OBJECTIVE: A Benchmark for Object-Oriented Active Database Systems

Uğur Çetintemel[1], Jürgen Zimmermann[2], Özgür Ulusoy[3], and Alejandro Buchmann[2]

[1] Department of Computer Science, University of Maryland, College Park, Maryland, U.S.A.
[2] Department of Computer Science, University of Darmstadt, Darmstadt, Germany
[3] Department of Computer Engr. and Information Sci., Bilkent University, Ankara, Turkey

Abstract

Although much work in the area of Active Database Management Systems (ADBMSs) has been done, it is not yet clear how the performance of an active DBMS can be evaluated systematically. In this paper, we describe the OBJECTIVE Benchmark for object-oriented ADBMSs, and present experimental results from its implementation in an active database system prototype. OBJECTIVE can be used to identify performance bottlenecks and active functionalities of an ADBMS, and to compare the performance of multiple ADBMSs.

1 Introduction

An Active Database Management System (ADBMS) detects certain situations and performs corresponding user defined actions in the form of Event-Condition-Action (ECA) rules (Dayal et al., 1988a). ADBMSs have received great attention lately, and several prototypes of object-oriented ADBMSs are already available (e.g., ACOOD (Berndtsson, 1991), NAOS (Collet et al, 1994), Ode (Agrawal and Gehani, 1989), REACH (Buchmann et al., 1995), SAMOS (Gatziu et al., 1994), SENTINEL (Chakravarthy et al., 1994)). We are currently in a position to evaluate the performance of ADBMSs by concentrating on

- the performance requirements of different architectural approaches; i.e., integrated versus layered,
- different techniques used for standard tasks of an ADBMS; i.e., rule maintenance, event detection, and
- a variety of functionalities provided by an ADBMS; e.g., garbage collection and parameter passing.

Benchmarking is a very important process in the sense that database users base their purchasing decisions partially relying on benchmark results, and database designers measure the performance of their systems by using an appropriate benchmark. There has been much work in the area of database benchmarking; e.g., the Wisconsin Benchmark (Boral and DeWitt, 1984), the OO1 Benchmark (Catell and Skeen, 1992), and the OO7 Benchmark (Carey et al., 1993). However, there have been only a few attempts to evaluate the performance of ADBMSs, the most important of which are the BEAST Benchmark (Geppert et al., 1995), and the ACT-1 Benchmark (Zimmermann et al., 1995).

In this paper, we describe the OBJECTIVE[1] Benchmark which is a simple but comprehensive test of active functionalities provided by an *object-oriented* ADBMS, and give performance results of its implementation in an ADBMS prototype. OBJECTIVE can be used to identify performance bottlenecks and active functionalities of an ADBMS, and compare the performance of multiple ADBMSs. The philosophy of OBJECTIVE is to isolate components providing active functionalities, and concentrate only on the performance of these components while attempting to minimize the effects of other factors (e.g., underlying platform). OBJECTIVE operates on a very simple database structure consisting of completely synthetic classes, events, and rules. Although the design is very simple (for ease of reproducibility and portability), we believe that this simplicity does not contribute negatively to the benchmark in any manner.

The OBJECTIVE Benchmark addresses the following issues with respect to object-oriented ADBMS performance and functionality:

- method wrapping penalty,
- detection of primitive and composite events,
- rule firing,
- event-parameter passing,
- treatment of semi-composed events, and
- rule administration tasks.

The OBJECTIVE Benchmark comprises a number of operations that evaluate the issues stated above, and those operations were first run on REACH (Buchmann et al., 1995). REACH is a full-fledged operational object-oriented ADBMS which is tightly integrated in Texas Instruments' Open OODB (Wells et al., 1992). The results reported in this paper reveal that REACH combines advanced features of current ADBMS proposals from the functionality point of view. As for its performance, a single bottleneck operation is identified.

The remainder of the paper is organized as follows. Section 2 discusses the main features of ADBMSs and Section 3 discusses previous related work. The OBJECTIVE Benchmark is introduced along with performance results of its implementation in REACH in Section 4. Finally, Section 5 concludes and gives directions for future work.

2 Active Database Management Systems

ADBMSs extend passive DBMSs with the ability to specify and implement reactive behavior which is typically specified in terms of ECA rules. The general form of an ECA rule is: on *event* if *condition* do *action*. The semantics of such a rule is that *when* the *event* occurs, the *condition* is checked, and *if* it is satisfied *then* the *action* is executed. Therefore, an ADBMS has to monitor events (of

[1] OBJECT-oriented actIVE database systems benchmark

interest) and detect their occurrences. After an event is detected, it is *signaled*. This signaling is a notification that an interesting event has occurred, and rule execution should take place. ECA rules require, at least, the operations *insert*, *delete*, and *fire*. These operations are used to insert a new rule into the database, delete an existing rule from the database, and trigger a rule, respectively. For some applications it may be useful to *disable* rules temporarily, which can afterwards be *enabled* when necessary (Dayal et al., 1988b).

This section discusses main issues in ADBMSs to an extent which is necessary for the comprehension of the rest of the paper.

2.1 Events

ECA rules are triggered on the occurrence of particular events. An event can be either *primitive* or *composite*. Primitive events are atomic events which can be associated with a point in time. The most commonly referred primitive event types are (Buchmann et al., 1995; Gatziu and Dittrich, 1993; Chakravarthy and Mishra, 1993):

- *method events*
  A method invocation can be defined as an event of interest. In such a case, an event occurs when its corresponding method is executed. Since a method execution corresponds to an interval rather than a point in time, usage of time modifiers like BEFORE or AFTER is mandatory. The semantics of BEFORE and AFTER modifiers, respectively, is that the method event is to be raised just before the invocation of the method, and immediately after the execution of the method.

- *state transition events*
  A change in the state of the object space can be an event; e.g., modification of an object attribute. It is then necessary to define operators to access old and new values of relevant entities.

- *temporal events*
  Basically, two types of temporal events exist; *absolute* and *relative*. Absolute temporal events are defined by giving a particular point in time (e.g., 01.10.1996, 11:23), whereas relative temporal events are defined relative to other events (e.g., 10 minutes after commit of a particular transaction). The latter type can also include events which occur periodically (e.g., every day at 17:30).

- *transaction events*
  Transaction events correspond to standard transaction operations like *begin of transaction* (BOT), *end of transaction* (EOT), *abort of transaction* (ABORT), and *commit of transaction* (COMMIT).

- *abstract events*
  Abstract events are user-defined events whose occurrences are directly signalled. Therefore, the underlying system does not need to monitor abstract events; i.e., they are explicitly raised by the user and associated with a point in time.

