for embedded real-time systems, in a short time with reduced development costs, and high quality of software.

There are many embedded databases on the market, but, as we show in this report, they vary widely form vendor to vendor. Existing commercial embedded database systems, e.g., Polyhedra [91], RDM and Velocis [70], Pervasive.SQL [88], Berkeley DB [99], and TimesTen [109], have different characteristics and are designed with specific applications in mind. They support different data models, e.g., relational vs object-oriented model, and operating system platforms. Moreover, they have different memory requirements and provide different types of interfaces for users to access data in the database.

Application developers must carefully choose the embedded database their application requires, and find the balance between the functionality an application requires and the functionality that an embedded database offers. Thus, finding the right embedded database, in addition of being a quite time consuming, costly and difficult process, is a process with lot of compromises. Although a significant amount of research in real-time databases has been done in the past years, it has mainly focussed on various schemes for concurrency control, transaction scheduling, and logging and recovery. Research projects that are building real-time database platforms, such as ART-RTDB [52], BeeHive [103], DeeDS [9] and RODAIN [61], mainly address real-time performance, have monolithic structure, and are built for a particular real-time application. Hence, the issue of how to enable development of an embedded database system that can be tailored for different embedded real-time applications arises. The development costs of such database system must be kept low, and the development must ensure good software quality. We show that exploiting component-based software engineering in the database development seem to have a potential, and examine how component-based software engineering can enable database systems to be easily tailored, i.e., optimized, for a particular application. By having well-defined reusable components as building blocks, not only development costs are reduced, but more importantly, costs related to verification and certification are reduced since components will only need to be checked once. Although some major database vendors such as Oracle, Informix, Sybase, and Microsoft have recognized that component-based development offers significant benefits, their component-based solutions are limited in terms of tailorability with, in most cases, inadequate support for development of such component-based database systems. Furthermore, commercial component-based databases do not enforce real-time behavior and are

not, in most cases, suitable for environments with limited resources. We argue that in order to compose a reliable embedded real-time system out of components, component behavior must be predictable. Thus, components must have well defined temporal attributes. However, existing component-based database systems do not enforce real-time behavior. Also, issues related to embedded systems such as low-resource consumption are not addressed at all in these solutions. We show that real-time properties are preserved only in a few of the existing component-based real-time systems. In addition to temporal properties, components in component-based embedded systems must have explicitly addressed memory needs and power consumption requirements. At the end of this report, we outline initial ideas for preserving real-time and embedded properties in components used for building an embedded database platform that can be tailored for different applications.

The report is organized as follows. In chapter 2 we survey commercial embedded and research real-time database systems. We then examine the component-based software engineering paradigm, and its integration with databases for embedded real-time systems in chapter 3. In chapter 4 we discuss a possible scenario for developing an embedded database platform that can be tailored for different applications, and give conclusions.

Chapter 2

Database systems

2.1 Traditional database systems

Databases are used to store data items in a structured way. Data items stored in a database should be persistent, i.e., a data item stored in the database should remain there until either removed or updated. Transactions are most often used to read, write, or update data items. Transactions should guarantee serialization. The so-called database manager is accessed through interfaces, and one database manager can support multiple interfaces like C/C++, SQL, or ActiveX interfaces.

Most database management systems consist of three levels [27]:

- The internal level, or physical level, deals with the physical storage of the data onto a media. Its interface to the conceptual level abstracts all such information away.
- The conceptual level handles transactions and structures of the data, maintaining serialization and persistence.
- The external level contains interfaces to users, uniforming transactions before they are sent to the conceptual level.

One of the main goals for many traditional database systems are transaction throughput and low average response time [50], while for real-time databases the main goal is to achieve predictability with respect to response times, memory usage, and CPU utilization. We can say that when a worst case response time or maximum memory consumption for a database can be guaranteed, the system is predictable. However, there can be different levels of predictability. For a system that can guarantee a certain response time with some defined confidence, is said to have a certain degree of predictability.

2.2 Embedded database systems

2.2.1 Definitions

A device embedded database system is a database system that resides in an embedded system. In contrast, an application-embedded database is hidden inside an application and is not visible to the application user. Application-embedded databases are not addressed further in this survey. This survey focuses only on databases embedded in real-time and embedded systems. The main objectives of a traditional enterprise database system often is throughput, flexibility, scalability, functionality etc., while size, resource usage, and processor usage are not as important, since hardware is relatively cheap. In embedded systems these issues are much more important. The main issues for an embedded database system can be summarized as [96, 80, 94]:

- Minimization of the memory footprint: The memory demand for an embedded system are most often, mainly for economical reasons, kept as low as possible. A typical footprint for an embedded database is within the range of some kilobytes to a couple of megabytes.
- Reduction of resource allocations: In an embedded system, the database management system and the application are most often run on the same processor, putting a great demand on the database process to allocate minimum CPU bandwidth to leave as much capacity as possible to the application.
- Support for multiple operating systems: In an enterprise database system, the DBMS is typically run on a dedicated server using a normal operating system. The clients, that could be desktop computers, other servers, or even embedded systems, connect to the server using a network connection. Because a database most often run on the same piece of hardware as the application in an embedded system, and that embedded systems often use specialized operating systems, the database system must support these operating systems.
- High availability: In contrast to a traditional database system, most embedded database systems do not have a system administrator present during run-time. Therefore, an embedded database must be able to run on its own.

Depending on the kind of system the database should reside in, some additional objectives might be more emphasized, while others are less important. For example, Pervasive, which manufactures Pervasive.SQL DBMS system, has identified three different types of embedded systems and has therefore developed different versions of their database to support these systems [89]:

- Pervasive.SQL for smart cards is intended for systems with very limited memory resources, typically a ten kilobytes. This version has traded off sophisticated concurrency control and flexible interfaces for size. Typical applications include banking systems like cash-cards, identification systems, health-care, and mobile phones.
- Pervasive.SQL for embedded systems is designed for small control systems like our example system in figure 1.1. It can also be used as a data pump, which is a system that reads data from a number of sensors, stores the data in a local database, and continuously “pumps” data to a large enterprise database in an unidirectional flow. Important issues here are predictability, with respect to both timing and system resources, as we are approaching a real-time environment. The interfaces are kept rather simple to increase speed and predictability. Since the number of users and the rate of transactions often can be predicted, the need for a complex concurrency control system might not be necessary.
- Pervasive.SQL for mobile systems is used in mobile devices like cellular phones or PDAs, where there is a need for higher degree of concurrency control. Consider that a user browses through e-mails on a PDA while walking into his/hers office, then the synchronizer updates the e-mail list using a wireless communication without interrupting the user. The interfaces support more complex ad-hoc queries than the embedded version. The interfaces are modular and can be included or excluded to minimize memory footprint.

2.2.2 An industrial case study

In 1999, ABB Robotics who develops and manufactures robots for industrial use, wanted to exchange the existing in-house developed configuration storage management system

into a more flexible and generic system, preferably some commercial-of-the- shelf (COTS) embedded database.

An industrial robot is a complex and computer-controlled system, which consists of many sub-systems. In each robot some configuration information about its equipment is stored. Today, this amounts to about 600 kilobytes of data. This data needs to be stored in a structured and organized way, allowing easy access.

The current system, called CFG, is a custom-made application and resembles in many ways to a regular database application. However, it is nowadays considered to be too inflexible and the user-interface is not user friendly enough. Furthermore, the internal structures and the choice of index system have lead to much longer response times as the amount of data has increased.

ABB Robotics decided to investigate the market for an existing suitable database system that would fulfill their demands. The following requirements were considered to be important[63]:

- Connectivity. It should be possible to access the database both directly from the robot controlling application and from an external application via a network connection. Furthermore the database should support *views* so data could be queried from a certain perspective. It would be preferable if some kind of standard interface, e.g., ODBC, could be used. If possible, the database should be backward compatible with the query language used in the CFG system. The database should also be able to handle simultaneous transactions and queries, and be able to handle concurrency.
- Scalability. The amount of data items in the system must be allowed to grow as the system evolves. Transaction times and database behavior must not change due to increase in data size.
- Security. It should be possible to assign different security levels for different parts of the data in the database. That is, some form of user identification and some security levels must exist.
- Data persistence. The database should be able to recover safely after a system failure. Therefore some form of persistent storage must be supported, and the database should remain internally consistent after recovery.
- Memory requirements. To keep the size of the DBMS system low is a key issue.

2.3 Commercial embedded DBMS: a survey

In this section we discuss and compare a number of commercial embedded database systems. We have selected a handful of systems with different characteristics. These databases are compared with each other with respect to a set of criteria, which represent the most fundamental issues related to embedded databases.

2.3.1 Criteria investigated

- DBMS model, which describes the architecture of the database system. Two DBMS architectures for embedded databases are the client/server model and the embedded library model.
- Data model, which specifies how data in the database is organized. Common models are the relational, the object-oriented and the object relational.
- Data indexing, which describes how the data indexes are constructed.

- Memory requirements, which describes the memory requirements for the system, both data overhead and the memory footprint, which is the size of the database management system.
- Storage media, which specifies the different kinds of storage medias that the database supports.
- Connectivity, which describes the different interfaces the application can use to access the database. Network connectivity is also specified, i.e. the ability to access the database remotely via a network connection.
- Operating system platforms, which specifies the operating systems supported by the database system.
- Concurrency control, which describes how concurrent transactions are handled by the system.
- Recovery, which specifies how backup/restoration is managed by a system in case of failures.
- Real-time properties, which discusses different real-time aspects of the system.

2.3.2 Databases investigated

In this survey we have selected a handful of systems that together represents a somewhat complete picture of the types of products currently on the market.

- Pervasive.SQL by Pervasive Software Inc. This database has three different versions for embedded systems: Pervasive.SQL for smart-card, Pervasive.SQL for mobile systems, and Pervasive.SQL for embedded systems. All three versions integrate well with each other and also with their non embedded versions of Pervasive.SQL. Their system view and the fact that they have, compared to most embedded databases, very small memory requirements was one reason for investigating this database [88].
- Polyhedra by Polyhedra Plc. This database was selected for three reasons, first of all it is claimed to be a real-time database, secondly it is a main memory database, and third, it supports active behavior [91].
- Velocis by Mbrane Ltd. This system is primarily intended for e-Commerce and Web applications, but has some support for embedded operating systems [70].
- RDM by Mbrane Ltd. Like Polyhedra, RDM also claims to be a real-time database. It is however fundamentally different from the Polyhedra system, by, for example, being an embedded library and, thus, does not adopt the client/server model [70].
- Berkeley DB by Sleepycat Software Inc. This database, which also is implemented as a library, was selected for the survey because it is distributed as open source and therefore interesting from a research point of view [99].
- TimesTen by TimesTen Performance Software. This relational database is, like the Polyhedra system a main memory real-time database system [109].

2.3.3 DBMS model

There are basically two different DBMS models supported (see table 2.1). The first model is the client/server model, where the database server can be considered to be an application running separately from the real-time application, even though they run on the same processor. The DBMS is called using a request/response protocol. The server is accessed either

DBMS system	Client/server	Library
Pervasive.SQL	x	
Polyhedra	x	
Velocis	x	
RDM		x
Berkeley DB		x
TimesTen	x	

Table 2.1: DBMS models supported by different embedded database systems.

through inter-process communication or some network. The second model is to compile the DBMS together with the system application into one executable system. When a task wants to access the database it only performs function calls to make the request. Transactions execute on behalf of their tasks' threads, even though internal threads in the DBMS might be used. There are advantages and drawbacks with both models. In a traditional enterprise database, the server most often run on a dedicated server machine, allowing the DBMS to use almost 100% of the CPU. However in an embedded client/server system, the application and the server process often share the same processor. This implies that for every transaction at least two context switches must be performed, see figure 2.1. More complex transactions, like an update transaction might require even more context switches. Consider for example a real-time task that executes an update transaction that first reads the data element x, then derives a new value of x from the old value and finally writes it back to the database. This transaction would generate four context switches. Two while fetching the value of x and two while writing x back to the database.

These context switches adds to the system overhead, and can furthermore make worst case execution time estimations of transactions more complex to predict. The network message passing or inter-process communication between the server and the client also add to the overhead cost. A drawback with an embedded library is that they lack standardized interfaces like ODBC [80].

The problem with message passing overhead has been reduced in the Velocis system. They allow the process to be compiled together with the application, thus reducing the overhead since shared memory can be used instead.

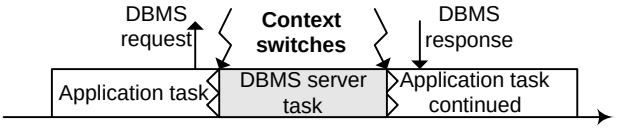

Figure 2.1: *In an embedded client/server DBMS at least two context switches are necessary for each transaction.*

2.3.4 Data model

The data model concerns how data is logically structured. The most common model is the relational model where data is organized in tables with columns and rows. Databases implementing relational data model are referred to as relational databases (RDBMS). One advantage with a relational model is that columns in the table can relate to other tables so arbitrary complex logical structures can be created. From this logical structure queries can be used to extract a specific selection of data, i.e. a view. However, one disadvantage with the relational model is added data lookup overhead. This is because that data elements are

DBMS system	Relational	Object-oriented	Object-relational	Other
Pervasive.SQL	x			
Polyhedra	x		(x)	
Velocis	x			
RDM	x			
Berkeley DB				x
TimesTen	x			

Table 2.2: Data models supported by different embedded database systems.

organized with indexes in which pointers to the data is stored, and to perform the index lookup can take significant amount of time, especially for databases stored on hard drives. This can be a serious problem for databases that resides in time critical applications like our example application in figure 1.1.

The relational model, which was developed by Codd, only supports a few basic data types, e.g., integers, floating point numbers, currency, and fixed length arrays. This is nowadays considered a limitation which has lead to the introduction of databases which supports more complex data types, e.g., Binary Large Objects (BLOBs). BLOB is data, which is treated as a set of binary digits. The database does not know what a BLOB contains and therefore cannot index anything inside of the BLOB.

The object-oriented database (ODMBS) is a different kind of data model, which is highly integrated with object-oriented modeling and programming, The ODBMS is an extension to the semantics of an object-oriented language. An object-oriented database stores objects that can be shared by different applications. For example, a company which deals with e-Commerce has a database containing all customers. The application is written in an object-oriented language and a customer is modeled as an object. This object has methods, like `buy` and `changeAdress`, associated with it. When a new customer arrives, an instance of the class is created and then stored in the database. The instance can then be retrieved at any time. Controlled access is guaranteed due to concurrency control.

A third data model, which has evolved from both the relational and the object-oriented model, incorporates objects in relations, thus is called object-relational databases (ORDBMS).

As shown in table 2.2, all systems in the survey except Berkeley DB are relational. Furthermore the Polyhedra has some object-relational behavior through its control language described below. However it is not fully object-relational since objects itself cannot be stored in the database. To our knowledge, no pure object-oriented embedded database exists on the market today. There are however some low requirements databases that are object-oriented, such as the Objectivity/DB [79] system. Furthermore some object-oriented client-libraries exist that can be used in an embedded system, such as the PowerTier [87] and the Poet Object system [90]. These client-libraries connect to an external database server. Since the actual DBMS and the database is located on a non-embedded system we do not consider them embedded databases in this survey.

Most systems have ways to “shortcut” access to data, and therefore bypassing the index lookup routine. Pervasive, for example, can access data using the Btrieve transactional engine that bypasses the relational engine. Mbrane uses a different approach in the RDM system. In many real-time systems data items are accessed in a predefined order (think of a controlling system where some sensor values are read from the database and the result is written back to the database). By inserting shortcuts between data elements such that they are directly linked in the order they are accessed, fast accesses can be achieved. As these shortcuts point directly to physical locations in memory, reorganization of the database is much more complex since a large number of pointers can become stale when a single data is dropped or moved.

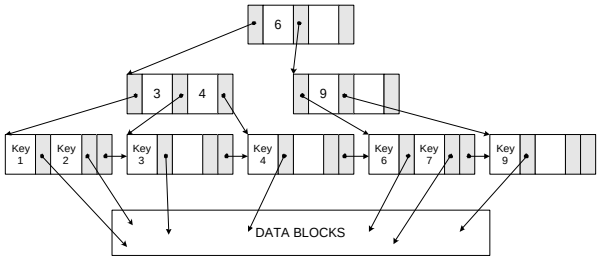

Figure 2.2: *The structure of a B^+-tree with fanout of 3. The inner nodes are only used to index down to the leave node containing the pointer to the correct data. Figure partially from [58].*

The Polyhedra DBMS system is fundamentally different compared to the rest of the relational systems in this survey, because of its active behavior. This is achieved through two mechanisms, active queries and by the control language (CL). An active query looks quite like a normal query where some data is retrieved and/or written, but instead the query stays in the database until explicitly aborted. When a change in the data occurs that would alter the result of the query, the application is notified. The CL, which is a fully object-oriented script language that supports encapsulation, information hiding and inheritance, can determine the behavior of data in the database. This means that methods, private or public, can be associated with data performing operations on them without involving the application. In our example application, the CL could be used to derive the valve setting y' from the level indicator x' and the PID parameters, thus removing the need for the PID task. The CL has another important usage, since it can be used in the display manager (DM) to make a graphical interface. The DM, which is implemented as a Polyhedra Client, together with the control language is able to present a graphical view on a local or remote user terminal. This is actually exactly the user interface in our example application, since DM also can take user input and forward that to the database. Active database management will be further discussed in section 2.4.4.

As mentioned above, Berkeley DB is the only non-relational system in this survey. Instead it uses a key-data relationship. One data is associated with a key. There are three ways to search for data, from key, part of key or sequential search. The data can be arbitrary large and of virtually any structure. Since the keys are plain ASCII strings the data can contain other keys so complex relations can be built up. In fact it would be possible to implement a relational engine on top of the Berkeley DB database. This approach claims for a very intimate relationship between the database and the programming language used in the application.

2.3.5 Data indexing

To be able to efficiently search for specific data, an efficient index system should exist. To linearly search through every single key from an unsorted list would not be a good solution since transaction response times would grow as the number of data elements in the database increases. To solve this problem two major approaches are used, tree structures and hashed lists. Both approaches supply similar functionality, but differ somewhat in performance. Two major tree structures are used, B^+-tree indexing, which suits disk based databases, and T-tree indexing, which is primarily used in main-memory databases.

The main issue for B^+-tree indexing is to minimize disk I/O, thus trading disk I/O for added algorithm complexity. B^+-tree indexing sorts the keys in a tree structure in which every node has a predefined number of children, denoted p. A large value of p results in a wide but shallow tree, thus the tree has a large fanout. A small value of p results in the

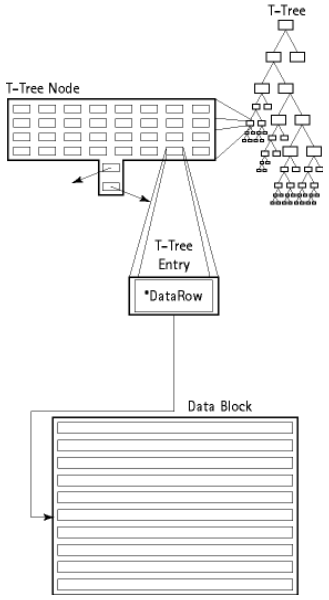

Figure 2.3: *The structure of a T-tree. Each node contains a number of entries that contains the key and a pointer to the corresponding data. All key values that are less then the boundary of this node is directed to the left child, and key values that are above the boundary are directed to the right. Figure from TimesTen White paper.*

opposite. For $p = 2$ tree is a binary tree. Since all data indexes reside in the leave nodes of the tree, while only the inner nodes are used to navigate to the right leave, the number of visited nodes will be fewer for a shallow tree. This results in fewer nodes that have to be retrieved from disk, thus reducing disk I/O. In figure 2.3.5 we see an example of a B^+-tree . The root directs requests to all keys that are less than six to the left child and keys from six to its middle child. As the value of p increases, the longer time it takes to pass the request to the correct child. Thus, when deciding the fanout degree, the time it takes to fetch a node from disk must be in proportion to the time it takes to locate which child to redirect to. Proposals on real-time concurrency control for B^+-tree indexing have been made [58].

DBMS system	**B^+-tree**	**T-tree**	**Hashing**	**Other**
Pervasive.SQL	x			
Polyhedra			x	
Velocis	n/a	n/a	n/a	n/a
RDM	x			
Berkeley DB	x		x	x
TimesTen	x	x	x	

Table 2.3: Data indexing strategies used by different embedded database systems.

For main-memory databases a different type of tree can be used, the T-tree structure [60]. The T-tree uses a deeper tree structure than the B^+-tree since it is a balanced binary tree, see figure 2.3.5, with a maximum of two children for every node. To locate data, more nodes are normally visited, compared to the B^+-tree , before the correct node is found. This is not a problem for a main-memory database since memory I/O is significantly faster than disk I/O. The primary issue for T-tree indexing is that the algorithms used to traverse a T-tree have a lower complexity and faster execution time than corresponding algorithms

for the B-tree. However, it has been pointed out that a T-tree traversal combined with concurrency control might perform worse than B-tree indexes due to the fact that more nodes in the T-tree had to be locked to ensure tree integrity during traversal [65] . They also proposed an improvement called T-tail, which reduces the number of costly re-balancing operations needed. Furthermore, they proposed two concurrency algorithms for T-tail and T-tree structures, one optimistic and one pessimistic (for a more detailed discussion about pessimistic and optimistic concurrency control, see section 2.3.10).

The second approach, hashing, uses an hash list in which keys are inserted according to a value derived from the key itself (different keys can be assigned the same value) [62]. Therefore, each entry in the list is a bucket that can contain a predefined number of data keys. If a bucket is full a rehash operation must be performed. The advantage with hashing is that the time to search for certain data is constant independent of the amount of data entries in the database. However, hashing often cause more disk I/O than B-trees. This is because data is often used with locality of reference, i.e., if data element d_1 is used, it is more probable that data element d_2 will be used shortly. In a B-tree both elements would be sorted close to each other, and when d_1 is retrieved from disk, then it is probable that also d_2 will be retrieved. However in a hash list d_1 and d_2 are likely to be in different buckets, causing extra disk I/O if both data are used. Another disadvantage with hashing compared to tree-based indexing is that non-exact queries is time consuming to perform. For example, consider a query for all keys greater than x. After that x has been found, all keys are found to the right of x in a B^+-tree . For a hashed index, all buckets need to be searched to find all matching keys.

We can notice that main-memory databases, namely Polyhedra and TimesTen use hashing (see table 2.3). Additionally, TimesTen supports T-trees. Pervasive, RDM and Berkeley DB use B-tree indexing.

It is also noteworthy that Berkeley DB uses both B^+-tree and hashing. They claim that hashing is suitable for database schemes that are either so small that the index fits into main memory or when the database is so large that B^+-tree indexing will cause disk I/O upon most node fetching due to that the cache can only fit a small fraction of the nodes. Thus making the B^+-tree indexing suitable for database schemes with a size in between these extremes. Furthermore, Berkeley DB supports a third access method, queue, which is used for fast inserts in the tail, and fast retrieval of data from the head of the queue. This approach is suitable for the large class of systems that consume large volumes of data, e.g., state machines.

2.3.6 Memory requirements

Memory requirement of the database is an important issue for embedded databases residing in environments with small memory requirements. For mass-produced embedded computer systems like computer nodes in a car, minimizing hardware is usually a significant factor for reducing development costs. There are two interesting properties to consider for embedded databases, first of all the memory footprint size, which is the size of the database without any data elements in it. Second, data overhead is of interest, i.e., the number of bytes required to store a data element apart from the size of the data element itself. An entry in the index list with a pointer to the physical storage address is typical data overhead. Typically client/server solutions seems to require significantly more memory than embedded libraries, with Pervasive.SQL being the exception (see table 2.4). This exception could partially be explained by Pervasive's Btrieve native interface being a very low-level interface (see section 2.3.8), and that much of the functionality is placed in the application instead. In the case of Berkeley DB, there is a similar explanation to its small memory footprint. Additional functionality (not provided by the vendor) can be implemented on top of the database.

[1] The values given in the table are the "footprint"-values made available by database vendors.

