The COMET Database Management System

Dag Nyström, Aleksandra Tešanović,
Christer Norström, and Jörgen Hansson

1 Introduction

The COMET DBMS (component-based embedded real-time database management system) is an experimental database platform. COMET is intended for resource-constrained embedded vehicle control-systems.

The complexity of modern vehicle control systems is rapidly increasing [1], and so is also the amount of data that needs to be handled and maintained. Thus, a need has emerged for a structured way of handling this data. A real-time database management system would fulfill this need by providing a higher level of abstraction of the data management.

Different vehicular control-systems have different requirements, i.e., some might be static and non-preemptive while other might be much more flexible, allowing preemption. Many vehicular systems are distributed, but some are not. This puts different requirements on the database. The natural approach would then be to develop an in-house DBMS that fulfills the requirements of the application. However, implementing a real-time database management system, and getting it certified as a component in a vehicular system is not a small task. A different approach would be to develop a tailorable database management system that can be configured to meet different kind of requirements.

COMET is, too a high degree, tailorable to meet different kinds of requirements, such as different task-models, architectures, temporal constraints, and resource constraints. To achieve this, COMET is designed using both a component-based and aspect-oriented approach.

Component-based software development enables systems to be built by previously built components. Each component is responsible for a well-defined task. By selecting a set of components that meet the requirements of the application, a specialized database can be made. In COMET we have identified the following components, user-interface component, transaction management component, index management component, memory management component, lock management component, check pointing and recovery management. The tasks of each of these components are described in section 5.

Aspect-oriented software development allows some concerns of the system to be separated into an aspect. We have identified a number of aspects in COMET. Some aspects, such as the concurrency control-aspect change the behavior of the system, while other, such as the temporal aspect, are used to describe non-functional properties of the system. Aspect-oriented software development is further described in section 2.

The COMET concept currently consists of a set of components, a set of aspects, a DBMS architecture and a tool to calculate worst-case execution times. Future versions of COMET will contain, configuration tools, more components and aspects, as well as further analysis tools.

In this report we describe the underlying concepts of COMET. The architecture of COMET, the individual components and aspects identified in COMET, as well as our component-model, RTCOM, and our design model ACCORD is furthermore described. Finally we describe the COMET BaseLine implementation, which is the first instance of COMET intended for a vehicle control-system described in [2].

2 Aspect-Oriented Software Development

Aspect-Oriented Software Development (AOSD) has emerged as a new principle for software development, and is based on the notion of separation of concerns [3]. Typically, AOSD implementation of a software system has the following constituents [3]:

- components, written in a language, which has support for aspect-oriented programming, e.g., C, C++, Java,
- aspects, written in the corresponding aspect language[1], e.g., AspectC [4], AspectC++ [5], AspectJ [6], and
- an aspect weaver, which is a special compiler that combines the components and aspects.

Components used for system composition in AOSD are not black box components (as they are in CBSE), rather they are *white box* components as we can modify their internal behavior by weaving different aspects in the code of a component.

Aspects are commonly considered to be a property of a system that affects its performance or semantics, and that crosscuts the system's functionality [3]. Aspects of software such as persistence and debugging can be described separately and exchanged independently of each other without disturbing the modular structure of the system. Each aspect declaration consists of advices and pointcuts.

A *pointcut* in an aspect language consists of one or more join points, and is described by a pointcut expression. Pointcuts define points in the static structure or a dynamic control flow of the program. A *join point* refers to a point in the component code where aspects should be weaved, e.g., a method, a type (struct or union).

Figure 1 shows the definition of a named pointcut `getLockCall`, which refers to all calls to the function `getLock()` and exposes a single integer argument of that call[2].

```
pointcut getLockCall(int lockId) =
   call(‘‘void getLock(int)’’) && args(lockId);
```

Fig. 1: An example of the pointcut definition

```
advice getLockCall(lockId):
   void after (int lockId)
{
   cout << ‘‘Lock requested is’’ << lockId << endl;
}
```

Fig. 2: An example of the advice definition

An *advice* is a declaration used to specify the code that should run when the join points, specified by a pointcut expression, are reached. Different kinds of advices can be declared, such as: (i) *before advice*, which is executed before the join point, (ii) *after advice*, which is executed immediately after the join point, and (iii) *around advice*, which is executed in place of the join point. Figure 2 shows an example of an after advice. With this advice each call to `getLock()` is followed by the execution of the advice code, i.e., printing of the lock id.

[1] All existing aspect languages are conceptually very similar to AspectJ, developed for Java.
[2] The examples presented are written in AspectC++.