Different techniques are used for the detection of method events. A straightforward approach is to modify the body of the method for which an event is to be defined with an explicit raise of an event

(Gatziu and Dittrich, 1993). Another technique, *method wrapping*, is to replace the original method with a *method wrapper* that contains an explicit event raise operation and a call to the original method (Buchmann et al., 1995). When a method for which an event is defined is called, actually its wrapper is invoked (i.e, the wrapper gets the name of the original method, and original method is renamed). The wrapper then raises the event and calls the original method (or vice versa depending on the time modifier used in the event).

Unlike primitive events which are atomic, composite events are defined as a combination of primitive (and possibly other composite) events. The meaningful ways to build composite events from its constituent events are usually specified through an *event algebra* that defines certain *event constructors*. Some useful event constructors are (Dayal et al., 1988b; Gatziu et al., 1994):

- The *disjunction* of two events, event1 and event2, is raised when either of event1 or event2 occurs.
- The *conjunction* of two events, event1 and event2, is raised when both event1 and event2 occur.
- The *sequence* of two events, event1 and event2, is raised when event1 and event2 occur in that order.
- The *closure* of an event, event1, is raised exactly once regardless of the number of times event1 occurs (provided that event1 occurs at least once).
- The *negation* of an event, event1, is raised if event1 does not occur in a given time interval.
- The *history* of an event, event1, is raised if event1 occurs a given number of times.

For the last three event constructors, it is appropriate to define time intervals in which composition of events should take place. The definition of a time interval is mandatory for negation, and optional for history and closure.

Composite events can further be grouped into *aggregating* composite events and *non-aggregating* composite events (Zimmermann and Buchmann, 1995). The former group contains composite events that are constructed with the operators sequence, disjunction, and conjunction, whereas the latter group comprises composite events constructed with history, negation, and closure.

Several different approaches are used for composite event detection including syntax graphs (Deutsch, 1994; Chakravarthy et al., 1993), Petri nets (Gatziu and Dittrich, 1994), finite state automata (Gehani et al., 1992), and arrays (Eriksson, 1993).

2.2 Conditions

The condition part of a rule is usually a boolean expression, a predicate, or a set of queries, and it is satisfied if the expression evaluates to true, or all the queries return non-empty results, respectively. In addition to the current state of the database, the condition may access the state of the database at the time of event occurrence by the use of event parameters.

2.3 Actions

The action part of a rule is executed when the condition is satisfied. In general, an action can be database operations, transaction commands (e.g., abort transaction), or arbitrary executable routines. Therefore, during the execution of an action some events may also occur. This may lead to the triggering of other rules which is called *cascaded rule triggering*. The action may access, besides the current database state, the database state at the time of event occurrence and the time of condition evaluation which can be accomplished by parameter passing.

2.4 Execution Model

An *execution model* specifies the semantics of rule execution in a transaction framework. A transaction which triggers rules is called a *triggering transaction*, and the (sub-)transaction which executes the triggered rule is called the *triggered (sub-)transaction*. An important issue which is determined by an execution model is the coupling between the triggered transaction and the triggering transaction. Additionally, an execution model also describes concurrency control and recovery mechanisms used to achieve a correct and reliable rule execution. These two issues are discussed in more detail in the rest of this subsection.

Coupling modes determine the execution of rules with respect to the transaction which triggers them. The Event-Condition (EC) and Condition-Action (CA) coupling modes, respectively, determine when the rule's condition is evaluated with respect to the triggering event, and when the rule's action is executed with respect to the condition evaluation. Three basic coupling modes are introduced in (Dayal, 1988): *immediate*, *deferred*, and *decoupled*.

For EC coupling, the intended meaning of each mode is:

- In *immediate* EC coupling mode, the condition is evaluated in the triggering transaction, immediately after the detection of the triggering event.
- In *deferred* EC coupling mode, the condition is evaluated after the triggering transaction executes but before it commits.
- In *detached* EC coupling mode, the condition is evaluated in a separate transaction which is independent from the triggering transaction.

For CA coupling, the semantics of each mode can be given analogously.

If several triggered rules have to be executed at the same point in time, they form a *conflict set* (Hanson and Widom, 1992). In this case, some sort of *conflict resolution* (e.g. priorities) must be employed to control their execution order. The ability to do such a resolution is especially desirable if we want to impose a particular serial order of execution.

Since condition and action parts of a rule may act on database objects, the execution of rules must be done in a transaction framework. The nested transaction model (Moss, 1985) is the most prevalent approach for rule execution in ADBMSs, primarily due to the fact that it captures the semantics of (cascaded) rule triggering well. In this model, the triggered rules are either executed as subtransactions of the triggering transaction, in case of immediate and deferred coupling modes, or as an independent transaction in case of detached coupling mode.

3 Related Work

Although much work in the area of ADBMSs has been done, it is not yet clear how the performance of an ADBMS can be evaluated systematically. In fact there have been very few attempts including (Geppert et al., 1995; Zimmermann et al., 1995; Brant and Miranker, 1993; Kersten, 1995). In this section, we discuss these efforts in some detail.

3.1 The BEAST Benchmark

The BEAST is the first benchmark proposed for testing the performance of object-oriented ADBMSs (Geppert et al., 1995). It was presented as a designer's benchmark; i.e., the designers of an ADBMS can use it to determine performance bottlenecks of their systems. It uses the database and schema of the OO7 Benchmark (Carey et al., 1993). The BEAST Benchmark runs a series of tests to determine the functionality of each component. It consists of:

- *Tests for event detection* concentrate on the time to detect particular events. A set of primitive and composite events are tested. Tests for primitive event detection consist of the detection of value modification, the detection of method invocation, the detection of transaction events, and the detection of a set of primitive events. The BEAST tests for composite event detection comprise the detection of a sequence of primitive events, the detection of a non-occurrence of an event within a transaction, the detection of a repeated occurrence of a primitive event, the detection of a sequence of composite events, the detection of a conjunction of method events sent to the same object, and the detection of a conjunction of events belonging to the same transaction.

- *Tests for rule management* evaluate the rule management component of an ADBMS by measuring the retrieval time of rules.

- *Tests for rule execution*
  The tests for rule execution consider both the execution of single and multiple rules. For the execution of single rules, a rule is executed with different coupling modes. In the case of multiple rule execution, the tests concentrate on the overhead of enforcing an ordering on the triggered rules, optimization of condition evaluation and raw rule execution power of the underlying system.