DBMS system	Memory requirements[1]
Pervasive.SQL for smart cards	8kb
Pervasive.SQL for embedded systems	50kb
Pervasive.SQL for mobile systems	50 - 400kb
Polyhedra	1.5 - 2Mb
Velocis	4Mb
RDM	400 - 500kb
Berkeley DB	175kb
TimesTen	5Mb

Table 2.4: Different memory needs of investigated embedded database systems.

DBMS system	Hard-drive	Flash	Smart card	FTP
Pervasive.SQL	x	x	x	
Polyhedra	x	x		x
Velocis	x			
RDM	x			
Berkeley DB	x	x		
TimesTen	x			

Table 2.5: Storage medias used by investigated embedded database systems.

Regarding data overhead, Polyhedra has a data overhead of 28 bytes for every record. Pervasive's data overhead is not as easily calculated since it uses paging. Depending on record and page sizes different amount of fragmentation is introduced in the data files. There are however formulas provided by Pervasive for calculating exact record sizes. The other systems investigated in this survey supplies no information about actual data overhead costs.

For systems where low memory usage is considered more important than processing speed, there is an option of compressing the data. Data will then be transparently compressed and decompressed during runtime. This, however, requires free working memory of 16 times the size of the record being compressed/decompressed.

2.3.7 Storage media

Embedded computer systems support different storage medias. Normally, data used on the computer is stored on a hard-drive. Hand-held computer use Flash or other non-volatile memory for storage of both programs and data. Since data in a database needs to be persistent even upon a power loss, some form of stable storage technique must be used. As can be seen from table 2.5, most systems support Flash in addition to a hard-drive. The Polyhedra system also supports storage via a FTP-connection. This implies that the Polyhedra system can run without persistent storage, since data can be loaded via FTP-connection upon system startup.

2.3.8 Connectivity

Database interfaces are the users' way to access the database. An interface normally contains functions for connecting to the server, making queries, performing transactions, and making changes to the database schema. However different interfaces can be specialized on specific types of operations, e.g., SQL is used mainly for schema modifications and queries, while a C/C++ API normally are specialized for transactional access. Figure 2.4

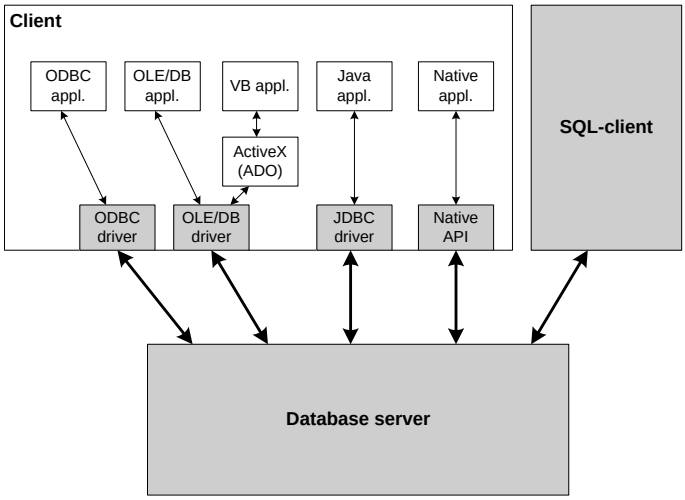

Figure 2.4: *Figure showing the architecture of applications that uses different types of database interfaces. The shaded boxes are components that normally are provided by the database vendor.*

shows a typical interface configuration for an embedded database system.

The most basic interface is the native interface. This interface, which is for a specific programming language, e.g., C or C++, is often used by the application running on the embedded system. In our example in figure 1.1, tasks and I/O management would typically use the native interface to perform their transactions.

Microsoft has developed the Open Database Connectivity (ODBC) interface in the late 80's. This interface uses the query language SQL, and is today one of the largest standards of database interfaces. The advantage with ODBC is that any database that supports ODBC can connect with any application written for ODBC. ODBC is a low-level interface, that requires more implementation per database access then a high-level interface like ActiveX Data Object (ADO) interface, explained below in more detail. The ODBC interface cannot handle non-relational databases very well. Main benefits, beside the interchangeability, is that the performance is generally higher than high-level interfaces, and the interface gives the possibility of detailed tuning.

A newer standard, also developed by Microsoft, is the OLE DB interface which is based on the Microsoft COM technology (for more detailed description of OLE DB's functionality see section 3.2.5). OLE DB is, as ODBC, a low-level interface. The functionality of OLE DB is similar to ODBC, but object-oriented. However, one advantage is the possibility to use OLE DB on non-relational database systems. There are also drivers available to connect the OLE DB interface on top of an ODBC driver. In this case the ODBC driver would act as a server for the application but as a client to the DBMS.

On top of OLE DB, the high-level ADO interface could be used. Since both OLE DB and ADO are based on Microsoft COM, they can be reached from almost any programming language.

For Java applications, the JDBC interface can be used. It builds on ODBC but has been developed with the same philosophy as the OLE DB and ADO interfaces. For databases that do not directly support JDBC, an ODBC driver can be used as intermediate layer, just like OLE DB. Pervasive.SQL uses the Btrieve interface for fast reading and updating transactions. Btrieve is very closely connected to the physical storage of data. In Btrieve data is written as records. A record can be described as a predefined set of data elements, similar to a *struct* in the C programming language. Btrieve is not aware of the data elements

DBMS system	C/C++	ODBC	OLE DB	ADO	JDBC	Java	Other
Pervasive.SQL		x	x	x	x		Btrieve RSI
Polyhedra	x	x	x	x	x		
Velocis	x	x				x	Perl
RDM	x						
Berkeley DB	x	x				x	TCL Perl
TimesTen					x		

Table 2.6: Interfaces supported by different embedded database systems.

DBMS system	Network connectivity
Pervasive.SQL	x
Polyhedra	x
Velocis	x
RDM	x
Berkeley DB	
TimesTen	x

Table 2.7: Network connectivity in investigated embedded database systems.

in the record, but treats data elements as string of bytes that can be retrieved or stored. An index key is attached to every record. Records can then be accessed in two ways, physical or logical. When a record is located logically, an index containing all keys is used to lookup the location of the data. The indexes are sorted in either ascending or descending order in a B-tree. Some methods to access data, except by keyword, are `get first`, `get last`, `get greater than`, etc. However for very fast access, the physical method can be used. Then the data is retrieved using a physical location. This technique is useful when data is accessed in a predefined order, this can be compared to the pointer network used by RDM discussed previously. Physical access is performed using four basic methods, `step first`, `step last`, `step next`, and `step previous`.

One limitation with the Btrieve interface is that it is not relational. Pervasive has therefore introduced their RowSet interface (RSI) that combines some functionality from Btrieve and some from SQL. Every record is a row and the data elements in the records can be considered columns in the relation. This interface is available for the smart card version, embedded version, and mobile version, but it is not available for non-embedded versions of Pervasive.SQL.

The Btrieve and RSI interfaces are quite different from the native interface in Polyhedra, which does not require the same understanding of the physical data structure. The Polyhedra C/C++ interface uses SQL, thus operating on a higher level. Basically, three different ways to access the database are provided in the interface: queries, active queries, and transactions. A query performs a retrieval of data from the database immediately, using the SQL `select` command, and the query is thereafter deleted. An active query works the same way as an ordinary query but will not be deleted after the result has been delivered, but will be reactivated as soon as any of the involved data is changed. This will continue until the active query is explicitly deleted. One problem with active queries is that if data changes very fast, the number of activations of the query can be substantial, thus risking to overload the system. To prevent this a minimum inter-arrival time can be set, and thus the query cannot be reactivated until the time specified has elapsed. From a real-time perspective, the activation could then be treated as a periodic event, adding to the predictability of

the system.

Using transactions is the only way to make updates to the database, queries are only for retrieval of data. A transaction can consist of a mixture of active update queries and direct SQL statements. Note that a query might update the database if it is placed inside of a transaction. Active update queries insert or update a single data element and also delete on single row. Transactions are, as always, treated as atomic operations and are aborted upon data conflict. To further speed up transaction execution time, SQL procedures can be used, both in queries and transactions. These procedures are compiled once and cannot be changed after that, only executed. This eliminates the compilation process and is very useful for queries that run often. However, procedures do not allow schema changes.

Noteworthy is that TimesTen only supports ODBC and JDBC, and it has no native interface like the rest of the systems. To increase the speed of ODBC connections, an optimized ODBC driver is developed that connects only to TimesTen. Like the Polyhedra DBMS, TimesTen uses precompiled queries, but in addition supports parameterized queries. A parameterized query is a precompiled query that supports arguments passed to it. For example the parameterized query

```
query( X )
    SELECT * from X
```

would return the table specified by `X`.

The Berkeley DB, as mentioned earlier, is not a client/server solution, but is implemented as an embedded library. It is, furthermore, not relational but uses, similar to the Btrieve interface, a key that relates to a record. The similarities to Btrieve go further than that; data can be accessed in two ways, by key or by physical location, just as Btrieve. The methods `get()` and `put()` access data by the key, while cursor methods can be used to get data by physical location in the database. As in Btrieve, the first, last, next, current, and previous record number can be retrieved. A cursor is simply a pointer that points directly to a record. The functionality and method names are similar to each other independent of which of the supported programming language that is used. There are three different modes that the database can run in:

- Single user DB. This version only allows one user to access the database at a time, thus, removing the need for concurrency control and transaction support.
- Concurrent DB. This version allows multiple users to access the database simultaneously, but does not support transactions. This database is suitable for systems that has very few updates but multiple read-only transactions.
- Transactional database. This version allows both concurrent users and transactions.

The RDM database, which is implemented as a library, has a C/C++ interface as its only interface. This library is, in contrast to Berkeley DB, relational. The interface is a low-level interface that does not comply with any of the interface standards. As mentioned earlier, one problem with databases that does not use the client/server model is the lack of standard interfaces. However, since RDM is relational, a separate SQL-like interface *dbquery* is developed. It is a superset of the standard library, which makes use of a subset of SQL.

2.3.9 Operating system platforms

Different embedded systems might run on various operating systems due to differences in the hardware environment. Also the nature of the application determines what operating systems might be most useful. Thus, most embedded databases must be able to run on different operating system platforms.

DBMS system	Desktop OS				
	Windows	**UNIX**	**Linux**	**OS/2**	**DOS**
Pervasive.SQL for smart cards					
Pervasive.SQL for embedded systems					
Pervasive.SQL for mobile systems	x				
Polyhedra	x	x	x		
Velocis	x	x	x		
RDM	x	x	x	x	x
Berkeley DB	x	x	x		
TimesTen	x	x	x		

Table 2.8: Desktop operating systems supported by investigated embedded database systems.

DBMS system	Real-Time OS						Hand-held OS		Smart card OS		
	VxWorks	***OSE***	***pSOS***	***LynxOS***	***QNX***	***Phar-lap***	***Win CE***	***Palm OS***	***Java Card***	***Mult OS***	***Win[0]***
Pervasive.SQL[1]									x	x	x
Pervasive.SQL [2]	x				x	x					
Pervasive.SQL [3]							x	x			
Polyhedra	x	x	x	x							
Velocis											
RDM	x				x						
Berkeley DB	x										
TimesTen	x			x							

Table 2.9: Real-time and embedded operating systems supported by investigated embedded database systems.

Tables 2.8 and 2.9 give an overview of different operating systems supported by embedded databases we investigated. There are basically four categories of operating systems that investigated embedded databases support:

- Operating systems traditionally used by desktop computers. In this category you find the most common operating systems like, Microsoft Windows, different versions of UNIX and Linux.
- Operating systems for hand-held computers. In this category you find Palm OS and Windows CE/PocketPC. These operating systems demand small memory requirements but still have most of the functionality of the ordinary desktop computer operating systems. A good interaction and interoperability between the operating systems on the hand-held computer and a standard desktop is also important. This is recognized by all databases, if you consider the complete Pervasive.SQL family as one database system.
- Real-time operating systems. In this category you find systems like VxWorks and QNX.
- Smart card operating systems. These systems, like Java Card and MultOS, are made for very small environments, typically no more than 32kb. The Java Card operating

[0] Smart card operating system for Windows.
[1] Pervasive.SQL for smart cards.
[2] Pervasive.SQL for embedded systems.
[3] Pervasive.SQL for mobile systems.

system is simply an extended Java virtual machine.

Most commercial embedded database systems in this survey support a real-time operating system (see table 2.9). Additionally, Pervasive.SQL supports operating systems for hand-held computers and smart card operating systems, in the Pervasive.SQL for mobile systems version and the Pervasive.SQL for smart card version, respectively.

2.3.10 Concurrency control

Concurrency control (CC) *serializes* transactions, retaining the *ACID properties*. All transactions in a database system must fulfill all four ACID properties. These are:

- Atomicity: A transaction is indivisible, either it is run to completion or it is not run at all.
- Consistentency: It must not violate logical constraints enforced by the system. For example, a bank transaction must follow the law of conservation of money. This means that after a money transfer, the sum of the receiver and the sender must be unchanged.
- Isolation: A transaction must not interfere with any other concurrent transaction. This is also referred to as serialization of transactions.
- Durability: A transaction is, once committed, written permanently to the database.

If two transactions have some data elements in common and are active at the same time, a data conflict might occur. Then it is up to the concurrency control to detect and resolve this conflict. This is most commonly done with some form of locking mechanism. It substitutes the semaphore guarding a global data in a real-time system. There are two fundamentally different approaches on achieving this serialization, an optimistic and a pessimistic approach.

The most common pessimistic algorithm is two-phase-locking (2PL) algorithm proposed in 1976 by Eswaran et al. [34]. This algorithm consists, as the name indicates, of two phases. In the first phase all locks are collected, no reading or writing to data can be performed before a lock has been obtained. When all locks are collected and the updates have been done, the locks are released in the second phase.

Optimistic concurrency control (OCC) was first proposed by Kung and Robinson [57] in 1981. This strategy takes advantage of the fact that conflicts in general are rather rare. The basic idea is to read and update data without regarding possible conflicts. All updates are, however, done on temporary data. At commit-time a conflict detection is performed and the data is written permanently to the database if no conflict was detected. However the conflict detection (verification phase) and the update phase need to be atomic, implying some form of locking mechanism. Since these two phases take much shorter time than a whole transaction, locks that involves multiple data, or the whole database, can be used. Since it is an optimistic approach, performance degrades when congestion in the system increases.

For real-time databases a number of variants of these algorithms have been proposed that suit these databases better [114, 10]. These algorithms try to find a good balance between missed deadlines and temporally inconsistent transactions. Song and Liu showed that OCC algorithms performed poorly with respect to temporal consistency in real-time systems that consist of periodic activities [100], while they performed very well in systems where transactions had random parameters, e.g., event-driven systems. However, it has been shown that the strategies and the implementation of the locking and abortion algorithms significantly determine the performance of OCC [46]. All databases except the Polyhedra DBMS use pessimistic CC (see table 2.10). Since Polyhedra is an event-driven

DBMS system	Pessimistic CC	Optimistic CC	No CC
Pervasive.SQL	x		
Polyhedra		x	
Velocis	x		
RDM	x		
Berkeley DB	x		x
TimesTen	x		

Table 2.10: Concurrency control strategies used in different embedded database systems.

system, OCC is a natural choice. Furthermore, Polyhedra is a main-memory database with very fast execution of queries, the risk of a conflict is thus reduced.

Pervasive.SQL has two kind of transactions: exclusive and concurrent. An exclusive transaction locks a complete database file for the entire duration of the transaction, allowing only concurrent non-transactional clients to perform read-only operations on the file. A concurrent transaction, however, uses read and write locks with much finer granularity, e.g., page or single data locks.

The Berkeley DB has three configurations: (i) The non-concurrent configuration allows only one thread at a time to access the database, removing the need for concurrency control. (ii) The concurrent version allows concurrent readers and concurrent writers to access the database simultaneously. (iii) The concurrent transactional version allows for full transactional functionality with concurrency control, such as fine grain locking and database atomicity.

Similar to Berkeley DB, TimesTen also has a “no concurrency” option, in which only a single process can access the database. In addition, TimesTen supports two lock sizes: data-store level and record level locking.

2.3.11 Recovery

One important issue for databases is persistence, i.e., data written and committed to the database should remain until it is overwritten or explicitly removed, even if the system fails and have to be restarted. Furthermore, the state of all ongoing transactions must be saved to be able to restore the database to a consistent state upon recovery, since uncommitted data might have been written to the database. There is basically two different strategies for backup restoration: roll-back recovery, and roll-forward recovery, normally called “forward error recovery”. During operation continuous backup points occurs where a consistent state of the database is stored on a non-volatile media, like a hard-drive. These backup points are called checkpoints.

In roll-back recovery you simply restart the database using the latest checkpoint, thus guaranteeing a consistent database state. The advantage with this approach is that it does not require a lot of overhead, but the disadvantage is that all changes made to the database after the checkpoint are lost.

When roll-forward recovery is used, checkpoints are stored regularly, but all intermediate events, like writes, commits and aborts are written to a log. In roll-forward recovery, the database is restored to the last checkpoint, and all log entries are performed in the same order as they have been entered into the log. When the database has been restored to the state it was at the time of the failure, uncommitted transactions are roll-backed. This approach requires more calculations at recovery-time, but will restore the database to the state it was in before the failure. Pervasive.SQL offers three different types of recovery mechanisms, as well as the option to ignore recovery (see table 2.11). This can also be selected parts of the database. Ignoring check-pointing for some data can be useful for systems where some data must be persistent while other data can be volatile. Going back to our example

DBMS system	Roll-forward	Roll-back	Journalling	No recovery
Pervasive.SQL	x			x
Polyhedra			x	x
Velocis	x			
RDM	x			
Berkeley DB		x		
TimesTen		x		x

Table 2.11: Strategies for recovery used in different embedded database systems.

in section 1.1, the desired level given by the user interface, and the regulator parameters need to be persistent, while the current reading from the level indicator must not, since by the time the database is recovered the data will probably be stale. The second option is shadow-paging, which simply makes a copy of the database file and makes the changes there, and when the transaction is complete, the data is written back to the original page. A similar approach is the delta-paging, which creates an empty page for each transaction, and only writes the changes (delta-values) to it. At the end of a transaction the data is written back to the original, just as for shadow-paging. With both of these techniques, the database is consistent at all times. The last option is to use logging and roll-forward recovery.

The default configuration for Polyhedra is non-journalling, which means that no automatic backup is performed, but the application is responsible for saving the database to non-volatile memory. This is particularly important since this system is a main memory database, and if no backups is taken all data is, of course, lost. When journalling is used, the user can select which tables that should be persistent and the journaller will write an entry in the log when a transaction involving persistent data is committed. There are two approaches when data is persistently written, direct journalling and non-blocking journalling. The direct journalling approach writes data to persistent storage when the load on the DBMS is low. However, the database is blocked during this process and this can cause problems for systems with time-critical data. The second approach can then be used, and that is to use a separate journalling process responsible for writing entries persistent.

TimesTen supports three different levels of recovery control. The most stringent guarantees transaction atomicity and durability upon recovery, in which case the transaction is not committed until the transaction is written onto disk. The second level of control guarantees transaction atomicity but not durability. Upon commit, the log entry is put into a queue and is later written to disk. Upon recovery the database will be consistent, but committed transactions that have not yet been written to disk might be lost. The lowest level of recovery control is no recovery control. Neither atomicity nor durability is guaranteed. This option might not be as inappropriate as it might seem at a first glance, since TimesTen is a main-memory database, often used in systems with data that would be stale upon recovery, e.g., a process controller.

2.3.12 Real-time properties

Even though none of the commercial database systems in this survey can be classified as a real-time database from a hard real-time systems perspective, most of the systems are successfully used in time-critical systems (to different extent). Ericsson uses Polyhedra in their 3G platform for wireless communication. Delcan Corporation uses Velocis DB in a traffic control system that handles about 1200 traffic lights simultaneously even though this database primarily is used in web and e-Commerce applications. The database systems are simply so fast and efficient and have so many options for fine tuning their performance that the application systems work, even though the DB systems cannot itself guarantee predictability.

A question one could ask is: How would we use one of these systems in a hard real-time system? Are there any ways to minimize the unpredictability to such a degree that a pessimistic estimation would fulfill hard real-time requirements? In our opinion it is. For an event triggered real-time system, Polyhedra would fit well. Let us use our example application again. When new sensor values arrive the I/O management, the database is updated. If there is a change in the alarm data the CL code that is connected to that data is generating an event to trigger the alarm task. Built into the system is also the minimum inter-arrival interval mentioned in the interfaces section. So by using the Polyhedra database to activate tasks, that will then be scheduled by a priority based real-time operating system, an acceptable degree of predictability is achieved. To further increase guarantees that critical data will be updated is to use a so called watchdog, that activates a task if it has not been activated for a predefined time. The memory and size predictability would be no problem since they have specified the exact memory overhead for every type of object. However, temporal validity is not addressed with this approach.

For statically scheduled real-time systems the RDM "Network access" could be able to run with predictable response times. This because the order of the data accesses is known a priori and can therefore be linked together in a chain, and the record lookup index is bypassed.

2.4 Current state-of-the-art from research point of view

There exists a number of databases that could be classified as pure real-time databases. However these databases are research project and are not yet on the commercial market. We have selected a number of real-time database systems and compared them with respect to the following criteria:

- Real-time properties. The criteria enables us to discuss real-time aspects of the systems and how they are implemented.
- Distribution. The criteria enables us to elaborate about different aspects with respect to distributing the database.
- Transactions workload characteristics. The criteria enables us to discuss how the transaction system is implemented.
- Active behavior. The criteria enables us to elaborate about the active aspects for some of the systems.

The systems selected in this survey represent some of the more recent real-time database platforms developed. These systems are representative of the ongoing research within the field.

STRIP The STanford Real-time Information Processor (STRIP) [6] is a soft real-time database system developed at Stanford University, US. STRIP is built for the UNIX operating system, is distributed and uses streams for data sharing between nodes. It is an active database that uses SQL3-type rules.

DeeDS The Distributed active, real-time Database System (DeeDS) [9] supports both hard and soft real-time transactions. It is developed at the University of Skövde, Sweden. It uses extended ECA rules to achieve an active behavior with real-time properties. It is built to take advantage of a multiprocessor environment.

BeeHive The BeeHive system [103] is a virtual database developed at the University of Virginia, Charlottesville, US, in which the data in the database can be located in multiple locations and forms. The database supports four different interfaces namely, a real-time interface, a quality of service interface, a fault-tolerant interface, and a security interface.

REACH The REACH system [116], developed at the Technical University of Darmstadt, Germany, is an active object-oriented database implemented on top of the object-oriented database openOODB. Their goal is to archive active behavior in an OO database using ECA rules. A benchmarking mode is supported to measure execution times of critical transactions. The REACH system is intended for soft real-time systems.

RODAIN The RODAIN system [108] developed at the University of Helsinki, Finland, is a firm real-time database system that primarily is intended for telecommunication. It is designed for high degree of availability and fault-tolerance. It is tailored to fit the characteristics of telecommunication transactions identified as short queries and updates, and long massive updates.

ARTS-RTDB The ARTS-RTDB system [52], developed at the University of Virginia, Charlottesville, US, supports both soft and hard real-time transactions. It uses imprecise computing to ensure timeliness of transactions. It is built on top of the ARTS real-time operating system [110].

2.4.1 Real-time properties

STRIP

The STRIP [6] system is intended for soft real-time systems in which the ultimate goal is to make as many transactions as possible commit before their deadlines. It is developed for the UNIX operating system which might seem odd, but since STRIP is intended to run in open systems UNIX was selected. Even though UNIX is not a real-time operating system it has, when used in the right way, turned out to be a good operating system for soft real-time systems since a real-time scheduling emulator can be placed on top of the UNIX scheduler which sets the priorities according to some real-time scheduling algorithm, a real-time behavior can be achieved. However, there have been some problems that needed to be overcome while implementing STRIP that is related to real-time scheduling in UNIX. For example to force a UNIX schedule not to adjust the priorities of processes as they are running. Non-real-time operating systems often have mechanisms to dynamically adjust the priorities of processes during run time to add to throughput and fairness in the system. By using Posix UNIX, a new class of processes whose priorities are not adjustable by the scheduler was provided. In [4] versions of the earliest deadline and least slack algorithms was emulated on top of a scheduler in a traditional operating system, and the results showed that earliest deadline using absolute deadline and least slack for relative deadline performed equal or better than their real-time counterparts, while the emulated earliest deadline for relative deadline missed more deadlines than the original EDF algorithm for low systems load. However, during high load the emulated algorithm outperformed the original EDF. In STRIP EDF, highest value first, highest density value first or a custom scheduling algorithm can be used to schedule transactions. To minimize unpredictability, the STRIP system is a main-memory database, thus removing disk I/O.

