OBJECTIVE: A BENCHMARK FOR OBJECT-ORIENTED ACTIVE DATABASE SYSTEMS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER

ENGINEERING AND INFORMATION SCIENCE

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Uğur Çetintemel

July, 1996

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Özgür Ulusoy (Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Erol Arkun

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Hakan Karaata

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet Baray
Director of Institute of Engineering and Science

ABSTRACT

OBJECTIVE: A BENCHMARK FOR OBJECT-ORIENTED ACTIVE DATABASE SYSTEMS

Uğur Çetintemel

M.S. in Computer Engineering and Information Science

Supervisor: Asst. Prof. Özgür Ulusoy

July, 1996

Although much work in the area of Active Database Management Systems (ADBMSs) has been done, there have been only a few attempts to evaluate the performance of these systems, and it is not yet clear how the performance of an active DBMS can be evaluated systematically.

In this thesis, 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 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.

Key words: Active database systems, database benchmarks, object-oriented database systems.

ÖZET

OBJECTIVE: NESNE YÖNELİMLİ AKTİF VERİ TABANI SİSTEMLERİ İÇİN BİR DEĞERLENDİRME

Uğur Çetintemel

Bilgisayar ve Enformatik Mühendisliği

Yüksek Lisans

Tez Yöneticisi: Yrd. Doç. Özgür Ulusoy

Temmuz, 1996

Aktif veri tabanı yönetim sistemleri alanında pek çok çalışma yapılmış olmasına karşın, bu sistemlerin performanslarının değerlendirilmesine dair sadece bir kaç çalışma vardır, ve halen bir aktif veri tabanı yönetim sistemi performansının değerlendirmesinin sistematik olarak nasıl yapılacağı açık değildir.

Bu tezde nesne yönelimli aktif veri tabanı yönetim sistemleri için OBJECTIVE Değerlendirmesi'ni tanımlıyoruz, ve bu değerlendirmenin bir aktif veri tabanı sisteminde gerçekleştirilmesinden elde edilen sonuçları sunuyoruz. OBJECTIVE bir aktif veri tabanı yönetim sisteminin performans dar boğazlarını ve aktif fonksiyonlarını belirlemek, ve birden çok aktif veri tabanı yönetim sisteminin performanslarını karşılaştırmak için kullanılabilir. OBJECTIVE'in amacı aktif fonksiyonları sağlayan parçaları ayırmak, ve diğer faktörlerin etkilerini en aza indirgeyerek sadece bu parçaların performansları üzerinde yoğunlaşmaktır.

Anahtar sözcükler: Aktif veri tabanı sistemleri, veri tabanı değerlendirmesi, nesne yönelimli veri tabanı sistemleri.

ACKNOWLEDGEMENTS

I am grateful to my supervisor, Assistant Professor Özgür Ulusoy, for his guidance and motivating support. It was a real pleasure to work with him.

I would like to thank Jürgen Zimmermann for helping me implement the benchmark in REACH. The implementation would not be possible without his help. I also would like to thank Prof. A. Buchmann and A. Deutsch for their valuable comments and suggestions during the development of the thesis.

I would like to thank Prof. Erol Arkun and Asst. Prof. Hakan Karaata for reading and commenting about the thesis.

Last, but definitely not least, I want to thank my colleagues G. Tunalı, Y. Saygın, T. Kurç, and K. Yorulmaz for their moral support.

Finally, I want to express my deepest gratitude to my family, for everything they did to bring me to this position. I dedicate this thesis to them.

Contents

1 Introduction **1**

1.1 The Need for Reactive Behavior in Database Systems 1

1.2 Motivation and Outline of the Thesis 2

2 Background **5**

2.1 Active Database Management Systems 5

2.1.1 Event-Condition-Action (ECA) Rules 5

2.1.2 Events 6

2.1.3 Conditions 8

2.1.4 Actions 8

2.1.5 Execution Model 9

2.1.6 Architectural Aspects 10

2.2 State-of-the-Art of Object-Oriented Active Database Systems . 11

2.2.1 ACOOD 11

2.2.2 NAOS 12

2.2.3 Ode 12

2.2.4 SAMOS 13

2.2.5 Sentinel 14

2.3 The Benchmarking of Database Systems 14

3 Related Work **16**

3.1 The BEAST Benchmark 16

3.2 The ACT-1 Benchmark 17

3.3 Other ADBMS Benchmarking Related Work 18

4 The OBJECTIVE Benchmark **19**

4.1 The OBJECTIVE Operations 20

4.2 Description of the OBJECTIVE Database 22

4.3 The OBJECTIVE Benchmark Implementation 24

5 The OBJECTIVE Results for REACH **35**

5.1 REACH 35

5.2 Results 36

5.2.1 Results for the Method Wrapping Operation 37

5.2.2 Results for the Event Detection Operations 37

5.2.3 Results for the Rule Firing Operations 38

5.2.4 Results for the Event Parameter Passing Operations . . . 39

5.2.5 Results for the Garbage Collection Operation 39

5.2.6 Results for the Rule Administration Operations 39

6 Conclusions and Future Work **42**

A The Complete Results for REACH **48**

B A Sample Event Creation Program **64**

C A Sample Rule Creation Program **68**

List of Figures

4.1 A class example 23

4.2 The events related to event detection operations 25

4.3 The OBJECTIVE Benchmark rules 26

4.4 An example dummy class 27

4.5 The Method Wrapping program 29

4.6 The Primitive Event Detection program 30

4.7 The Composite Event Detection program 30

4.8 The Rule Firing program 31

4.9 The Event Parameter Passing program 32

4.10 The Garbage Collection program 33

List of Tables

4.1 The OBJECTIVE operations 21

4.2 The OBJECTIVE database configurations 28

5.1 The OBJECTIVE results for REACH 41

A.1 The Method Wrapping results (EMPTY and SMALL Database Configurations) 48

A.2 The Method Wrapping results (MEDIUM Database Configuration) 49

A.3 The Method Wrapping results (LARGE Database Configuration) 49

A.4 The Primitive Event Detection results (EMPTY and SMALL Database Configurations) 50

A.5 The Primitive Event Detection results (MEDIUM Database Configuration) 51

A.6 The Primitive Event Detection results (LARGE Database Configuration) 51

A.7 The Composite Event Detection results (EMPTY and SMALL Database Configurations) 52

A.8 The Composite Event Detection results (MEDIUM Database Configuration) 53

A.9 The Composite Event Detection results (LARGE Database Configuration) 54

A.10 The Rule Firing results (EMPTY and SMALL Database Configurations) 55

A.11 The Rule Firing results (MEDIUM Database Configuration) . . 56

A.12 The Rule Firing results (LARGE Database Configuration) . . . 57

A.13 The Event Parameter Passing results (EMPTY and SMALL Database Configurations) 58

A.14 The Event Parameter Passing results (MEDIUM Database Configuration) 59

A.15 The Event Parameter Passing results (LARGE Database Configuration) 59

A.16 The Garbage Collection results (EMPTY and SMALL Database Configurations) 60

A.17 The Garbage Collection results (MEDIUM Database Configuration) 61

A.18 The Garbage Collection results (LARGE Database Configuration) 61

A.19 The Rule Administration results (EMPTY and SMALL Database Configurations) 62

A.20 The Rule Administration results (MEDIUM Database Configuration) 63

A.21 The Rule Administration results (LARGE Database Configuration) 63

Chapter 1

Introduction

1.1 The Need for Reactive Behavior in Database Systems

Active database systems have recently been proposed as an alternative to *passive* (conventional) database systems which can only provide unsatisfactory solutions to a number of monitoring applications. These applications include integrity control, access control, derived data handling, workflow management, network management, and computer-integrated manufacturing. Common to all these applications is the necessity to react to certain situations of interest—with almost unpredictable occurrence patterns—in a timely manner.

Passive database systems can support such applications by using either polling or embedding monitoring code in applications. *Polling* indicates the running of situation monitoring code periodically. This method allows for the detection of monitored situations only at discrete points in time determined by the frequency of polling. If the frequency is set too low, then timely response may not be achieved as event occurrences will not be detected till the next poll. On the other hand, if the frequency is set too high, then the system may be overloaded by queries that do not detect any interesting situations.

The alternative to polling is to *embed monitoring code in applications*. Unfortunately, this method is poor in terms of modularity and maintainability. Any modifications done to the situations being monitored, or to the reactions to situations require that all relevant application programs be modified accordingly. Furthermore, there will be repeated—and possibly inconsistent—specifications of the same situation monitoring and reaction code, making the application programs hard to maintain.

In order to overcome these problems, active database management systems (ADBMSs) are proposed to provide timely response and modularity by extending passive DBMSs with the ability to specify and implement reactive behavior.

