14

REACH

Jürgen Zimmermann
Alejandro P. Buchmann

ABSTRACT
REACH is an active OODBMS that was developed as a platform to experiment both with the issues arising from the implementation of advanced active functionalities, and as a platform for the development of applications that are potential beneficiaries of active database technology. To achieve the former, we chose an experimental OODBMS, Texas Instruments' OpenOODB, for which the source code was available to us, and tried to implement the full range of active functionality with multiple coupling modes, a complete event algebra, full transaction management, and support features, such as garbage collection of events. To achieve the latter goal, a rich set of tools was implemented to facilitate the use of the system by application programmers. This chapter gives a brief overview of the REACH system implemented on OpenOODB. Currently, efforts are underway to port the REACH functionality to ObjectStore.

14.1 Introduction

The REACH project set out to build an active OODBMS with full active functionality [BBKZ93, BBKZ92, BZBW95]. This was considered important since previous projects had set ambitious goals and specified a broad range of functionality [DBB+88], but no robust system with full active functionality that could be used for actual implementation of applications was available. The initial goal was to use a commercial OODBMS and build REACH on top of it to speed up development and to benefit from the stability of a commercial platform. However, soon the limitations of building on top of a closed system became evident. Problems appeared when trying to implement some of the coupling modes, and when modifying the pre-processor. Therefore, it was decided in 1993 to begin a new implementation on top of Texas Instruments' OpenOODB [WBT92]. OpenOODB is an experimental platform that was conceived as an extendible OODBMS in which individual functions are specified in a modular way as policy managers that can be exchanged or extended as needed. The policy managers can be invoked in response to low-level events. This appeared to be philosophically very close to the paradigm proposed by active databases

and it was decided to use OpenOODB as the implementation platform. This was particularly attractive, as we were able to obtain the source code and were included in the list of Alpha test sites. The use of OpenOODB proved to be a fortunate choice in that it gave us a good head start. It was particularly useful to have access to the source code of the precompiler that was modified to include the wrappings for method events. However, OpenOODB was built on top of the Exodus storage manager 2.2 [CDRS86], and since Open OODB used extensively the Exodus transaction manager with its page locking and recovery mechanisms, it turned out quite difficult to implement necessary changes to the transaction manager, especially parallel nested transactions. In addition, since OpenOODB is implemented as a client to the Exodus server, the whole REACH functionality is only available on the client side. On balance, the use of OpenOODB gave us a significant initial boost but did not live up completely to our expectations, in part also because of the growing pains of an experimental system.

In spite of the limitations encountered, REACH implements a fairly complete range of active functionality. The event algebra is a superset of the HiPAC event algebra and was adapted from the SAMOS project [GD93a, GD93b]. The six coupling modes provided are also a superset of the HiPAC modes [HLM88] and the most comprehensive set of coupling modes implemented to our knowledge. Special emphasis was placed in the efficient and specialized implementation of event detectors, the correct composition of events relative to transaction boundaries, the passing of events and parameters, the garbage collection of semi-composed events, and the possibility for future distribution, i.e., no design decision should preclude future use in a distributed environment.

The second major area of concern was to provide a set of tools for the administration of rules and as a support for the application programmer. These tools include static termination checkers, detailed event histories, rule browsers, a graphical interface for rule specification by the naive user, and the organization of the rule space in analogy to the directory structure of the UNIX file system [ZBB+96].

In this chapter section 14.2 briefly describes OpenOODB and its philosophy. Section 14.3 describes the functionality of REACH and how it was realized, section 14.4 describes the tools, and section 14.5 summarizes the features of REACH in accordance with the criteria formulated in Chapter 1. Section 14.6 presents conclusions and lessons learned.

14.2 The OpenOODB Platform

Texas Instruments' OpenOODB is an extensible OODBMS whose computational model transparently extends the behavior of operations in application programming languages. Invocations of these operations are examples