DeeDS

DeeDS [9] is intended for both hard and soft real-time systems. It is built for the OSE delta real-time operating system developed by ENEA DATA [31]. This distributed hard real-time operating system is designed for both embedded uniprocessor and multiprocessor systems.

If used in a multiprocessor environment the CPUs are loosely coupled. A more detailed description of OSE delta can be found in [43]. The DeeDS system consists of two parts, one part that handles non critical system services and one that handles the critical. All critical systems services are executed on a dedicated processor to simplify overhead cost and increase the concurrency in the system. The DeeDS system is, as the STRIP system, a main memory database.

BeeHive

BeeHive utilizes the concept of data deadline, forced-wait and the data deadline based scheduling algorithms [113]. These concepts assure that transactions are kept temporally consistent by assigning a more stringent deadline to a transaction based on the maximum allowed age of the data elements that are used in the transaction, thus the name data deadline. The key concept of forced wait is to, if a transaction cannot complete before its data deadline, postpone it until data elements get updated, thus giving the transaction a later data deadline. These transactions can then be scheduled using, for example, the earliest data-deadline first (EDDF) or data deadline based least slack first (DDLSF) algorithms [113].

For real-time data storage a four level memory hierarchy is assumed: main memory, non-volatile RAM (e.g., Flash), persistent disk storage, and archival storage (e.g., tape storage) [103]. The main memory is used to store data that is currently in use by the system. The non-volatile RAM is used as a disk-cache for data or logs that are not yet written to disk. Furthermore, system data structures like lock tables and rule bases can be stored in non-volatile RAM. The disk is used to store the database, just like any disk-based database. Finally, the tape storage is used to make backups of the system. By using these four levels of memory management a higher degree of predictability can be achieved than for entirely disk-based databases, since the behavior can be somewhat like a main-memory database even though it is a disk-based system.

REACH

REACH is built on top of Texas Instruments OpenOODB system, which is an open database system. By an open system we refer to a system that can be controlled, modified, and partly extended by developers and researcher [112]. The OpenOODB system is by itself not a real-time database system, but an object-oriented database. Some efforts has been done in the REACH project to aid the user with respect to system predictability [18] such as milestones, contingency plans, a benchmarking tool, and a trace tool.

Milestones are used to monitor the progress of transactions. If a milestone is missed, the transaction can be aborted and a contingency plan is released instead. This contingency plan should be implemented to handle a missed deadline situation by either degrading the system in a safe way, or produce a good-enough result before the original transaction's deadline. One could say that a contingency plan in cooperation with milestones works as a watch-dog that is activated when a deadline is about to be missed.

The REACH system has a benchmarking tool that can be used to measure execution times for method invocations and event triggering. The REACH trace tool can trace the execution order in the system. If the benchmarking tool is used together with the trace tool, system behavior and timeliness could be determined.

RODAIN

The RODAIN system is a real-time database system intended for telecommunication applications. The telecommunication environment is a dynamic and complex environment that deals with both temporal data and non-temporal data. A database in such a system must, support both soft and firm real-time transactions [108]. However, the belief is that

hard real-time databases will not be used in telecommunication in the near future since they generally are to too expensive to develop or use to suit the market [108].

One of the key issues for RODAIN is availability, thus fault-tolerance is necessary. The RODAIN system is designed to have two nodes, thus giving full database replication. This is especially important since the database is a main-memory system. It uses a primary and a mirror node. The primary database sends logs to the mirror database which in turn acknowledges all calls. The communication between the nodes is assumed to be reliable and have a bounded message transfer delay. A watchdog monitor keeps track of any failing subsystem and can instantly swap the primary node and the mirror node in case of a failure, see figure 2.5. The failed node always recover as a mirror node and loads the database image from permanent storage. Furthermore, the User Request Interpreter system (URIS) can keep track of all ongoing transactions and take appropriate actions if a transaction fails. Since both the primary and the mirror node have a URIS no active transactions will be lost if a failure in the primary system leads to a node swap between the primary and the mirror. Figure 2.5 shows the architecture of RODAIN. The subsystem OODBMS handles all traditional database activities, like concurrency control, physical storage, schema management etc. There are three interfaces to the database, the traditional user interface, referred to as the Applications interface, the DBMS nodes interface, and the Mirror interface. Noteworthy is the distinction between mirror nodes and distributed DBMS nodes. The purpose of the mirror nodes is to ensure a fault-tolerant running, while the distribution nodes are used for normal database distribution.

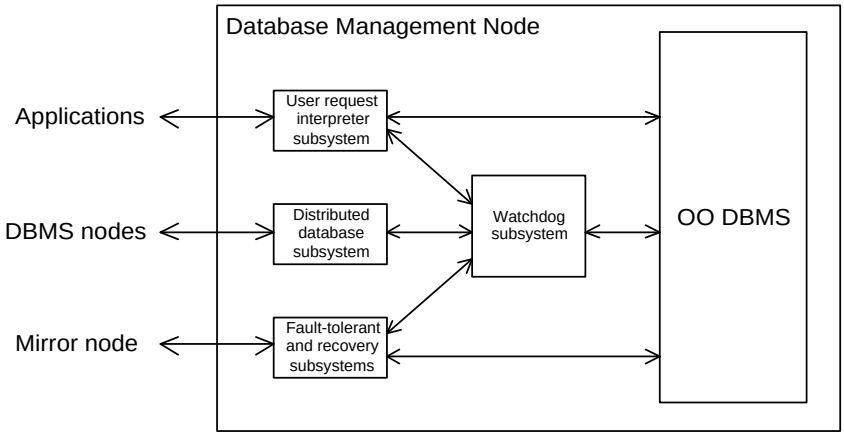

Figure 2.5: *RODAIN DBMS node. Figure from [108]*

ARTS-RTDB

The relational database ARTS-RTDB [52] incorporates an interesting feature called imprecise computing. If a query, when its deadline expires, is not finished, the result produced so far can be returned to the client if the result can be considered meaningful. For example, a query that calculates the average value of hundreds of values in a relation column. If an adequate amount of values has been calculated at the time of deadline, the result could be considered meaningful but imprecise and it is therefore returned to the client anyway.

ARTS-RTDB is built on top of the distributed real-time operating system ARTS, developed by Carnegie Mellon University [110]. ARTS schedules tasks according to a time-varying value function, which specifies both criticality and importance. It does support both soft and hard real-time tasks.

In ARTS-RTDB the most critical data operations has been identified to be the INSERT, DELETE, UPDATE, and SELECT operations. Special care has therefore been made to

optimize the system to increase the efficiency for the four basic operations: insert, delete, update, and select. According to [52] many real-time applications almost only use these operations at run-time.

2.4.2 Distribution

To increase concurrency in the system, distributed databases can be used. Distributed in a sense that the database copies reside on different computer nodes in a network. If the data is fully replicated over all nodes, applications can access any node to retrieve a data element. Unfortunately, maintaining different database nodes consistent with each other is not an easy task, especially if timeliness is crucial. One might have to trade consistency for timeliness. A mechanism that continuously tries to keep the database nodes as consistent with each other as possible is needed. Since the system is distributed all updates to a database node must be propagated via message passing, or similar, thus adding significantly to the database overhead because of the added amount of synchronization transactions in the system. One of the most critical moments for a transaction deployed in a distributed database is the point of commit. At this point all the involved nodes must agree upon committing the transaction or not.

One of the most commonly used commit protocols for distributed databases is the two-phase commit protocol [39]. As the name of the algorithm suggests the algorithm consists of two phases, the prepare-phase and the commit-phase. In the prepare phase, the coordinator for the transaction sends a `prepare` message to all other nodes and then waits for all answers. The nodes then answers with a `ready` message if they can accept the commit. If the coordinator receives `ready` from all nodes a `commit` message is broadcasted, otherwise the transaction is aborted. The coordinator then finally waits for a `finished` message from all nodes to confirm the commit. Hereby is consistency guaranteed between nodes in the database.

DeeDS

The DeeDS [9] system has a less strict criteria for consistency, in order to enforce timeliness. They guarantee that each node has a local consistency, while the distributed database might be inconsistent due to different views of the system on the different nodes. This approach might be suitable for systems that mostly rely on data that is gathered locally, but sometimes uses data imported from other subsystems. Consider an engine that has all engine data, e.g., ignition control, engine speed and fuel injection, stored in a local node of a distributed database. The timeliness of these local data items are essential in order to run the engine. To further increase the efficiency of the engine, remotely gathered data, like data from the transmission box, with less critical timing requirements can be distributed to the local node.

STRIP

The concept of streams communicating between nodes in a distributed system has been recognized in STRIP [6]. Nodes can stream views or tables to each other on a regular basis. The user can select if whole tables/views or delta tables, which only reflects changes to the tables/views, should be streamed. Furthermore it is selectable if the data should be streamed, periodically when some data has reached a predefined age or only after an explicit instruction to stream. In figure 2.6 we can see the process architecture of STRIP. The middle part of the figure shows the execution process, the transaction queue, the result queue and the request process. This part makes the query layer of the database and can be compared to an ordinary database that processes transactions and queries from the application. Remote application addresses queries or transactions to the requesting process, which in turn places the transaction in the transaction queue. However, local applications can

access the transaction queue directly via a client library, thus minimizing overhead. Each execution processes can execute a transaction from the transaction queue or an update transaction from one of the update queues. The update queues are fed from the incoming update streams. When a table/view needs to be sent to an outgoing stream, one of the execution processes enqueues it to the update queue read by the export process. It has been shown that when the update frequency is low, the import and export processes can run at a high priority without significantly affecting transaction response times but when the frequency increases the need for a different update scheduling policy is necessary [6], see section 2.4.3.

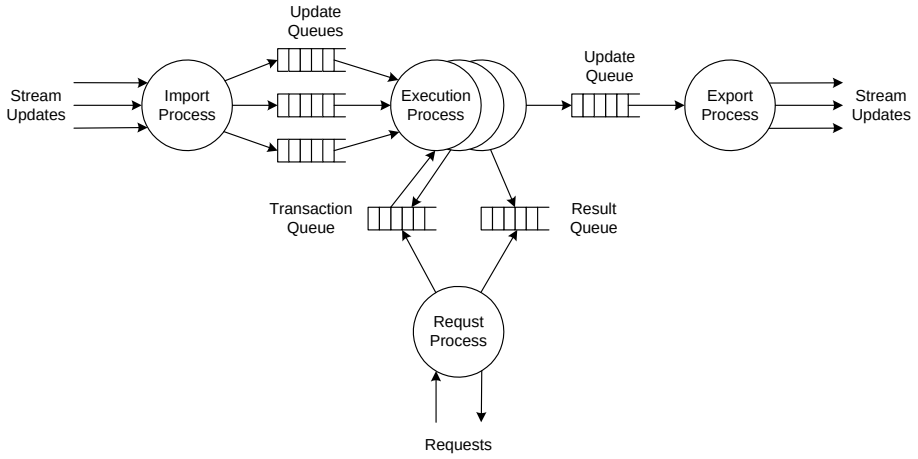

Figure 2.6: *The process architecture of the STRIP system. Figure from [6].*

BeeHive

Unlike most databases BeeHive is a virtual database, in the sense that in addition to its own data storage facility it can use external databases and incorporate them as one database. These databases can be located either locally or on a network, e.g., the Internet.

A virtual database is defined as a database that organizes data scattered through the Internet into a queriable database [40]. Consider a web service that compares prices of products on the Internet. When you enter the database you can search for a specific product, and the service will return a list of Internet retailers and to what price they are selling the product. The database system consists of four parts [40]:

- Wrappers. A wrapper is placed between a web-page or database and the virtual database. In the web-page case, the wrapper converts the textual content of the web-page into a relational format understandable by the virtual database. When using a wrapper in the database case, the wrapper converts the data received from the database into the virtual database format. Such a wrapper can be implemented using a high level description language, e.g., Java applets.
- Extractors. The relations received from the wrapper most often contains unstructured textual data in which the interesting data needs to be extracted.
- Mappers. The mapper maps extracted relations into a common database schema.
- Publishers. The publisher is used to present the database to the user, it has similar functionality as a traditional database.

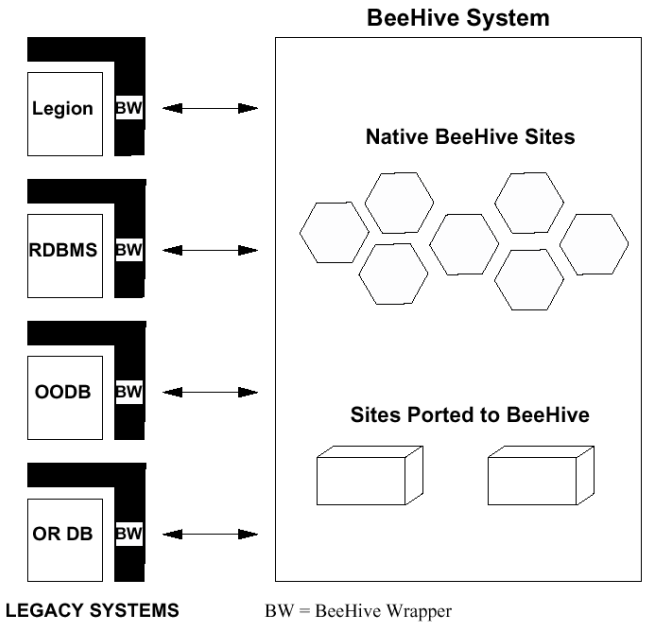

Figure 2.7: *The BeeHive virtual database. Figure from [103]*

BeeHive can communicate with other virtual databases, web-browsers or independent databases using wrappers. These wrappers are Java applets and may use the JDBC standard, see figure 2.7.

RODAIN

RODAIN system supports data distribution. The nodes can be anything between fully replicated to completely disjoint. Their belief is that only a few requests in telecommunications need access to more than one database node and that the request distributions among the databases in the system can be arranged to be almost uniform [108].

ARTS-RTDB

ARTS-RTDB has been extended to support distribution. The database nodes use a shared file that contains information binding all relations to the server responsible for a particular relation. Since the file is shared between the nodes, it is to be treated as a shared resource and must therefore be accessed using a semaphore. It is believed that if relations are uniformly distributed between the nodes and if no hot-spot relations exist, an increase in performance will be seen. A hot-spot relation is a relation that many transactions use, and such a relation can lead to a performance bottleneck in the system.

2.4.3 Transaction workload characteristics

DeeDS

DeeDS supports both sporadic (event-triggered) and periodic transactions. There are two classes of transactions: critical (hard transactions) and non-critical (soft transactions). To ensure that a deadline is kept for a critical transaction, milestone monitoring and contingency plans are used. A milestone can be seen as a deadline for a part of the transaction.

If a milestone is passed, it is clear that the transaction will not make its deadline, a contingency plan can be activated and the transaction is aborted. A contingency plan should, if implemented correctly, deal with the consequences of a transaction aborting. One way to compute less accurate result and present it in time, similar to imprecise computing used by ARTS-RTDB (see section 2.4.1). Milestones and contingency plans are discussed further in [32]. The timing requirements for a transaction is passed to the system as a parameterized value function, this approach reduces the computational cost and the storage requirements of the system.

The scheduling of transactions is made online in two steps. First, a sufficient schedule is produced. This schedule meets the minimum requirements, for example all hard deadlines and 90 percent of the soft deadlines are met. Secondly, the event monitors worst-case execution time is subtracted from the remaining allowed time for the scheduler during this time slot, and this time is used to refine and optimize the transaction schedule [72].

STRIP

STRIP supports firm deadlines on transactions [7], thus when a transaction misses its deadline it will be aborted. There are two classes of transactions in STRIP, low value and high value transactions. The time when transactions are scheduled for execution is determined by transaction class, value density, and choice of scheduling algorithm. The value density is the ratio between the transactions value and the remaining processing time. In the STRIP system there is a distinction between transactions and updates. Transactions are considered to be read, write and update operations initiated by the user or the application, whereas updates are considered to be update operations initiated from other database nodes in a distributed system. The scheduling of transactions is in competition with the scheduling of updates, therefore updates also have the same properties as transactions with respect to classes, values, and value density. STRIP has four scheduling algorithms, Updates First (UF), Transactions First (TF), Split Updates (SU), and Apply Updates on Demand (OD).

- Updates First schedules all updates before transactions, regardless of the value density of the transactions. It is selectable if a running transaction is preempted by an update of if it should be allowed to commit before the update is executed. For a system with a high load of updates this policy can cause long and unpredictable execution times for transactions. However, for systems where consistency between nodes and where temporal data consistency is more important than transaction throughput, this can be a suitable algorithm. Further, this algorithm is also suitable for systems where updates are prioritized over transactions. Consider a database running stock exchange transactions. A change in price for a stock must be performed before any pending transactions in order to get the correct value of the transaction.

- Transactions First is the opposite of update first. Transactions are always scheduled in favor of updates. However a transactions cannot preempt a running update. This because updates most often have short execution time compared to transactions. This algorithm suits systems that values throughput of transactions higher than temporal consistency between the nodes. Transactions First might be useful in a industrial controlling system. If, in an overloaded situation, the most recent version of a data element is not available, the system might gain in performance from being able to run its control algorithms using an older value.

- Split Updates make use of the different classes of updates. It schedules updates and transactions according to the following order: high value updates, transactions, and low value updates. This algorithm combines the two previous algorithms and would suit a system where one part of the data elements is crucial with respect to temporal consistency, at the same time as transaction throughput is important.

- Apply Updates on Demand execute transactions before updates, but with the difference that when a transaction encounters stale data, the update queue is scanned for an update that is waiting to update the stale data element. This approach would enable a high throughput of transactions with a high degree of temporal consistency. However, calculating the execution time for a transaction is more difficult since the worst case is that all data elements used in the transaction has pending updates. Applying updates on demand resembles about the ideas of triggered updates [8].

Since transaction deadlines are firm, thus transactions are valueless after their deadline has expired, a mechanism called feasible deadlines can be used for transactions. The mechanism aborts transactions that has no chance of committing before their deadline.

BeeHive

The fact that the database is virtual and may consist of many different kinds of underlying databases is transparent. The user accesses the database through at least one of the four interfaces provided by BeeHive: FT API, RT API, Security API, and QoS API (see figure 2.8). FT API is the fault-tolerant interface. Transaction created with this interface will have some protection against processor failures and transient faults due to power glitches, software bugs, race conditions, and timing faults. RT API is the real-time interface, which enables the user to specify time constraints and typical real-time parameters along with transactions. The security interface can be used by application that for instance demands authorization for users and applications. Furthermore encryption can be supported. The Quality of Service (QoS) interface is primarily used for multimedia transactions. The user can specify transactions demands on quality of service.

When a transaction arrives from any interface, it is sent to the service mapper, which transforms it to a common transaction format used for all transactions in the system, regardless of its type, e.g., real-time transaction. It is then passed on to the resource planner, which determines what resources will be needed. Furthermore, it is passed to the admission controller, which in cooperation with the resource planner, decides if the system can provide a satisfactory service for the transaction. If that is the case, it is sent to the resource allocation module, which globally optimizes the use of resources in the system. Finally the transaction is sent to the correct resource, e.g., a network or the operating system etc.

RODAIN

Five different areas for databases in telecommunication is identified [93]:

1. Retrieval of persistent customer data.
2. Modifications of persistent customer data.
3. Authorization of customers, e.g., PIN codes.
4. Sequential log writing.
5. Mass calling and Tele voting. This can be performed in large blocks.

From these five areas, three different types of transactions can be derived [77]: short simple queries, simple updates, and long massive updates. Short simple queries are used when retrieving customer data and authorizing customers.Simple updates are used when modifying customer data and writing logs. Long massive updates are used when tele voting and mass calling is performed.

The concurrency control in RODAIN supports different kinds of serialization protocols. One protocol is the τ -serialization in which a transaction may be allowed to read old data as long as the update is not older than a predefined time [108].

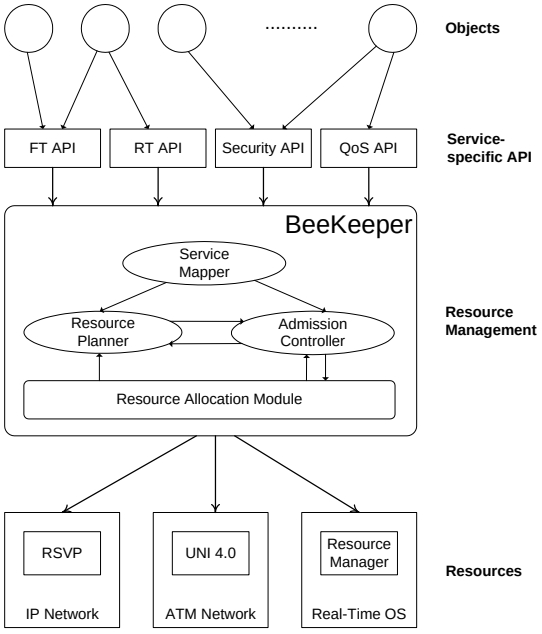

Figure 2.8: *The resource manager in BeeHive. Figure from [103]*

Apart from a transaction deadline, an isolation level can also be assigned to a transaction. A transaction running on a low isolation level accepts that transactions running on a higher isolation level are accessing locked objects. However, it cannot access objects belonging to a transaction with a high degree of isolation.

ARTS-RTDB

ARTS-RTDB uses a pessimistic concurrency control, strict two phase locking with priority abort [52]. This means that as soon as a higher prioritized transaction wants a lock that is owned by a transaction with low priority, the low-level transaction is aborted. To avoid the costly process of rolling back the aborted transaction, all data writing is performed on copies. At the point of commit, the transaction asks the lock-manager if a commit can be allowed. If this is the case, the transaction invokes a subsystem that writes all data into the database.

2.4.4 Active databases

Active databases can perform actions not explicitly requested from the application or environment. Consider a data element z, which is derived from two other data elements, x and y. The traditional way to keep z updated would be to periodically poll the database to determine if x or y have been altered. If that is the case, a new value of z would be calculated and the result written back to the database. Periodic polling is often performed at a high cost with respect to computational cost and will increase the complexity of the application task schedule. A different approach would be to introduce a trigger that would cause an event as soon as either x or y has been changed. The event would in its turn start a transaction or similar that would update z. This functionality would be useful for other purposes than to update derived data elements. It could, for example, be used to enforce boundaries of data elements. If a data element, e.g., data that represents a desired water level in our example application in figure 1.1, has a maximum and minimum allowed value than this restriction can be applied in the database instead of in the application. An attempt

to set an invalid value could result in some kind of action, e.g., an error exception or a correction of the value to the closest valid value. By applying this restriction in the database it is abstracted away from the application, thus reducing the risk of programming errors or inconsistency between transactions that use this data element.

One way to handle such active behavior is through the use of ECA rules [35], where ECA is an abbreviation of Event-Condition-Action-rules, see figure 2.9. The events in an

```
ON <event E>
   IF <condition C>
      THEN <action A>
```

Figure 2.9: *The structure of an ECA rule.*

active database need to be processed by an event manager, which sends them to the rule manager. The rule manager in its turn locates all ECA rules that are affected by the newly arrived event and checks their conditions. For those ECA rules whose conditions are true, the action is activated. There are, however, three distinct approaches on when and how to execute the actions [32]:

- Immediate: When an event arrives, the active transaction is suspended and condition evaluation and possibly action execution is done immediately. Upon rule completion, the transaction is resumed. This approach affects the response time of the transaction.
- Deferred: When an event arrives, the active transaction is run to completion and then condition evaluation and possibly action execution is done. This approach does not affect the response time of the triggering transaction.
- Decoupled: When using this approach, condition evaluation and possibly action execution are performed in one or several separate transactions. This has the advantage that these updates will be logged even if the triggering transaction aborts.