1.2 Motivation and Outline of the Thesis

An ADBMS detects certain situations and performs corresponding user defined actions typically in the form of Event-Condition-Action (ECA) rules [16]. ADBMSs have received great attention lately, and several prototypes of object-oriented ADBMSs are already available (e.g., ACOOD [4], NAOS [14], Ode [1], REACH [7], SAMOS [23], SENTINEL [13]). 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 [5], the OO1 Benchmark [10], and the OO7 Benchmark [8]. However, there have been only a few attempts to evaluate the performance of ADBMSs, the most important of which are the BEAST Benchmark [26], and the ACT-1 Benchmark [37].

In this thesis, 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

[1] OBJECT-oriented actIVE database systems benchmark

very simple (for ease of reproducibility and portability), 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 [7]. REACH is a full-fledged operational object-oriented ADBMS which is tightly integrated in Texas Instruments' Open OODB [34]. The results reported in this thesis reveal that REACH combines the most 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 thesis is organized as follows. Chapter 2 presents background information about this work to an extent which is necessary for the comprehension of the rest of the text. In particular, main issues related to ADBMSs, important features of several object-oriented ADBMS prototypes, and the concept of database benchmarking are discussed in respective sections of this chapter.

Chapter 3 discusses previous research on the performance of ADBMSs emphasizing the BEAST Benchmark and the ACT-1 Benchmark for object-oriented ADBMSs.

Chapter 4 describes the OBJECTIVE Benchmark (operations, database, and implementation) in full detail.

Chapter 5 gives an overview of the REACH ADBMS prototype, and reports the results obtained from the implementation of the OBJECTIVE Benchmark in REACH.

Finally, Chapter 6 concludes the thesis and gives directions for future research.

In addition, Appendix A presents the complete set of OBJECTIVE results for REACH. Appendix B and Appendix C present sample programs used for the creation of benchmark events and rules, respectively.

Chapter 2

Background

2.1 Active Database Management Systems

2.1.1 Event-Condition-Action (ECA) Rules

ECA rules have become a standard to specify reactive behavior. 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 interest) and detect their occurrences. After an event is detected, it is *signalled*. This signalling is a notification that an event of interest 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 [17].

2.1.2 Events

ECA rules are triggered on the occurrence of particular events. An event can be either *primitive* or *composite*.

Primitive Events

Primitive events are atomic events which can be associated with a point in time. The most commonly referred primitive event types are [7, 21, 12]:

- *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 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.

Several 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 [21]. 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 [7].

Composite Events

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 [17, 23]:

- 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 [38]. 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 [18, 11], Petri nets [22], finite state automata [24], and arrays [20].

An *event composition policy* identifies which event occurrence of a particular event type will be used in the event composition process. Consider the

composite event defined as the sequence of event types event1 and event2, and the sequence of event occurrences e_{11}, e_{12}, e_2 (first two events are instances of event1 and the last one is an instance of event2) in this order. In this case, a choice must be made about whether to use e_{11} or e_{12} as the initiator event of the composite event under consideration. Motivated by a number of application types, four useful *parameter contexts* are proposed [12]:

- In *recent* context, the most recent occurrence of a primitive event is used.
- In *chronicle* context, the occurrences are used in chronological order; i.e., in the order they are generated.
- In *continuous* context, each occurrence of an initiator event marks the beginning of a different composite event, and the occurrence of a single terminator event is sufficient to signal all of these composite events.
- In *cumulative* context, all occurrences of a primitive event till the occurrence of the respective composite event are consumed.

2.1.3 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, the predicate is satisfied, 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.

2.1.4 Actions

The action part of a rule is executed when the condition is satisfied. In general, actions 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.

2.1.5 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 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 correct and reliable rule execution. These two issues are discussed in more detail in the rest of this subsection.

Coupling Modes

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 were introduced in [15]: *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 at the end but before the commit of the triggering transaction.
- 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 as (provided that the condition is satisfied):

- In *immediate* CA coupling mode, the action is executed right after the condition evaluation within the same transaction.
- In *deferred* CA coupling mode, the action is executed at the end but before the commit of the triggering transaction.
- In *detached* CA coupling mode, the action is executed in a separate independent transaction.

If several triggered rules have to be executed at the same point in time, they form a *conflict set* [29]. 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.

Transaction Model

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 [32] 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.

2.1.6 Architectural Aspects

Three architectural approaches for implementing ADBMSs are [27]:

- *implementation from scratch*
  All the passive components as well the active ones are implemented from the beginning. Although this implementation approach is very costly in terms of development time and effort, all the required functionality can be implemented without restrictions rooted from the usage of existing software. Ode [1] is a very good example of an ADBMS that is developed from scratch.

- *integrated architecture*
  An existing passive DBMS is modified and customized to offer active functionality. This approach seems to be a compromise between development cost and functionality. REACH [7] and SENTINEL [13] are two object-oriented ADBMS prototypes that have integrated architectures. Both utilize the extensible DBMS Open OODB [34] as the underlying passive database system. NAOS [14] is another prototype ADBMS which is integrated in the O_2 object-oriented database system [3].

- *layered architecture*
  An existing passive DBMS is used as a black-box and active functionality is implemented on top using the external interfaces of the system. This approach is the least costly one, however there are arguments about the level of active functionality that can be provided by such a system.

SAMOS [23] is an ADBMS that is implemented as a layer on top of ObjectStore [31]. ACOOD [4] also has a layered architecture; i.e., it is built on top of ONTOS object database [2].

2.2 State-of-the-Art of Object-Oriented Active Database Systems

Recently, a lot of work has been done in the area of incorporating reactive behavior into database systems, and restricted reactive capability, typically in the form of simple trigger mechanisms, is already being offered by some commercial products [33, 35], all of which are based on the relational model.

HiPAC project [16] pioneered most of the de-facto standard features of object-oriented ADBMSs, e.g., ECA rules, coupling modes, composite events. Since that project (which has not been fully implemented), a large number of object-oriented ADBMS prototypes have emerged [7, 13, 23, 14].

This section presents a number of object-oriented ADBMS prototypes, emphasizing the distinguishing features of each, with the aim to give a state-of-the-art overview of reactive processing in those systems. Another operational object-oriented ADBMS prototype, REACH, which is of special interest to our work is discussed in Section 5.1.

2.2.1 ACOOD

ACOOD [4, 19] was built as a layer on top of ONTOS [2] at the University of Skövde, Sweden. Event definitions and rules are treated as first-class objects, and ECA rules are adopted to specify reactive behavior.

ACOOD currently supports method and abstract event types as its primitive event types, and conjunction, disjunction, sequence, negation, and iteration as its composite event constructors. It is possible to create, delete, or modify events and rules at runtime (by using the programmatic type interface of ONTOS which facilitates access to database schema at runtime).

In order to optimize rule checking when an event occurs, a *subscription* mechanism is employed. This mechanism associates each event with a list of rules that are subscribed to it, restricting the search space of rules in case of event occurrence.

2.2.2 NAOS

NAOS [14] is the active rule component of the O_2 object-oriented DBMS [3]. The current implementation of NAOS supports primitive events (excluding temporal events) but not composite events. Primitive event types include *entity manipulation event types* (e.g., creation, deletion, or modification of objects), and *application event types* which are related to the execution of applications, programs, or transactions (e.g., begin/end of an application). Rules triggered by entity manipulation events can be executed either in immediate or deferred coupling mode, whereas rules triggered by application events can only be executed in immediate coupling mode (deferred execution is meaningless in such a case). In addition to these primitive event types, NAOS also provides user-defined (abstract) events that are defined as stand-alone event objects.

In NAOS, rules are executed in *cycles*, i.e., regardless of its coupling mode, a triggered rule is executed in a new cycle different from the one which contains the triggering operation. More specifically, rules triggered in immediate mode are executed with the initial cycle comprising the operations executed up to the first rule triggering, and the following cycles are determined similarly. For deferred rules, the initial cycle contains the operations of the transaction, whereas all rules triggered in this cycle will be executed in the next cycle, and so on. This *cycling* mechanism for cascading rule execution was first introduced in the HiPAC project.

Delta elements (for immediate rules) and *delta collections* (for deferred rules) are used to realize the execution environment for rules. A delta element contains, besides the object related to the event occurrence, the inserted, deleted, or updated data, or the actual parameters of a method or program (depending on the event type). A delta collection is basically a set of delta elements used for deferred rules. A single delta element is insufficient for a deferred rule, because such a rule may correspond to multiple events of the same type, as regardless of the number of occurrences of a particular event, the corresponding deferred rule is executed only once. Therefore data related to each of these events should be stored. The operators *new*, *old*, *current*, *delta*, and *arg* are proposed to access the contents of these delta elements and collections so as to provide event-condition-action binding.