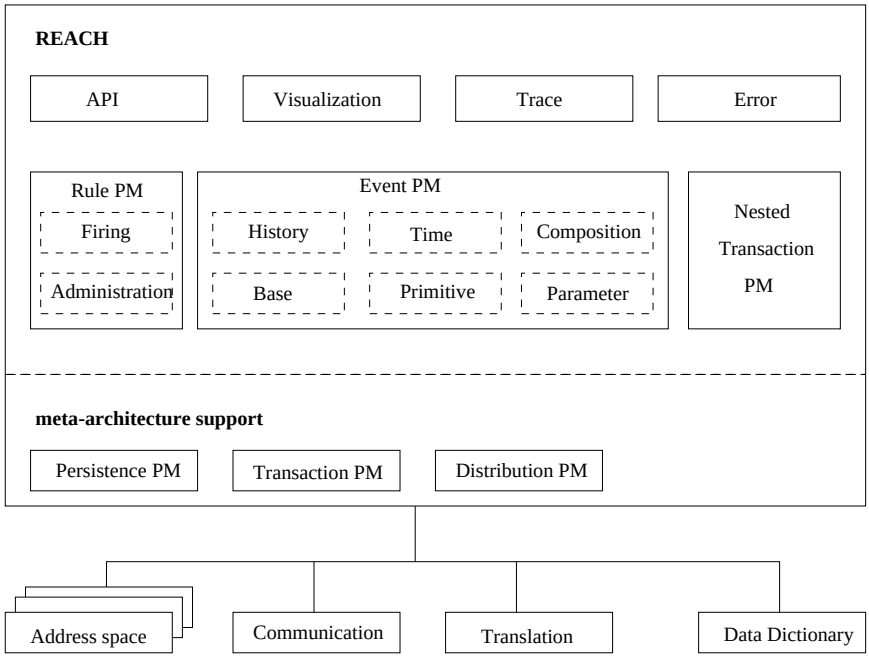

FIGURE 14.1. OpenOODB architecture with REACH extensions.

of primitive method events. OpenOODB uses the C++ type system.

OpenOODB consists of two layers: the lower layer consists of support modules for address space management, communication, translation, and the data dictionary. The upper level is the meta-architecture module that implements the computational model by providing events, sentries, and policy manager interfaces. The meta-architecture module provides the extensibility mechanisms for OpenOODB and plays the role of a software bus into which the database components can be plugged. Each database component is realized as a policy manager. Figure 14.1 shows the OpenOODB architecture with the REACH extensions.

The meta-architecture is philosophically close to the active database paradigm. Any operation performed within the context of a programming language can be an event. A sentry mechanism tracks primitive events and invokes the appropriate policy manager (PM) which implements the extended behavior. There must be at least one policy manager for each database function, and the Open OODB architecture provides for the possibility of exchanging a given policy manager, e.g., the flat transaction PM in favor of a nested transaction PM. While philosophically clean and attractive, it is not easy to exchange policy managers because of interactions and dependence on functionality of the Exodus storage manager or to add

a new policy manager to OpenOODB, e.g., for distribution.

14.3 REACH Goals, Design Principles, and Implementation Decisions

The long-term goal of REACH was to support complex applications and to provide a stable testbed for applications using active capabilities, to be extendible to open environments, and to allow for applications with timing constraints.

To satisfy the goals that were set for REACH, we formulated some design principles that can be summarized as follows:

- provide dynamic rule specification through orthogonality of monitoring and type,
- provide a rich event set with a clear definition of event semantics,
- provide flexible rule execution through a rich set of coupling modes,
- provide efficient rule invocation through fast basic event detection and low composition overhead,
- do not preclude future use in a distributed environment through central components that can become bottlenecks, and
- provide the necessary maintenance mechanisms for long-term stable operation.

Dynamic rule specification is essential in making an active OODBMS useful for application development. The orthogonality of monitoring and type makes it possible to treat all classes and their methods in a uniform manner. Therefore, it is not necessary to know at the time a class is defined which method event will eventually be relevant for a rule. Rules can be defined independently of the object classes and subscribe to a given event type. Since all methods are uniformly wrapped by the precompiler, no recompilation is needed when a new rule is defined.

14.3.1 Event Detection and Composition

It was not a goal of the REACH project to define new event types and event algebras. Therefore, the event hierarchy that was used is similar to that of Sentinel [CKAK94] and SAMOS [GD93a, GD93b, GD94]. It includes method events; flow-control events that include Begin, End, Commit and Abort of transactions; and absolute, relative, periodic, and aperiodic time events. State change events are not implemented, as they would require a

different basic event detection mechanism. However, in an attempt to allow for time-constrained processing in an inherently non-real-time environment, we defined the notion of milestone events. A milestone event is raised when a transaction passes a certain point. If the corresponding milestone is not reached by a certain time relative to a deadline, a rule specifying an alternative action can be invoked. Detached exclusive coupling is required to avoid two different results from becoming valid. In detached exclusive coupling mode, the rule is executed in a separate transaction that commits and thus becomes visible only when the triggering transaction aborts. As to event algebra, REACH implements the three operators proposed in HiPAC, sequence, disjunction, and closure–with their original semantics [DBM88]. In addition, it implements conjunction, negation, and history as defined in the SAMOS project with the same semantics [GD93a, GD93b, GD94].

To allow for future use in a distributed environment, no single central component should act as a bottleneck. This is particularly true for event detectors/composers and the logging of event histories. Therefore, specialized event detectors exist for each type of composite event, and for each instance of a composite event, a separate event hierarchy is constructed. The event detectors create an event object. Logging of event histories occurs in parallel by writing the event objects into partial logs with asynchronous merging of the partial logs.

A stable active DBMS requires maintenance facilities, such as garbage collection, to remove semicomposed events or event-log consolidation. Through the implementation of a separate event graph for each composite event, it is relatively easy to eliminate events for which it becomes obvious that they will not be completed. Through careful definition of the scope of an event, it is possible to eliminate useless events and their parameters.