In all these tests response time was accepted as the sole performance metric. In the experiments, the number of defined events (primitive and composite), and the number of rules were used as benchmark parameters, and a set of quantitative results were obtained for each particular setting of these parameters. To date, BEAST has been run on four object-oriented ADBMS prototypes, and the performance results are presented in (Geppert et al., 1996).

3.2 The ACT-1 Benchmark

The ACT-1 Benchmark (Zimmermann et al., 1995) concentrates on the minimal features of object-oriented ADBMSs. Four basic issues are addressed in this benchmark:

1. *Method wrapping penalty* measures the useless overhead of method wrapping for the detection of method events.

2. *Rule firing cost* measures the cost of raising an event and firing the corresponding rule.
3. *Minimal event composition cost* aims to measure the cost of a simple event composition (the sequence of two events).
4. *Sequential rule firing cost* concentrates on the overhead of serialization of a set of rules that have to be executed at the same time (two rules that are triggered by the same event at the same coupling mode).

ACT-1 uses a simple database with objects and rules modeling the operation of a power plant. Four operations, WRAPPING PENALTY, FIRING COST, BUILD UP, and SEQ EXEC, were implemented in REACH and some preliminary results based on response times of these operations were presented.

3.3 Other ADBMS Benchmarking Related Work

There are several other performance evaluation studies on ADBMSs. Actually, these are not devoted performance evaluation works; rather, they present a rule (sub)system and then evaluate its performance. For instance (Brant and Miranker, 1993) mainly addresses the problem of handling large rule sets. It argues that the techniques used in current active database prototypes are not appropriate for handling large rulebases. It proposes a novel indexing technique for rule activation and gives performance results of DATEX, a database rule system, which uses this particular technique. Storage size and number of disk accesses are used as the cost metrics in this evaluation.

(Kersten, 1995) presents another performance study on active functionality in DBMSs. It gives a performance evaluation of the rule system of MONET—a parallel DBMS kernel aimed to be used as a database back-end—by using a simple benchmark. This simple *core* benchmark is designed mainly for testing the implementation of MONET, and it consists of three basic experiments. The *countdown* experiment tries to measure the cost of handling a single event and the subsequent firing of a single rule (i.e., an abstract event is signalled and a rule is fired by this event. This fired rule notifies the same event which further leads to the triggering of the same rule. This is repeated a predetermined number of times). The *dominoes* experiment is aimed to determine the cost of isolating a firable rule instance. The *pyramid* experiment has the purpose of investigating the performance of the system under high active workloads.

4 The OBJECTIVE Benchmark

After a brief discussion of some restrictions of previous benchmarks for ADBMSs in Section 4.1, we introduce the operations of the OBJECTIVE Benchmark along with a requirements analysis in Section 4.2. We then describe the synthetic database of OBJECTIVE in Section 4.3. In Section 4.4, we describe the implementation of the OBJECTIVE operations in detail while presenting experimental results of their implementation in REACH.

4.1 Why Another Benchmark for Object-Oriented ADBMSs?

The ACT-1 Benchmark specifies a small but important set of operations. However, as discussed by its designers, it needs to be extended with new operations to evaluate ADBMSs properly.

The BEAST Benchmark is also a very good initial step towards evaluating ADBMS performance and functionality. However, there are some drawbacks associated with the BEAST benchmark. The primary limitation of BEAST is that it ignores some issues that are very important from the functionality point of view in an ADBMS, e.g., event-parameter passing and rule administration. Typically, a system with little functionality can be implemented more efficiently than a system with more functionality. As an example, consider the (useless) overhead of method wrapping. At one extreme, there are systems that hand-wrap only those methods on which a rule is defined, and at the other extreme there are systems that do automatic wrapping of all the methods. The latter systems allow the definition of new rules without requiring the recompilation of classes, but pay for the wrapping when a method that is not an event type for any rule is invoked. Likewise, a system that allows event parameters to be passed to condition and action parts of rules will be much more flexible than the one which does not support such a functionality, but at the same time it will face an overhead in event composition and rule execution in non-immediate coupling modes. Therefore, a fair benchmark cannot ignore features that provide flexibility, but potentially affect the performance of the system.

BEAST uses the fairly complex database and schema of 007. This is definitely an advantage if one wants to evaluate the performance of both the passive and active components of an ADBMS (i.e., 007 and BEAST can be run on the same database). However, if one is interested only in the active functionality of an ADBMS, usage of a complex database like that of 007 is not justified if the operations of the benchmark does not make use of that complexity (which is true for the BEAST Benchmark). In addition, BEAST has a subtle limitation with respect to its implementation; i.e., BEAST does not distinguish between cold and hot execution times (see section 4.4) for its operations. The proper usage and interpretation of hot and cold times might be particularly useful in isolating certain tasks when access to database system internals is not allowed to do that explicitly.

In the OBJECTIVE Benchmark, we aim to overcome the limitations of the ACT-1 Benchmark and the BEAST Benchmark by:

- providing a comprehensive set of operations that address some critical functionality of ADBMSs besides performance in order not to skew results in favor of systems with less functionality,
- using simple to implement operations and a simple database, and,
- using both hot and cold execution times for better understanding of the performance of components providing active functionalities.

As discussed in the sequel, the OBJECTIVE Benchmark is simpler than BEAST, and more comprehensive than both BEAST and ACT-1.

4.2 The OBJECTIVE Operations

The aim of the OBJECTIVE Benchmark is to identify the bottlenecks and functionalities of an object-oriented ADBMS, and to create a level-playing field for comparison of multiple object-oriented ADBMSs. The OBJECTIVE Benchmark addresses the following issues (Zimmermann and Buchmann, 1995) by the operations which are described briefly in Table 1:

1. *Method wrapping penalty*
   In an object-oriented database system where method wrapping is used for method event detection, there is a *useless* overhead which is generated when a method which does not generate

TEST	DESCRIPTION
MW1	Method wrapping *penalty*
PED1	Detection of a *method invocation* event
PED2	Detection of a *BOT* event
PED3	Detection of a *COMMIT* event
CED1	Detection of a *sequence* of primitive events
CED2	Detection of a *conjunction* of primitive events
CED3	Detection of a *negation* of a primitive event
CED4	Detection of a *history* of a primitive event
CED5	Detection of a *closure* of a primitive event
RF1	*Retrieval* of a rule
RF2	Rule firing in *deferred* coupling mode
RF3	Rule firing in *decoupled* coupling mode
RF4	Rule *execution*
RF5	*Conflict resolution* of triggered rules
RF6	*Cascaded* rule triggering
EPP1	The passing of event parameters in *immediate* coupling mode
EPP2	The passing of event parameters in *deferred* coupling mode
EPP3	The passing of event parameters in *decoupled* coupling mode
GC1	The *garbage collection* of semi-composed events
RA1	*Creating* a rule
RA2	*Deleting* a rule
RA3	*Enabling* a rule
RA4	*Disabling* a rule
RA5	*Modifying* a rule

Table 1: The OBJECTIVE operations

any event or which generates an event that does not contribute to the triggering of any rule is invoked (i.e., such an event is neither a primitive event for a rule, nor a part of a composite event for a rule). Ideally, the introduction of active capabilities should not deteriorate the performance when they are not in effect. In other words, ADBMS users should not pay for active functionality when they do not use it. Therefore, an ADBMS must keep such a (useless) overhead minimal.