One disadvantage with active databases is that the predictability, with respect to response times, is decreased in the system. This is because of the execution of the rules. For immediate or deferred rules the response time of the triggering transaction is prolonged because of the execution of rules prior to its point of commit. To be able to calculate a response time for such a transaction, the execution time of all possible rules that the transaction can invoke. These rules can in their turn activate rules, so called cascading, which can create arbitrary complex calculations. When decoupled rules are used, transactions can generate new transactions in the system. These transactions and their priorities, response times, and possible rule invocations must be taken in account when calculating transaction response time, resulting in very pessimistic worst-case-execution times.

Some extensions to ECA rules which incorporates time constraints have been studied in [50]. For a more detailed description about ECA rules, active databases and active real-time databases see [32, 98]

STRIP

The STRIP rule system checks for events at commit-time of each transaction T, and for all events whose condition is true a new transaction T' is created, which executes the execution clause of the rule. Thus, the STRIP system uses a decoupled approach. T' is activated for execution after the triggering transaction has committed. By default T' is released immediately after commit of the triggering transaction, but it can be delayed using the optional `after` statement, see figure 2.10.

The `after` statement enables all rule execution clauses, activated during the time window between rule evaluation and the execution clause specified by the `after` statement, to

```
WHEN <eventlist>
   [IF <condition>]
   THEN
      EXECUTE <action>
      [AFTER <time-value>]
```

Figure 2.10: *A simplified structure of a STRIP-rule. The optional* `AFTER` *-clause enables for batching of several rule-executions in order to decrease computational costs.*

	Sequential	Parallel
Independent	T_r independent of Tt	
Dependent	T_r cannot *start* until Tt *commits*, otherwise it is *discarded*.	T_r cannot *complete* until Tt *commits*, otherwise it is *aborted*.
Exclusive	T_r cannot *start* until Tt *aborts*, otherwise it is *discarded*.	T_r cannot *complete* until Tt *aborts*, otherwise it is *aborted*.

Table 2.12: *Table showing all decoupled execution modes supported by the DeeDS system.* Tt *is the triggering transaction and* T_r *is the rule transaction. Table from [33].*

be batched together. If multiple updates on a single data element exist in the batch, only the ones necessary to derive the resulting value of that data element are executed, thus saving computational cost [5]. To have the execution clause in a separate transaction simplifies rule processing. The triggering transaction can commit regardless of the execution of the executions clauses, since the condition clause of the rules cannot alter the database state in anyway. Possible side-effects of the rules are then dealt with in a separate transaction. One disadvantage with decoupled execution clauses is that the condition results and transition data are deleted when the triggering transaction commits, thus, data is not available for the triggered transaction. This has been solved in STRIP by the `bind as` statement. This statement binds a created table to a relation that can be accessed by the execution clause.

DeeDS

Since DeeDS supports hard real-time systems the rule processing must be designed to retain as much predictability with respect to timeliness as possible. This is achieved by restricting rule processing. Cascading, for example, is limited so that unbounded rule triggering is avoided when a rule is executed. The user can define a maximum level of the cascade depth. If the cascading goes deeper than that an exception is raised. Furthermore, condition evaluation is restricted to logical expressions on events and object parameters, and method invocations. DeeDS uses extended ECA rules [72] that also allows for specifying time constraints to rules, e.g., deadlines.

Event monitoring has also been designed for predictability. Both synchronous and asynchronous monitoring is available. In synchronous mode the event monitor is invoked on time-triggered basis and results have shown that throughput is higher than if the asynchronous (event-triggered) event monitoring is used, but at the cost of longer minimum event delays. When synchronous mode is used, event bursts will not affect the system in an unpredictable way [72]. In DeeDS, the immediate and deferred coupling modes are not allowed. Instead when a rule is triggered and the condition states that the rule should be executed, a new transaction is created, which is immediately suspended. If a contingency plan exists for this rule, a contingency transaction is also created and suspended. All combinations of decoupled execution is supported in DeeDS [33], see table 2.12. Depending

on which mode that is selected, the scheduler activates the rule transactions according to the mode. The exclusive mode is used to schedule contingency plans.

REACH

REACH allows ECA rules to be triggered in all modes, immediate, deferred and decoupled. Decoupled rules are dependent or independent, furthermore they can be executed in parallel, sequential, or exclusive mode. The exclusive mode is intended for contingency plans. The ECA rules used in REACH have no real-time extensions.

There are however some restrictions on when to use the different coupling modes. Rules triggered by a single method event can be executed in any coupling mode, while a rule triggered by a composite event only cannot be run in immediate mode. This is because if the events that make up composite event originate in multiple transactions, no identification of the spawning transactions is possible [116]. Rules that are invoked by temporal events (time-triggered) can only be run in an independent decoupled mode, since they are not triggered by a transaction [19].

To simplify the creation of the rule-base a tool, GRANT, has been developed. This graphical tool prompts the user for necessary information, provides certain default choices, creates C++ language structures, and maps rules to C functions stored in a shared library [18].

The event manager in REACH consumes events according to two policies: chronological or most recent. The chronological approach executes all events in the order they arrive while the most recent strategy only executes the most recent instance of every event. What policy to use is dependent of the semantics of the application. Consider the example with the stock market database again. If, in an overload situation, the database is not able to immediately execute all stock price updates, only the most recent update of a stock is interesting since this update contains the current price of this stock. For a system like this, the most recent approach would be sufficient, while in a telephone billing system it might not. In a system like that, all updates to the database consist of a value to add to the existing record, therefore most recent is not sufficient here.

2.5 Observations

The databases presented represent a set of systems that basically have the same purpose, i.e., to efficiently store information in embedded systems or real-time systems. They mainly consist of the same subsystems, e.g., interfaces, index system, and concurrency control system. But yet the systems behave so differently. Some systems are primarily intended for event-driven system, while other suit statically scheduled systems more. Some systems are relational while other are object-oriented. Some systems are intended for a specific application, e.g., telecommunication, process control or multimedia application.

Furthermore, an interesting observation can be made about these systems. The research platforms focus more on functionality while the commercial systems are more interested in user friendliness, adaptability, and standards compliance. While the research platforms have detailed specification on new internal mechanisms that improve performance under certain circumstances, like new concurrency controls or scheduling policies, the commercial systems supports multiple interfaces to ease integration with the application and supports standards like ODBC, OLE DB, and JDBC. Looking back at the criteria mentioned by ABB in section 2.2.2, we can see that most of the criteria are fulfilled by the commercial products. To be able to combine these criteria with the technology delivered by the research platforms would be a contribution.

One important issue that arises is: Which embedded database system should one choose for a specific type of application? The task of choosing the best database system can be a long and costly process and probably not without compromises. A different approach

would be to have a more generic database that can be customized, possibly with aid of a tool, to fit a specific application. This will be further discussed in the next chapter.

Chapter 3

Component-based systems

Business enterprises and society in general are becoming increasingly dependent on computer systems. Almost every application domain is faced with the challenge of ensuring that data in the system is managed in a uniform manner and stored efficiently by the system, e.g., telecommunications, e-commerce, business enterprise, vehicle industry, embedded and mobile systems. As a result, database systems are used in a wide variety of application areas, and are required to provide efficient management of heterogeneous data of all shapes and sizes.

Database industry is today, more than ever, faced with requirements for rapid application development and deployment, delivery of data in the right form to the right place at the right time and, most importantly, minimized complexity for both end users and developers. Rapid development with low costs, high quality and reliability, which minimizes system complexity can be achieved if component-based software development paradigm is introduced in database development. This has been recognized and exploited by some major database vendors, e.g., Oracle, IBM, Informix and Sybase have all developed extensible database systems, which allow adding non-standard features and functionality to the extensible database system. However, existing component-based database solutions mainly focus on solving one problem, e.g., the Microsoft Universal Data Access architecture provides uniform manipulation of data from different data stores, the extensible Oracle8*i* server enables manipulation of non-standard data types.

It has been recognized that embedded and real-time systems could benefit from component-based development as well. Generally, by having well-defined reusable components as building blocks, not only development costs are reduced, but more importantly, verification and certification of components will only need to be done once. However, to efficiently reuse components for composing different systems, good composition rules for the system assembly must be provided, ensuring reliability of the composed system. Existing component-based embedded and real-time systems are mostly focused on ways of preserving real-time behavior of the system and/or enabling use of components in low-resource systems. A database system that can be tailored for different applications such that it provides necessary functionality but requires minimum memory footprint, and is highly integrated with the system would most likely be a preferable solution for integrating databases in embedded real-time systems. However, databases for real-time and embedded systems supporting fast development through reuse of components, do not exist, which makes these application domains deprived of the benefits that component-based development could bring. Thus, in this chapter we identify and contrast major issues in two component-based systems, database and embedded real-time, and inspect possibilities of combining these two areas.

Component-based systems can be analyzed only if certain knowledge of general component-based software engineering and its concepts such as components, architecture, and reuse are understood. Therefore, basic concepts of component-based software engi-

neering are introduced in section 3.1, and they are necessary in order to identify issues in different component-based systems and to obtain uniform criteria for surveying existing component-based database and embedded real-time systems. In section 3.2 we review components and component-based solutions in database systems. This is followed by real-time components and solutions in embedded real-time systems in section 3.3. Section 3.4 concludes this chapter with a tabular overview of investigated component-based systems and their characteristics.

3.1 Component-based software development

The need for transition from monolithic to open and flexible systems has emerged due to problems in traditional software development, such as high development costs, inadequate support for long-term maintenance and system evolution, and often unsatisfactory quality of software [17]. Component-based software engineering (CBSE) is an emerging development paradigm that enables this transition by allowing systems to be assembled from a pre-defined set of components explicitly developed for multiple usage. Developing systems out of existing components offers many advantages to developers and users. In component-based systems [17, 25, 26, 36]:

- Development costs are significantly decreased because systems are built by simply plugging in existing components.
- System evolution is eased because system built on CBSE concepts is open to changes and extensions, i.e., components with new functionality can be plugged into an existing system.
- Quality of software is increased since it is assumed that components are previously tested in different contexts and have validated behavior at their interfaces. Hence, validation efforts in these systems have to primarily concentrate on validation of the architectural design.
- Time-to-market is shortened since systems do not have to be developed from scratch.
- Maintenance costs are reduced since components are designed to be carried through different applications and changes in a component are, therefore, beneficial to multiple systems.

As a result, efficiency in the development for the software vendor is improved and flexibility of delivered product is enhanced for the user [59].

Component-based development also raises many challenging problems, such as [59]:

- Building good reusable components. This is not an easy task and a significant effort must be used to produce a component that can be used in different software systems. In particular, components must be tested and verified to be eligible for reuse.
- Composing a reliable system out of components. A system built out of components is in risk of being unreliable if inadequate components are used for the system assembly. The same problem arises when a new component needs to be integrated into an existing system.
- Verification of reusable components. Components are developed to be reused in many different systems, which makes the component verification a significant challenge. For every component use, the developer of a new component-based system must be able to verify the component, i.e., determine if the particular component meets the needs of the system under construction.

- Dynamic and on-line configuration of components. Components can be upgraded and introduced at run-time; this affects the configuration of the complete system and it is important to keep track of changes introduced in the system.

3.1.1 Software component

Software components are the core of CBSE. However, different definitions and interpretations of a component exist. Within a software architecture, a component is considered to be a unit of composition with explicitly specified interfaces and quality attributes, e.g., performance, real-time, reliability [17]. In systems where COM is used for a component framework, a component is generally assumed to be a self-contained, configurable binary package with precisely defined standardized interfaces [76]. A component can be also viewed as a software artifact that models and implements a well-defined set of functions, and has well-defined component interface (which do not have to be standard interfaces) [29].

Hence, there is no common definition of a component for every component-based system. The definition of a component clearly depends on the implementation, architectural assumptions, and the way the component is to be reused in the system. However, all component-based systems have one common fact, and that is: *components are for composition* [107].

While frameworks and standards for components today primarily focus on CORBA, COM, or JavaBeans, the need for component-based development has been identified in the area of operating systems (OSs). The aim is to facilitate OS evolution without endangering legacy applications and provide better support of distributed applications [73].

Common for all types of components, independent of their definition, is that they communicate with its environment through well-defined interfaces, e.g., in COM and CORBA interfaces are defined in an interface definition language (IDL), Microsoft IDL and CORBA IDL. Components can have more than one interface. For example, a component may have three types of interfaces: provided, required, and configuration interface [17]. Provided and required interfaces are intended for the interaction with other components, whereas configuration interfaces are intended for use by the user of the component, i.e., software engineer (developer) who is constructing an application out of reusable components. Each interface provided by a component is, upon instantiation of the component, bound to one or more interfaces required by other components. The component providing an interface may service multiple components, i.e., there can be one-to-many relation between provided and required interfaces. When using components in an application there might be syntactic mismatch between provided and required interfaces, even when the semantics of the interfaces match. This requires adaptation of one or both of the components or an adapting connector to be used between components to perform the translation between components

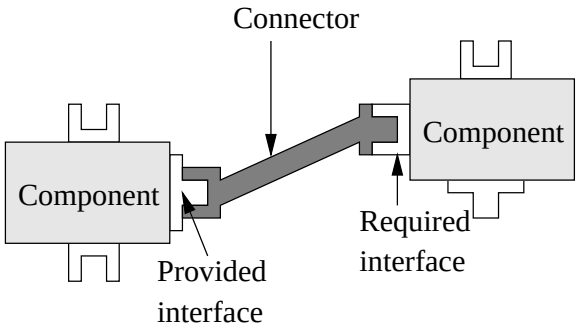

Figure 3.1: Components and interfaces

(see figure 3.1).

Independently of application area, a software component is normally considered to have black box properties [36, 29]: each component sees only interfaces to other components; internal state and attributes of the component are strongly encapsulated.

Every component implements some field of functionality, i.e., a domain [17]. Domains can be hierarchically decomposed into lower-level domains, e.g., the domain of communication protocols can be decomposed into several layers of protocol domains as in the OSI model. This means that components can also be organized hierarchically, i.e., a component can be composed out of subcomponents. In this context, two conflicting forces need to be balanced when designing a component. First, small components cover small domains and are likely to be reused, as it is likely that such component would not contain large parts of functionality unneeded by the system. Second, large components give more leverage than small components when reused, since choosing the large component for the software system would reduce the cost associated with the effort required to find the component, analyze its suitability for a certain software product, etc [17]. Hence, when designing a component, a designer should find the balance between these two conflicting forces, as well as actual demands of the system in the area of component application.

3.1.2 Software architecture

Every software system has an architecture, although it may not be explicitly modeled [71]. The software architecture represents a high level of abstraction where a system is described as a collection of interacting components [3]. A commonly used definition of a software architecture is [11]:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationship among them.

Thus, the software architecture enables decomposition of the system into well-defined components and their interconnections, and consequently provides means for building complex software systems [71]. Design of the software architecture is the first step in the design process of a software product, right after the specification of the system's requirements. Hence, it allows early system testing, that is, as pointed out by Bosch [17] assessment of design and quality attributes of a system. Quality attributes of a system are those that are relevant from the software engineering perspective, e.g., maintainability, reusability, and those that represent quality of the system in operation, e.g., performance, reliability, robustness, fault-tolerance. These quality attributes of a system could be further classified as functional (see table 3.1) and non-functional quality attributes (see table 3.2), and used for architectural analysis [111]. The software architectural design process results in component quality requirements, as well as several constraints and design rules component must obey, e.g., means of communication [17]. Hence, developing reusable components depends on the software architecture, because the software architecture to a large extent defines the way a component is to be developed, its functionality and required quality [17]. Thus, the software architecture represents the effective basis for reuse of components [86]. Moreover, there are indications that software evolution and reuse is more likely to receive higher payoff if architectures and designs can be reused and can guide low level component reuse [67, 75].

In the system development the issue of handling conflicts in quality requirements during architectural design must be explicitly addressed. That is because a solution for improving one quality attribute affects other quality attributes negatively, e.g., reusability and performance are considered to be contradicting, as are fault-tolerance and real-time computing. E.g., consider a software system fine-tuned to meet extreme performance requirements. In such a system, it could be costly to add new functionality (components), since adding

Performance	The system's capacity of handling data or events.
Reliability	The probability of a system working correctly over a given period of time.
Safety	The property of the system that will not endanger human life or the environment.
Temporal constraints	The real-time attributes such as deadlines, jitter, response time, worst case execution time, etc.
Security	The ability of a software system to resist malicious intended actions.
Availability	The probability of a system functioning correctly at any given time.

Table 3.1: Functional quality attributes

Testability	The ability to easily prove system's correctness by testing.
Reusability	The extent to which the architecture can be reused.
Portability	The ability to move a software system to a different hardware and/or software platform.
Maintainability	The ability of a system to undergo evolution and repair.
Modifiability	Sensitivity of the system to changes in one or several components.

Table 3.2: Non-functional quality attributes

new parts could result in degraded performance of the upgraded system. In such a scenario, additional efforts would have to be made to ensure that the system meets the initial performance requirements.

3.1.3 The future of CBSE: from the component to the composition

So far we have examined software components and software architectures, as they are the core of the component-based software development. However, the research in the component-based software engineering community increasingly emphasizes composition of the system as the way to enable development of reliable systems, and the way to improve reuse of components. In this section we give an introduction to new techniques dealing with the problem of system composition. Figure 3.2 provides hierarchical classification of component-based systems [69].

Component-based systems on the first level, e.g., CORBA, COM and JavaBeans, represent the first generation of component-based systems, and are referred to as "classical" component-based systems. Frameworks and standards for components of today in industry primarily focus on classical component-based systems. Components are black boxes and communicate through standard interfaces, providing standard services to clients, i.e., the component is standardized. Standardization eases adding or exchanging components in the software system, and improves reuse of components. However, classical component-based systems lack rules for the system composition, i.e., composition recipe.

The next level represents architecture systems, e.g., RAPIDE [66] and UNICON [115]. These systems provide an architectural description language (ADL). ADL is used to specify architecture of the software system. In an architecture system, components encapsulate application-specific functionality, and are also black boxes. Components communicate through connectors [3]. A connector is a specific module that encapsulates communication between application-specific components. This gives significant advancement in the composition as compared to classical component-based systems, since communication and the architecture can be varied independently of each other. Thus, architecture systems separate

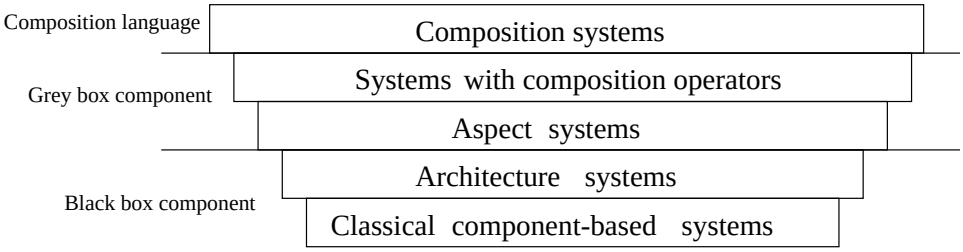

Figure 3.2: Classes of component-based systems

three major aspects of the software system: architecture, communication, and application-specific functionality. One important benefit of an architecture system is possibility of early system testing. Tests of the architecture can be performed with "dummy" components leading to the system validation in the early stage of the development. This also enables the developer to reason about the software system at an abstract level. Classical component-based systems, adopted in the industry, can be viewed as a subset of architecture systems (which are not yet adopted by the industry), as they are in fact simple architecture systems with fixed communication.

The third level represents aspect systems, based on aspect-oriented programming (AOP) paradigm [51]. Aspect systems separate more aspects of the software system than architecture systems. Beside architecture, application, and communication, aspects of the system can be separated further: representation of data, control-flow, and memory management. Temporal constraints can also be viewed as an aspect of the software system, implying that a real-time system could be developed using AOP [13]. Only recently, several projects sponsored by DARPA (Defense Advanced Research Projects Agency), have been established with an aim to investigate possibilities of reliable composition of embedded real-time systems using AOP [92]. Aspects are separated from the core component and are recombined automatically, through process called weaving. In AOP, a core component is considered to be a unit of system's functional decomposition, i.e., application-specific functionality [51]. Weavers are special compilers which combine aspects with core components at so-called joint points statically, i.e., at compile time, and dynamically at run-time. Weaving breakes the core component (at joint points) and cross-cuts aspects into the component, and the weaving process results in an integrated component-based system. Hence, core components are no longer black boxes, rather they are grey boxes as they are cross-cut with aspects. However, aspect weavers can be viewed as black boxes since they are written for a specific combination of aspects and core components, and for each new combination of aspects and core components a new aspect weaver needs to be written. Compared to architecture systems, aspect systems are more general and allow separation of various additional aspects, thus, architecture systems can be viewed as a subset of the class of aspect systems. Having different aspects improves reusability since various aspects can be combined (reused) with different core components. The main drawback of aspect systems is that they are built on special languages for aspects, requiring system developers to learn these languages.

At the fourth level are systems that provide composition operators by which components can be composed. Composition operators are comparable to component-based weaver, i.e., a weaver that is no longer a black box, but is also composable out of components, and can be, in a sense, re-composed for every combination of aspects and components, further improving the reuse. Subject-oriented programming (SOP) [83], an example of systems with composition operators, provide composition operators for classes, such as merge (merges two views of a class), equate (merges two definition of classes into one), etc. SOP is a powerful technique for compositional system development, since it provides

a simple set of operators for weaving aspects or views, and SOP programs support the process of system composition. However, SOP focuses on composition and does not provide any special component. Components in SOP are C++ classes.

Finally, at the last level are systems that contain full-fledged composition language, and are called composition systems. A composition language should contain basic composition operators to compose, glue, adopt, combine, extend and merge components. The composition language should also be tailorable, i.e., component-based, and provide support for composing (different) systems, in the large. Invasive software composition [69] is one approach that aims to provide a language for the system composition. Components in the invasive composition technique are more general. The component may consist of a set of arbitrary program elements, and is called a box [69]. Boxes are connected to the environment through very general connection points, called hooks. Composition of the system is encapsulated in composition operators (composers), which transform the component with hooks into a component with code. The process of system composition using composers is more general than aspect weaving and composition operators, since invasive composition allows composition operators to be collected in libraries and to be invoked by the composition programs (recipes) in a composition language. Composers can be realized in any programming or specification language. Invasive composition supports software architecture, separation of aspects, and provides composition receipts, allowing production of families of variant systems [68]. Reuse is improved, as compared to systems in the lower levels, since composition recipes can also be reused, leading to easy reuse of components and architectures. An example of the system that supports invasive composition is COMPOST [23]. However, COMPOST appears to be more suitable for complex non-real-time software systems, rather than systems that have limited amount of resources and enforce real-time behavior. Also, the system is not general enough, since it only supports Java source-to-source transformations.

The given overview illustrates current efforts to enable efficient system composition. In particular, techniques at the last three layers in figure 3.2 focus on the composition process. Component-based systems that we investigate in more detail in the following sections do not cover all classes of component-based systems shown in figure 3.2. Systems we investigate represent current componentization efforts in the database and the embedded real-time community, and are mainly subsets of component-based systems in the first two layers in figure 3.2.

3.2 Component-based database systems

A database system consists of a software called database management system (DBMS) and one or more databases managed by the DBMS [97]. Essential part of the database system is the DBMS.

In this section we focus on database systems that can be partially or completely assembled from a pre-defined set of components, i.e., that exploit component-based development. We refer to these systems as component-based database systems. First, the main granularity level of a component used for the composition of a database system is established. Then, in sections that follow, the classification and detail analysis of existing component-based database systems is given. Of a particular interest is to show how and to which degree different component-based database systems allow customization, i.e., to which degree a designer can tailor the database system for a particular application.