2.2.3 Ode

Ode [1] was developed at AT&T Bell Laboratories. It uses O++ for definition, querying, and manipulation of the underlying database. O++ extends C++ with facilities for the creation and manipulation of persistent objects.

Ode implements active functionality in terms of *constraints* and *triggers*.

Both are associated with class definitions and are not regarded as objects. Events are not defined explicitly in Ode, i.e., constraints and triggers are specified by conditions and actions. Only object updates caused by public member functions are regarded as events.

A constraint in Ode consists of a predicate and a handler (action), and the handler of the constraint is executed when the predicate is *not* satisfied. *Hard* and *soft* constraints are supported: Hard constraints are checked immediately after event occurrence, and the checking of the soft constraints are delayed till the end of transaction. If a constraint is not satisfied and no handler is specified for that constraint, or a handler is provided but the execution of the handler does not result in the fulfillment of the constraint (i.e., condition is re-evaluated after the execution of the handler), then the transaction is aborted.

Like constraints, triggers are also defined by condition-action pairs. Although triggers are also specified inside class definitions, they must be explicitly activated for particular instances of their classes. In this respect, triggers are different from constraints as constraints are defined for all instances of their classes. Two basic types of triggers, *once-only* and *perpetual*, are provided by Ode. If a trigger is defined once-only, then it will be automatically *deactivated* after it fires. On the other hand, perpetual triggers do not need explicit reactivation after each firing. In addition *timed* triggers can be specified, which must fire within a given period. The condition of a trigger is checked immediately after the event occurrence, and its action (unlike a constraint handler) is always executed in a separate transaction which is started after the commit of the current transaction.

2.2.4 SAMOS

SAMOS [23] is an object-oriented ADBMS prototype built on top of ObjectStore [31]. It was developed at the University of Zurich, Switzerland.

SAMOS employs ECA rules for specification and implementation of active behavior, and treats them as first-class objects. One of the noteworthy features of SAMOS is its complex event algebra. SAMOS supports the composite event constructors *conjunction*, *negation*, and *times*, along with *disjunction*, *sequence*, and *closure* which were inherited from the HiPAC project [16, 17].

SAMOS allows for the passing of a *fixed* set of event parameters which (partially) allows it to see the state of the database at the time of event occurrence. These parameters include event occurrence time, transaction identifier of the transaction in which the event occurred, the owner of the transaction, and the object whose method has been invoked (in case of method events).

2.2.5 Sentinel

Sentinel [13] was developed at the University of Florida by extending the extensible object-oriented DBMS Open OODB [34]. Its design was heavily influenced by the HiPAC project.

Sentinel has a comprehensive event specification language called Snoop. Snoop defines *four parameter contexts*: recent, chronicle, continuous, and cumulative, in order to specify the order in which successive event occurrences of a particular event are consumed as constituents of a composite event. These contexts are defined within a rule rather than within an event, primarily to maintain the reusability of events which are defined as stand-alone objects.

Sentinel supports concurrent and nested rule execution, and the order of rule execution is determined by using priorities. Priority classes are defined for *global* conflict resolution, and rules of the same priority class are executed concurrently.

2.3 The Benchmarking of Database Systems

Benchmarking, broadly speaking, is a systematic evaluation of the system under consideration. In that sense, DBMS benchmarks can be considered as a way to measure performance and/or functionality of a DBMS. In general, a benchmark is designed to examine DBMS performance in a specific domain of applications, or to evaluate the performance of particular components of a DBMS. Typically, users are interested in the former type of *domain-specific* benchmarks as they reflect end-to-end performance. On the other hand, database designers and implementors are more interested in isolating certain components and focusing on the performance of each separately. This helps highlight problematic components of the system which can then be optimized, or, at the worst case, be reimplemented to become more performant.

A good domain-specific benchmark must meet four important criteria [28]:

- A domain-specific benchmark should be *relevant* to its domain, i.e., it should evaluate the system performance when performing operations that are typical of the target domain.
- A benchmark must be *portable* to several different systems, i.e, it should be easy to implement the benchmark on different systems.
- A benchmark should be *scalable* to small and large computer systems. It is necessary that the benchmark be scalable as the capabilities of systems increase.

- A benchmark should be *simple* to understand and implement.

When considering *non* domain-specific benchmarks, the first criteria, namely relevance, is not applicable; thus, portability, scalability, and simplicity should be taken into account when evaluating such benchmarks.

Chapter 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 (e.g., [26, 37, 6, 30]). In this chapter, we discuss these efforts in some detail.

3.1 The BEAST Benchmark

BEAST is a benchmark for testing the performance of active object-oriented database management systems [26]. It is 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 [8].

The BEAST Benchmark focuses on event detection, rule management, and rule execution aspects of an ADBMS. These three aspects represent the whole active behavior and should be covered by any ADBMS.

The BEAST Benchmark runs a series of tests to determine the functionality of each component. It consists of:

- *Tests for event detection*
  These tests 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 message sending, 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*
  The BEAST Benchmark tests 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, namely SAMOS, ACOOD, ODE and REACH, and the performance results are presented in [25].

3.2 The ACT-1 Benchmark

The ACT-1 Benchmark [37] 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 asses 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, are implemented in REACH, and some preliminary results based on response times of these operations are presented [37].

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; [6] 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.

[30] 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 asses the cost of handling a single event and subsequent firing of a single rule. In this experiment, 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.

Chapter 4

The OBJECTIVE Benchmark

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 BEAST Benchmark is a very good initial step towards a benchmark which will cover a bigger set of functionalities of an ADBMS. However, we require a more generic benchmark to be able to test *both* performance and functionality.

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, in order not to skew results in favor of systems with less functionality, OBJECTIVE also concentrates on some critical functionality of ADBMSs besides performance.

After introducing the operations of the OBJECTIVE Benchmark along with a requirements analysis in Section 4.1, we describe the synthetic database of OBJECTIVE in Section 4.2. In Section 4.3, we describe the implementation of the benchmark operations.

4.1 The OBJECTIVE Operations

The OBJECTIVE Benchmark addresses the following issues [38] by the operations which are described briefly in Table 4.1:

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 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

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 4.1: The OBJECTIVE operations

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 [7]. 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 online. The ability to maintain rules dynamically is very important because of well-known reasons of availability and extensibility. Although execution speeds of these tasks are not of great importance, a comprehensive benchmark should take them into account.

4.2 Description of the OBJECTIVE Database

Generation of a synthetic database is an important issue in all benchmarks for database systems [28]. 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 4.1 for a generic class definition[1]), and it has a very simple schema. The rationale for this decision is twofold: First, a benchmark should be easily reproducible and portable, and second OBJECTIVE is designed to be a generic benchmark, not 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

[1] 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 4.1: A class example

application. Thus, we do not want to add extra complexity which will not contribute to the benchmark in any manner, but will make the implementation more difficult.

Several events and rules are defined (Figure 4.2 and Figure 4.3) to be used in the benchmark operations. The rules are defined in the rule programming language REAL (REAch rule Language) [36]. 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. The benchmark events, however, are defined in a hypothetical language based on the event definition notation of REAL[2].

The naming convention used for objects, events, and rules are based on 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.

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.

The dummy objects, events, and rules are generated as follows:

- *Dummy objects*:
  Dummy classes (e.g., Figure 4.4) with the same methods and attributes

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

are generated, and the instances of these dummy classes form the (dummy) database objects.

- *Dummy primitive events*:
  The methods of the dummy classes are used to generate before/after (dummy) method events.

- *Dummy composite events*:
  The event constructors *sequence* and *history* are used to generate non-aggregating and aggregating (dummy) composite event types, respectively. For each composite event, the number of component events is selected at random[3] from the set {2, 3, ..., 10}. Likewise, the component event types for a composite event are selected randomly from the already generated (dummy) primitive event types.

- *Dummy rules*:
  A dummy rule chooses its event type at random from the already generated dummy primitive and composite event types. Both the condition and action parts of dummy rules are defined as empty.

4.3 The OBJECTIVE Benchmark Implementation

In this section, we discuss and illustrate the implementation of the benchmark operations which are described briefly in Section 4.1 by using simplified codes.

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 operations 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

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

```
event PED1 { PED1::doNothing(); };

event PED2 { BOT('PED2'); };

event PED3 { COMMIT('PED3'); };

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 4.2: The events related to event detection operations

```
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 4.3: The OBJECTIVE Benchmark rules

```
class Dummy0{
      double data[125];
   public:
      void doNothing0    {; }
      void doNothing1    {; }
      ...
      ...
      void doNothing9    {; } };
```

Figure 4.4: An example dummy class

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 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[4].