2. *Event detection*
   An ADBMS should support primitive and composite events and response times for event detection, both primitive and composite, are crucial for the performance of an ADBMS. The primitive event types should minimally include method events and transaction events. For composite events, at least, the detection time for an aggregating event and a non-aggregating event should be measured.

3. *Rule firing*
   Rules typically reside in secondary storage and have to be fetched into main memory for execution. Therefore, efficient retrieval of rules whose events are signalled is indispensable for an

ADBMS. As well as for capturing the semantics of some applications, (non-immediate) coupling modes are introduced primarily for increased performance with respect to execution of rules. If different coupling modes cannot be supported effectively, then there will hardly be any point in keeping them. Therefore, efficient firing of rules in different coupling modes is a crucial issue. Different approaches can be taken in the storage of condition/action parts of a rule (e.g., compiled code). Regardless of their internal representation, efficient access and execution of these parts is mandatory. Another pragmatic issue is the conflict resolution of a set of rules that are to be executed at the same point in execution flow. In addition, the ability to treat application/program execution and rule execution uniformly is also significant. Extra overhead should not be introduced for detection of events and firing of rules during rule execution.

4. *The handling of event parameters*
   For some applications, e.g., consistency-constraint checking and rule-based access control, event parameters must be passed to the condition-action part of the rule. Otherwise, expressing conditions and actions with proper bindings is not possible. This requires the usage of some intermediate storage in case the rule is executed in either deferred or detached coupling mode. In immediate coupling mode it may be sufficient to pass a pointer to the parameters instead of passing the parameters themselves. However, this approach may not be applicable in deferred and detached coupling modes, because the parameters to be passed might be transient objects rather than persistent ones. The way event parameters are handled, thus, has a great impact on the performance of the system.

5. *Garbage collection of semi-composed events*
   The problem of garbage collection exists for some composite events that are not fully composed, and whose extents have expired (Buchmann et al., 1995). If no garbage collection is done for such semi-composed events, the database size will increase unnecessarily which will lead to a further increase in response time. So, an efficient mechanism for garbage collection of semi-composed events must be employed from the performance point of view.

6. *Rule administration*
   An ADBMS should be able to create, destroy, enable and disable rules on-line. The ability to maintain rules dynamically is very important because of well-known reasons of availability and extensibility. Although the execution speeds of these tasks are not of great importance (as they are executed rather seldomly), a comprehensive benchmark should take them into account.

4.3 Description of the OBJECTIVE Database

Generation of a synthetic database is an important issue in all benchmarks for database systems (Gray, 1991). In a benchmark for active database systems, the most interesting part of database specification is the specification of events and rules, because tests of the benchmark will typically concentrate more on rules and events than particular objects in the database.

The database for the OBJECTIVE Benchmark consists of completely synthetic object classes with the same methods and attributes (see Figure 1 for a generic class definition[2]), and it has a very simple

[2]We use a notation for our class definitions and test routines which is the de facto standard for object-oriented languages, namely the notation of C++ programming language.

```
  class Name{
        int attribute;
        double data;
public:
        void doNothing()                          {; }
        void setAttribute(int i)                  {attribute = i;}
        int getAttribute()                        {return attribute;}
        void setData(double d1, double d2)        {data = d1 - d2;}
        void setMinData()                         {data = 0.0;}};
```

Figure 1: A class example

schema. The rationale for this decision is twofold: First, a benchmark should be easily reproducible and portable, and second OBJECTIVE is *not* designed to be a domain-specific benchmark; i.e., the aim of OBJECTIVE is to test important aspects of system performance and functionality, not to model a particular application. Thus, we do not want to add extra complexity which will not contribute to the benchmark in any manner, but which will make the implementation more difficult.

Several events and rules are defined (Figure 2 and Figure 3) to be used in benchmark operations. The rules are defined in the rule programming language REAL[3] (REAch rule Language) (Zimmermann et al., 1996). The events, however, are defined in a hypothetical language based on the event definition notation of REAL[4]. These events and rules are discussed in detail in Section 4.4 where we describe the implementation of the benchmark operations. The naming convention used for objects, events, and rules are based the name of the relevant operation; e.g., the objects, events, and rules of name EPP1 are the ones that will be utilized in operation EPP1. Apparently, these events and rules can easily be reproduced in any ADBMS employing ECA rules.

In addition to these events, rules, and classes which are used in the benchmark tests, we also utilize dummy event, rule, and class types. By changing the number of instances of these dummy types, we can run our operations for different database configurations, and see their effects on system performance.

We define dummy classes with the same methods and attributes, and the instances of these dummy classes form the (dummy) database objects. The methods of these dummy classes are used to generate before/after (dummy) method events. The event constructors *sequence* and *history* are used to generate non-aggregating and aggregating (dummy) composite event types, respectively. The number of component events to form a composite event is selected at random[5] from range {2, 3, ..., 10}. Likewise, the component event types are selected randomly from the already generated method event types. The dummy rules choose their event types at random from the existing dummy primitive and composite event types. Both the condition and action parts of dummy rules are defined as empty.

We include four parameters for the OBJECTIVE Benchmark; *NumEvents*, *FracCompEvents*, *Num-*

[3] Rules in REAL consist of parts for defining a rule's event, condition and action along with EC and CA coupling modes and priorities. The default value for a coupling mode is *imm*(ediate) and the default values for method event modifiers and priorities (priority range is {1, 2, ...,10}) are *after* and *5*, respectively. In addition, there is a *decl*(laration) section in which variables are specified in a C++ manner.

[4] REAL does not consider the definition of stand-alone event types.