3.2.1 Granularity level of a database component

In general, two main granularity levels of a database component in component-based systems exist:

- DBMS (database system) as a component, and

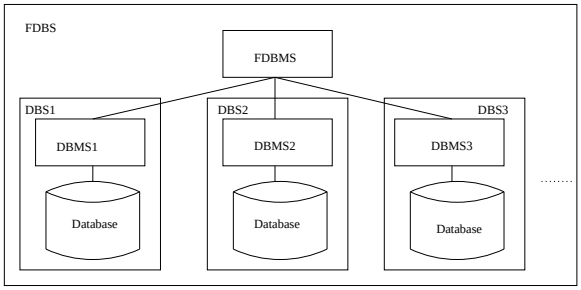

Figure 3.3: Components in FDBS

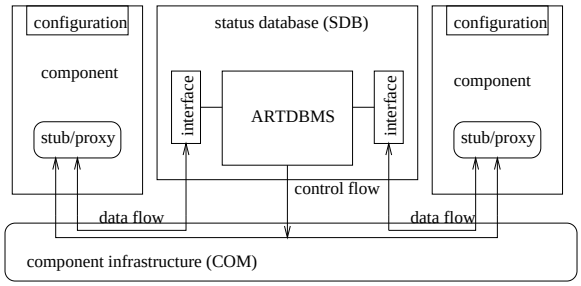

Figure 3.4: Component-based system with ARTDBS as a component

- part of the DBMS as a component.

The former component type is not exactly a component-based solution of a database system, rather DBMS used as a component in larger component-based systems. For example, a database system may be viewed as a component in a federated database system (FDBS), which is a collection of cooperating but autonomous database systems [97]. Database systems are controlled and managed by a software called federated database management system (FDBMS) (see figure 3.3). Hence, the database system is not made out of components but is rather a component of a larger component-based federated database system. In this respect, CORBA can be used to achieve interoperability among DBMSs in a multidatabase approach. CORBA enables the basic infrastructure that can be used to provide database interoperability [30]. When using CORBA as a middleware in a multidatabase system, the DBMS itself is not customizable; rather the DBMS is a component (CORBA object) registered to the CORBA infrastructure. Another example of a database system used as a component is an active real-time database system (ARTDBMS) that can be used as a component in an architecture with standardized components [76] (see figure 3.4). In this solution, infrastructure of the component technology (in this case COM) is used for separating the exchange of control information between components and data transfer via the status database system. The status database system is implemented as an active real-time database system and contains all the information about the current status of an application that is used by at least two different components. This architecture supports application composition out of components, and aims to enable adding or removing components without additional programming. A component in this system is assumed to be a self-contained, configurable binary software package with precisely defined standardized interfaces that can be combined with other components. Time consistency of saved data, in addition to the logical consistency, is ensured in the composed system by having a real-time database. However, this type of component-based solution is applicable for tailoring component-based systems where time consistency of data is needed and not for tailoring a

database system. A database system in this solution is a monolithic system, and changes to that type of systems are not easily done.

It is our opinion that more emphasis and detailed analysis is needed for the second category of database components, i.e., a component being a part of the DBMS. DBMS made out of components would indeed be an actual component-based solution of a database system. We motivate this further in the following section.

3.2.2 Component vs. monolithic DBMS

Database systems are today used in various application areas, such as telecommunications [1], e-commerce [37, 2], embedded systems [80], real-time systems [44], etc. Expansion of database systems to new application domains unavoidably results in new requirements a database system must fulfill [29]. Consider Internet applications where data is accessed from a variety of sources such as audio, video, text, image data, etc. Therefore, many web-based applications require a database that can manage vide variety of different types of data. Also, new application-specific data types have emerged that are not supported by traditional databases. Examples of non-standard data types are temporal data, spatial data, multimedia data. Hence, DBMSs are required to provide support for modeling and storing these new, non-standard, data types [82]. Broad variety of data from different data sources makes heterogeneous data management a very important challenge, since a large number of applications require a database that provides them with the uniform and homogeneous view of the entire system [21]. Business computing environment poses additional set of requirements on database systems, such as rapid application development, and minimized complexity both for end users and developers [81]. In addition, it is common that a company needs to produce a database system for a bigger spectra of different application areas. Such scenarios require cost-effective solutions of DBMSs. During the development of a DBMS, it is very hard to exactly predict in which direction system will evolve, i.e., which additional functionality the database system will have to provide in the future. Thus, a DBMS must be developed such that it is open to new, future, uses. Having a database that can be extended with new functionality or modified during its life-time is beneficial both for users and database vendors.

Meeting new requirements implies that traditional, monolithic DBMS must be extended to include new functionality. However, adding functionality to a monolithic DBMS has several drawbacks [29]:

- Parts in the monolithic DBMS are connected to one another and dependent on one another to such a degree that changes made in one part of the DBMS structure would lead to a domino effect in other parts of the monolithic system, i.e., extensions and modifications are not easily done.
- Adding new functionality in the monolithic system would result in increased complexity of a DBMS and in turn, increased maintenance costs of the system.
- Applications would have to pay performance and cost penalty for using unneeded functionality.
- System evolution would be more complex, since a DBMS vendor might not have the resources or expertise to perform such extensions in a reasonable period of time.

One approach to solve these problems could be componentization of a DBMS [29]. A component DBMS (CDBMS), also called a component-based database system, allows new functionality to be added in a modular manner, that is, system can be easily modified or extended by adding or replacing new components. Components and their interconnections should be well-defined in the CDBMS architecture. The architecture defines places in the system where components can be added, thus, specifying and restricting the ways in which DBMS can be customized. Ideally, applications would no longer have to pay performance

and cost penalty for using unneeded functionality because unnecessary components do not have to be added to a system. However, it has to be noted that components needed by the application might contain some unnecessary functionality themselves. Also, complexity of the system and maintenance cost would be reduced if the system could be composed only out of components (functionality) application needs. Finally, evolution of such a system would be simplified, since new components can be plugged into the system without having to recompile (or shut down) the entire system. Of course, this implies that new components have previously been tested and have a verified behavior.

Note that component-based database systems are faced with the same challenges as a component-based technology in general, such as performance degradation of a component-based system as compared to a monolithic system, complex development process of a reusable component, etc.

3.2.3 Components and architectures in different CDBMS models

Four different categories of CDBMSs have been identified in [29]:

- Extensible DBMS. The purpose of systems falling into this category is to extend existing DBMS with non-standard functionality, e.g., Oracle8*i* [82], Informix Universal Server with its DataBlade technology [48], Sybase Adaptive Server [81] and DB2 Universal Database [21].
- Database middleware. The purpose of systems falling into this category is to integrate existing data stores into a database system and provide users and applications with a uniform view of the entire system, e.g., Garlic [22] and DiscoveryLink [42], and OLE DB [74], .
- DBMS service. The purpose of systems falling into this category is to provide database functionality in standardized form unbundled into services, e.g., CORBAService [84].
- Configurable DBMS. The purpose of systems falling into this category is to enable composition of a non-standard DBMS out of reusable components, e.g., KIDS [38].

Sections that follow focus on characteristics of systems in each of these four different categories.

3.2.4 Extensible DBMS

The core system in this group is formed from fully functional DBMSs that implement all standard DBMS functionality [29]. Non-standard features and functionality not yet supported can be plugged into this DBMS (see figure 3.5). Components in this category of CDBMSs are families of base and abstract data types or implementations of some DBMS function such as new index structures. The architecture defines a number of plugs that components can use and formulates expectations concerning interfaces that component must meet in order to be integrated successfully.

Oracle8*i*

Oracle8*i* allows developers to create their own application-domain-specific data types [82]. Capabilities of the Oracle data server can be extended by means of data cartridges which, in this case, represent components in the Oracle8*i* architecture. Data cartridges are stored in an appropriate library for reusability. A data cartridge consists of one or more domain-specific types and can be integrated with the server. For example, a spatial data cartridge may provide comprehensive functionality for a geographical domain such as being able to store spatial data, perform proximity/overlap comparisons on such data, and also integrate

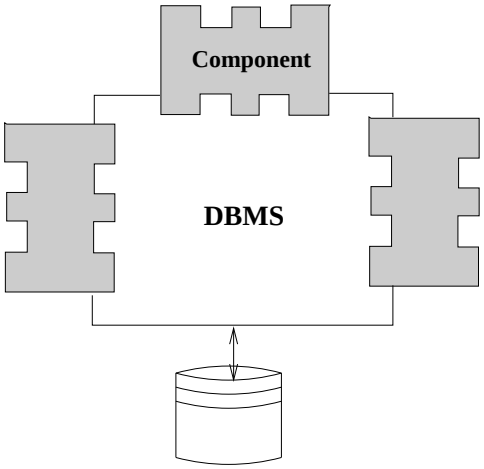

Figure 3.5: The principle of systems falling into an extensible DBMS category

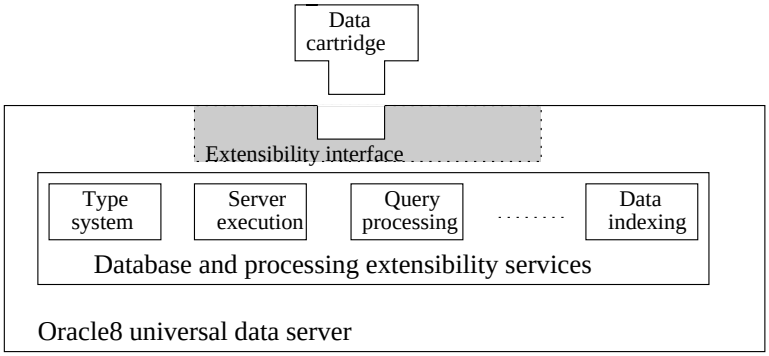

Figure 3.6: The Oracle extensibility architecture

spatial data with the server by providing the ability to index spatial data. In addition to the spatial cartridge, text, image and video data cartridge can be integrated in the architecture. Data cartridges can be integrated into a system through extensibility interfaces. There are three types of these interfaces: DBMS and data cartridge interfaces, used for communication between components and the DBMS, and service interfaces used by the developers of a component. In the Oracle8*i* architecture, data cartridges can provide their own implementation of database services such as storage service, query processing service, services for indexing, query optimization, etc. These services are extended by the data cartridge implementation and are called database extensibility services (see figure 3.6). Extensibility of a database service can be illustrated with the example of the spatial data cartridge, which provides the ability to index spatial data, and consequently extends indexing service of the server. Configuration support is provided for the development, packaging and deployment of data cartridges, as illustrated in figure 3.7. The Oracle Designer family of products has modeling and code generation tools which enable development of data cartridges. The Cartridge Packager module, part of the Oracle Enterprise Manager family, assists developer in packaging data cartridges into installable units. Components (data cartridges) are then installed into the server by end users using the Oracle Installer.

The architecture of the Oracle8*i* is fixed and defines the places where extensions can be made, i.e., components added, as well as interfaces to these components. Designer

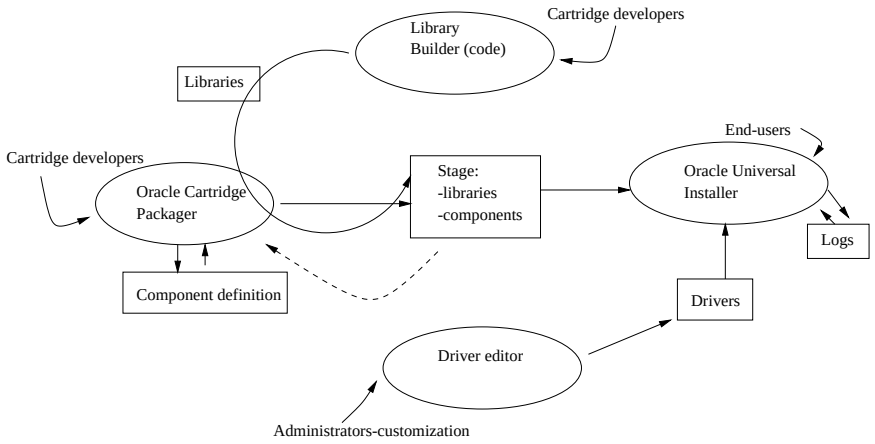

Figure 3.7: Packaging and installing cartridges

is allowed to customize database server only by plugging-in new components, i.e., the system has low degree of tailorability. Provided configuration support is adequate, since the system already has a fixed architecture and pre-defined extensions, and that extensions are allowed only in well-defined places of the architecture. This type of system emphasizes on satisfying only one requirement - handling non-standard data types. If a system needs to provide another (non-standard) service beside managing abstract data types, or if some internal part of the infrastructure needs to be changed, we are faced with the same difficulties as when using monolithic systems (see section 3.2.2). Also, this type of system can not easily be integrated in all application domains, e.g., real-time system, since there is no analysis support for checking temporal behavior.

The concept of the Informix DataBlade technology, DB2 Universal Database, and Sybase Adaptive Server is the same as the above described concept of the extensible Oracle8*i* server architecture for manipulation of non-standard data types. Moreover, conclusions made for the Oracle8*i* with respect to configuration tools, tailorability for satisfying only one requirement (handling non-standard data types), and lack of real-time properties and analysis tools, are applicable for all systems that fall in this category, i.e., extensible DBMS. Therefore, analysis of the Informix DataBlade technology, DB2 Universal Database, and Sybase Adaptive Server is kept very short.

Informix DataBlade technology

DataBlade modules are standard software modules that can be plugged into the Informix Universal Server database to extend its capability [48]. DataBlade modules are components in the Informix Universal Server. These components are designed specifically to enable users to store, retrieve, update, and manipulate any domain-specific type of data. Informix allows development of DataBlade modules in Java, C, C++, J++ or stored procedure language (SPL) by providing the application programming interface (API) for those languages. Same as Oracle, Informix has provided low degree of tailoring, since the database can only be extended with standardized components that enable manipulation of non-standard data types. Configuration support is provided for development and installation of DataBlade modules, e.g., BladeSmith, BladePack, and BladeManager.

DB2 Universal Database

As previously described database servers, DB2 Universal Database [21, 28] also allows extensions in the architecture to provide support for comprehensive management of application-specific data types. Application-specific data types and new index structures for that data types are provided by DB2 Relational Extenders, reusable components in the DB2 Universal Database architecture.There are DB2 Relation Extender for text (text extender), image (image extender), audio and video (extender). Each extender provides the appropriate functions for creating, updating, deleting, and searching through data stored in its data type. An extender developers kit with wizards for generating and registering extenders provides support for the development and integration of new extenders in DB2 Universal Database.

Sybase Adaptive Server

Similar to other database systems in this category, the Sybase Adaptive Server [81] enables extensions in its architecture, called Sybase's Adaptive Component Architecture (ACA), to enable manipulation of application-specific data types. Components that enable manipulation of these data types are called Speciality Data Stores, e.g., speciality data stores for text, time series and geospatial data. The Sybase Adaptive Server differs from other database systems in the extensible DBMS category in that it provides a support for standard components in the distributed computing environment, as the name of the ACA architecture indicates. Through open (Java) interfaces Sybase's Adaptive Component Architecture (ACA) provides mechanisms for communication with other database servers. In addition of providing the interoperability of database server, Sybase enables interoperability with other standardized components in the network, such as JavaBeans.

3.2.5 Database middleware

The aim of systems falling into this category is to integrate existing data stores, into a common DBMS-style framework [29]. A CDBMS acting as a middleware between different data stores and the application of integration provides users and applications with a uniform and integrated view of the entire system (see figure 3.8). In this model the architecture introduces a common intermediate format into which the local data formats can be translated. Components in this category of CDMSs perform this kind of translation. Common interfaces and protocols define how the database middleware system and the components interact. Components, also known as wrappers, are located between the DBMS and each data source. Each component mediates between the data source and the DBMS, i.e., it wraps the data source.

Garlic

Garlic [22] is an IBM research project that aims to integrate and unify data managed by multiple, disparate, data sources. The primarily goal of the Garlic project is to build a heterogeneous multimedia information system capable of integrating data from a broad range of data sources. Each data source (also known as a repository) has its own data model, schema, programming interface and query capability. Data models for these sources vary widely, e.g., relational, object-oriented, a simple file-system, and a specialized molecular search data model. The Garlic data model is based on the Object Database Management Group (ODMG) standard. Garlic can provide access to a data source only if appropriate wrapper for that data source exists. Thus, associated with each data source is a wrapper, a reusable component in the Garlic architecture (see figure 3.9). In addition to data sources containing legacy data, Garlic allows users to create their own data source, i.e., data source for Garlic complex objects. Wrappers for new data sources can be integrated into an existing Garlic database without disturbing legacy applications, other wrappers and Garlic

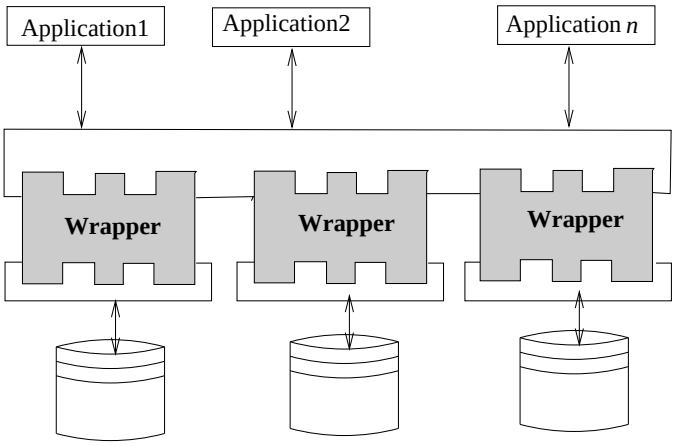

Figure 3.8: Principle of systems falling into a DBMS middleware category

applications. Garlic applications interact with query services and run-time system through Garlic's object query language and a C++ API. The query processor develops plans to efficiently decompose queries that span multiple data sources into pieces that individual data sources can handle. As shown in figure 3.10 wrapper provides four major services to the

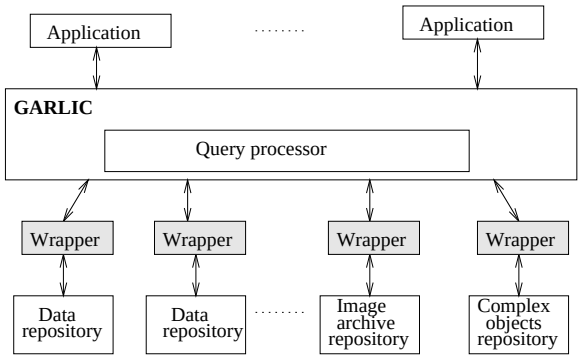

Figure 3.9: The Garlic architecture

Garlic system [95]:

1. A wrapper models the contents of its data source as Garlic objects, and allows Garlic to retrieve references to these objects. The Garlic Data Language (GDL), a variant of the ODMG's Object Description Language (ODMG-ODL), is used to describe the content of a data source. Interfaces that describe the behavior of objects in a data source (repository) are known collectively as a repository schema. Repositories are registered as parts of the Garlic database and their individual repository schemas are merged into the global schema that is presented to Garlic users.

2. A wrapper allows Garlic to invoke methods on objects and retrieve their attributes. Method invocation can be generated by Garlic's execution engine, or by a Garlic application that has obtained a reference (either as a result of a query or by looking up root object by name).

3. A wrapper participates in query planing when a Garlic query processor ranges over multiple objects in the repository. The goal of the query planning is to develop

alternative plans for answering a query, and then to choose the most efficient one.

4. A wrapper's finale service is to participate in plan translation and query execution. A Garlic query plan is presented as a tree of operators, such as FILTER, PROJECT, JOIN, etc. The optimized plan must be translated into a form suitable for execution. During every execution, the wrapper completes the work which was assigned to it in the query planing phase. Operators are mapped into iterators, and each wrapper provides a specialized `Iterator` subclass that controls execution of the work described by one of its plans.

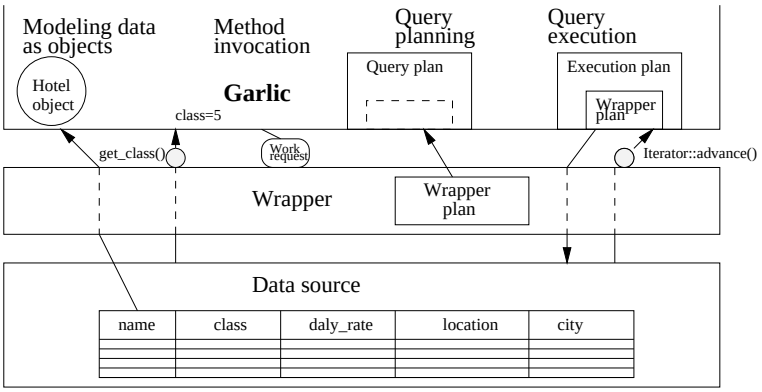

Figure 3.10: Services provided by the wrapper

Having defined services that a wrapper provides to the Garlic architecture, simplifies the development process of the wrapper. A designer of the wrapper needs to perform specific tasks to ensure that a wrapper provides a certain service. After defining the wrapper services, i.e., performing tasks specific to each service, the wrapper designer has a final task of packaging written pieces in a complete wrapper. To keep the cost of communication between the Garlic and a wrapper low, the wrapper code is packaged as a dynamically loadable library and resides in the same address space. Every wrapper includes interface files that contain one or more interface definitions written in GDL, environment files that contain name/value pairs to encode repository-specific information for use by the wrapper, and libraries that contain dynamically loadable code to implement schema registration, method invocation, and the query interface. Decomposing a wrapper into interface files, libraries, and environment files gives the designer of a wrapper additional flexibility when designing a wrapper for a particular repository or family of repositories. For each repository, an interface file describes the objects in the corresponding database, and an environment file encodes the name of the database to which the wrapper must connect, the names of the roots exported by the repository and the tables to which they are bound, etc. Wrapper designers are provided with a library of schema registration tools, query-plan-construction routines, a facility to gather and store statistics, and other useful routines in order to automate writing wrappers as much as possible.

To conclude, the well-defined Garlic wrapper architecture with wrapper services and tasks for a wrapper designer associated with each service, enables easy and fast (cost-effective) development of wrappers. The architecture also provides stability in a sense that new wrappers can be easily installed into an existing system without effecting the other wrappers or the system itself. The Garlic system is suitable for large distributed database systems. In terms of tailorability, one can observe that the Garlic wrapper architecture provides a moderate degree of customization, and can be tailored for different applications, provided that appropriate wrappers needed by the application exist or are developed.

However, the Garlic cannot be used with ease in every application domain, e.g., a real-time domain, where correctness of a system depends on the time when results are produced. The Garlic wrapper architecture unifies access to different data source, but it is not clear if such access can be guaranteed in a timely predictable manner.

DiscoveryLink

DiscoveryLink [42] is a commercial middleware DBMS, completely based on the Garlic technology, and incorporated into the IBM's DB2 database server. Discovery Link integrates life science data from heterogeneous data sources. As in Garlic, components, i.e., wrappers, in the DiscoveryLink are C++ programs, packaged as shared libraries. Wrappers can be created dynamically, allowing a number of data sources to grow (and shrink) on the fly. Most of the wrappers in DiscoveryLink are required to have only minimum functionality. In particular, the wrapper must be able to return the tuples of any relation that it exports. The wrapper architecture and the notion of having reusable components that can be added or exchanged makes DiscoveryLink unique among commercial database middleware systems that provide query across multiple relational sources, e.g., DataJoiner [24] and Sybase [47].

OLE DB

OLE DB [14, 15] is designed as a Microsoft Component Object Model (COM) interface. COM provides a framework for integrating components. This framework supports interoperability and reusability of distributed objects by allowing developers to build systems by assembling reusable components from different vendors which communicate via COM. COM defines an application programming interface (API) to allow for the creation of components for use in integrating custom applications or to allow diverse components to interact. However, in order to interact, components must adhere to a binary structure specified by Microsoft. As long as components adhere to this binary structure, components written in different languages can interoperate. COM supports concepts such as aggregation, encapsulation, interface factoring, inheritance, etc.