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

1. clear the system buffer,
2. open the database,
3. perform cold and hot time measurements, and
4. close the database.

We include four parameters for the OBJECTIVE Benchmark:

- *NumEvents* denotes the number of (dummy) events.

[4]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, as parameters.

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 4.2: The OBJECTIVE database configurations

- *FracCompEvents* denotes the ratio of the number of composite events to the number of all events (i.e., *NumEvents*).
- *NumRules* denotes the number of (dummy) rules.
- *NumObjects* denotes the number of (dummy) data objects.

The database configurations based on these parameters are summarized in Table 4.2.

We do not include every step of the implementation in our codes which illustrate the benchmark operations; as they may vary widely from system to system, and will not contribute much to the understanding of the benchmark. In particular, we skipped the code for handling the database, flushing the system buffer and measuring the elapsed time. For ease of illustration, all the tests of a particular group are shown in the same main program, although each is implemented as an independent operation comprising all the implementation steps outlined above. Each block of statements written in bold font indicates the interesting parts of a particular test and is wrapped around by the code for time measurement. In the implementation, each such block is executed eleven times; we take the cold time to be the time elapsed for the first iteration and the hot time to be the average of the elapsed time for the last ten iterations.

Method Wrapping

The purpose of operation MW1 (Figure 4.5) 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.

Event Detection

The primitive event detection operations (Figure 4.6) examine how efficiently an ADBMS detects primitive events of interest. The aim of operation PED1 is

```
void main(){
    MW1 *objectMW1=new MW1;

    Transaction::begin();
      objectMW1->doNothing();
                                        \\ no event generation
    Transaction::commit();  }
```

Figure 4.5: The Method Wrapping program

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. In 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 is not interesting to us. We chose the BOT operation since it seems to contain minimum irrelevant 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 which, we believe, is the most representative of all transaction operations. The primary focus of this operation, unlike that of PED2, is 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[5] management).

The composite event detection operations (Figure 4.7) examine the event composition of an ADBMS. In order to concentrate on composition costs only, we used minimum (meaningful) 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 relevant to event detection operations are defined in Figure 4.2.

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

```
void main(){
    PED1 *objectPED1=new PED1;

    Transaction::begin();
                                        \\ generate event PED1
      objectPED1->doNothing();
    Transaction::commit();
                                        \\ generate event PED2
    Transaction::begin('PED2');
    Transaction::commit();
    Transaction::begin('PED3');
                                        \\ generate event PED3
    Transaction::commit(); }
```

Figure 4.6: The Primitive Event Detection program

```
void main(){
    Transaction::begin();
                                        \\ generate event CED1
      AbstractEvent::raise('CED1_1');
      AbstractEvent::raise('CED1_2');
                                        \\ generate event CED2
      AbstractEvent::raise('CED2_1');
      AbstractEvent::raise('CED2_2');
                                        \\ generate event CED3
      AbstractEvent::raise('CED3_1');
      AbstractEvent::raise('CED3_3');
                                        \\ generate event CED4
      AbstractEvent::raise('CED4_1');
      AbstractEvent::raise('CED4_2');
      AbstractEvent::raise('CED4_3');
                                        \\ generate event CED5
      AbstractEvent::raise('CED5_1');
      AbstractEvent::raise('CED5_2');
      AbstractEvent::raise('CED5_3');
    Transaction::commit();  }
```

Figure 4.7: The Composite Event Detection program

```
void main(){
    RF6 *objectRF6=new RF6;

    Transaction::begin();
                                        \\ trigger rule RF1
      AbstractEvent::raise('RF1');
                                        \\ trigger rule RF2
      AbstractEvent::raise('RF2');
                                        \\ trigger rule RF3
      AbstractEvent::raise('RF3');
                                        \\ trigger rule RF4
      AbstractEvent::raise('RF4');
                                        \\ trigger rules RF5_1
                                        \\ and RF5_2
      AbstractEvent::raise('RF5');
                                        \\ trigger rule RF6 twice
      objectRF6->setAttribute(1);
    Transaction::commit();  }
```

Figure 4.8: The Rule Firing program

Rule Firing

The rule firing operations (Figure 4.8) 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 (which is not interesting to us in this operation) minimal, 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 coupling 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

```
void main(){
    EPP1 *objectEPP1=new EPP1;
    EPP2 *objectEPP2=new EPP2;
    EPP3 *objectEPP3=new EPP3;

    Transaction::begin();
                                        \\ trigger rule EPP1
      objectEPP1->setData(1.0, 2.0);
                                        \\ trigger rule EPP2
      objectEPP2->setData(1.0, 2.0);
                                        \\ trigger rule EPP3
      objectEPP3->setData(1.0, 2.0);
    Transaction::commit(); }
```

Figure 4.9: The Event Parameter Passing program

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 4.3.

Event Parameter Passing

The event parameter passing operations (Figure 4.9) 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 4.3.

```
void main(){
    Transaction::begin('GC1');
                                \\ create a semi-composed event
    for(int i=0; i < 999; i++)
      AbstractEvent::raise('GC1');
    Transaction::commit(); }
```

Figure 4.10: The Garbage Collection program

Garbage Collection of Semi-Composed Events

The purpose of operation GC1 (Figure 4.10) 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.

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.

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. Note here that we do not illustrate the rule administration operations, because the implementation of these operations may change radically from system to system. The important thing is that, in all these operations, the relevant rules should be kept very simple, e.g., rule RF1 (Figure 4.3), in order to focus on the efficiency of the provided rule administration facility.

Chapter 5

The OBJECTIVE Results for REACH

In this chapter, first the basic features of the REACH object-oriented active database system prototype are given in Section 5.1, and then the results of the application of the OBJECTIVE Benchmark to this system are presented in Section 5.2.

5.1 REACH

REACH (REaltime, ACtive and Heterogeneous mediator system) [7] is an object-oriented ADBMS prototype which is being developed at the Technical University of Darmstadt, Germany. It is one of the first operational prototypes which combines the most advanced features of ADBMSs. REACH is implemented as an extension of Texas Instruments' Open OODB [34]. Open OODB is an extensible object-oriented database system, and it uses EXODUS [9] as its storage manager.

REACH expresses ECA rules in REAL (REAch rule Language) which allows for the specification of the event, condition, action, EC and CA coupling modes, and priority of a rule. When a rule is defined, two C functions (one for the condition and one for the action) are stored in a shared library. GRANT (Graphical Rule AdmiNistration Tool) is a graphical user interface which simplifies the definition and maintenance of rules (see [36] for more about definition and maintenance of rules in REACH).

REACH supports both primitive and composite events. As primitive events,

it supports method events, transaction events, abstract events, and temporal events. Inline method wrapping technique is used for the detection of method events.

For the definition of composite events, REACH takes the sequence, disjunction, and closure composite event constructors from HiPAC [16], and negation, conjunction, and history from SAMOS [23]. In addition, the notion of validity interval of a composite event is also inherited from SAMOS. REACH uses syntax graphs, more specifically extended syntactic trees [18], for composite event detection. Two alternative contexts are provided for event consumption; recent and chronicle.

REACH supports immediate, deferred, and detached coupling modes which determine the semantics of rule execution. A distinguishing feature of REACH is that it does not allow rules to be triggered in immediate coupling mode by composite events. This decision was made to make sure that the event composition process does not delay normal execution of an application.

The current implementation of REACH allows sequential rule execution, but the transaction model is being extended to incorporate parallel rule execution. Priority mechanism is used for conflict resolution of a set of rules, and in the case of a tie, the oldest rule first (default) or the newest rule first (optional) tie-break policies that utilize the creation time of rules can be employed.

5.2 Results

During the implementation of OBJECTIVE in REACH, we were not able to follow our assumption of using just the external interfaces of a system to be tested. This is mainly because of the fact that REACH does not allow the definition of stand-alone event types (i.e., events that do not take part in the event type of any rule) through its rule definition interface. Such stand-alone event types (Figure 4.2) are utilized in event detection operations. For this reason, we created these events manually (see Appendix B for a sample program used to create a stand-alone event). 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.

In addition, we had to define the benchmark rules (Figure 4.3) manually (see Appendix C for a sample program used to create a rule), because at the time of the benchmark implementation, the rule compiler of REACH was not able to support the definition of rules with coupling modes other than immediate.