3 ACCORD: Aspectual Component-Based Real-Time System Development

The COMET database is developed using ACCORD (aspectual component-based real-time system development)[7], which is a method to design aspectual component-based real-time systems.

In ACCORD the system is first decomposed in a set of components, and then further decomposed into a set of aspects. A component performs a specific task and is accessed using one or more well-define interfaces. Aspects on the other hand are semantic functions or properties that crosscuts the system, influencing several components.

ACCORD consider three different categories of aspects, namely, (i) Application aspects which changes the functional behavior of the application, (ii) run-time aspects which describe extra-functional properties of the compiled application, and (iii) composition aspects which describe which components and aspects that can be combined. For further reading on these categories of aspects, we refer to [7].

Application aspects are aspects that change the functional behavior of one or more components. This is done by writing aspect-code that is weaved into components using an aspect weaver. An example of a functional aspect is, an earliest deadline first policy that is weaved into a scheduling component, and debug information that is weaved into a component during systems testing.

In this report, the application aspects are within the scope, we will not discuss run-time and composition aspects further. In sections 5,7.1, and 7.2 we will further elaborate on aspects derived during the development of the COMET DBMS.

4 RTCOM: Real-Time Component-Model

To support the different kinds of aspects used in ACCORD, the component-model RTCOM [8] (real-time componenent model) is used in COMET.

The component consists of three parts, each corresponding to one type of aspect in ACCORD, the different parts can be seen in figure 3. The functional part, which is the actual code, corresponds to application aspects, the run-time part corresponds to run-time aspects and the compositional part corresponds to compositional aspects.

The functional part consists of two sub parts, the policy part and the mechanism part. The mechanisms part consists of mechanisms that are fixed, while the policy part consists of operations that can be modified by application aspects. A component is accessed through a number of required and provided interfaces. Each interface consists of a subset of the component's operations.

For example, consider a component that implements a fifo-list. Internally the list is implemented using a linked list. In the mechanism part a number of mechanisms exist that are used to modify the linked list, i.e., `linkIn`, `linkOut`, `getFirst` and `getNext`. However the policy part consists of two operations, namely `enqueue` and `dequeue`.

The operations can call all the mechanisms in the component, as well as operations of other components, but not other operations of the same component. It is important to write the operations as structured and modular as possible.

Further, consider the aspect `priority` is weaved into the component. The aspect contains aspects code that is weaved into the operations in such a way that the list is sorted, based on a priority, i.e., the `enqueue` operation is modified to link the new element in its appropriate place in the list.

The run-time part contains information about different run-time aspects of the component, such as memory requirements, platform compatibility and worst-case execution time etc. Several of these aspects are dependant of the application aspects, such as the memory requirements and the worst-case execution time aspects. If an application aspect is weaved

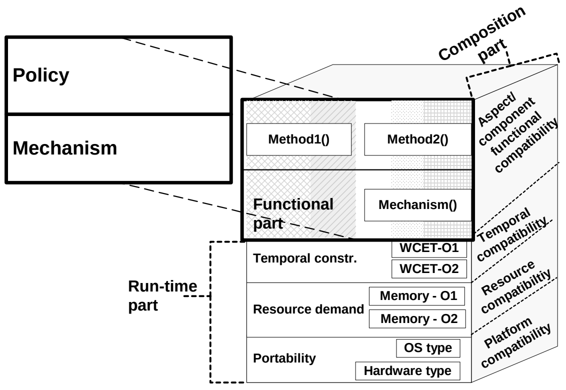

Fig. 3: The real-time component-model

into the operations of the components, it will affect the worst-case execution time and memory requirements of the component. To handle this, a component description language that supports adding application aspects to components is used [9].

The formal description of the RTCOM component-model presentented below is somewhat relaxed compared to the description in [8] in which mechanisms are not allowed to call operations in other components. This extension to the model has been made in order to implement the COMET BaseLine described in section 8.

Definition 1: A **component-type** t consists of the following:

1. A set of operations $O = \{o_1, o_2, \dots, o_n\}$
2. A set of state variables $S = \{s_1, s_2, \dots, s_m\}$
3. A set of mechanisms $M = \{m_1, m_2, \dots, m_p\}$
4. A set of provided interfaces $P = \{p_1, p_2, \dots, p_q\}$
5. A set of required interfaces $R = \{r_1, r_2, \dots, r_r\}$

Definition 2: A **component** c is an instance of a component-type t.

$$c = \texttt{inst}(t)$$

Definition 3: An **operation** o in a component-type t is constructed from the following:

1. A set of mechanisms $M_o \subseteq M \in t$.
2. A set of operations $O_o \subseteq O \in t$, such that no operation on O_o calls o.
3. A set of operations provided by other component-types that does not call any operations in t.[3]
4. A set of state variables $S_o \subseteq S \in t$, that can be manipulated by o.
5. Glue code

Definition 4: A **mechanism** m in a component-type t is constructed from the following:

1. A set of mechanisms $M_m \subseteq M \in t$.

[3] Recursive calls among operations in RTCOM is not allowed in order to enable calculation of properties such as WCET.