OLE DB is a specification for a set of data access interfaces designed to enable a variety of data stores to work together [74]. OLE DB provides a way for any type of data store to expose its data in a standard and tabular form, thus unifying data access and manipulation. In Microsoft's infrastructure for component-based computing, a component is thought of as [74]:

> "...the combination of both process and data into a secure, reusable object..."

and as a result, both consumers and providers of data are treated as components. A data consumer can be any piece of the system or the application code that needs access to a broad range of data. In contrast, data providers are reusable components that represent data source, such as Microsoft ODBC, Microsoft SQL server, Oracle, Microsoft Access, which are all standard OLE DB providers. Thus, OLE DB enables building component-based solutions by linking data providers and data consumers through providing services that add functionality to existing OLE DB data and where the services are treated as components in the system (see figure 3.11). The architecture in figure 3.11 is called the Universal Data Access (UDA) architecture. It is possible to develop new, custom, data providers that reuse existing data providers as the underlying component or a component building block of more complex (data provider) components. This enables developers to expose custom views of data and functionality without rewriting the entire data store or the access methods. Note that only reusable components in this system are data providers. These components can be viewed as wrappers in the UDA architecture. Note that architectural requirements get more complex as compared to the systems in the extensible category. In contrast to the Garlic wrapper architecture, OLE DB allows the UDA architecture to have more variable

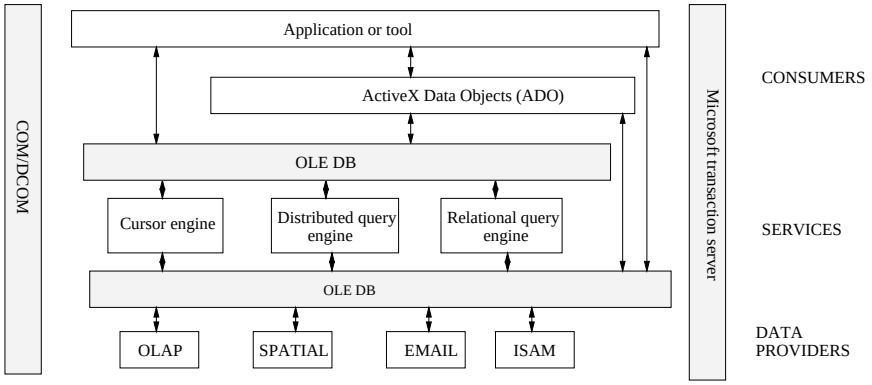

Figure 3.11: The Universal Data Access (UDA) architecture

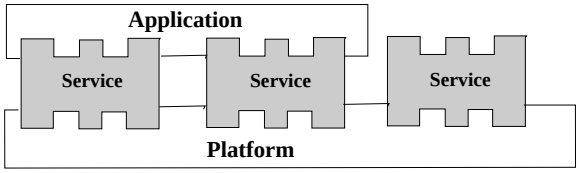

Figure 3.12: The principle of systems falling into a DBMS service category

parts and more customization of the system is allowed. The UDA architecture provides a common format into which all local data formats can be translated as well as standardized interfaces that enable communication between the applications and different data stores, i.e., OLE DB interface.

Although OLE DB provides unified access to data and enables developers to build their own data providers, there is no common implementation on either the provider or consumer side of the interface [16]. Compatibility is provided only through the specification and developers must follow the specification exactly to make interoperable components, i.e., adequate configuration support for this is not yet provided. To make up for inadequate configuration support, Microsoft has made available, in Microsoft's Software Developer's Kit (SDK), tests that validate conformance of the specification. However, analysis of the composed system is missing.

OLE DB is not applicable for a real-time domain, since temporal behavior is not of an importance. Additionally, OLE DB is limited with respect to software platforms, since it can only be used in Microsoft software environments.

3.2.6 DBMS service

In this type of a CDBMS, the DBMS and related tasks are unbundled into services [29], and as a result the monolithic DBMS is transformed into a set of stand-alone services (see figure 3.12). Applications no longer operate on full fledged DBMSs, but instead use those services as needed. Services are defined in a common model or language and are implemented using a common platform in order to render the service implementations exchangeable and freely combinable. Systems in this category have components that are database services and their implementations. CORBA with its CORBAservices could be an example of a system falling into this category. Although there are no DBMS service systems that are implemented solutions, the objectives of this category can be illustrated through a CORBAservices example-system.

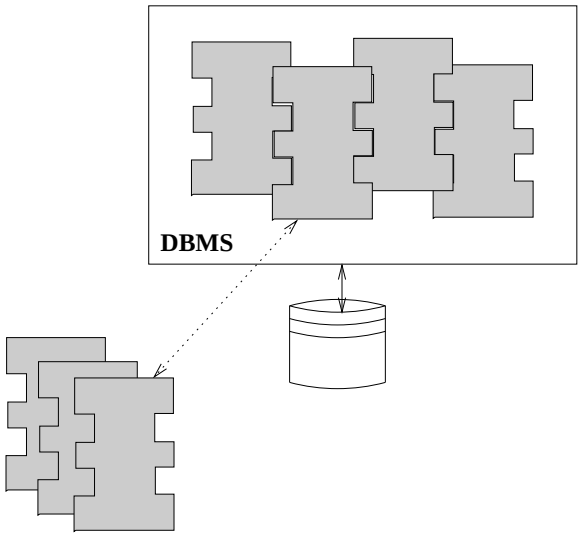

Figure 3.13: The principle of systems falling into a configurable DBMS category

CORBAservices

One single DBMS could be obtained by gluing together CORBAservices which are relevant for databases, such as transaction service, backup and recovery service, concurrency service. CORBAservices are implemented on the top of the Object Request Broker (ORB). Service interfaces are defined using the Interface Definition Language (IDL) [30]. In this scenario a component would be one of the database relevant CORBAservices. This would mean that applications could choose from a set of stand-alone services those services, i.e., components, which they need. However, this approach is (still) not viable because it requires writing significant amount of glue code. In addition, performance overhead could be a problem due to the inability of an ORB to efficiently deal with fine-granularity objects [84]. Also, an adequate value-added framework that allows development of components and use of these components in other applications is still missing. Configuration as well as analysis tools to support this process are missing.

3.2.7 Configurable DBMS

One step further away from the DBMS service category of CDBMSs, where the set of services have been standardized and fixed, is a configurable DBMS category that allows new DBMS parts to be developed and integrated into a DBMS (see figure 3.13) [29]. Here, components are DBMS subsystems defined in the underlying architecture which is no longer fixed.

KIDS

The KIDS (Kernel-based Implementation of Database management Systems) [38] approach to constructing CDBMSs is an interesting research project at the University of Zürich, since it offers a methodical, open, and cost-effective construction of a DBMS. The approach offers high level of reusability, where virtually any result obtained in a previous system construction is reused (designs, architectures, specifications, etc.). The architecture is well-defined, and once created it can be saved in a library and reused. The DBMS architecture is based on the broker/service model, i.e., it consists of brokers, services, and responsibilities. A service represents a specific task to be provided by the DBMS, and

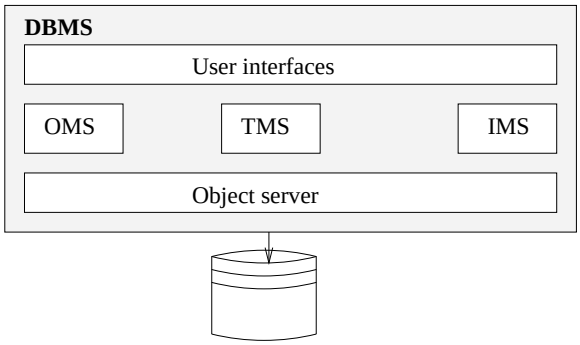

Figure 3.14: The KIDS subsystem architecture

each service is provided by at least one component in the system and can be requested by raising events. Services are provided by reactive processing elements called brokers (brokers are comparable to objects in object-oriented programming). Components in KIDS are DBMS subsystems that are collections of brokers. Brokers are responsible for a related set of tasks, e.g., object management, transaction management, and integrity management. For example, the transaction management subsystem can consist of a transaction broker, scheduler, log management broker, and recovery broker. Transaction broker is responsible for starting and terminating transactions, i.e., services such as begin transaction, commit, and abort. The scheduler is responsible for serializing concurrent transactions, i.e., validate service which checks whether the database access is legitimate with the respect to the execution of concurrent transactions. A structural view of the KIDS architecture is shown in figure 3.14. The DBMS architecture consists of two layers. The first layer is the object server component, which supports the storage and retrieval of storage objects. The object server provides a fixed interface, which hides implementation details and operating system characteristics from upper layers. The object server component is reused in its entirety, and belongs to the fixed part of the DBMS architecture (this is because the object server implements functionality needed by any DBMS). The second layer is variable to a large extent, and can be decomposed into various subsystems. In the initial decomposition of KIDS, three major subsystems exist in the second layer:

- The object management subsystem (OMS), which implements the mapping from data model objects into storage objects, retrieval of data model objects, and meta data management.
- The transaction management subsystem (TSM), which implements the concept of a transaction, including concurrency control, recovery, and logging.
- The integrity management subsystem (IMS), which implements the (DBMS-specific) notion of semantic integrity, and is responsible for checking whether database state transitions result in consistent states.

Each of these subsystems is implemented using dedicated languages and reuse techniques. These three subsystems (OMS, TMS, and IMS) implement basic database functionality. Additional functionality can be provided by adding new subsystems in the second layer of the KIDS architecture, i.e., expanding decomposition of this layer to more than three subsystems.

KIDS is composed out of components (subsystems) carefully defined in an underlying architecture. The construction process of the CDBMS is well-defined, and starts after a requirement analysis of the desired DBMS, for a specific domain, has been performed and relevant aspects, i.e., functionality that DBMS needs to provide, have been determined (see figure 3.15). The process of DBMS construction is done in phases:

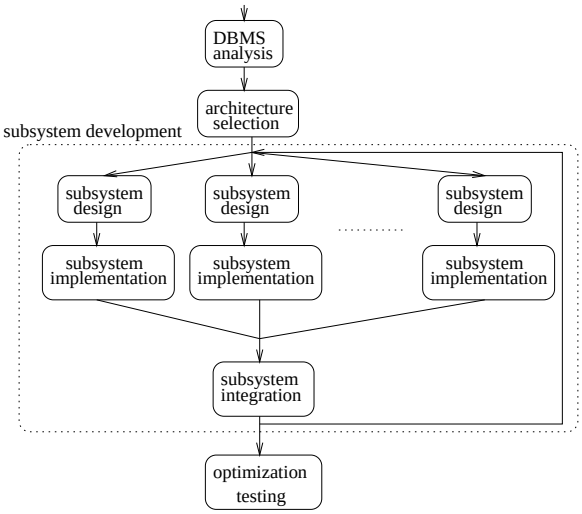

Figure 3.15: Phases of the DBMS construction process

- The selection of an architecture skeleton from the library, or a development of a new architecture skeleton, i.e., a (possibly still incomplete) collection of partially defined subsystems.
- The subsystem development, which consists of subsystem design, implementation, and subsystem integration. At this phase the process branches in a sense of parallelism for a design and implementation of each subsystem. Integration of subsystems into previously defined architectural skeleton is then performed. Since it might not always be possible to integrate the subsystem after the implementation of all subsystems has been completed, it is possible to perform integration as far as necessary and continue with the subsystem development, i.e., this phase is iterative and incremental.
- Testing and optimization follow the subsystem development. The KIDS construction process focuses on the development of subsystems and their integration. Analysis of the system is not investigated in greater details.

KIDS allows new components to be developed and integrated into the system, thus, enables tailoring of a DBMS for different applications. Expanding the initial set of components in the KIDS architecture with the functionality (components) needed by a particular application, one could be able to design “plain” object-oriented DBMS, a DBMS video-server, or a real-time plant control DBMS. Of course, in the proposed initial design of KIDS, real-time properties of the system or components are not considered. For a real-time application, the proposed construction process could be used if a component is constructed such that its timing behavior is known, and the consumption of memory by each component is taken into account. KIDS offers good basis for the early system testing; in the first phase of the construction process, at the architecture level of the system.

A defined process of a DBMS construction, reusability of components and architectures, and high degree of componentization (tailorability) of a system differentiate this CDBMS from all others.

3.3 Component-based embedded and real-time systems

Embedded real-time systems are being used widely in today's modern society. Thus, agile and low-cost software development for embedded real-time systems has become critically important. Successful deployment of embedded real-time systems depends on low development costs, high degree of tailorability and quickness to market. Thus, the use of the component-based software development paradigm for constructing and tailoring embedded real-time systems has premise. We elaborate this in more detail by listing the major benefits of using component-based software engineering for development of embedded real-time systems [106, 64]:

- Rapid development and deployment of the system. Component-based software development aims to reduce the amount of new code that must be written each time new application is being developed. Many software components, if properly designed and verified, can be reused in different embedded real-time applications. Embedded real-time systems built out of components may be readily adopted and tuned for new environments as opposed to monolithic systems.

- Changing software "on-the-fly". Embedded real-time sensor-based control systems may be designed to have software resources that can change on the fly, e.g., controllers, device drivers. This feature is desirable in autonomous and intelligent control applications, as well.

- Evolutionary design. In a component-based system software components can be replaced or added in the system. This is appropriate for complex embedded real-time systems that require continuous hardware and software upgrades in response to technological advancements, environmental change, or alteration of system goals.

- Product lines. It is common that manufacturers create several different, but still similar, embedded real-time products. Having reusable components stored in a library would enable the entire product line to use same software components from the library when developing products. This eliminates the need to develop and maintain separate software for each different product.

- Increased reliability. It is easier to develop, test and optimize individual components of limited functionality than when the same functionality is embedded within a large monolithic system. In addition, individual components can be specified and verified more easily, and provided that the rules for the composition of a system are well-defined, safety-critical applications can be composed from such components.

- Fine-tuning. Having components in the system that can be replaced offers considerable flexibility for fine-tuning of an embedded real-time application, e.g., components can be introduced that enable switching between static and dynamic scheduling algorithms.

- Reduced system complexity. Components are designed to have different functionality. Choosing components from the library that provide functionality needed by the system should reduce system's complexity. This is true since it is likely that a component (as compared to the monolithic system) will not contain large parts of functionality unneeded by the system.

In the remainder of this section we give analysis of existing component-based embedded real-time systems. However, some of the component-based systems analyzed only embedded or only real-time systems. To obtain a good understanding of how component-based software development can be successfully integrated in the area of embedded real-time systems, it is necessary to investigate both real-time and embedded system, in addition to those system that can be classified as embedded real-time (definitions of embedded, real-time,

and embedded real time systems are given in chapter 1). In order to keep the notation as simple as possible and, at the same time, easily understandable, when referring to multiple systems that can be classified either as real-time or as embedded or as embedded real-time, we use the notation embedded real-time systems.

3.3.1 Components and architectures in embedded real-time systems

We have identified three distinct types of component-based embedded real-time systems:

- Extensible systems. An example of this type of the system is SPIN [12], an extensible microkernel. Extensions in the system are possible by plugging components, which provide non-standard features or functionality, into an existing system. Extensions are allowed only in well-defined places of the system architecture.
- Middleware systems. These are characterized by their aim of providing efficient management of resources in dynamic heterogeneous environments, e.g., 2K [53] is a distributed operating system that is specifically developed for management of resources in a distributed environment, which consists of a different software and hardware platforms.
- Configurable systems. An architecture of a configurable system allows new components to be developed and integrated into the system. Components in such systems are true building parts of the system. A variety of configurable systems exists, e.g., VEST [102], Ensemble [64], the approach to system development with real-time components introduced by Isović [49], and systems based on the port-based object (PBO) model [105].

3.3.2 Extensible systems

SPIN

SPIN [12, 85] is an extensible operating system that allows applications to define customized services. Hence, SPIN can be extended to provide so-called application-specific operating system services [12]. An application-specific service is one that precisely satisfies the functional and performance requirements of an application, e.g., multimedia applications impose special demands on the scheduling, communication and memory allocation policies of an operating system. SPIN provides a set of core services that manage memory and processor resources, such as device access, dynamic linking, and events. All other services, such as user-space threads and virtual memory are provided as extensions. Thus, SPIN provides possibility to customize, i.e, tailor, the system according to the needs of a specific application. However, the system has a low degree of tailorability, since the architecture can only be extended with components, called extensions.

A reusable component, an extension, is a code sequence that can be installed dynamically into the operating system kernel by the application or on behalf of it. Thus, an application can dynamically add code to an executing system to provide a new service. The application can also replace or augment old code to exchange existing services, i.e., components can be added, replaced, or modified in the system. E.g., an application may provide a new in-kernel file system, replace an existing paging policy, or add compression to network protocols. Extensions are written in Modula-3, a modular, ALGOL-like programming language. The mechanism that integrates extensions (components) with the core system are events, i.e., communication in SPIN is event-based. An event is a message that is raised to announce a change in the state of the system or a request for a service. Events are registered in an event handler, which is a procedure that receives an event. An event dispatcher oversees event-based communication. The dispatcher is responsible for enabling services such as conditional execution, multicast, asynchrony, and access control.

An extension installs a handler with the event thorough a central dispatcher that routes events to handlers. Event-based communication allows considerable flexibility of the system composition. All relationships between the core system and components are subject to changes by simply changing the set of event handlers associated with any given event.

The correctness of the composed system depends only on the language's safety and encapsulation mechanisms; specifically interfaces, type safety, and automatic storage management. Analysis of the composed system is not performed, since it is assumed that the configuration support provided within the Modula-3 language is enough to guarantee the correct and safe system. SPIN allows applications to implement their own scheduling policies. Provided the right extension for real-time scheduling policy this operating system can be used for soft real-time applications such as multimedia applications.

3.3.3 Middleware systems

2K

2K [55, 53] is an operating system specifically developed for manipulation of resources in a distributed heterogeneous environment (different software systems on different hardware platforms). 2K is based on a network-centric model and CORBA component infrastructure. In the network centric model all entities, i.e., user applications (labeled as 2K applications), software components and devices, exist in the network and are represented as CORBA objects (see figure 3.16). When a particular service is instantiated, the entities that constitute that service are assembled. Software components (CORBA objects) communicate through IDL interfaces. As shown in figure 3.16, 2K middleware architecture is realized using standard CORBA services such as naming, trading, security and persistence, and extending the CORBA service model with additional services, such as QoS-aware management, automatic configuration, and code distribution. The 2K automatic configura-

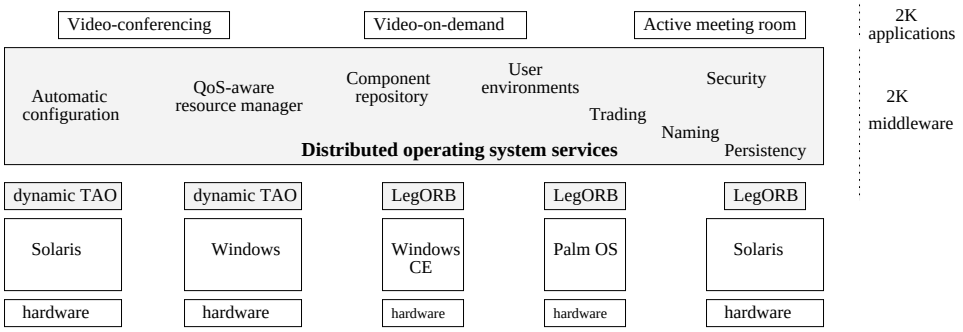

Figure 3.16: The 2K middleware architecture

tion service supports loading components into run-time system. A CORBA object, called ComponentConfigurator, stores inter-component dependencies that must be checked before components can be successfully installed into a system by the automatic configuration service. An Internet browser, for example, could specify that it depends upon components implementing an X-Windows system, a local file service, the TCP/IP protocol, and the Java virtual machine version 1.0.2 or later. The automatic configuration service and ComponentConfigurator enable automated installation and configuration of new components provided the system has access to the requirements for installing and running a software component, i.e, inter-component dependencies.

Integration of components into the middleware is done through a component called dynamic TAO (The Adaptive Communication Environment ORB). The dynamic TAO is a CORBA compliant reflective ORB as it allows inspection and reconfiguration of its internal

engine [54]. This is important since the middleware must be configurable and able to adopt to dynamic changes in resource availability and in the software and hardware infrastructure.

However, the dynamic TAO component has a memory footprint greater than a few megabytes, which makes it inappropriate for use in environments with limited resources. A variant to the dynamic TAO, a LegORB component, is developed by the 2K group and it has a small footprint and is appropriate for embedded environments, e.g., 6 Kbytes on the PalmPilot running on PalmOS. Dynamic TAO and LegORB components are not reusable, but are the key enabler for reuse of other CORBA objects. Even though this system provides automated installation and configuration of new components, it does not specify the development of new components. The development of new components is done only based on CORBA component model specifications. Software components in this system can be reused. However, those are components which already exist in the network, and are reused in the sense that they can be dynamically installed into the system whenever some application needs a specific component. Also, it is is assumed that inter-component dependencies provide good basis for the system integration, and guarantee correct system behavior (other guarantees of the system behavior, obtained by appropriate analysis, do not exist).

As figure 3.16 shows, 2K provides services for the applications such as video-on-demand and video-conferencing, which can be viewed as soft real-time applications. Considering that 2K uses CORBA component infrastructure with real-time CORBA extensions, i.e., TAO ORB, implies that hard real-time applications can also be supported by the 2K middleware architecture.

3.3.4 Configurable systems

Ensemble

Ensemble is a high performance network protocol architecture designed to support group membership and communication protocols [64]. Ensemble does not enforce real-time behavior, but is nevertheless interesting because of the configurable architecture and the way it addresses the problem of configuration and analysis of the system. Ensemble includes a library of over sixty micro-protocol components that can be stacked, i.e, formed into a protocol in a variety of ways to meet communication demands of an application. Each component has a common event-driven Ensemble micro-protocol interface, and uses the message-passing way of communication. Ensemble's micro-protocols implement basic sliding window protocols, and functionality such as fragmentation and re-assembly, flow control, signing and encryption, group membership, message ordering, etc. The Ensemble system provides an algorithm for calculating the stack, i.e., composing protocol out of micro-protocols, given the set of properties that an application requires. This algorithm encodes knowledge of protocol designers and appears to work quite well, but does not assure generation of a correct stack (the methodology for checking correctness is not automated yet). Thus, Ensemble can be efficiently customized for different protocols, i.e., it has a high level of tailorability. In addition, Ensemble gives the possibility of formal optimization of the composed protocol. This is done in Nuprl [64] and appears to give good results as far as the optimization of a protocol for a particular application goes.

System development with real-time components

Isović et al. [49] defined a development method specifically targeted towards real-time systems regardless of its complexity. The development method they propose is an extension of the method used for developing real-time systems adopted by Swedish car industry [78]. In this development method it is assumed that:

- The component library contains binaries of components-of-the-shelf (COTS), and description of them, e.g., memory consumption, identification, environment assumptions (processor family on which component operates), and functional description.

- The component library also contains dependencies to other components.
- Components are mapped to tasks, or multiple tasks for more complex components.
- Component communication is done through shared memory, and interfaces are called ports.

The development process for real-time systems is divided in several stages as shown in figure 3.17, and starts with a system specification as an input to the top-level design. At the top-level design stage decomposition of a system into components is performed. A designer browses through the library and designs the system having in mind possible component candidates. At the detailed design stage temporal attributes are assigned to components: period, release times, precedence constraints, deadline, mutual exclusion, and time-budget (a component is required to complete its execution within its time-budget). At the following stage checks are performed to determine if selected components are appropriate for the system, or, if the adaptation of components is required, or new components need to be developed. Also, at this stage, component interfaces are checked to see if input ports are connected and if their type matches, and that way system integration is performed. If this stage shows that selected components are not appropriate for the system under construction, then a new component needs to be developed. Developed components are placed in libraries for future reuse. The detail design stage, the scheduling/interface check stage, and the top-level design stage can be repeated until proper components for system integration are found (or designed). When the system finally meets the requirements from the specification, the temporal behavior of components must be tested on the target platform to verify if they meet temporal constrains defined in the design phase, i.e., verification of worst case execution time is performed. The method described provides a high degree of tailorability

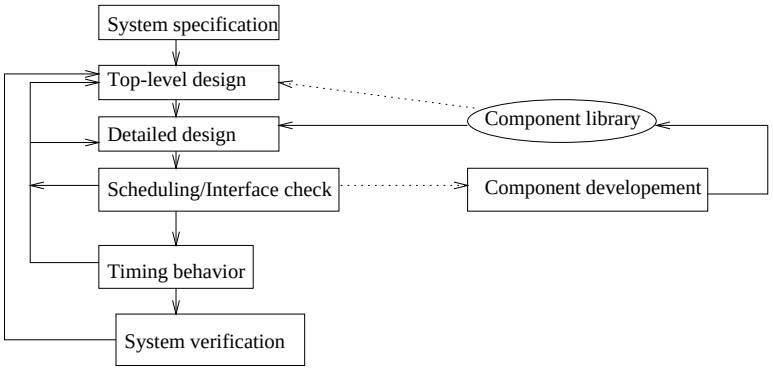

Figure 3.17: Design model for real-time components

for the developer, since the architecture of the system is not fixed, and components in the system can be exchanged during the design of the system in order to obtain the most suitable components for that particular system. The developer does have some support while choosing the appropriate component for reuse, given the check stages of the design and dependency checks among components that are stored in libraries. However, such process is not automated.