Another difference between the benchmark specification and its implementation in REACH was in the way database objects were used in benchmark operations. In Section 4.3, we require that the database objects to be used in the benchmark operations be transient rather than persistent to exclude any database overhead. In REACH, this was not possible for certain reasons, so we had to make each such object persistent. However, as we created the objects right in the beginning of each test (e.g., Figure 4.9), and then make them persistent by using the *persist*() member function (which is added to every class by the Open OODB preprocessor); no database access was made (i.e., although the objects were persistent, they were already created and known to be in main memory).

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 2.2. Each operation was run about 50 times for the same setting of database parameters in a normal operating user environment (i.e., not in an isolated machine).

Table 5.1 depicts the mean values[1] of the OBJECTIVE Benchmark results for REACH. All the values are given in milliseconds. In order to keep the exposition manageable and clear, we skip some of the results in this chapter. All results along with 90% confidence intervals and standard deviations are presented in Appendix A.

5.2.1 Results for the Method Wrapping Operation

The useless overhead paid by REACH is quite acceptable (Figure 5.1), because REACH minimizes 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.

5.2.2 Results for the Event Detection Operations

REACH optimizes useful overhead of method event detection as well as its useless overhead. This is accomplished by 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

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

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 poor event history 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 Table A.4, Table A.5, and Table A.6) 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, since 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.

5.2.3 Results for the Rule Firing Operations

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 somewhat 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 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 a 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.

5.2.4 Results for the Event Parameter Passing Operations

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 the 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 intermediate secondary storage.

5.2.5 Results for the Garbage Collection Operation

As in the case of operation PED3, we encounter very poor results for operation GC1. It is suggested in Section 4.3 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 5.2.2, 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.

5.2.6 Results for the Rule Administration Operations

All of the rule administration operations are implemented using the rule management commands of REACH from its command line interface [36]. These commands are programs with names resembling well-known UNIX commands. In addition, they have a prefix indicating the context in which they are to be used.

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 prefix `rl` stands for *rule library*) . The other operations, RA2, RA3, and RA4, are implemented using REACH commands `r_delete`, `r_enable`, and `r_disable` (the prefix `r` stands for *rule*), respectively. Unfortunately, we were not able to get results for operation RA5 (although it is possible 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.

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	3.56	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	5.39	7.64	7.92	8.85
$RA2_c$	2.18	2.13	2.25	2.27	2.22	2.21	2.41	2.23	2.34	3.71
$RA3_c$	2.07	2.05	2.17	2.08	2.17	2.16	2.40	2.52	2.39	2.55
$RA4_c$	2.24	2.14	2.22	2.07	2.07	2.48	2.58	2.46	2.51	2.66

Table 5.1: The OBJECTIVE results for REACH

Chapter 6

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.

The results obtained from the implementation of OBJECTIVE on REACH reveal that REACH supports a high level of active functionality efficiently. Almost all components of REACH perform and 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 generic benchmarks. This is 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.

Bibliography

[1] R. Agrawal and N. H. Gehani. ODE (object database and environment): The language and the data model. In *Proc. 1989 ACM-SIGMOD Conference on Management of Data*, Portland, Oregon, June 1989.

[2] T. Andrews. The ONTOS object database. manuscript.

[3] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database - The story of O_2*. Morgan Kaufmann, 1992.

[4] Mikael Berndtsson. ACOOD: An approach to an active object oriented DBMS. Master's thesis, University of Skövde, Department of Computer Science, Skövde, Sweden, September 1991.

[5] H. Boral and D. J. DeWitt. A methodology for database system performance evaluation. In *Proc. of the 1984 SIGMOD Conference*, pages 176–185, Boston, June 1984.

[6] D. A. Brant and D. P. Miranker. Index support for rule activation. In *ACM SIGMOD Conference on Management of Data*, pages 42–48, Washington D.C., May 1993.

[7] A. P. Buchmann, J. Zimmermann, J. Blakeley, and D. L. Wells. Building an integrated active OODBMS: Requirements, architecture and design decisions. In *Proceedings of Data Engineering Conference*, pages 117–128, 1995.

[8] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. In *ACM SIGMOD Conference*, pages 12–21, May 1993.

[9] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Object and file management in the EXODUS extensible database system. In *Proceedings of the 12th International Conference on Very Large Databases*, 1986.

[10] R. Catell and J. Skeen. Object operations benchmark. *ACM Transactions on Database Systems*, 17(1):1–31, March 1992.

[11] S. Chakravarthy, V. Krishnaprasad, E. Abwar, and S. K. Kim. Anatomy of a composite event detector. Technical Report UF-CIS-TR-93-039, CIS Department, University of Florida, December 1993.

[12] S. Chakravarthy and D. Mishra. SNOOP: an expressive event specification language for active databases. Technical Report UF-CIS-TR-93-007, CIS Department, University of Florida, March 1993.

[13] S. Chakravarthy, Z. Tamizuddin V. Krishnaprasad, and R. H. Badani. ECA rule integration into an OODBMS: Architecture and implementation. Technical Report 94-023, CIS Department, University of Florida, 1994.

[14] C. Collet, T. Coupaye, and T. Svensen. NAOS: Efficient and modular reactive capabilities in an object-oriented database system. In *Proceedings of the 20th International Conference on Very Large Databases*, Santiago, Chile, September 1994.

[15] U. Dayal. Active database management systems. In *Proc. 3rd International Conference on Data and Knowledge Bases*, pages 150–169, Jerusalem, June 1988.

[16] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ladin, D. McCarthy, and A. Rosenthal. The HiPAC project: Combining active databases and timing constraints. *ACM SIGMOD Record*, 17(1):51–70, March 1988.

[17] U. Dayal, A. Buchmann, and D. McCarthy. Rules are objects too: a knowledge model for an active, object-oriented database system. In *Proceedings of 2nd International Workshop on Object-Oriented Database Systems, Lecture Notes in Computer Science 334*. Springer, 1988.

[18] Alin Deutsch. Detection of method and composite events in the active DBMS REACH. Master's thesis, Technical University Darmstadt, July 1994.

[19] Andreas Eklund. Performance evaluation of an active database system. Master's thesis, University of Skövde, Department of Computer Science, Skövde, Sweden, September 1995.

[20] J. Eriksson. Cede: Composite event detector in an active object-oriented database. Master's thesis, Department of Computer Science, University of Skövde, 1993.

[21] S. Gatziu and K. R. Dittrich. Events in an active object-oriented database system. In *Proc. of the 1st International Workshop on Rules in Database Systems*. Springer, September 1993.

[22] S. Gatziu and K. R. Dittrich. Detecting composite events in active database systems using Petri nets. In *IEEE RIDE Proc. 4th International*

Workshop on Research Issues in Data Engineering, Houston, Texas, USA, 1994.

[23] S. Gatziu, A. Geppert, and K. R. Dittrich. The SAMOS active DBMS prototype. Technical Report 94.16, CS Department, University of Zurich, 1994.

[24] N. Gehani, H. V. Jagadish, and O. Shumeli. Composite event specification in active databases: Model and implementation. In *Proc. 18th International Conference on Very Large Data Bases*, Vancouver, Canada, August 1992.

[25] A. Geppert, M. Berndtsson, D. Lieuwen, and J. Zimmermann. Performance evaluation of active database management systems using the BEAST benchmark. Technical Report 96.01, CS Department, University of Zurich, February 1996.

[26] A. Geppert, S. Gatziu, and K. R. Dittrich. A designers benchmark for active database management systems: OO7 meets the BEAST. Technical Report 95.18, Dept. of Computer Science, University of Zurich, 1995.

[27] A. Geppert, S. Gatziu, K. R. Dittrich, H. Fritschi, and A. Vaduva. Architecture and implementation of the active object-oriented database management system SAMOS. Technical Report 95.29, CS Department, University of Zurich, 1995.

[28] Jim Gray. *The Benchmark Handbook for Database and Transaction Processing*. Morgan Kaufmann, 1991.

[29] E. Hanson and J. Widom. An overview of production rules in database systems. Technical Report 92-031, CIS Department, University of Florida, 1992.

[30] M. L. Kersten. An active component for a parallel database kernel. In *Rules in Database Systems*, Workshops in Computing, pages 277–291. Springer, September 1995.

[31] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10), 1991.

[32] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, 1985.

[33] M. Stonebraker and G. Kennitz. The POSTGRES next-generation database management system. *Communications of the ACM*, 34(10):78–92, October 1991.

[34] D. L. Wells, J. A. Blakeley, and C. W. Thompson. Architecture of an open object-oriented database management system. *IEEE Computer*, 25(10):74–81, October 1992.

[35] J. Widom and S. Finkelstein. Set-oriented production rules in relational database systems. In *Proceedings of ACM-SIGMOD*, pages 259–270, May 1990.

[36] J. Zimmermann, H. Branding, A. P. Buchmann, A. Deutsch, and A. Geppert. Design, implementation and management of rules in an active database system. In *Proceeedings of Database and Expert System Application*, 1996. To be published.