2. A set of operations provided by other component-types that does not call any operations in t.

3. A set of state variables $S_m \subseteq S \in t$, that can be manipulated by m.

4. Glue code.

Definition 5: A **provided interface** p in a component-type t is constructed from a subset of the operations in t.

Definition 6: A **required interface** r **in a component-type** t **as a provided interface of a component-type** $t_{prov} \neq t$

Definition 7: An **application aspect** a is constructed of the following:

1. A number of component-types $T_{asp} = \{t_1, t_2, \dots, t_s\}$ which a crosscuts.

2. A set of join points $J = \{j_1, j_2, \dots, j_t\}$, where each j_x is a location in a specific operation o in a component-type $t \in T_{asp}$.

3. A set of advices $D = \{d_1, d_2, \dots, d_u\}$, where each d_x is a piece of code that can be inserted before, after or instead of the location pointed out by a join point j.

4. A set of code weavings $W = \{(j_1, d_1), \dots, (j_1, d_x), \dots, (j_n, d_m)\}$

Definition 8: A set of component-types T weaved with a set of aspects A will result in a new set of component-types T'.

$$T \times A \rightarrow T'$$

5 The Architecture of COMET

When we designed COMET, we used the ACCORD approach, and thus we started by identifying the different activities in a database management system. We came up with the following activities:

1. User interface management
2. Transaction management
3. Scheduling management
4. Index management
5. Memory management
6. Lock management
7. Concurrency controlling
8. Checkpoint and recovery management
9. Logging management

One could argue that a database management system might consist of more activities, e.g., authorization management, query-optimizing management, and distribution management. We drew the conclusion, however, that the above-mentioned activities where the most central for vehicular control-systems, based on the case study presented in [2]. However COMET could, without too much effort, be extended to include these additional activities too.

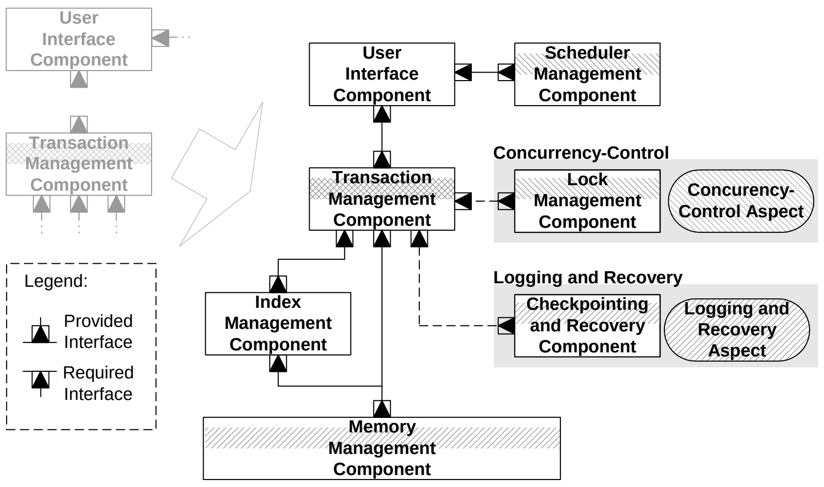
User Interface Component
Transaction Management Component
Legend:
Provided Interface
Required Interface
User Interface Component
Scheduler Management Component
Concurrency-Control
Transaction Management Component
Lock Management Component
Concurrency-Control Aspect
Logging and Recovery
Index Management Component
Checkpointing and Recovery Component
Logging and Recovery Aspect
Memory Management Component

Fig. 4: The Architecture of COMET

The second step was to extract components from these activities, and we found that activity 1 to 5 was candidate components, since these activities are well defined and isolated. Furthermore, the use of these components, assembled together, would result in a functional database management system. The limitations on such a database would be that it could not handle concurrent transactions, as well as recovery.

In the third step, concurrency controlling and logging/recovery management were identified as aspects, because both influences, i.e., crosscuts, several components.