[5] Uniform distribution is used in all random selections.

```
event PED1 { PED1::doNothing(); };
event PED2 { BOT('PED2'); };
event PED3 { COMMIT('PED2'); };
event CED1 { ABSTRACT(CED1_1) then ABSTRACT(CED1_2); };
event CED2 { ABSTRACT(CED2_1) and ABSTRACT(CED2_2); };
event CED3 { not ABSTRACT(CED3_2) in (ABSTRACT(CED3_1) , ABSTRACT(CED3_3)); };
event CED4 { 1 times ABSTRACT(CED4_2) in (ABSTRACT(CED4_1) , ABSTRACT(CED4_3)); };
event CED5 { all ABSTRACT(CED5_2) in (ABSTRACT(CED5_1) , ABSTRACT(CED5_3)); };
```

Figure 2: The events related to event detection operations

Rules, and *NumObjects*, which define the number of (dummy) events, the fraction of composite events, the number of (dummy) rules, and the number of (dummy) data objects, respectively. The database configurations based on these parameters are summarized in Table 2.

4.4 Implementation and Results

In this section, we discuss the implementation of the benchmark operations which are described briefly in Section 4.2, and present the results of their application to REACH.

In all the operations described in this section, we assume that access to the internals of an ADBMS is not possible. This assumption is made due to two primary reasons: First, this is generally the case in reality, and second we want our benchmark to be a general one so that it can be applied to different ADBMSs through their external interfaces. Although this assumption makes accurate time measurement impossible for certain tests, we can circumvent it to a certain extent by keeping all the other non-interesting phases as small as possible by using appropriate events and rules. Actually, we assume that we can run our tests by just using the application programming interface of an ADBMS.

We make use of two time measures for the OBJECTIVE operations (whenever appropriate); *cold* and *hot* times representing the elapsed times when a measurement is done beginning with empty buffer, and beginning with completely initialized buffer, respectively. However, we do not present both cold and hot time results for all operations. Instead, we prefer to present the more meaningful and informative time measure for a given operation according to the focus of that operation. As a case in point, it is more meaningful to concentrate on the cold times for an operation concerned with rule retrieval, while one should emphasize the hot time results for conflict resolution of triggered rules.

We consider the *CPU time* used by the process running an operation instead of wall-clock time, because we ran the benchmark in a normal operating environment (i.e., not in an isolated machine), and we do not want to include the effects of certain operating system tasks in our results. Another important point to note is that we always use *transient* objects rather than persistent ones in order to exclude any database overhead[6].

The following general order of execution is used for the implementation of each operation:

1. clear the system buffer,

2. open the database,

[6] Only exceptions are the operations that require the passing of objects as event parameters in non-immediate modes. In such a case, it only makes sense to pass persistent objects, not transient ones.

```
rule RF1{
  decl    ;
  event   ABSTRACT(RF1);
  cond    FALSE;
  action  ;
};

rule RF2{
  decl    ;
  event   ABSTRACT(RF2);
  cond    def  FALSE;
  action  ;
};

rule RF3{
  decl    ;
  event   ABSTRACT(RF3);
  cond    dep  FALSE;
  action  ;
};

rule RF4{
  decl    ;
  event   ABSTRACT(RF4);
  cond    TRUE;
  action  ;
};

rule RF5_1{
  decl    ;
  event   ABSTRACT(RF5);
  cond    FALSE;
  action  ;
  prio    1;
};

rule RF5_2{
  decl    ;
  event   ABSTRACT(RF5);
  cond    FALSE;
  action  ;
  prio    2;
};

rule RF6{
  decl    RF6 *obj;
          int i;
  event   obj->setAttribute(i)
  cond    obj->getAttribute() > 0;
  action  obj->setAttribute(i-1);
};

rule EPP1{
   decl    EPP1 *obj;
           double d1;
           double d2;
   event   obj->setData(d1,d2);
   cond    d1 < d2;
   action  obj->setMinData();
};

rule EPP2{
   decl    EPP2 *obj;
           double d1;
           double d2;
   event   obj->setData(d1,d2);
   cond    def  d1 < d2;
   action  obj->setMinData();
};

rule EPP3{
   decl    EPP3 *obj;
           double d1;
           double d2;
   event   obj->setData(d1,d2);
   cond    dep  d1 < d2;
   action  obj->setMinData();
};

rule GC1{
   decl    ;
   event   1000 times ABSTRACT(GC1) in
                    BOT('GC1') ,
                    COMMIT('GC1')
  cond    FALSE;
  action  ;
};
```

Figure 3: The OBJECTIVE Benchmark rules

3. perform cold and hot time measurements, and
4. close the database.

(Cetintemel et al., 1996) presents simplified codes that illustrate the implementation of the OBJECTIVE operations.

The environment in which we benchmark REACH is a SUN-SPARC 10/512 with 112 MB of RAM under Solaris 2.5, Open OODB 0.2.1, and the EXODUS Storage Manager[7] 2.2. Each operation has been run about 50 times for the same setting of database parameters. Table 3 depicts the mean values of the REACH results. All results along with 90% confidence intervals and standard deviations are presented in (Cetintemel et al., 1996).

4.4.1 Method Wrapping

The purpose of operation MW1 is to asses the cost of the *useless* overhead generated by the invocation of a method which is wrapped to provide active functionality whenever required. In operation MW1, a method is invoked which does not generate any event. Although this test seems to be applicable only to those systems that use method wrapping, the primary aim of it is to determine whether an

[7] Open OODB uses EXODUS as its storage manager.

Parameter	Empty	Small	Medium	Large
NumEvents	0	100	500	1000
FracCompEvents		0.3,0.6,0.9	0.3,0.6,0.9	0.3,0.6,0.9
NumRules	0	100	500	1000
NumObjects	0	5000	25000	50000

Table 2: The OBJECTIVE database configurations

overhead is incurred or not when active functionality is *not* used. In this sense, the name of this test may be somewhat misleading.

REACH attempts to minimize this overhead by assigning a global variable to each method indicating the presence/absence of a detector for that method in the database; thereby reducing it to a memory look-up rather than a database access. Nevertheless, the database must be scanned for the relevant event detectors and the corresponding variables must be set in the memory before the start of an application program.

4.4.2 Event Detection

The primitive event detection operations examine how efficiently an ADBMS detects primitive events of interest. The aim of operation PED1 is to measure the time it takes to detect a method event. We invoke a method which generates a primitive event which is not an event type for any rule. By this way, we try to discard the time for rule execution, and concentrate on event detection only. Operation PED2 tries to measure the time it takes to detect a transaction event. Unfortunately, in any transaction operation the underlying system does certain bookkeeping operations which are not interesting to us. We chose the BOT operation since it seems to contain minimum uninteresting operations when compared with the other transaction operations. This transaction operation generates an event which does not trigger any rule. On the other hand, operation PED3 considers the COMMIT operation whose primary focus is, unlike that of PED2, not only on the detection of a transaction event, but also on getting an insight about the influence of the support for some active functionalities (e.g., event history management).