[37] J. Zimmermann, A. Buchmann, and A. Deutsch. The ACT-1 benchmark. Technical report, Technical University Darmstadt, Germany, 1995.

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

Appendix A

The Complete Results for REACH

TEST		CONFIGURATION			
		EMPTY	**SMALL**		
			0.3	**0.6**	**0.9**
$MW1_h$	average	0.03	0.04	0.03	0.04
	std. dev.	0.01	0.02	0.01	0.02
	conf. int.	[0.02,0.05]	[0.02,0.06]	[0.02,0.05]	[0.03,0.06]

Table A.1: The Method Wrapping results (EMPTY and SMALL Database Configurations)

TEST		**CONFIGURATION**		
		MEDIUM		
		0.3	**0.6**	**0.9**
$MW1_h$	average	0.03	0.03	0.03
	std. dev.	0.01	0.02	0.01
	conf. int.	[0.02,0.04]	[0.02,0.05]	[0.02,0.04]

Table A.2: The Method Wrapping results (MEDIUM Database Configuration)

TEST		**CONFIGURATION**		
		LARGE		
		0.3	**0.6**	**0.9**
$MW1_h$	average	0.03	0.03	0.03
	std. dev.	0.01	0.02	0.01
	conf. int.	[0.03,0.05]	[0.02,0.05]	[0.03,0.04]

Table A.3: The Method Wrapping results (LARGE Database Configuration)

TEST		**CONFIGURATION**			
		EMPTY	**SMALL**		
			0.3	**0.6**	**0.9**
$PED1_h$	average	2.04	2.19	2.27	2.15
	std. dev.	0.05	0.14	0.05	0.06
	conf. int.	[1.97,2.15]	[1.87,2.40]	[2.19,2.38]	[2.08,2.27]
$PED2_c$	average	12.72	13.79	14.67	14.19
	std. dev.	0.7	1.01	1.18	1.07
	conf. int.	[11.67,13.67]	[11.32,15.44]	[12.75,17.37]	[12.12,15.31]
$PED3_c$	average	318	1005	1062	5447
	std. dev.	4	203	1907	13219
	conf. int.	[313,325]	[333,1097]	[1036,1093]	[993,45106]

Table A.4: The Primitive Event Detection results (EMPTY and SMALL Database Configurations)

TEST		**CONFIGURATION**		
		MEDIUM		
		0.3	**0.6**	**0.9**
$PED1_h$	average	2.31	2.57	3.07
	std. dev.	0.87	0.16	0.10
	conf. int.	[0.37,3.31]	[2.44,2.63]	[2.98,3.24]
$PED2_c$	average	13.37	13.22	15.03
	std. dev.	1.81	0.75	1.17
	conf. int.	[11.17,15.98]	[12.02,13.85]	[13.75,17.34]
$PED3_c$	average	10069	20921	42758
	std. dev.	1268	23190	70541
	conf. int.	[329,23601]	[4484,70043]	[4615,207086]

Table A.5: The Primitive Event Detection results (MEDIUM Database Configuration)

TEST		**CONFIGURATION**		
		LARGE		
		0.3	**0.6**	**0.9**
$PED1_h$	average	3.50	3.80	3.70
	std. dev.	0.20	0.03	0.12
	conf. int.	[3.32,3.87]	[3.77,3.84]	[3.61,3.99]
$PED2_c$	average	16.97	17.70	17.37
	std. dev.	0.82	0.65	0.58
	conf. int.	[15.88,18.35]	[17.28,19.00]	[16.71,18.52]
$PED3_c$	average	35436	46321	74865
	std. dev.	32020	16039	77360
	conf. int.	[10572,97230]	[30302,73728]	[9013,222630]

Table A.6: The Primitive Event Detection results (LARGE Database Configuration)

TEST		**CONFIGURATION**			
		EMPTY	**SMALL**		
			0.3	**0.6**	**0.9**
$CED1_h$	average	3.50	3.72	3.77	3.56
	std. dev.	0.10	0.21	0.21	0.12
	conf. int.	[3.35,3.73]	[3.51,4.17]	[3.65,4.40]	[3.40,3.82]
$CED2_h$	average	4.16	4.31	4.30	4.37
	std. dev.	0.14	0.18	0.09	0.27
	conf. int.	[3.99,4.51]	[4.08,4.71]	[4.17,4.46]	[4.11,5.05]
$CED3_h$	average	3.60	3.68	3.69	3.56
	std. dev.	0.10	0.12	0.14	0.11
	conf. int.	[3.46,3.76]	[3.61,3.80]	[3.50,3.95]	[3.38,3.82]
$CED4_h$	average	4.69	4.86	4.84	4.84
	std. dev.	0.20	0.10	0.09	0.10
	conf. int.	[4.50,5.24]	[4.62,4.99]	[4.74,4.99]	[4.70,5.03]
$CED5_h$	average	4.73	4.87	4.88	4.79
	std. dev.	0.10	0.15	0.19	0.20
	conf. int.	[4.59,4.94]	[4.62,5.24]	[4.59,5.25]	[4.56,5.33]

Table A.7: The Composite Event Detection results (EMPTY and SMALL Database Configurations)

TEST		**CONFIGURATION**		
		MEDIUM		
		0.3	**0.6**	**0.9**
$CED1_h$	average	4.01	3.82	4.45
	std. dev.	0.22	0.16	0.14
	conf. int.	[3.70,4.23]	[3.77,3.94]	[4.25,4.67]
$CED2_h$	average	4.48	4.51	5.31
	std. dev.	0.44	0.15	0.11
	conf. int.	[3.95,5.24]	[4.44,4.58]	[5.14,5.44]
$CED3_h$	average	3.97	3.91	4.53
	std. dev.	0.36	0.12	0.15
	conf. int.	[3.39,4.24]	[3.75,4.08]	[4.33,4.71]
$CED4_h$	average	5.17	5.21	6.15
	std. dev.	0.45	0.16	0.14
	conf. int.	[4.52,5.62]	[5.04,5.51]	[5.96,6.25]
$CED5_h$	average	5.11	5.12	6.13
	std. dev.	0.43	0.13	0.10
	conf. int.	[4.51,5.49]	[5.02,5.40]	[5.98,6.25]

Table A.8: The Composite Event Detection results (MEDIUM Database Configuration)

TEST		**CONFIGURATION**		
		LARGE		
		0.3	**0.6**	**0.9**
$CED1_h$	average	5.42	5.49	5.35
	std. dev.	0.32	0.16	0.11
	conf. int.	[5.00,5.97]	[5.22,5.71]	[5.22,5.49]
$CED2_h$	average	6.43	6.80	6.39
	std. dev.	0.21	0.39	0.39
	conf. int.	[6.16,6.71]	[6.08,7.28]	[6.02,7.08]
$CED3_h$	average	5.52	5.81	5.50
	std. dev.	0.19	0.36	0.26
	conf. int.	[5.36,5.88]	[5.12,6.11]	[5.26,5.92]
$CED4_h$	average	7.47	7.50	7.49
	std. dev.	0.44	0.42	0.24
	conf. int.	[6.84,8.05]	[6.75,7.99]	[7.19,7.81]
$CED5_h$	average	7.02	7.36	7.68
	std. dev.	0.31	0.25	0.2
	conf. int.	[6.61,7.42]	[6.98,7.74]	[7.38,8.07]

Table A.9: The Composite Event Detection results (LARGE Database Configuration)

TEST		**CONFIGURATION**			
		EMPTY	**SMALL**		
			0.3	**0.6**	**0.9**
$RF1_c$	average	10.58	12.21	12.38	12.79
	std. dev.	0.24	0.10	0.38	0.64
	conf. int.	[10.24,10.94]	[10.35,14.24]	[11.80,12.96]	[11.72,14.02]
$RF2_h$	average	1.68	1.92	1.94	1.92
	std. dev.	0.01	0.03	0.01	0.01
	conf. int.	[1.63,1.72]	[1.82,1.94]	[1.91,1.98]	[1.91,1.99]
$RF3_h$	average	2.38	2.61	2.65	2.66
	std. dev.	0.15	0.28	0.18	0.22
	conf. int.	[2.00,2.50]	[2.01,2.89]	[2.16,2.82]	[2.11,2.88]
$RF4_h$	average	1.50	2.04	2.02	1.91
	std. dev.	0.04	0.24	0.08	0.02
	conf. int.	[1.46,1.55]	[1.48,2.56]	[1.96,2.10]	[1.89,1.95]
$RF5_h$	average	1.46	2.44	2.44	2.33
	std. dev.	0.05	0.32	0.07	0.15
	conf. int.	[1.39,1.54]	[2.01,2.97]	[2.33,2.55]	[2.21,2.78]
$RF6_h$	average	2.40	3.02	3.04	2.96
	std. dev.	0.03	0.20	0.04	0.19
	conf. int.	[2.34,2.44]	[2.39,3.21]	[2.97,3.12]	[2.82,3.51]