This method supports analysis of temporal behavior of components (focus is on worst-case execution time checks), but how the temporal behavior of the entire system is checked is not clear. Also, this approach is constrained by the assumption that the components in library can be COTS and that temporal behavior of components is not known beforehand. This allows temporal constraints of the component to be defined (or better to say predicted) only at the design time of the system.

VEST

While previously described design method is targeted towards systems that, in most cases, need to be composed out of COTS, and suitable for more complex real-time systems, VEST [102] aims to enable the construction of the OS-like portion of an embedded real-time system with strengthened resource needs. The VEST development process is fairly well-defined with an emphasis on configuration and analysis tools is recognized in VEST. The development process offers more flexibility than the one presented by Isović et. al. [49], since components in the library are passive and real-time attributes of components are known. System development starts with the design of the infrastructure, which can be saved in a library and used again (see figure 3.18). The infrastructure consists of micro-components such as interrupt handlers, inidirection tables, dispatchers, plug and unplug primitives, proxies for state mapping, etc. The infrastructure represents a framework for composing a system out of components. Configuration tools permits the user to create an embedded real-time system by composing components into a system, i.e., mapping passive components into run-time structures (tasks). After a system is composed, dependency checks are invoked to establish certain properties of the composed system. If the properties are satisfied and the system does not need to be refined, the user can invoke analysis tools to perform real-time, as well as reliability analysis. As can be seen, VEST offers high degree of tailorability for the designer, i.e., a specific system can be composed out of appropriate components as well as infrastructure from the component library. Note that components in

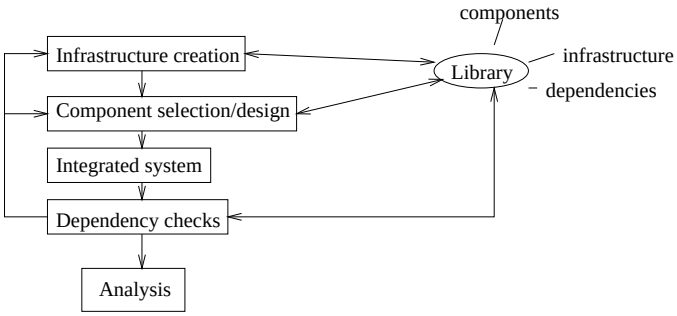

Figure 3.18: Embedded system development in VEST

VEST are passive (collection of code fragments, functions and objects), and are mapped into run-time structures (tasks). Each component can be composed out of subcomponents. For example, the task management component can be made of components such as create task, delete task, and set task priority. Components have real-time properties such as worst case execution time, deadline, and precedence and exclusion constraints, which enable real-time analysis of the composed system. In addition to temporal properties, each component has explicit memory needs and power consumption requirements, needed for efficient use in an embedded system. Selecting and designing the appropriate component(s) is fairly complex process, since both real-time and non-real-time aspects of a component must be considered and appropriate configuration support has to be available. Dependency checks proposed in VEST are one good way of providing configuration support and the strength of the VEST approach. Due to its complexity dependency checks are broken into 4 types:

- factual: component-by-component dependency checks (worst case execution time, memory, importance, deadline, etc.),
- inter-component: pairwise component checks (interface requirements, version compatibility, is a component included in another, etc.),

- aspects: checks that include issues which affect the performance or semantic of components (real-time, concurrency synchronization and reliability issues), and
- general: checks of global properties of the system (the system should not experience deadlocks, hierarchical locking rules must be followed, etc.).

Having well defined dependency checks is vital since it minimizes possible errors in the system composition. Interface problems in VEST are only identified but are not addressed at all; thus it is not obvious how components can be interconnected. Also, analysis of the system is proposed, but some more concrete solutions are not presented. Finally, VEST is an ongoing project developing the platform for configuration and analysis of embedded real-time systems.

PBO model

A component-based system based on the port-based object (PBO) model can be classified as configurable, and is suitable for development of embedded real-time control software system [105]. Components from the component library, in addition to newly created ones, can be used for the system assembly. A component is the PBO that is implemented as an independent concurrent process. Components are interconnected through ports, and communicate through shared memory. The PBO defines module specific code, including input and output ports, configuration constants (for adopting components for different applications), the type of the process (periodic and aperiodic), and temporal parameters such as deadline, frequency, and priority. Support for composing a system out of components is limited to general guidelines that are given to the designer for composing a system out of PBO components, and is not automated at all. This approach to componentization is somewhat unique since it gives methods for creating a framework that handles the communication, synchronization and scheduling of each component. Any C programming environment can be used to create components with minimal increase in performance or memory usage. Creating code using PBO methodology is an "inside out" programming paradigm as compared to a traditional coding of real-time processes. The PBO method provides consistent structure for every process and OS system services, such as communication, synchronization, scheduling. Only when necessary, OS calls PBO's method to execute application code. Analysis of the composed system is not considered.

3.4 A tabular overview

This chapter concludes with a tabular summary of investigated component-based systems and their characteristics. The tables 3.3 and 3.4 provide an additional instrument for comparing and analyzing component-based database and component-based embedded real-time systems.
The following symbols are used in the table:

x	—	feature is supported in/true for the system, and
x/p	—	feature is partially supported in/true for the system, i.e, the system fulfills the feature to a moderate extent.

Below follows a description of the criteria.

[1]The system aims to support the feature, but only the draft of how the feature should be supported by the system exists, i.e., there is no concrete implementation.

Platforms		DBMS platforms											Embedded and real-time platforms						
Characteristics of platforms		[30]	[76]	Oracle8i[82]	DataBlade[48]	UDB2[21]	Sybase[81]	Garlic[95]	DiscoveryLink[42]	OLE DB[74]	CORBAService[84]	KIDS[38]	SPIN[12]	2K[53]	Ensemble[64]	[49]	VEST[102]	PBO [105]
A. Type of the system	1) database	x	x	x	x	x	x	x	x	x	x	x							
	2) embedded												x		x				
	3) real-time																		
	4) embedded real-time													x		x	x	x	x
B. Status	1) research platform							x			x	x	x	x	x	x	x	x	
	2) commercial product	x	x	x	x	x	x		x	x									x
C. Component granularity	1) system	x	x																
	2) part of the system	x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
D. Category of the system	1) extensible			x	x	x	x						x						
	2) middleware							x	x	x				x					
	3) service										x								
	4) configurable											x			x	x	x	x	x
E. Component type	1) domain-specific data type or a new index			x	x	x	x												
	2) wrapper							x	x	x									
	3) service										x		x						
	4) DBMS subsystem											x							
	5) CORBA object	x												x					
	7) microprotocol		x												x				
	8) binary															x			
	9) passive																x		
	10) PBO																		x

Table 3.3: Characteristics of component-based database, embedded and real-time systems

Platforms		DBMS platforms												Embedded and real-time platforms						
Characteristics of platforms		[30]	[76]	Oracle8i[82]	DataBlade[48]	UDB2[21]	Sybase[81]	Garlic[95]	DiscoveryLink[42]	OLE DB[74]	CORBAService[84]	KIDS[38]	SPIN[12]	2K[53]	Ensemble[64]	[49]	VEST[102]	PBO [105]	
F.	Real-time properties	1) not preserved 2) preserved	x	x	x	x	x	x	x	x	x	x	x	x/p	x	x	x	x[1]	x/p	
G.	Interface/ communication	1) standardized 2) system specific 3) ports/unbuffered	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x			x
H.	Configuration tools	1) not available 3) available	x	x	x	x	x	x	x/p	x/p	x/p	x	x	x/p	x	x	x/p	x[1]	x/p	
I.	Analysis tools	1) not available 3) available	x	x	x	x	x	x	x	x	x	x	x/p	x	x/p	x	x	x/p[1]	x	
J.	Reusability	1) component 2) architecture	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x[1] x[1]	x	
K.	Tailoring ability	1) none 2) low 3) moderate 4) high	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x[1]	x	

Table 3.4: Characteristics of component-based database, embedded and real-time systems

A. Type of the system We investigated integration of the component-based software engineering for the system development in the following areas:

1. database,
2. embedded,
3. real-time, and
4. embedded real-time.

This criteria illustrates lack of component-based solutions in the area of embedded real-time database systems. As can be seen from table 3.3, there are few component-based systems that can be classified as embedded real-time; all component-based systems are either embedded and real-time systems, or database systems. There does not exist a component-based database system that can be classified as embedded real-time.

B. Status Analyzed component-based platforms are:

1. research platforms, and
2. commercial products.

As can be seen from the table 3.3, most component-based database systems are commercial products, while embedded real-time platforms are all research projects. It is important to notice that the database industry has embraced the component-based software paradigm and that the need for componentization is increasingly important, since almost all major database vendors provide some component-based solution of their database servers. Also, it is clear that real-time and embedded issues are not integrated in the commercial component-based development, thus, implying that there is still a lot of open research questions that need to be answered before component-based embedded real-time systems could be commericalized.

C. Component granularity There are two major granularity levels of components:

1. system as a component, and
2. part of the system as a component.

It is noticeable that a database component can be anything from a database system, a component of large granularity, to lightweight components that are composing parts of the DBMS. In contrast, most embedded real-time systems have lightweight components, i.e., parts of the system, and are building an operating system-like portion of embedded real-time systems. An exception is 2K where a component (CORBA object) is of larger granularity and can be an embedded system itself.

D. Category of the system If we emphasize that a database as a component cannot represent a real component-based solution of a database system, then our view is narrowed to a component DBMS (CDBMS) and parts or extensions of the database system as a component. Same can be stated for embedded real-time systems. Hence, investigated component-based systems are classified as follows:

1. extensible,
2. middleware,
3. service, and
4. configurable.

Note that service is the category of CDBMSs, and not of embedded real-time systems.

E. Component type A component in a component-based database system and a component-based embedded real-time system is one of the following:

1. domain-specific data type or a new index,
2. wrapper,
3. service,
4. DBMS subsystem,
5. CORBA object,
6. microprotocol,
7. binary component,
8. passive component, and
9. PBO (port-based object).

The first four component types are typically components found in database systems. We refer to these as database components. The last five types of components are mostly found in embedded real-time systems.

CDBMSs mainly consist of components that implement a certain database functionality (or extend existing functionality), usually mapped into services. Thus, a database component provides certain database service or function. For example, in Oracle8*i* the spatial data cartridge component implements non-standard DBMS functionality such as spatial index. Also, in KIDS, the DBMS subsystem component, transaction management, can provide several related services such as transaction, serialization and validation services, and in CORBAservices one component is one database service.

Components in embedded real-time systems are more diverse. In some systems components are not explicitly defined, and can only be classified as passive components (VEST) or as binary components (in development methodology introduced by Isović et al. [49]). Systems such as SPIN and Ensemble have more specific components, extensions for application-specific operating system services (SPIN) and microprotocols (Ensemble). Only 2K has standardized components (CORBA objects).

Note that almost every system has its own notion and a definition of a component, which suites the system's purpose and requirements.

F. Real-time properties Component-based database and embedded real-time systems may:

1. not preserve, or
2. preserve

real time properties. Component-based database systems do not enforce real-time behavior (see table 3.4). In addition, issues related to embedded systems such as low-resource consumption are not addressed at all. Accent in a database component is on providing a certain database functionality. In contrast, a component in existing component-based embedded real-time systems is usually assumed to be mapped to a task, i.e., passive components [102], binary components [49] and PBO components [105] are all mapped to a task. This comes naturally in the real-time research community because a task is the smallest schedulable unit in a real-time system. Therefore, analysis of real-time components in these solutions addresses the problem of managing temporal attributes at a component level by considering them as task attributes [49, 102, 56]:

- worst-case execution time (WCET) - the longest possible time it takes to complete a task,
- release time - the time at which all data that are required to begin executing are available,
- deadline - the time by which a task must complete its execution,
- precedence constraints and mutual exclusion - specify if a task needs to precede or exclude other tasks, and
- period - if a task is periodic.

Components in the PBO model [105] have only two temporal attributes, frequency and deadline. Components, in the development method for component-based real-time systems introduced by Isović et al. [49], are assumed to have all of the above listed temporal attributes, and these attributes are predicted in the design phase of the system development. VEST includes full list of temporal attributes in its components and expands that list with attributes essential for correct behavior of components in an embedded environment:

- power consumption requirements, and
- memory consumption requirements.

This makes the VEST approach the most flexible embedded real-time approach, since component attributes include real-time and embedded needs, and are known for every component in the library (this was not the case in the development method for component-based real-time systems introduced by Isović et al. [49]).

A component in an embedded real-time system must have well defined temporal properties, i.e., component behavior must be predictable, in order to be able to compose a reliable embedded real-time system. Hence, to be able to develop a database suitable for an embedded real-time system out of components, such components would have to have well-defined temporal properties and resource requirements, e.g., memory and power consumption, issues not addressed in current database components. Thus, we conclude that a database component used in an embedded real-time system would have to be a real-time component, and as such, mapped to a task in order to ensure predictability of the component and of the composed embedded real-time system, i.e., we need to know, for example, worst-case execution time for a database component to be able to ensure predictability of the component, and, in turn, of the database system composed out of such components.

G. Interfaces / communication The component communicates with its environment (other components) through well-defined interfaces. Generally, existing component-based database and embedded real-time solutions use:

1. standard interfaces,
2. system specific, and
3. unbuffered interfaces (ports), in which case they use communication through shared memory.

For example, standardized interfaces defined in the IDL are used in CORBAservices and in 2K. Also, OLE DB interface is used in the Microsoft's Universal Data Access architecture. Interfaces developed within the system are used in other systems, e.g., Oracle8*i* has extensibility interfaces and KIDS has component-specific interface. Systems that aim to allow more flexible composition use event-based communication, e.g., KIDS, SPIN, Ensemble, and 2K. Inter-component communication in database systems and embedded real-time systems have different goals. Interfaces in embedded real-time systems must be such that inter-component communication can be performed in a timely predictable manner. There are two possible ways of a real-time component communication:

- Buffered communication. The communication is done through message passing, e.g., [64].
- Unbuffered communication. Unbuffered data is accessed through shared memory, e.g., [104, 49].

Note that most component-based database systems use buffered communication since predictability of communication is not of importance in such systems. Systems enforcing real-time behavior use unbuffered communication (an exception is VEST where interfaces of components are not defined), due to several disadvantages of buffered communication [45, 49]:

- Sending and receiving messages incur significant overhead.
- Tasks waiting for data might block for an undetermined amount of time.
- Crucial messages can get lost as a result of the buffer overflow if tasks do not execute at the same frequency.
- Sending messages in control systems, which have many feedback loops, creates risk for deadlock.
- The upper bound on the number of produced/consumed messages must be determined to enable guarantee of temporal properties.

Generally, a real-time system using buffered communication is difficult to analyze due to dependencies among tasks. Unbuffered communication eliminates direct dependencies between tasks, since they only need to bind to a single element in the shared memory. Communication through shared memory incurs fewer overheads as compared to a message-passing system. Also, it is easier to check system's temporal behavior if unbuffered communication is used [49]. Hence, unbuffered style of communication is preferred style of communication in embedded real-time systems. It is suggested that interfaces in (hard) real-time systems should be unbuffered [49].

H. Configuration tools The development process must be well defined to enable efficient system assembly out of existing components from the component library or newly created ones. Adequate and automated configuration support must exist to help system designer with this process, e.g., rules for composing a system out of components, support for selection of an appropriate component from the component library, and support for the development of new components. However, in some systems configuration tools are not available. Hence, we identify that configuration tools in a component-based database and embedded real-time system are:

1. not available, and
2. available.

Observe (table 3.4) that most extensible CDBMSs have available configuration support. Since extensible systems already have fixed architecture and pre-defined extensions, provided configuration tools are sufficient to enable development and integration of components into an extensible system. The Garlic middleware technology provides a simple and fairly easy development of new components (wrappers), but these components are with simple functionality and structure. The situation gets more complicated if we consider OLE DB middleware, where not only components, data providers, can be added or exchanged in the architecture, but the architecture can also be extended to use customizable data providers. Hence, adequate configuration support is needed to ensure interoperability of different data providers. However, when developing a data provider component the

developer is only encouraged to follow specification exactly in order to ensure interoperability with other data providers, i.e, limited configuration support available. OLE DB is also limited in terms of use, it can only be used in Microsoft's computing environments.

On the other hand, extensible and middleware embedded real-time systems do not provide good configuration support. 2K, for example, does not have reusable components in its architecture and provides automated configuration (and reuse) of components from different systems. The correctness of integrated 2K middleware is only assumed (based on checks of inter-component dependencies). Also, SPIN offers very little configuration support, since correctness, configuration, and integration of components in the core system is based on features of the extension language in which components are developed. The rules for composition of a system are not defined in 2K and SPIN, and these systems can be viewed as the first generation of a component-based embedded real-time systems.

In general, demands on development support in configurable systems are high. KIDS has met these demands to some extent, with a well-defined development process. In most configurable embedded real-time systems, some configuration support is also provided. For example, PBO model gives good guidelines to help designer when composing system out of components.The development method for component-based real-time systems introduced by Isović et al. [49] provides configuration support for choosing appropriate component for system composition, i.e., checks of temporal properties and interfaces of components are performed to ensure that the component from the library is suitable for the system under development. In VEST, the necessity of having good configuration tools is recognized. Composition rules are defined through four types of dependency checks. Note that the development introduced by Isović et al. [49] and 2K have only inter-component dependency checks, which are just one out of four types of checks proposed in VEST. This makes the VEST approach the most appropriate one with respect to the correct system composition, because the more dependencies are checked in the system, within components, the probability of errors in the composed system, i.e., compositional errors, is minimized.

I. Analysis tools Since the reliability of the composed system depends on the level of correctness of the component, analysis tools are needed to verify the behavior of the component and the composed system. In particular, real-time systems must meet their temporal constraints, and adequate analysis to ensure that a system has meet temporal constraints is required. Thus, analysis tools are:

1. not available, and
2. available.

The problem of analysis of the composed component-based database system is rather straightforward, in most cases, analysis of the composed system is unavailable (see table 3.4). Importance of having good analysis of the composed system is recognized in KIDS, but is not pursued beyond that, i.e., analysis tools are not provided. Component-based embedded real-time systems do not provide analysis of the composed system as well. That is true for SPIN, 2K, and systems based on the PBO model. VEST introduces notion of reliability and real-time analysis of the system, but does not give more detailed description of such analysis. In the development method introduced by Isović et al. [49] checks of the WCET of components are performed.

G. Reusability Systems are composed out of reusable components from the library. Architecture of the system can be reused in the system's development as well. Thus, in component-based database and embedded real-time systems parts that can be reused are:

1. component, and
2. architecture.

As can be seen from the table 3.4, the part that can be reused in all systems is the component. KIDS and VEST can also reuse the architecture, as opposed to other systems. If we consider that reusability of architecture as a way to receive a higher pay-off [67, 75], KIDS and VEST have significantly higher degree of reusability as compared to others. The fact that an architecture of a system can be reused is also important from a different aspect, early system testing. For example, even though it does not have any analysis tools, KIDS offers good basis for the early system testing, in the first phase of the construction process, at the architecture level of the system development.

H. Tailoring ability The benefit of using component-based development in database systems is customization of the database for different applications. There are four degrees of tailorability in component-based database and embedded real-time systems:

1. none,
2. low,
3. moderate, and
4. high.

It is noticeable that extensible systems have low tailorability, middleware moderate, while configurable systems have high tailorability (see table 3.4). Since the goal is to provide an optimized database for a specific application with low development costs and short time-to-market, it is safe to say that configurable systems are the most suitable in this respect. In particular, VEST and KIDS, since they allow architectures to be saved and reused. At the same time, the methodology introduced in KIDS and VEST should enable tailoring of one generic system for a variety of different applications.

Chapter 4

Summary

This final chapter starts with a summary of the main issues we have identified in the report with respect to current state-of-the-art in the area of embedded databases for embedded real-time systems (section 4.1). In the last section (section 4.2) we identify challenges for future work.

4.1 Conclusions

Embedded systems, real-time systems, and database systems are research areas that have been actively studied. However, research on embedded databases for embedded real-time systems, explicitly addressing the development and design process, is sparse. A database that can be used in an embedded real-time system must handle transactions with temporal constraints, and must, at the same time, be suitable for embedded systems with limited amount of resources, i.e., the database should have small footprint, be portable to different operating system platforms, have efficient resource management, and be able to recover from a failure without external intervention.

As we have shown in this report, there are a variety of different embedded databases on the market. However, they vary significantly in their characteristics. Differences in products are typically the way data are organized in the database (data model), the architecture of the database (DBMS model), and memory usage. Commercial embedded databases also provide different interfaces for applications to access the database, and support different operating system platforms. Application developers must choose carefully the embedded database their application requires. This is a difficult, time consuming and costly process, with a lot of compromises. One solution could be to have a more generic embedded database platform that can be tailored and optimized such that it is suitable for different applications. Existing real-time database research platforms are unsuitable in this respect, since they are mainly monolithic systems, and as such, they are not easily tailored for new applications having different or additional requirements.

Satisfying requirements put on the embedded database in an embedded real-time system calls for a new way of designing and developing an embedded database for application specific system. For this purpose we have examined the component-based software engineering paradigm since it has been successfully applied in conventional non-real-time environments.

Traditional component-based systems, e.g., systems based on COM or CORBA component framework, normally view an entire database system as one component. We emphasize that the component-based database system is the database system, where the building parts are components, and refer to such building parts as database components. Components in a database system usually implement certain database functionality (or extend existing functionality), and these are normally mapped into services. In contrast, compo-

nents in embedded real-time systems are developed to have well-defined temporal properties, e.g., worst-case execution time, deadline, and release time, and are mapped to one or multiple tasks. We have shown that a real-time database component, i.e., a database component with temporal constraints that represents the intersecting type of the database and real-time component, does not exist. Existing component-based database systems do not enforce real-time behavior, and issues related to embedded systems such as low-resource consumption are not addressed at all in these solutions. However, in order to compose a reliable embedded real-time system out of components, each component's behavior must be predictable. This is the reason why components in existing component-based real-time systems are usually assumed to be mapped to a task with well-defined temporal properties. To ensure predictability and reliability of a database in an embedded real-time system, such database would have to be composed out of components, with well-defined and known temporal properties of a component must be well-defined and known. This, in turn, would require such a component to be mapped to a task.

We have observed that the architecture of component-based systems vary from fairly fixed (extensible systems), to completely configurable. In the extensible systems, which have fixed architecture, extensions are allowed only in well-defined places of the architecture. In the configurable systems components can be freely added or exchanged in the architecture. Configurable systems allow significant amount of flexibility to the developer and the user, in comparison to other systems. Hence, configurable systems represent the preferred type of component-based systems, as they allow the embedded database to be tailored for new applications with different requirements. Since configurable systems offer the highest degree of tailorability, and are the most flexible ones, we focus on these systems; in particular KIDS (component-based database system) and VEST (component-based embedded real-time system). Higher pay-off and quickness to market can be achieved if the architecture can be stored in a library and reused, as well as components (this feature can also be found in configurable systems such as KIDS and VEST). Moreover, to have an architectural abstraction of a system would enable early system testing. Temporal analysis of systems and components could be done in an early stage of the system development, thus, reducing development costs, and enhancing reliability and predictability of the system. For example, in KIDS the architecture is reusable, and even though it does not have any analysis tools, KIDS offers good basis for the early system testing; in the first phase of the construction process, at the architecture level of the system.