The composite event detection operations examine the event composition of an ADBMS. In order to concentrate on composition costs only, we used the smallest possible number of component primitive events for testing different composition types. It would be just as easy to use a larger number of component events, but then it would be very hard to justify a particular number, and more importantly there would be a relatively high risk that the composition costs be overshadowed. To stress the composition costs even more, abstract events are used as component events to exclude event detection and parameter passing time. As in the case of primitive event detection operations, the composite event detection operations generate composite events which are not event types for any rules. It is important to note here that, in all the event detection operations, there is also an overhead for looking up rules to be fired. The primitive and composite event types[9] relevant to event detection operations

[8] The subscripts $_c$ and $_h$ represent cold and hot time results, respectively.

[9] Since REACH does not allow the creation of events without any associated rules through its rule definition interface,

TEST	**CONFIGURATION**									
	EMPTY	**SMALL**			**MEDIUM**			**LARGE**		
		0.3	**0.6**	**0.9**	**0.3**	**0.6**	**0.9**	**0.3**	**0.6**	**0.9**
$MW1_h$	0.03	0.04	0.03	0.04	0.03	0.03	0.03	0.03	0.03	0.03
$PED1_h$	2.04	2.19	2.27	2.15	2.31	2.57	3.07	3.50	3.80	3.70
$PED2_c$	12.72	13.79	14.67	14.19	13.37	13.22	15.03	16.97	17.70	17.37
$PED3_c$	318	1005	1062	5447	10069	20921	42758	35436	46321	74865
$CED1_h$	3.50	3.72	3.77	3.76	4.01	3.82	4.45	5.42	5.49	5.35
$CED2_h$	4.16	4.31	4.30	4.37	4.48	4.51	5.31	6.43	6.80	6.39
$CED3_h$	3.60	3.68	3.69	3.56	3.97	3.91	4.53	5.52	5.81	5.50
$CED4_h$	4.69	4.86	4.84	4.84	5.17	5.21	6.15	7.47	7.50	7.49
$CED5_h$	4.73	4.87	4.88	4.79	5.11	5.12	6.13	7.02	7.36	7.68
$RF1_c$	10.58	12.21	12.38	12.79	13.37	13.77	14.64	16.48	16.54	16.84
$RF2_h$	1.68	1.92	1.94	1.92	2.48	2.47	2.60	3.31	3.20	3.26
$RF3_h$	2.38	2.61	2.65	2.66	2.54	2.75	3.08	3.95	3.71	4.26
$RF4_h$	1.50	2.04	2.02	1.91	2.11	2.53	2.70	2.48	2.59	2.53
$RF5_h$	1.46	2.44	2.44	2.33	2.21	2.71	3.17	3.03	3.84	3.48
$RF6_h$	2.40	3.02	3.04	2.96	3.28	3.84	4.05	4.09	4.58	4.37
$EPP1_h$	2.12	2.86	2.84	2.75	2.89	3.05	3.58	3.81	3.86	3.78
$EPP2_h$	2.84	3.07	3.05	2.96	3.44	3.73	4.06	5.16	5.18	5.90
$EPP3_h$	3.40	3.44	3.84	3.57	3.66	3.96	4.53	5.81	5.93	6.32
$GC1_c$	19712	19423	19785	26483	18981	26010	48674	102149	171610	272112
$RA1_c$	4.48	4.53	4.60	4.57	4.63	4.71	5.40	7.64	7.92	8.85
$RA2_c$	2.18	2.13	2.25	2.27	2.22	2.21	2.42	2.23	2.34	3.71
$RA3_c$	2.07	2.06	2.17	2.08	2.17	2.16	2.40	2.52	2.39	2.55
$RA4_c$	2.22	2.14	2.22	2.07	2.07	2.48	2.58	2.46	2.51	2.66

Table 3: The OBJECTIVE results[8]for REACH (in milliseconds)

are defined in Figure 2.

REACH optimizes useful overhead of method event detection as well as useless overhead. This is accomplished by a using a prefetching mechanism. This mechanism, by examining the relevant application programs and header files, prefetches the necessary primitive method event detectors, composite event detectors containing those primitive event types as constituents, and the rules to be triggered by the occurrences of these event types. However, this prefetching is done only for method event types, not for transaction or abstract events. This explains why the PED2 results are worse than the PED1 results. The comparison of results of operation PED2 and those of operation PED3 reveals that the COMMIT operation itself, not its detection, shows very poor performance. A more through investigation leads us to the fact that this behavior is primarily a consequence of REACH's

we had to create these events manually. REACH takes a different approach in this respect, because almost all object-oriented ADBMSs (e.g., SAMOS, NAOS, SENTINEL, and ACOOD) encourage the stand-alone definition of events for reusability reasons.

poor event history[10] maintenance; i.e., at commit time REACH updates the event history with the events occurred in that transaction. However, it is also evident (from the dependency of the results on database configuration) that this update is implemented inefficiently. In addition, it can be inferred from the large standard deviations of PED3 results (see (Cetintemel et al., 1996)) that, duration of the event history maintenance task (thus duration of the commit operation) depends on the size of the event history which increases at each run of the benchmark operations. Another contributing factor is the underlying platform, Open OODB, which always writes back the whole buffer at commit time.

Results for the composite event detection operations show almost no dependency on database configuration. This is a direct consequence of the use of extended syntactic trees for event composition. For each composite event type, a specialized event detector object is constructed; hence, the overhead of using more generic models (e.g., Petri Nets) is eliminated; making the event composition process very fast. The results for operations CED1 and CED2 are slightly better, as they do not require the confirmation of a validity interval as is done in operations CED3, CED4, and CED5. In general, composite event detection process scales very well; even the most crucial parameters for this test, *NumEvents* and *FracCompEvents*, do not have a notable effect on the results.