Table A.10: The Rule Firing results (EMPTY and SMALL Database Configurations)

TEST		**CONFIGURATION**		
		MEDIUM		
		0.3	**0.6**	**0.9**
$RF1_c$	average	13.37	13.77	14.64
	std. dev.	1.12	0.70	0.39
	conf. int.	[11.67,14.55]	[13.24,15.18]	[14.11,15.06]
$RF2_h$	average	2.48	2.47	2.60
	std. dev.	0.09	0.08	0.04
	conf. int.	[2.41,2.62]	[2.33,2.54]	[2.52,2.64]
$RF3_h$	average	2.54	2.75	3.08
	std. dev.	0.31	0.05	0.10
	conf. int.	[2.42,2.68]	[2.72,2.78]	[3.01,3.20]
$RF4_h$	average	2.11	2.53	2.70
	std. dev.	0.11	0.18	0.05
	conf. int.	[2.01,2.22]	[2.31,2.76]	[2.68,2.74]
$RF5_h$	average	2.21	2.71	3.17
	std. dev.	0.63	0.03	0.02
	conf. int.	[1.45,2.86]	[2.67,2.75]	[3.11,3.31]
$RF6_h$	average	3.28	3.84	4.05
	std. dev.	0.54	0.04	0.08
	conf. int.	[2.43,3.79]	[3.81,3.86]	[3.97,4.21]

Table A.11: The Rule Firing results (MEDIUM Database Configuration)

TEST		**CONFIGURATION**		
		LARGE		
		0.3	**0.6**	**0.9**
$RF1_c$	average	16.48	16.54	16.84
	std. dev.	0.46	0.75	0.35
	conf. int.	[15.86,16.95]	[15.51,17.53]	[16.59,17.44]
$RF2_h$	average	3.31	3.20	3.26
	std. dev.	0.18	0.09	0.16
	conf. int.	[3.15,3.66]	[3.10,3.30]	[3.08,3.47]
$RF3_h$	average	3.95	3.71	4.26
	std. dev.	0.16	0.02	0.07
	conf. int.	[3.82,4.19]	[3.69,3.73]	[4.12,4.31]
$RF4_h$	average	2.48	2.59	2.53
	std. dev.	0.07	0.12	0.18
	conf. int.	[2.41,2.59]	[2.47,2.70]	[2.32,2.78]
$RF5_h$	average	3.03	3.84	3.48
	std. dev.	0.18	0.54	0.06
	conf. int.	[2.87,3.28]	[3.51,4.93]	[3.41,3.55]
$RF6_h$	average	4.09	4.58	4.37
	std. dev.	0.09	0.28	0.01
	conf. int.	[3.99,4.21]	[4.36,5.12]	[4.36,4,38]

Table A.12: The Rule Firing results (LARGE Database Configuration)

TEST		**CONFIGURATION**			
		EMPTY	**SMALL**		
			0.3	**0.6**	**0.9**
$EPP1_h$	average	2.12	2.86	2.84	2.75
	std. dev.	0.14	0.37	0.18	0.05
	conf. int.	[2.03,2.50]	[2.17,3.47]	[2.70,3.35]	[2.69,2.84]
$EPP2_h$	average	2.84	3.07	3.05	2.96
	std. dev.	0.19	0.10	0.07	0.06
	conf. int.	[2.75,3.40]	[2.81,3.17]	[2.95,3.19]	[2.91,3.10]
$EPP3_h$	average	3.40	3.44	3.84	3.57
	std. dev.	0.19	0.02	0.10	0.08
	conf. int.	[3.22,3.93]	[3.38,3.44]	[3.80,3.91]	[3.51,3.61]

Table A.13: The Event Parameter Passing results (EMPTY and SMALL Database Configurations)

TEST		**CONFIGURATION**		
		MEDIUM		
		0.3	**0.6**	**0.9**
$EPP1_h$	average	2.89	3.05	3.58
	std. dev.	0.48	0.05	0.15
	conf. int.	[2.10,3.39]	[3.00,3.13]	[3.49,3.90]
$EPP2_h$	average	3.44	3.73	4.06
	std. dev.	0.42	0.10	0.07
	conf. int.	[2.82,3.87]	[3.65,3.87]	[3.99,4.19]
$EPP3_h$	average	3.66	3.96	4.53
	std. dev.	0.37	0.13	0.20
	conf. int.	[3.33,4.17]	[3.88,4.11]	[4.35,4.67]

Table A.14: The Event Parameter Passing results (MEDIUM Database Configuration)

TEST		**CONFIGURATION**		
		LARGE		
		0.3	**0.6**	**0.9**
$EPP1_h$	average	3.81	3.86	3.78
	std. dev.	0.28	0.06	0.05
	conf. int.	[3.39,4.17]	[3.80,3.95]	[3.73,3.85]
$EPP2_h$	average	5.16	5.18	5.90
	std. dev.	0.14	0.12	0.06
	conf. int.	[5.03,5.43]	[5.10,5.38]	[5.82,5.98]
$EPP3_h$	average	5.81	5.93	6.32
	std. dev.	0.10	0.32	0.17
	conf. int.	[5.66,5.93]	[5.59,6.45]	[6.12,6.49]

Table A.15: The Event Parameter Passing results (LARGE Database Configuration)

TEST		**CONFIGURATION**			
		EMPTY	**SMALL**		
			0.3	**0.6**	**0.9**
$GC1_c$	average	19712	19423	19785	26483
	std. dev.	1780	1238	1032	22577
	conf. int.	[16953,22963]	[17829,21327]	[17926,72951]	[17677,92111]

Table A.16: The Garbage Collection results (EMPTY and SMALL Database Configurations)

TEST		**CONFIGURATION**		
		MEDIUM		
		0.3	**0.6**	**0.9**
$GC1_c$	average	18981	26010	48674
	std. dev.	2582	34177	36241
	conf. int.	[16579,24199]	[22206,28280]	[31481,129709]

Table A.17: The Garbage Collection results (MEDIUM Database Configuration)

TEST		**CONFIGURATION**		
		LARGE		
		0.3	**0.6**	**0.9**
$GC1_c$	average	102149	171610	272112
	std. dev.	62925	75462	89994
	conf. int.	[43224,169073]	[37115,228039]	[40254,449263]

Table A.18: The Garbage Collection results (LARGE Database Configuration)

TEST		**CONFIGURATION**			
		EMPTY	**SMALL**		
			0.3	**0.6**	**0.9**
$RA1_c$	average	4.48	4.53	4.60	4.57
	std. dev.	1.03	1.22	1.76	0.37
	conf. int.	[3.79,6.88]	[3.83,6.95]	[3.22,7.08]	[4.22,5.04]
$RA2_c$	average	2.18	2.13	2.25	2.27
	std. dev.	0.05	0.07	0.02	0.10
	conf. int.	[2.15,2.29]	[2.02,2.21]	[2.22,2.27]	[2.17,2.42]
$RA3_c$	average	2.07	2.05	2.17	2.08
	std. dev.	0.07	0.06	0.18	0.04
	conf. int.	[1.97,2.19]	[1.99,2.16]	[2.00,2.27]	[2.03,2.16]
$RA4_c$	average	2.24	2.14	2.22	2.07
	std. dev.	0.15	0.12	0.02	0.05
	conf. int.	[2.09,2.50]	[2.03,2.33]	[2.18,2.23]	[2.02,2.15]

Table A.19: The Rule Administration results (EMPTY and SMALL Database Configurations)

TEST		**CONFIGURATION**		
		MEDIUM		
		0.3	**0.6**	**0.9**
$RA1_c$	average	4.63	4.71	5.39
	std. dev.	0.33	0.05	0.17
	conf. int.	[4.20,5.01]	[4.67,4.78]	[5.23,5.57]
$RA2_c$	average	2.22	2.21	2.41
	std. dev.	0.13	0.15	0.16
	conf. int.	[2.07,2.39]	[2.01,2.64]	[2.26,2.58]
$RA3_c$	average	2.17	2.16	2.40
	std. dev.	0.04	0.05	0.01
	conf. int.	[2.12,2.21]	[2.10,2.21]	[2.39,2.41]
$RA4_c$	average	2.07	2.48	2.58
	std. dev.	0.10	0.06	0.03
	conf. int.	[1.97,2.20]	[2.41,2.58]	[2.56,2.61]