However, in both the case of concurrency control and in the case of logging/recovery, there are parts of these activities that can be viewed as isolated activities. In the concurrency control management, a lock-manager is an isolated activity that was considered best implemented as a component. In the logging and recovery management, making periodic checkpoints, and recover a restarted system also is considered isolated activities, and was therefore implemented into the checkpointing and recovery component. These two components however, are optional, since they are not needed if the system does not use the corresponding aspect.

The resulting architecture of COMET, which is shown in figure 4, show all components and their interrelations in the architecture. Furthermore, the concurrency control, and logging/recovery aspects, as well as which components each aspect affects is shown. Since it is possible to have multiple transaction-types in a DBMS simultaneously it is possible to have multiple types of user interface components and transaction management components in the architecture.

By using this architecture, COMET will have a high degree of flexibility, and tailorability. All components in the system are exchangeable, so that a particular algorithm can be used in a particular activity, e.g., data indexing, transaction management, user interface management, memory management, etc. Furthermore, it is possible to extend the functionality of the DBMS by applying an aspect, such as concurrency control, or logging and recovery. This implies that COMET can consist only of the functionality needed by the application, and thereby minimizing the DBMS footprint.

6 The Components in COMET

6.1 User Interface Component

The User Interface Component (UIC) provides an interface towards the user, or application. All operations to the DBMS are received by the UIC. Examples of requests are; (i)Begin of transaction, which indicates that a new transaction is initiated, (ii) Commit/Rollback, which ends a transaction, either by completing it or aborting it, (iii) A DML/DDL statement, which can be written in the supported language, e.g., structured query language, SQL.

The main activities of the UIC are to:

- receive and parse incoming requests from users and applications.
- maintain the list of all active transactions, i.e., all initiated transactions that is not yet committed or roll backed, and
- parse incoming DML/DDL statements, and convert these into execution plans.

The user interface component will perform syntactical checking (parsing) of the incoming DML/DDL statements. The UIC does not know anything about the structure of the database, and therefore it cannot perform any semantical checks. This approach has both advantages and disadvantages. One disadvantage is that the execution plans cannot be optimized. Unoptimized execution of queries might take significantly longer time than optimized. An advantage is that the UIC might be executed on an external computer, thus performing the parsing on the external CPU. Consider the following example:

The COMET DBMS is installed in an electronic control unit (ECU) that resides in a vehicle. A real-time control application is continuously updating data elements within the database. Consider a service computer is connected to the vehicle during run-time. If the UIC, and thus the parser, is located on the service computer, the ECU will not have to handle the parsing.

The interfaces of the user interface component are presented in table 1

Interface	**Type**	**Description**
`IDbOperation`	Provided	Provides all operations reachable for the user
`IParse`	Provided	Provides the Parse Operations.
`ISchedule`	Required	Interface for scheduling new transactions.
`IExecute`	Required	Interface to execute transactions.

Tab. 1: The interfaces of the user interface component

6.2 Scheduling Management Component

The scheduling management Component (SMC) is responsible for scheduling the active transactions for execution, as well as maintaining a ready-queue containing all active unscheduled transactions. The SMC will schedule transactions based on the scheduling policy chosen. A scheduled transaction will be assigned to one instance of the transaction management component, e.g., one transaction manager from the transaction process pool, see section 6.3.

The interface of the scheduling management component is presented in table 2

Interface	**Type**	**Description**
`ISchedule`	Required	Interface for scheduling new transactions.

Tab. 2: The interfaces of the scheduling management component

6.3 Transaction Management Component

The transaction management component (TMC) executes execution-plans received from the UIC. It ensures transaction serialization by using the lock management component; see section 6.5. Which concurrency and lock policy to use is defined by the applied concurrency-control aspect, described more in detail in section 7.1. The tuples needed for the transaction are then located and fetched, using the index management component and the memory management component, see sections 6.4 and 6.6.

The flow of events in the TMC is easiest described using an example. Consider a TMC that executes relational transactions using a pessimistic concurrency control algorithm. For such a configuration, the flow of events could be as follows:

1. The execution-plan is received from the user interface component.
2. The execution of the transaction is started.
3. At some point of the execution, a relation must be fetched from the database. This is done by:
   - Issuing a request that the appropriate locks should be obtained. This request is sent to the lock management component.
   - Requiring the address of the tuples by calling the index management component.
   - Fetching the tuples from the database by calling the memory management component.
4. When all tuples needed by the transaction have been fetched, the data elements in the tuples can be manipulated according to the specification in the execution-plan.
5. After the execution of the transaction, the user interface component is notified.
6. The TMC will now wait until a command to commit (or abort) the transaction is received from the user interface.
7. If a commit-request is received, any updated tuples are permanently written to the database.
8. The locks obtained for the transaction are released.
9. The user interface component is notified that the transaction is completed.