4.4.3 Rule Firing

The rule firing operations of the OBJECTIVE Benchmark focus on different aspects of rule firing in an ADBMS. Operation RF1 measures the cost of fetching a rule from the rulebase by triggering a rule in immediate coupling mode. In order to keep the elapsed time for rule execution minimal, which is not interesting to us in this operation, the triggered rule has a FALSE condition part, so that condition evaluation is relatively cheap, and no action is executed. Operations RF2 and RF3 trigger rules in deferred and decoupled coupling modes, respectively. These operations do not measure the time to fire and execute rules in different coupling modes; rather, they examine the cost of storing the information that the triggered rule will be fired just before commit, and in a new transaction, respectively. Although the task measured by RF3 is similar to that measured by RF2 (i.e., abstract event signalling and notification of the current transaction to store a particular bit of information), the contribution of operation RF3 is mainly with respect to functionality (i.e., is decoupled mode supported?). The focus of operation RF4 is on determining how efficiently a rule's condition/action parts are accessed and executed (or interpreted). This operation triggers a rule with a TRUE condition part, so that its action part (though empty) is executed. Operation RF5 reveals the overhead when an event occurs and two rules have to be fired. Different priorities are assigned to these rules to force a particular serialization order. Operation RF6 invokes a method event which triggers a rule that generates the same event in its action part. Therefore the same rule is triggered a second time, but with a condition which evaluates to FALSE; stopping this cascading rule firing. The rules which are triggered by the rule firing operations are defined in Figure 3.

As REACH treats rules as first-class objects, rules are fetched just like ordinary objects by using their names. The (cold time) results for operation RF1 suggest a dependency of rule retrieval time on database configuration. It is important to emphasize here that cold times make sense for this operation as no prefetching mechanism is used for abstract events. Results for operations RF2 and RF3 show that it is slower to initialize the triggering of a rule in decoupled mode than to initialize it in deferred mode. Such a behavior is not surprising at all, since operation RF3 contains the initialization

[10] Event history is the log of all event occurrences since system startup.

of a new transaction to execute the rule. The results for operation RF4 indicate mainly the time for accessing and executing the action part of the rule. These results are almost constant for all database configurations, because the condition and action parts of a rule are stored as compiled code in shared library allowing very fast access and execution independent of database parameters. The figures for operation RF6 are slightly worse than those for operation RF5. Although both operations contain two rule triggerings, RF6 generates two method event occurrences, whereas in RF5 rules are triggered by a single abstract event.

4.4.4 Event Parameter Passing

The event parameter passing operations test how efficiently an ADBMS passes event parameters to the condition and action parts of the rules in different coupling modes. The operation EPP1 measures the cost of parameter passing as well as rule execution in immediate coupling mode, whereas the operations EPP2 and EPP3 measure just the cost of using an intermediate storage for passing event parameters. From the point of view of the triggered rules, there is a similar overhead due to the retrieval of the event parameters from the storage where they reside temporarily; but this overhead is not measured by our operations. The rules triggered by the event parameter passing operations are defined in Figure 3.

REACH supports the ability to pass all arguments of a method invocation that triggers a rule to condition and action parts of that rule. In immediate mode all arguments are stored in a bag (i.e., bytestring) and access to the arguments is accomplished by using an array of pointers that store addresses of the arguments. The same mechanism is used in deferred mode, but the dereferenced value of a pointer argument is stored in the array instead of the pointer itself. In detached mode, as the execution of the rule will take place in a different address space, the bag and the pointer array are written in a file. Different requirements for the implementation of these approaches show their effects in the results, making parameter passing somewhat expensive in detached mode due to the inevitable use of an intermediate secondary storage.

4.4.5 Garbage Collection of Semi-Composed Events

The purpose of operation GC1 is to examine the overhead of flushing an event composition structure that is used in the detection of a composite event. In this operation, we first produce garbage (i.e., create a semi-composed event), and then try to measure the time for collecting the garbage. Such a garbage collection can typically be accomplished at two different points (from a black-box point of view):

- immediately after the monitoring interval is finished, or
- at commit time.

In the former case, the time for the operation generating the end-of-interval event, and in the latter case, the time for commit operation should be measured. For generality of the test, we take COMMIT to be also the end-of-interval event so that garbage collection can only be accomplished during commit for this operation. Unfortunately, isolation of garbage collection inside commit is not possible by using the results of this operation only. However, we can circumvent this problem to a certain extent by using the difference of the results of this operation and those of operation PED3 (i.e., detection of

COMMIT) in which no time for garbage collection is involved. In this manner, we may have an *estimation* of the times indicating the duration of the garbage collection task, which is the best we can do with our black-box view of the system.

As in the case of operation PED3, we encounter very poor results for operation GC1. It is argued above that results of operation PED3 be used in the interpretation of the results of GC1. Unfortunately, it is out of question to get an understanding of the performance of the system under the intended task even by using results of PED3. The reason is that, as mentioned in Section 4.4.2, commit time is dependent on the size of the event history in REACH, and the size of the event history is not the same in respective runs of operation PED3 and operation GC1; making it impossible to interpolate the time for garbage collection by using the results of these two operations.

4.4.6 Rule Administration

The rule administration operations are somewhat different from the other OBJECTIVE operations in the sense that they are more likely to be included in a *feature* benchmark. However, we deem the functionalities examined by these operations so important from the functionality point of view that they must be included in a comprehensive benchmark for ADBMSs. Although execution time results are also presented for these operations, for the purposes of this benchmark it is more important to observe whether these tasks are supported than to concentrate on the numbers obtained.

Operation RA1 creates a new rule and stores it in the rulebase, and operation RA2 deletes an existing rule from the rulebase. Operation RA3 and RA4, enables and disables a rule, respectively. Operation RA5 changes the action part of a rule. In all these operations, the relevant rules are kept very simple in order to focus on the efficiency of the provided rule administration facility.

All of the rule administration operations are implemented using the rule management commands of REACH from its command line interface (Zimmermann et al., 1996). The implementation of operation RA1 in REACH consists of the creation of a rule and compilation of the shared library containing the condition/action parts of rules in the form of two C functions by using the REACH command `rl_cc`. The other operations, RA2, RA3, and RA4, are implemented using REACH commands `r_delete`, `r_enable`, and `r_disable`, respectively. Unfortunately, we were not able to get results for operation RA5 (although it is possible to modify rules dynamically in REACH) because of a bug in the system. The results for the presented rule administration operations, except RA1, show a constant behavior under all database configurations. The exceptional results for operation RA1 are possibly due to the compilation time of the shared library whose size is directly proportional to the number of rules.

5 Conclusions and Future Work

We presented the OBJECTIVE Benchmark for object-oriented ADBMSs, and illustrated it with the results obtained from its implementation in REACH. Although OBJECTIVE is designed to be very simple in nature, it is also very comprehensive in its coverage of active functionalities; it is simpler and more comprehensive than previous competitor benchmarks.