Table A.20: The Rule Administration results (MEDIUM Database Configuration)

TEST		**CONFIGURATION**		
		LARGE		
		0.3	**0.6**	**0.9**
$RA1_c$	average	7.64	7.92	8.85
	std. dev.	0.12	0.08	0.21
	conf. int.	[7.51,7.76]	[7.83,8.02]	[8.61,8.99]
$RA2_c$	average	2.23	2.34	3.71
	std. dev.	0.02	0.13	0.17
	conf. int.	[2.21,2.25]	[2.23,2.45]	[3.52,3.90]
$RA3_c$	average	2.52	2.39	2.55
	std. dev.	0.43	0.20	0.35
	conf. int.	[2.08,2.95]	[2.27,2.56]	[2.21,2.81]
$RA4_c$	average	2.46	2.51	2.66
	std. dev.	0.02	0.22	0.14
	conf. int.	[2.44,2.48]	[2.33,2.69]	[2.42,2.79]

Table A.21: The Rule Administration results (LARGE Database Configuration)

Appendix B

A Sample Event Creation Program

As mentioned in Section 5.2, REACH does not support the definition of stand-alone event types. In order to circumvent this exceptional design decision, we wrote small programs, and created the benchmark events (Figure 4.2) manually.

Here, we present the event creation program written to create the event type CED1 (a composite event; i.e., a sequence of two abstract events) for illustration.

```
// ----------------------------------------------------------------
// UGUR CETINTEMEL
// Implementation of OBJECTIVE on REACH
// ----------------------------------------------------------------
// definition of event CED1
// event CED1{
//               event:     ABSTRACT(CED11) then ABSTRACT(CED12)
// }
// ----------------------------------------------------------------

#include "Reach.hh"
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char** argv)
{
   RCREACH              rc;

   CompositeDetector*   compDetectorCED1;
   ThenNodeRecent*      thenNode;

   AbstractDetector*    abstractCED11;
   CompositionStart*    abstractCED11Start;

   AbstractDetector*    abstractCED12;
   CompositionStart*    abstractCED12Start;

   Rule*                CED1;

#ifdef __O3DB__
   int                  db = OODB_DEFAULT_SG;
#endif

   if (argc > 2)
   {
      printf("usage: %s [db]\n", argv[0]);
      exit(-1);
   }

   if (argc == 2)
      db = atoi(argv[1]);

   SET_TRACELEVEL;
   TRACE_(RULE_ADMIN_REACH, BEGIN_, argv[0], NULL);

   Reach::init(db, B_FALSE);
```

```
ReachTransaction::begin();

// create detector for abstract event CED1
// ----------------------------------------
rc = AbstractDetector::create(&abstractCED11, "CED11");

if ((rc != REACHOK) && (rc != ABSTRACTDETECTOR_CREATE_NAME_EXISTS))
   MsgReach::print(rc, fatalError, "abstract CED11");


// create detector for abstract event CED2
// ----------------------------------------
rc = AbstractDetector::create(&abstractCED12, "CED12");

if ((rc != REACHOK) && (rc != ABSTRACTDETECTOR_CREATE_NAME_EXISTS))
   MsgReach::print(rc, fatalError, "abstract CED12");

// create the composite event detector for sequence
// ------------------------------------------------

compDetectorCED1= CompositeDetector::create(B_TRUE);

// create the then  node
// ------------------------------------------------

thenNode = ThenNodeRecent::create(NULL, NULL);

if ((rc = CompositionStart::create(&abstractCED11Start,
                                   compDetectorCED1,
                                   thenNode,
                                   cn_Left,
                                   abstractCED11)) != REACHOK)
  {
    MsgReach::print(rc, fatalError, NULL);
  }

if ((rc = CompositionStart::create(&abstractCED12Start,
                                   compDetectorCED1,
                                   thenNode,
                                   cn_Right,
                                   abstractCED12)) != REACHOK)
  {
    MsgReach::print(rc, fatalError, NULL);
  }

// set the composite event definition compDetector CED1
```

```
   // ------------------------------------------------

   compDetectorCED1->setCompositionNodeTree(thenNode);

   ReachTransaction::commit();
   Reach::stop();

   TRACE_(RULE_ADMIN_REACH, END_, argv[0], NULL);

   return 0;
}
```

Appendix C

A Sample Rule Creation Program

As mentioned in Section 5.2, the rule compiler of REACH was not able to support the definition of rules with non-immediate coupling modes at the time of benchmark implementation. Therefore, all the benchmark rules depicted in Figure 4.3 are created by hand.

Here, we give a sample program written to create rule EPP1 for illustration.

```
// ----------------------------------------------------------------
// UGUR CETINTEMEL
// Implementation of OBJECTIVE on REACH
// ----------------------------------------------------------------
// definition of rule EPP1
// rule EPP1{
//              event:     classEPP1->setData(double d1, double d2);
//              condition: Imm, (d1 < d2)
//              action:    Imm, classEPP1->setMinData();
//              priority:  5
// }
// ----------------------------------------------------------------


#include "Reach.hh"
#include <stdio.h>


int main(int argc, char** argv)
{
   RCREACH              rc;
   MethodDetector*      mdEPP1;
   ArgumentDescriptor*  desc;
   Rule*                EPP1;

#ifdef __O3DB__
   int                  db = OODB_DEFAULT_SG;
#endif


   if (argc > 2)
   {
      printf("usage: %s [db]\n", argv[0]);
      exit(-1);
   }

   if (argc == 2)
      db = atoi(argv[1]);

   SET_TRACELEVEL;
   TRACE_(RULE_ADMIN_REACH, BEGIN_, argv[0], NULL);

   Reach::init(db, B_FALSE);
   ReachTransaction::begin();
```

```
// describes the whole argument set
desc = new ArgumentDescriptor(3);

// 1st parameter: double
desc->add(at_value, sizeof(double));

// 2nd parameter: double
desc->add(at_value, sizeof(double));


// create the method detector for the method
// ----------------------------------------------------------------
rc = MethodDetector::create(
     &mdEPP1,                                // out: method det.
     "classEPP1",                            // class name
     "setData",                              // method name
     PTR_SIZE+sizeof(double)+sizeof(double), // bag size
     desc,                                   // argument descriptor
     "setData__classEPP1_vFdd_REACH_A__");

if ((rc != REACHOK) && (rc != METHODDETECTOR_CREATE_NAME_EXISTS))
   MsgReach::print(rc, fatalError,
              "setData__classEPP1_vFdd_REACH_A__");


// create rule EPP1
// ----------------------------------------------------------------
rc = RuleIncludes::appendFile("\"objective.h\"");
if ((rc != REACHOK) && (rc != INCL_FILE_ALREADY_INSERTED))
   MsgReach::print(rc, error, "objective.h");

rc = Rule::create(&EPP1,                  // out: rule object
     "/OBJECTIVE/EPP1",            // rule name
     mdEPP1,                 // detector
     cm_Imm,                 // coupling mode event-condition
     cm_Imm,                 // coupling mode condition-action
     5,                      // priority
     B_TRUE,                 // rule is enabled
     "objectEPP1->setData(d1, d2)",   // event def
     "d1 < d2",                       // cond
     "objectEPP1->setMinData;",       // action
                                      // variables
     "objectEPP1;classEPP1*;;setData;boolean_t;;d1;double;;
      d2;double;;;;",
     NULL,                            // methods
```

```
//-------------------
// action function
//-------------------
"boolean_t _OBJECTIVE_EPP1 (Event* event)\n\
{\n\
   RULE_ACTION(\"/OBJECTIVE/EPP1\", NULL);\n\
   return B_TRUE;\n\
\n\
   classEPP1* objectEPP1;\n\
   ((MethodEvent*) event)->getArgument(0, &objectEPP1,
                                       sizeof(objectEPP1));\n\
   objectEPP1->setMinData();\n\
}\n",

   //-------------------
   // condition function
   //-------------------
"boolean_t _OBJECTIVE_EPP1_ (Event* event)\n\
{\n\
   double     d1;\n\
   double     d2;\n\
   ((MethodEvent*) event)->getArgument(1, &d1, sizeof(d1));\n\
   ((MethodEvent*) event)->getArgument(2, &d2, sizeof(d2));\n\
   if (d1 < d2)\n\
     return B_TRUE;\n\
   return B_FALSE;\n\
}\n");

   if (rc != REACHOK)
      MsgReach::print(rc, error, "/OBJECTIVE/EPP1");

   ReachTransaction::commit();
   Reach::stop();

   TRACE_(RULE_ADMIN_REACH, END_, argv[0], NULL);

   return 0;
}
```