In order to process multiple transactions concurrently, several instances of the transaction manager can exist in the database. Each instance of the TMC is then assigned to a dedicated task, i.e., a transaction process. These processes make up the transaction process pool.

The interfaces of the transaction management component are presented in table 3

Interface	**Type**	**Description**
`IExecute`	Provided	Interface to execute transactions.
`IIndex`	Required	Interface to access the tuple index.
`ILock`	Required[4]	Interface to access database locks.
`IMemory`	Required	Interface to access the tuples and dynamic memory.
`ICheckpoint`	Required[5]	Interface to perform a checkpoint or recovery operation.

Tab. 3: The interfaces of the transaction management component

[4] This interface is required only if concurrency control is used.
[5] This interface is required only if logging/recovery is used.

6.4 Index Management Component

The index management component (IMC) is responsible for the indexing of tuples. It matches an alphanumerical key, i.e., a database key, to its corresponding tuple identifier (TID). A TID contains information about where a tuple is located in memory.

The interfaces of the index management component are presented in table 4

Interface	**Type**	**Description**
`IIndex`	Provided	Interface to access the tuple index.
`IMemory`	Required	Interface to access the database.

Tab. 4: The interfaces of the index management component

6.5 Lock Management Component

The lock management component (LMC) administers the database locks. Database locks are used to prevent data access conflicts, such as read write conflicts in which a transaction has read a data element and a second immediately after assigns a new value to it. At this point in time the two, still active, transactions has different views of the current value of the data element.

How database locks are used is determined of the concurrency control algorithm applied to the system by the concurrency control aspect, described in section 7.1.

The interface of the lock management component is presented in table 5

Interface	**Type**	**Description**
`ILock`	Provided	Interface to access database locks

Tab. 5: The interface of the lock management component

6.6 Memory Management Component

The memory management component (MMC) administers the database, as well as the meta-data, such as database indexes and lock information.

The interface of the memory management component is presented in table 6.

Interface	**Type**	**Description**
`IMemory`	Provided	Interface to access the tuples and dynamic memory.

Tab. 6: The interface of the memory management component

6.7 Checkpointing and Recovery Component

The checkpointing and recovery component (CRC) is responsible for making checkpoints of the database and its meta-data. This component can be invoked periodically, or on request. Furthermore it is responsible for restoring the system after a system failure.

The interface of the checkpointing and recovery component is presented in table 7.

7 The Aspects in COMET

7.1 Concurrency-Control Aspect

The concurrency control aspect (CCA) ensures that concurrent transactions are serialized. This is enforced using (i) a serialization policy, such as optimistic or pessimistic concur-

Interface	Type	Description
`ICheckpoint`	Provided	Interface to perform a checkpoint or recovery operation..

Tab. 7: The interface of the memory management component

rency control, (ii) the lock management component, (iii) the scheduling management component, and (iv) the transaction management component.

The CCA consists of the following:

- Aspect-code that is weaved into the transaction management component. This code is responsible for obtaining and releasing the appropriate locks, in accordance with the concurrency control policy used.
- Aspect-code that is weaved into the lock management component. This code instructs the component what to do if a lock conflict occurs.
- Aspect-code that is weaved into the scheduling management component. This code decides what scheduling policy to use.

7.2 Logging and Recovery Aspect

The logging and recovery aspect (LRA) is responsible for ensuring that volatile data can be safely stored and recovered in case of a system breakdown. The LRA uses (i) a logging and recovery policy, such as roll-forward recovery, (ii) the transaction management component, and (iii) the memory management component.

The LRA consists of the following:

- Aspect-code that is weaved into the memory management component. This code instructs the component to issue a log entry in a log, when a tuple is created, erased or modified.
- Aspect-code that is weaved into the transaction management component. This code will issue log entries whenever a transaction is started, committed, or aborted.
- Aspect-code that is weaved into the checkpointing and recovery component to perform the correct type of checkpointing and recovery.

8 COMET BaseLine

The COMET BaseLine is the first implementation instance of COMET DBMS. It implements a database management system suited for the control system described in [2].

The COMET BaseLine provides two types of database transactions:

- **Soft real-time transactions**. Non-critical tasks can use this transaction type, which provide a flexible and generic way to manipulate the database. The interface supports a subset of the SQL commands, such as `SELECT`, `INSERT` and `UPDATE`. A set of SQL statements can be grouped together into a transaction.
- **Hard real-time transactions**. Time-critical tasks can use this transaction type, which provide a deterministic and efficient way of manipulating individual data elements. The interface implements the concept of Database Pointers discussed in [10].