To ensure minimum errors in the system composition, the most efficient existing approach is to have a large number of dependency checks, as in VEST, where four types of dependency checks are introduced, e.g., factual, inter-component, aspects and general. Note that some systems, such as 2K, have only one type of dependency checks, i.e., inter-component dependency checks. Other systems do not have dependency checks at all, which makes them more vulnerable for compositional errors.

When composing an embedded database system out of components, the issue of inter-component communication should be carefully handled. Components in embedded real-time systems must communicate in a timely predictable manner, and for that reason most embedded real-time systems use unbuffered communication. In contrast, most existing component-based database systems have buffered communication (message passing), and are not concerned with predictability of inter-component communication. Furthermore, most component-based database systems use standardized interfaces, e.g., IDL, OLE DB, which makes them suitable for easier exchange and addition of new components (they must conform to a standard), while embedded real-time systems mostly have system-specific interfaces, i.e., interfaces defined within the system.

We have studied components and their definitions in component-based software engineering in particular, as well as in component-based database and embedded real-time systems. We found that every component-based system has its own notion and a definition of a component. Thus, the component is to a large extent an arbitrary notion, and is practically re-defined and re-invented in each system. However, there are at least three

common requirements that a component must satisfy in any component-based system (i) to be a composing part of the system; (ii) to be reusable with well-defined conditions and ways of reuse; and (iii) to have well-defined interfaces for inter-connection with other components. Furthermore, rules for composition of a system must be defined, and (automated) tools to assist the developer must be available. Although most of the component-based database and embedded real-time systems focus on components, their solutions lack good composition rules, and automated development support.

4.2 Future work

It is evident that research for embedded databases that explicitly addresses (i) development and design process, and (ii) limited amount of resources, inherits challenges from the traditional real-time systems, embedded systems, and component-based software engineering.

Key challenges in component-based software engineering include:

- developing a component that can be reused as many times as possible, and would conform to a standard,
- determining how a component should be verified to obtain appropriate information in order to determine suitability of a component for a specific system,
- defining rules for the composition of a reliable system, which satisfies specification requirements,
- analyzing the behavior of the composed system, as well as ensuring trade-off analysis between conflicting requirements in the early design stage, and
- providing adequate configuration support for development process, in particular determining what is doable by tools and what needs to be left to the designer.

The above research challenges in a component-based software engineering are further augmented for embedded real-time systems. The additional challenges include:

- defining a component with predictable behavior,
- determining how inter-component communication should be performed, and what are appropriate interfaces for real-time components,
- managing resources such as memory usage and required processing time for different operations of a component, possibly on different hardware platforms and operating system platforms, and
- determining what are the appropriate analysis tools for analysis of system resource demands, and analysis of the timing properties of the composed system.

Embedding a database in a real-time system also brings a set of challenges:

- determining what type of the DBMS architecture is appropriate (library vs client-server) to ensure predictability in a real-time system,
- determining the most appropriate data model for the embedded database, and
- determining which type of interfaces, through which user is accessing a database, is the most suitable.

Finally, fundamental research questions when developing an embedded database for an embedded real-time system using component-based software engineering include:

- defining a real-time database component, i.e., functionality and services that a component should provide, and temporal constraints a component should satisfy,
- determining which component interfaces are appropriate, and how inter-component communication should be performed (message passing vs shared memory), and
- integrating the composed database into an embedded real-time system.

Our research is focused on providing an experimental research platform for building embedded databases for embedded real-time systems. At a high-level, the platform consists of two parts. First, we intend to develop a component library, holding a set of methods, which can be used when building an embedded database. Initially, we will develop a set of components that deal with concurrency control, scheduling, and main-memory techniques. At the next step, we develop tools that, based on the application requirements, will support the designer when building an embedded database using these components. More importantly, we want to develop application tools and techniques that: (i) support the designer in the composition and tailoring of an embedded database for a specific system using the developed components, where the application requirements are given as an input; (ii) support the designer when analyzing the total system resource demand of the composed embedded database system; and (iii) help the designer by recommending components and methods if multiple components can be used, based on the application requirements. Further, such a tool will help the designer to make trade-off analysis between conflicting requirements early in the design phase.

The components should carry property descriptions of themselves with information about (i) what service and functionality the component provides; (ii) what quality of service guarantees facilitated by the component, e.g., techniques may be applicable to soft real-time applications but not hard real-time applications; (iii) their system resource demand, e.g, memory usage and required processing time for different operations and services provided by a component, possibly on different hardware platforms in order to simplify interoperability; and (iv) composition rules, i.e., which specifies how, and with which components, a component can be combined.

Our research should give better understanding of the specification of components to be used in an embedded and real-time setting. This includes functionality provided by the component, the resource demand required by a component when executed on different platforms, and rules for specifying how components can be combined and how the overall system can be correctly verified given that each component has been verified.

Bibliography

[1] *Proceedings of International Workshop, Database in Telecommunications*, number 1819 in Lecture Notes in Computer Science, Edinburgh, Scotland, UK, September 1999. Springer-Verlag.

[2] S. Abiteboul, B. Amann, S. Cluet, A. Eyal, L. Mignet, and T. Milo. Active views for electronic commerce. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases*, pages 138–149, Edinburgh, Scotland, UK, September 1999. Morgan Kaufmann. ISBN 1-55860-615-7.

[3] G. D. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architectures. *ACM Software Engineering Notes*, 18(5):9–20, 1993.

[4] B. Adelberg, H. Garcia-Molina, and B. Kao. Emulating soft real-time scheduling using traditional operating system schedulers. In *Proceedings of IEEE Real-Time System Symposium*, 1994.

[5] B. Adelberg, H. Garcia-Molina, and J. Widom. The STRIP rule system for efficiently maintaining derived data. In *SIGMOD*, pages 147–158, 1997.

[6] B. Adelberg, B. Kao, and H. Garcia-Molina. Overview of the STanford Real-Time Information Processor (STRIP). *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(1), 1996.

[7] B. S. Adelberg. *STRIP: A Soft Real-Time Main Memory Database for Open Systems*. PhD thesis, Stanford University, 1997.

[8] Q. N. Ahmed and S. V. Vrbsky. Triggered Updates for Temporal Consistency in Real-Time Databases. *Real-Time Systems*, 19(3):209–243, November 2000.

[9] 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 Sigmond Record*, Volume 25, 1996.

[10] F. Baothman, A. K. Sarje, and R. C. Joshi. On optimistic concurrency control for RT-DBS. *IEEE Region 10 International Conference on Global Connectivity in Energy, Computer, Communication and Control*, 1998.

[11] L. Bass, P. Clements, and R. Kazman. *Software Architecture In Practice*. SEI Series in Software Engineering. Addison Wesley, 1998.

[12] B. N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. G. Sirer. SPIN - an extensible microkernel for application-specific operating system services. Technical Report 94-03-03, Department of Computer Science and Engineering, University of Washington, Seattle, WA, USA, February 1994.

[13] L. Blair and G. Blair. A tool suite to support aspect-oriented specification. In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP '99*, pages 7–10, Lisbon, Portugal, June 1999. A position paper.

[14] J. A. Blakeley. OLE DB: A component DBMS architecture. In *Proceedings of the 12th International Conference on Data Engineering (ICDE)*, pages 203–204, New Orleans, Louisiana, USA, March 1996. IEEE Computer Society Press.

[15] J. A. Blakeley. Universal data access with OLE DB. In *Proceedings of the 42nd IEEE International Computer Conference (COMPCON)*, pages 2–7, San Jose California, February 1997. IEEE Computer Society Press.

[16] J. A. Blakeley and M. J. Pizzo. *Component Database Systems*, chapter Enabling Component Databases with OLE DB. Data Management Systems. Morgan Kaufmann Publishers, 2000.

[17] J. Bosch. *Design and Use of Software Architectures*. ACM Press in collaboration with Addison-Wesley, 2000.

[18] A. P. Buchmann, A. Deutsch, J. Zimmermann, and M. Higa. The REACH active OODBMS. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 476–476, 1995.

[19] A. P. Buchmann, J. Zimmermann, J. A. Blakeley, and D. L. Wells. Building an Integrated Active OODBMS: Requirements, Architecture, and Design Decisions. In *Proceedings of the 11th International Conference on Data Engineering*, pages 117–128. IEEE Computer Society Press, 1995.

[20] R. Camposano and J. Wilberg. Embedded system design. *Design Automation for Embedded Systems*, 1(1):5–50, 1996.

[21] M. J. Carey, L. M. Haas, J. Kleewein, and B. Reinwald. Data access interoperability in the IBM database family. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Interoperability*, 21(3):4–11, 1998.

[22] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. Luniewski, W. Niblack, D. Petkovic, J. Thomas II, J. H. Williams, and E. L. Wimmers. Towards heterogeneous multimedia information systems: The Garlic approach. In *Proceedings of the 5th International Workshop on Research Issues in Data Engineering: Distributed Object Management (RIDE-DOM)*, pages 124–131, Taipei, Taiwan, March 1995. IEEE Computer Society. ISBN 0-8186-7056-8.

[23] COMPOST. http://i44w3.info.uni-karlsruhe.de/ compost/. University of Karlsruhe, Germany.

[24] IBM Coorporation. http://www.software.ibm.com/data/datjoiner.

[25] I. Crnkovic and M. Larsson. A case study: Demands on component-based development. In *Proceedings of 22th International Conference of Software Engineering*, pages 23–31, Limeric, Ireland, June 2000. ACM.

[26] I. Crnkovic, M. Larsson, and F. Lüders. State of the practice: Component-based software engineering course. In *Proceedings of 3rd International Workshop of Component-Based Software Engineering*. IEEE Computer Society, January 2000.

[27] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley, 2000.

[28] J. R. Davis. Creating an extensible, object-relational data management environment: IBM's DB2 Universal Database. Database Associated International, InfoIT Services, November 1996. Available at http://www.dbaint.com/pdf/db2obj.pdf.

[29] K. R. Dittrich and A. Geppert. *Component Database Systems*, chapter Component Database Systems: Introduction, Foundations, and Overview. Data Management Systems. Morgan Kaufmann Publishers, 2000.

[30] A. Dogac, C. Dengi, and M. T. Öszu. Distributed object computing platform. *Communications of the ACM*, 41(9):95–103, 1998.

[31] ENEA Data. OSE Real-time system. http://www.enea.se.

[32] J. Eriksson. Real-time and active databases: A survey. In *Proceedings of the Second International Workshop on Active, Real-Time and Temporal Databases*, nr. 1553 in Lecture note series. Springer-Verlag, December 1998.

[33] J. Eriksson. *Specifying and Managing Rules in an Active Real-Time Database System*. Licenciate Thesis, Number 738. Linköping University, Sweden, December 1998.

[34] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *The communications of the ACM*, 19, 1976.

[35] S. Chakravarthy et al. HiPAC: A research project in active, time-constrained database management. Technical Report XAIT-8902, Xerox Advanced Information Technology, 1989.

[36] W. Fleisch. Applying use cases for the requirements validation of component-based real-time software. In *Proceedings of 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 75–84, Saint-Malo, France, May 1999. IEEE.

[37] A. Gal. Data management in ecommerce (tutorial session): the good, the bad, and the ugly. In *Proceedings of the 2000 ACM SIGMOD on Management of Data*, volume 34, page 587, Dallas, TX, USA, May 2000. ACM.

[38] A. Geppert, S. Scherrer, and K. R. Dittrich. KIDS: Construction of database management systems based on reuse. Technical Report ifi-97.01, Department of Computer Science, University of Zurich, September 1997.

[39] J. Gray. Notes on database operating systems. In *Operating Systems: An Advanced Course*, number 60 in Lecture Notes in Computer Science, pages 393–481. Springer-Verlag, 1978.

[40] A. Gupta, V. Harinarayan, and A. Rajaraman. Virtual database technology. In *Proceedings of 14th International Conference on Data Engineering*, pages 297–301, 1998.

[41] R. K. Gupta. *Co-Synthesis of Hardware and Software for Digital Embedded Systems*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1995.

[42] L. M. Haas, P. Kodali, J. E. Rice, P. M. Schwarz, and W. C. Swope. Integrating life sciences data-with a little Garlic. In *Proceedings of the IEEE International Symposium on Bio-Informatics and Biomedical Engineering (BIBE)*, pages 5–12. IEEE, 2000.

[43] J. Hansson. Dynamic real-time scheduling in OSE Delta. Technical Report HS-IDA-TR-94-007, Dept. of Computer Science, Univ. of Skövde, 1994.

[44] J. R. Haritsa and K. Ramamritham. Real-time database systems in the new millennium. *Real-Time Systems*, 19(19):205–208, November 2000. The International Journal of Time-Critical Computing Systems.

[45] M. Hassani and D. B. Stewart. A mechanism for communication in dynamically reconfigurable embedded systems. In *Proceedings of High Assurance Software Engineering (HASE) Workshop*, pages 215–220, Washington DC, August 1997.

[46] J. Huang, J. Stankovic, K. Ramamritham, and D. Towsley. Experimental Evaluation of Real-time Optimistic Concurrency Control Schemes. In *Proceedings of VLDB*, Barcelona, Spain, September 1991.

[47] Sybase Inc. http://www.sybase.com.

[48] Developing DataBlade modules for Informix-Universal Server. Informix DataBlade Technology. Informix Corporation, 22 March 2001. Available at http://www.informix.com/datablades/.

[49] D. Isovic, M. Lindgren, and I. Crnkovic. System development with real-time components. In *Proceedings of ECOOP Workshop - Pervasive Component-Based Systems*, France, June 2000.

[50] K. Ramamritham. Real-Time Databases. *International Journal of distributed and Parallel Databases*, pages 199–226, 1993.

[51] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.

[52] Y-K. Kim, M. R. Lehr, D. W. George, and S. H. Song. A database server for distributed real-time systems: Issues and experiences. In *Proceedings of the Second IEEE Workshop on Parallel and Distributed Real-Time Systems*, 1994.

[53] F. Kon, R. H. Campbell, F. J. Ballesteros, M. D. Mickunas, and K. Nahrsted. 2K: A distributed operating system for dynamic heterogeneous environments. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC'9)*, pages 201–208, Pittsburgh, August 2000.

[54] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhales, and R. H. Campbell. Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in Lecture Notes in Computer Science, pages 121–143, New York, USA, April 2000. Springer.

[55] F. Kon, A. Singhai, R. H. Campbell, and D. Carvalho. 2K: A reflective, component-based operating system for rapidly changing environments. In *Proceedings of the ECOOP Workshop on Reflective Object Oriented Programming and Systems*, volume 1543 of *Lecture Notes in Computer Science*, Brussels, Belgium, July 1998. Springer.

[56] C. M. Krishna and K. G. Shin. *Real-time Systems*. McGraw-Hill Series in Computer Science. The McGraw-Hill Companies, Inc., 1997.

[57] H. T. Kung and J. T. Robinsson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6, 1981.

[58] T-W. Kuo, C-H. Wei, and K-Y. Lam. Real-Time Data Access Control on B-Tree Index Structures. In *Proceedings of the 15th International Conference on Data Engineering*, 1999.

[59] M. Larsson and I. Crnkovic. New challenges for configuration management. In *Proceedings of System Configuration Management, 9th International Symposium (SCM-9)*, volume 1675 of *Lecture Notes in Computer Science*, pages 232–243, Toulouse, France, August 1999. Springer.

[60] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *Proceedings of the 12th Conference on Very Large Databases, Morgan Kaufmann pubs. (Los Altos CA), Kyoto*, 1986.

[61] 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*, pages 158–173, 1999.

[62] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th International Conference on Very Large Databases*, 1980.

[63] B. Liu. Embedded real-time configuration database for an industrial robot. Master's thesis, Linköping University, 2000.

[64] X. Liu, C. Kreitz, R. Renesse, J.Hickey, M. Hayden, K. Birman, and R. Constable. Building reliable, high-performance communication systems from components. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, volume 34, pages 80–92, December 1999. Published as Operating Systems Review.

[65] H. Lu, Y. Yeung Ng, and Z. Tian. T-Tree or B-Tree: Main Memory Database Index Structure Revisited. *11th Australasian Database Conference*, 2000.

[66] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995. Special Issue on Software Architecture.

[67] C. H. Lung, S. Bot, K. Kalaichelvan, and R. Kazman. An approach to software architecture analysis for evolution and reusability. In *Proceedings of CASCON*, Toronto, ON, November 1997. IBM Center for Advanced Studies.

[68] U. Aßmann. A component model for invasive composition. In *ECOOP 2000 Workshop on Aspects and Dimensions of Concerns*, Sophia, Antipolis and Cannes, June 2000.

[69] U. Aßmann. Invasive software composition with program transformation. Forthcoming habilitation, 2001.

[70] MBrane Ltd. http://www.mbrane.com.

[71] N. Medvedovic and R. N. Taylor. Separating fact from fiction in software architecture. In *Proceedings of the Third International Workshop on Software Architecture*, pages 10–108. ACM Press, 1999.

[72] J. Mellin, J. Hansson, and S. Andler. Refining timing constraints of application in DeeDS. In S. H. Son, A. Bestavros, and K-J. Lin, editors, *Real-Time Database Systems: Issues and Applications*, pages 325–343, 1997.

[73] B. Meyer and C. Mingins. Component-based development: From buzz to spark. *IEEE Computer*, 32(7):35–37, July 1999. Guest Editors' Introduction.

[74] Universal data access through OLE DB. OLE DB Technical Materials. OLE DB White Papers, 12 April 2001. Available at http://www.microsoft.com/data/techmat.htm.

[75] R. T. Monroe and D. Garlan. Style-based reuse for software architectures. In *Proceedings of the Fourth International Conference on Software Reuse*, pages 84–93. IEEE Computer Society Press, April 1996.

[76] A. Münnich, M. Birkhold, G. Färber, and P. Woitschach. Towards an architecture for reactive systems using an active real-time database and standardized components. In *Proceedings of International Database Engineering and Application Symposium (IDEAS)*, pages 351–359, Montreal, Canada, August 1999. IEEE Computer Society Press.

[77] T. Niklander and K. Raatikainen. RODAIN: A Highly Available Real-Time Main-Memory Database System. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium*, 1998.

[78] C. Norström, M. Gustafsson, K. Sandström, J. Mäki-Turja, and N-E. Bånkestad. Findings from introducing state-of-the-art real-time techniques in vehicle industry. In *Industrial session of the 12th Euromicro Conference on Real-Time Systems*, Stockholm, Sweden, June 2000.

[79] Objectivity inc. Objectivity/DB. http://www.objectivity.com.

[80] M. A. Olson. Selecting and implementing an embedded database system. *IEEE Computers*, 33(9):27–34, September 2000.

[81] S. Olson, R. Pledereder, P. Shaw, and D. Yach. The Sybase architecture for extensible data management. *Data Engineering Bulletin*, 21(3):12–24, 1998.

[82] All your data: The Oracle extensibility architecture. Oracle Technical White Paper. Oracle Corporation. Redwood Shores, CA, February 1999.

[83] H. Ossher and P. Tarr. Subject-oriented programming: a critique of pure objects. In *Proceedings of the eighth annual conference on object-oriented programming systems, languages, and applications*, pages 411–428, Washington, USA, September 26 - October 1 1993. ACM Press.

[84] M. T. Özsu and B. Yao. *Component Database Systems*, chapter Building Component Database Systems Using CORBA. Data Management Systems. Morgan Kaufmann Publishers, 2000.

[85] P. Pardyak and B. N. Bershad. Dynamic binding for an extensible system. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Operating Systems Review, Special Issue, pages 201–212, Seattle WA, USA, October 1996. ACM and USENIX Association. ISBN 1-880446-82-0.

[86] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering*, volume 17, 1992.

[87] Persistence software inc. PowerTier. http://www.persistence.com.

[88] Pervasive Software Inc. http://www.pervasive.com.

[89] Pervasive Software Inc. Pervasive.SQL 2000 Reviewers guide. Technical white paper.

[90] POET software corp. POET Object. http://www.poet.com.

[91] Polyhedra Plc. http://www.polyhedra.com.

[92] DARPA ITO projects. Program composition for embedded systems. http://www.darpa.mil/ito/research/pces/index.html, 7 August 2001.

[93] K. Raatikainen and J. Taina. Design issues in database systems for telecommunication services. In *Proceedings of IFIP-TC6 Working conference on Intelligent Networks, Copenhagen, Denmark*, pages 71–81, 1995.

[94] Raima Corporation. Databases in real-time and embedded systems. http://www.raimabenelux.com/, February 2001.

[95] M. T. Roth and P. M. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *Proceedings of 23rd International Conference on Very Large Data Bases (VLDB'97)*, pages 266–275, Athens, Greece, August 1997. Morgan Kaufmann. ISBN 1-55860-470-7. Available at http://dblp.uni-trier.de.

[96] S. Ortiz Jr. Embedded databases come out of hiding. In *IEEE Computer*, 2000.

[97] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[98] R. Sivasankaran, J. Stankovic, D. Towsley, B. Purimetla, and K. Ramamritham. Priority assignment in real-time active databases. *The VLDB Journal*, Volume 5, 1996.

[99] Sleepycat Software Inc. http://www.sleepycat.com.

[100] X. Song and J. Liu. Maintaining temporal consistency: Pessimistic vs. optimistic concurrency control. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, October 1995.

[101] J. Stankovic. Misconceptions about real-time computing: a serious problem for next-generation systems. *IEEE Computer*, 21, Oct 1988.

[102] J. Stankovic. VEST: A toolset for constructing and analyzing component based operating systems for embedded and real-time systems. Technical Report CS-2000-19, Department of Computer Science, University of Virginia, Charlottesville, VA, May 2000.

[103] J. A. Stankovic, S. H. Son, and J. Liebeherr. *Real-Time Databases and Information Systems*, chapter BeeHive: Global Multimedia Database Support for Dependable, Real-Time Applications, pages 409–422. Kluwer Academic Publishers, 1997.

[104] D. B. Stewart, M. W. Gertz, and P. K. Khosla. Software assembly for real-time applications based on a distributed shared memory model. In *Proceedings of the 1994 Complex Systems Engineering Synthesis and Assessment Technology Workshop*, pages 214–224, Silver Spring, MD, July 1994.

[105] D. S. Stewart. Designing software components for real-time applications. In *Proceedings of Embedded System Conference*, San Jose, CA, September 2000. Class 408, 428.

[106] D. S. Stewart and G. Arora. Dynamically reconfigurable embedded software - does it make sense? In *Proceedings of IEEE Intl. Conf. on Engineering of Complex Computer Systems (ICECCS) and Real-Time Application Workshop (RTAW)*, pages 217–220, Montreal, Canada, October 1996.

[107] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

[108] J. Taina and K. Raatikainen. RODAIN: A Real-Time Object-Oriented Database System for Telecommunications. In *Proceedings of the DART'96 workshop*, pages 12–15, 1996.

[109] TimesTen Performance Software. http://www.timesten.com.

[110] H. Tokuda and C. Mercer. ARTS: A Distributed Real-Time Kernel. In *ACM Operating Systems Review, 23(3)*, 1989.

[111] A. Wall. Software architecture for real-time systems. Technical Report 00/20, Mälarden Real-Time Research Center, Mälardalen University, Västerås, Sweden, May 2000.

[112] D. L. Wells, J. A. Blakeley, and C. W. Thompson. Architecture of an open object-oriented database management system. *IEEE Computer*, 25(10):74–82, 1992.

[113] M. Xiong, R. Sivasankran, J. Stankovic, K. Ramamritham, and D. Towsley. Scheduling transaction with temporal constraints: Exploiting data semantics. In *Proceedings of the 17th IEEE Real-time Systems Symposium*, 1996.

[114] P. S. Yu, K. Wu, K. Lin, and S. H. Son. On real-time databases: Concurrency control and scheduling. In *Proceedings of the IEEE*, volume 82, pages 140–157, 1994.

[115] G. Zelesnik. *The UniCon Language User Manual*. Carnegie Mellon University, Pittsburgh, USA, May 1996. Available at http://www.cs.cmu.edu/afs/cs/project/vit/www/unicon/reference-manual/.

[116] J. Zimmermann and A. P. Buchmann. REACH. In *N. Paton (ed): Active Rules in Database Systems, Springer-Verlag*, 1998.