The results obtained from the implementation of OBJECTIVE on REACH reveal that REACH supports a high level of active functionality and (almost) all components of REACH scale well. The only exception we encountered is the problematic commit operation of REACH. This operation is a real bottleneck as it is a *must* operation for all applications running inside a transaction framework, and this bottleneck must be surmounted to achieve acceptable overall system performance. The implementation phase also helped to disclose a number of bugs in the system. The results of REACH alone are sufficient to identify its bottleneck components. However, results to be taken from different systems (with possibly different approaches and architectures for supporting ADBMS tasks) would be highly welcome to make an objective judgment about the degree of efficiency with which these tasks are supported by a particular ADBMS.

We believe that the OBJECTIVE operations cover an important subset of issues with respect to ADBMS performance and functionality. The remaining issues are mainly the ones related to event consumption policies, condition optimization, and parallel rule execution.

An open related research area is the evaluation of ADBMS performance in multi-user environments. There is considerable performance difference between single-user and multi-user environments which results from issues of optimal system resource utilization. Therefore, the results obtained from a single-user benchmark do not necessarily represent the real performance of the system. It is especially interesting to investigate the effects of the number of concurrently running transactions to event detection and rule execution.

An interesting thing to note here is that all the benchmarks that have been proposed so far for ADBMSs, including OBJECTIVE, are *non* domain-specific benchmarks. This is, we think, a consequence of the lack of adequate information about the characteristics of ADBMS tasks (even the notion of an ADBMS task is elusive for now). As the application areas for ADBMSs mature, we expect to see the development of domain-specific benchmarks to evaluate end-to-end performance in order to have a better understanding of ADBMS performance.

As a final remark, we hope that the OBJECTIVE Benchmark finds acceptance as a useful yardstick for evaluating ADBMS performance and functionality.

References

Agrawal R. and Gehani N. H., ODE (object database and environment): The language and the data model, in *Proc. of the 1989 SIGMOD Conference*, Portland, Oregon, 1989.

Berndtsson M., ACOOD: An Approach to an Active Object Oriented DBMS, Master's Thesis, University of Skovde, Skovde, Sweden, 1991.

Boral H. and DeWitt D. J., A methodology for database system performance evaluation, in *Proc. of the 1984 SIGMOD Conference*, Boston, 1984.

Brant D. A. and Miranker D. P., Index support for rule activation, in *1993 SIGMOD Conference*, Washington D.C., 1993.

Buchmann A. P., Zimmermann J., Blakeley J., and Wells D. L., Building an integrated active OODBMS: requirements, architecture and design decisions, in *Proc. of the 1995 Data Engineering Conference*, Taipeh, 1995.

Carey M. J., DeWitt D. J., and Naughton J. F., The 007 benchmark, in *Proc. of the 1993 SIG-*

MOD Conference, Washington D.C., 1993.

Catell R. and Skeen J., Object Operations Benchmark, *ACM Transactions on Database Systems*, 17(1):1-31, 1992.

Cetintemel U., Zimmermann J., Ulusoy O., and Buchmann A., OBJECTIVE: A Benchmark for Object-Oriented Active Database Systems, Technical Report BU-CEIS-9610, Bilkent University, Ankara, Turkey, 1996.

Chakravarthy S. and Mishra D., SNOOP: An Expressive Event Specification Language for Active Databases, Technical Report UF-CIS-TR-93-007, University of Florida, Florida, 1993.

Chakravarthy S., Krishnaprasad V., Abwar E., and Kim S. K., Anatomy of a Composite Event Detector. Technical Report UF-CIS-TR-93-039, University of Florida, Florida, 1993.

Chakravarthy S., Tamizuddin Z., Krishnaprasad V., and Badani R. H., ECA Rule Integration into an OODBMS: Architecture and Implementation, Technical Report UF-CIS-94-023, University of Florida, Florida, 1994.

Collet C., Coupaye T., and Svensen T., NAOS: Efficient and modular reactive capabilities in an object-oriented database system, in *Proc. of the 20th VLDB*, Santiago, Chile, 1994.

Dayal U., Blaustein B., Buchmann A., Chakravarthy S., Hsu M., Ladin R., McCarthy D., and Rosenthal A., The HiPAC Project: Combining Active Databases and Timing Constraints, *ACM SIGMOD Record*, 17(1):51-70, 1988.

Dayal U., Active database management systems, in *Proc. of the 3rd International Conference on Data and Knowledge Bases*, Jerusalem, 1988.

Deutsch A., Detection of Method and Composite Events in the Active DBMS REACH, Master's Thesis, Technical University Darmstadt, Darmstadt, Germany, 1994.

Eriksson J., CEDE: Composite Event Detector in an Active Object-Oriented Database, Master's Thesis, University of Skovde, Skovde, Sweden, 1993.

Gatziu S. and Dittrich K. R., Events in an active object-oriented database system., in *Proc. of the 1st International Workshop on Rules in Database Systems*, (N.W. Paton, H.W. Williams, eds.), Springer-Verlag, 1994.

Gatziu S. and Dittrich K. R., Detecting composite events in active database systems using Petri nets, in *Proc. of the 4th IEEE RIDE*, Houston, Texas, 1994.

Gatziu S., Geppert A., and Dittrich K. R., The SAMOS Active DBMS Prototype, Technical Report CS 94.16, University of Zurich, Zurich, Switzerland, 1994.

Gehani N., Jagadish H. V., and Shumeli O., Composite event specification in active databases: Model and implementation, in *Proc. of the 18th VLDB*, Vancouver, Canada, 1992.

Geppert A., Gatziu S., and Dittrich K. R., A Designer's Benchmark for Active Database Management Systems: 007 Meets the BEAST, Technical Report CS 95.18, University of Zurich, Zurich, Switzerland, 1995.

Geppert A., Berndtsson M., Lieuwen D., and Zimmermann J., Performance Evaluation of Active Database Management Systems using the BEAST benchmark, Technical Report CS 96.01, University of Zurich, Zurich, Switzerland, 1996.

Gray J., *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press, 1985.

Wells D. L., Blakeley J. A., and Thompson C. W., Architecture of an Open Object-Oriented Database Management System, *IEEE Computer*, 25(10):74-81, 1992.

Zimmermann J. and Buchmann A. P., Benchmarking active database systems: A requirements analysis, in *OOPSLA'95 Workshop on Object Database Behavior, Benchmarks and Performance*, Austin, Texas, 1995.

Zimmermann J., Branding H., Buchmann A. P., Deutsch A., and Geppert A., Design, implementation and management of rules in an active database system, in *Proc. of the 7th DEXA*, Zurich, Switzerland, 1996.