Since the Volvo control-system is non-preemptive, there is no need for concurrency control in COMET BaseLine. Therefore the concurrency control aspect is not implemented in COMET BaseLine. However, the transaction manager component is prepared

SQL command	Description
`SELECT ... FROM`	Selects a subset of the tuples and attributes from a number of relations.
`CREATE TABLE`	Creates a new relation.
`DROP TABLE`	Removes a relation.
`INSERT ... INTO`	Inserts a new tuple into a relation.
`UPDATE ... WHERE`	Updates tuples in a relation.
`DELETE ... WHERE`	Deletes tuples from a relation

Tab. 8: The SQL-commands supported in COMET BaseLine

Relational API functions	Description
`BeginTransaction()`	Initializes a new transaction returns a transaction ID.
`Query()`	Takes a SQL query, parses it and prepares it for execution.
`CommitTransaction()`	Executes the SQL queries in the transaction, updates the database and closes the transaction.
`RollbackTransaction()`	Aborts the transaction.

Tab. 9: The relational API functions supported by COMET BaseLine

for the implementation of future aspects, e.g., the concurrency-control aspect, and the logging/recover aspect.

Furthermore, the logging/recovery management aspect has not yet been implemented in COMET BaseLine.

The current version of COMET BaseLine compiles down to an executable with the size of under 20kb.

9 Relational Processing

The relational transaction processing provides a generic and flexible way of accessing the database. It supports a subset of the commands in SQL; see table 8. These commands allow the user to retrieve any view of the data in the database, as well as add, delete and modify data elements. Finally the structure of the data can be changed, i.e., relations can be added and removed.

A set of queries can be bundled together to form a transaction. A transaction is considered an indivisible, e.g., atomic, operation, with a well-defined beginning and end. Table 9 shows the four API functions required to use a relational transaction in COMET BaseLine.

9.1 Database Pointer Processing

The database pointer processing provides a way to manipulate individual data elements within the database, in an efficient and predictable way. The database pointer processing engine is implemented in accordance with the concept presented in [10].

The database pointer processor supports the four interface operations presented in the paper, see table 10.

10 COMET BaseLine Architecture

10.1 Relational User Interface Component

The relational user interface component (RUIC) is responsible for administering the transactions as well as converting the SQL queries into execution-plans. The RUIC creates a

DB pointer API functions	Description
`bind()`	Binds the database pointer to the query specified.
`remove()`	Deletes the database pointer from the database pointer table.
`write()`	Writes a value to the data element pointed out by the database pointer.
`read()`	Reads the value from the data element pointed out by the database pointer.

Tab. 10: The database pointer API functions supported by COMET BaseLine

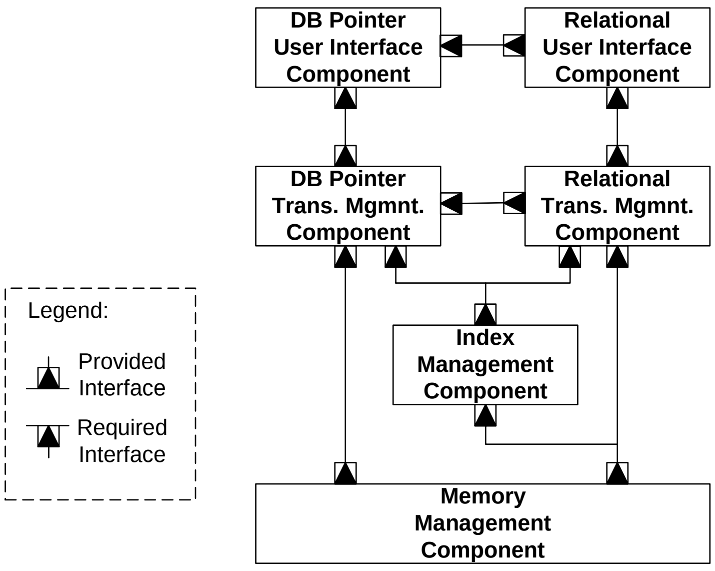
DB Pointer User Interface Component
Relational User Interface Component
DB Pointer Trans. Mgmnt. Component
Relational Trans. Mgmnt. Component
Index Management Component
Memory Management Component
Legend:
Provided Interface
Required Interface

Fig. 5: The architecture of COMET BaseLine

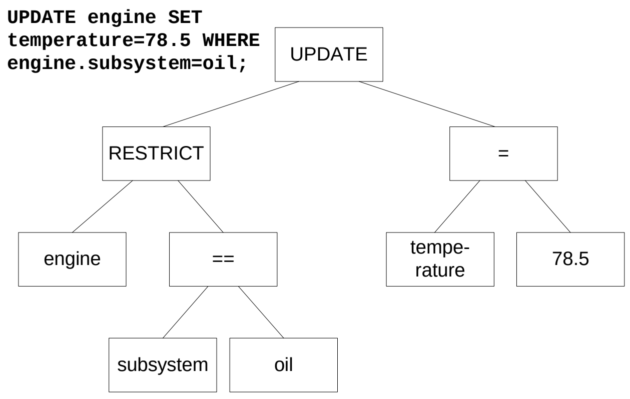

Fig. 6: Example of execution-plan

tree that represents the execution-plan, see example of an execution-plan in figure 6.

The execution-tree is stored until the `commitTransaction` operation is issued. At this stage, all execution-plans belonging to the committing transaction is sent to the relational transaction management component (RTMC) to be executed.

The nodes in the execution-tree can be of 5 different classes, namely:

1. **Relational Operations.** Originally C.F. Codd formulated 8 different relational operations, `select`, `project`, `join`, `product`, `union`, `intersect`, `difference`, and `divide`. In the subset of SQL supported in COMET BaseLine, only the `select`, `project`, `join`, and `product` operations are necessary. The relational operations create new relations by modifying existing, and are used to create new views of the data.

2. **Data Manipulating Operations.** These operations, namely `delete`, `insert`, and `update`, modify tuples in a relation.

3. **Data Definition Operations.** These operations are used to modify the structure of the database, i.e., `create table` and `drop table`, in which the former creates a new relation, while the latter deletes one.

4. **Operators.** Typical operators are, conditions and assignments.

5. **Operands.** Operand can be values, tuples, or relations.

10.2 Relational Transaction Management Component

The relational transaction management component (RTMC) executes the execution-plans created by the RUIC.

A central part of the RTMC is the buffer manager. The buffer manager is used to store intermediate, i.e., non-materialized, relations used during the execution of a query. The buffer manager, which handles the relational structure, the relational meta-data and the actual data, is of such a substantial complexity that it is implemented as an internal component inside the RTMC.

To illustrate how the execution of a query is performed in COMET BaseLine, an example, using the query and execution plan in figure 6, is given.

1. The execution begins using a bottom-up approach.

2. The actual relation `engine` is loaded. This is performed in the following steps:

   a) The meta-data for the relation is fetched from the index management component (IMC).
   b) From the meta-data, the size of the relation, i.e., the length of the tuples and the number of tuples, is determined.
   c) Using this information, a buffer is allocated.
   d) The location of the first tuple is fetched using the `search()` operation in the IMC.
   e) The tuple is loaded from the database using the memory management component (MMC), and is stored in the buffer.
   f) The location of the next tuple is fetched using the `searchGT()` operation in the IMC
   g) The steps e) and f) is repeated until the whole relation is loaded.

3. The relation is now restricted, i.e., all tuples not fulfilling the `WHERE subsystem=oil` is removed from the buffer.

4. The remaining tuples in the buffer is now updated according to the assignment `SET temperature=78.5`.

5. The last step is to write the updated tuples back to the database.

10.3 Database Pointer User Interface Component

The database pointer user interface component (DPUIC) provides the database pointer interface. The DPUIC uses the RUIC to create the execution-plan needed for the `bind` operation.

10.4 Database Pointer Transaction Management Component

The database pointer transaction management component (DPTMC) executes the incoming database pointer operations. It administers the database pointer table, in which information of all database pointers are stored.

The DPTMC uses the RTMC to execute the execution-plans received from the DPUIC. The difference, compared to relational query processing, is that the query does not return the value of the data element, but the address to the tuple that the data element resides in, and an offset.

Since the COMET BaseLine is a non-preemptive DBMS, the database pointer table does not include locking information. However it is implemented in such a way that this can easily be extended, or aspectualized.

The DPTMC can also receive requests from the RTMC to update the database pointer table. This is typically done, when tuples are moved or deleted from the relational query processor.

10.5 Index Management Component

The index management component (IMC) in COMET BaseLine implements the t-tree indexing algorithm. The information in the nodes in the tree has been extended to contain the database pointer flag, described [10]

The architecture of the IMC is shown in figure 7. The IMC consists of two layers of indexes, an upper layer, that indexes the different database files, and the lower layer that indexes the tuples in each database file. This approach was suggested in [11]. The architecture allows the index manager to be configurable to a high degree.

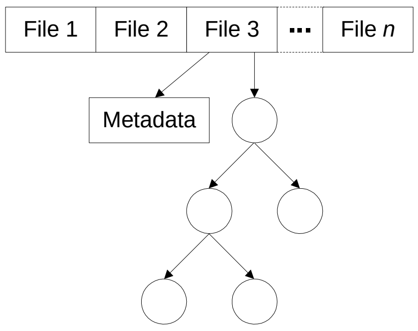

Fig. 7: The architecture of the interface management component

For systems where the number of relations is fixed, or bounded, the upper layer index can be implemented as a array of files, while for more flexible system a dynamic tree index might be suited. The lower layer index can implement any index structure, e.g., tree or hash index. Different index structures can be used simultaneously for different relations.

For the upper layer index, COMET BaseLine uses a static array with a pre-defined length. For the lower layer, the t-tree algorithm is used.

COMET BaseLine uses one file for each relation. Therefore a search for a tuple in a relation starts with locating the relation in the upper layer index. When the relation is located, the lower layer index is used to locate the tuple.

10.6 Memory Management Component

The memory manager component (MMC) manages the stored tuples, and acts as a dynamic memory manager. It delivers a hardware independent memory management interface to the database, even for systems that does not support dynamic memory allocations.

The MMC provides two different types of dynamic memory:

- Tuple ID's, that is a data structure that contains address information to one tuple in the database. The MMC provides four operations on tuple ID´s, namely `allocate`, `delete`, `write`, and `read`.
- Dynamic memory used for buffering and temporary data structures used during the execution of a transaction. The MMC provides two operations for dynamic memory, `malloc` and `free`.

11 Conclusions and Future Work

In this report, we have described the architecture of the COMET database management system, which is configurable to suit different kinds of embedded control-systems. Furthermore, we have described the first instance of COMET, called COMET BaseLine, which is a version of COMET suited for a vehicle control system developed at Volvo Construction Equipment Components in Eskilstuna, Sweden.

COMET has been developed using a design method called ACCORD [7], which combines component-based software development and aspect-oriented software development. COMET BaseLine is implemented using a component-model called RTCOM [8], which also is a part of ACCORD.

Since the application for which COMET BaseLine is developed is non-preemptive, there is no need for transaction synchronization and conflict resolution. This implies that there is no need for the transaction scheduling component, the locking management, and the concurrency control aspect.

The next step is to evaluate the performance of COMET BaseLine, by integrating it into the Volvo Construction Equipment Component control-system. Implementation of the scheduling management component, the locking management component, the concurrency control aspect and backup/recovery aspect is also planned.

References

[1] Lennart Casparsson, Antal Rajnak, Ken Tindell, and Peter Malmberg. Volcano - a revolution in on-board communications. Technical report, Volvo Technology Report, 1998.

[2] Dag Nyström, Aleksandra Tešanović, Christer Norström, Jörgen Hansson, and Nils-Erik Bånkestad. Data Management Issues in Vehicle Control Systems: a Case Study. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, 2002.

[3] 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.

[4] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, 2002.

[5] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, February 2002. Australian Computer Society. AspectC++ can be downloaded from: http://www.aspectc.org.

[6] The AspectJ programming guide. Xerox Corporation, September 2002. Available at: http://aspectj.org/doc/dist/progguide/index.html.

[7] Aleksandra Tešanović, Dag Nyström, Jörgen Hansson, and Christer Norström. Towards Aspectual Component-Based Development of Real-Time Systems. In *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications*, 2003.

[8] Aleksandra Tešanović, Dag Nyström, Jörgen Hansson, and Christer Norström. COMET: a COMponent-based Embedded real-Time database. Technical report, Dept. of Computer Science, Linköping University, and Dept. of Computer Engineering, Mälardalen University, 2002.

[9] Aleksandra Tešanović, Dag Nyström, Jörgen Hansson, and Christer Norström. Integrating Symbolic Worst-Caase Execution Time Analysis with Aspect-Oriented System Development. In *Proceedings of OOPSLA 2002 Workshop on Tools for Aspect-Oriented Software Development*, November 2002.

[10] Dag Nyström, Aleksandra Tešanović, Christer Norström, and Jörgen Hansson. Database Pointers: a Predictable Way of Manipulating Hot Data in Hard Real-Time Systems. In *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications*, 2003.

[11] Tony Bertilsson. Implementation Analysis for Databases in Embedded Systems. Master's thesis, Mälardalen University, 2002.