14.3.2 Rule Execution and Coupling Modes

To provide an adequate support for a wide variety of applications, it was decided to implement as complete a set of coupling modes as possible. REACH implements the coupling modes immediate, deferred, detached, and three variants of detached with causal dependencies, parallel, sequential, and exclusive [BBKZ93]. Detached parallel allows for parallel evaluation of a rule, but termination of rule execution depends on the commit of the spawning transaction. Detached sequential requires the beginning of rule execution to be delayed until the spawning transaction commits. Detached exclusive allows for parallel execution of a rule, but the rule may only commit if the spawning transaction aborts. This coupling mode is intended for the implementation of contingency plans through ECA rules. The coupling of the condition and of the action part are specified separately.

To speed up rule invocation, the event detectors are specialized. A separate event detector exists for each event type. The event detectors for primitive events know which rules are directly fired by that event and whether a

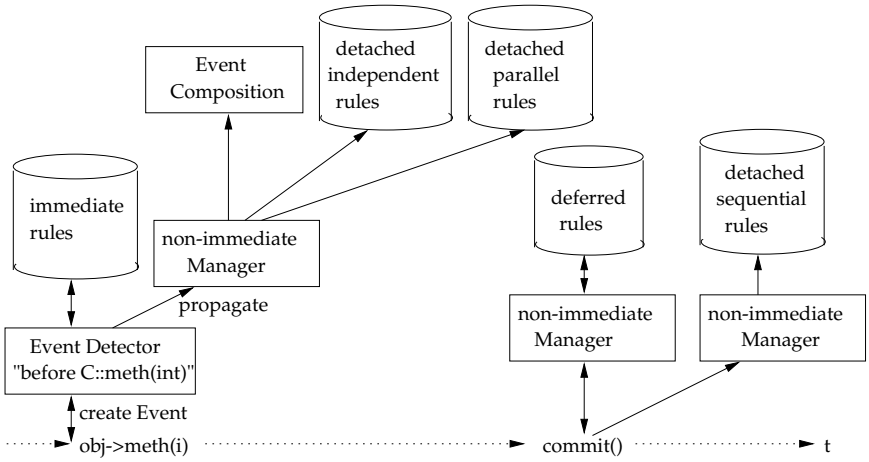

FIGURE 14.2. Rule execution.

complex event detector consumes a primitive event. This reduces the number of indirections. A specialized event detector exists also for each type of composite event, and for each instance of a composite event a separate event hierarchy is constructed. Whenever a new rule is defined and compiled, the corresponding event detector is created. An event detector also includes the necessary data structures for the parameters of the event and the necessary instructions for copying the state of the relevant objects depending on the coupling mode. If an event detector already exists because the triggering event of the new rule already is used to trigger an existing rule, the new rule only subscribes to that event detector. Subscription means that the corresponding rule is added to the set of rules that need to be fired by that event detector. For the condition and the action, two functions are generated and stored in a shared library.

Figure 14.2 illustrates the processing of events and execution of the triggered rules at runtime. A method detector traps a method event. Rules that are fireable by the method event in immediate coupling mode are fired, and the trapped method event is propagated to the non-immediate manager. The non-immediate manager either passes the method event to the corresponding composite event detector for composition, or fires the rules that are to execute in detached or detached parallel coupling modes. REACH does not allow composite events to trigger rules in immediate mode. The reason is that a composite event contains several basic events with different timestamps and parameters (see section 14.3.3). Since the parameters of the older events could be invalid or out of scope, immediate event composition could result in runtime errors during rule execution when the event parameters are accessed. There also exists a performance reason. If rules that are triggered by a composite event can be executed in immediate coupling

mode, the execution of any transaction must be interrupted while event composition occurs. This interruption lasts until it is clear that no rule must be fired by a just-completed composite event. Detached independent and detached parallel rules are executed in separate child processes. Whenever the transaction manager raises the End-Of-Transaction (EOT) event, i.e., wants to begin the commit process, the EOT event is signaled to the non-immediate manager to trigger the deferred rules. After execution of the deferred rules and completion of the commit process, the non-immediate manager is invoked once more by passing it the commit event to fire the detached sequential rules.

A separate temporal event detector handles the detection of temporal events and the firing of the corresponding rules and passing of temporal events to the event composers.

In REACH, the rules that are executed in immediate or in deferred coupling mode are executed as closed nested transactions. The subtransactions commit through the top with the commit and abort dependencies of Moss-style closed nested transactions [Mos85]. Subtransactions execute sequentially. When rules are executed in a detached coupling mode the possibility of a locking conflict between the spawning transaction and the rule transaction exists. In particular, if the rule is executed in a detached parallel mode that requires the spawning transaction to commit for the detached rule to commit, hidden deadlocks are possible. Since the underlying Exodus storage system doesn't know how to handle rule transactions, this must be solved at the REACH level.

Conflicts between the spawning transaction and a detached transaction are solved in REACH through the introduction of the notion of strong and weak transactions [Mar95]. Weak transactions are not allowed to wait for a lock. They are aborted as soon as a conflict occurs. Strong transactions are allowed to wait for resources and locks. Rule transactions are always started as weak transactions and therefore cannot cause a deadlock before finishing. However, they may be involved in a hidden deadlock where the rule transaction has finished and is waiting for the spawning transaction to commit. Since a rule transaction could have gained a lock on a resource later needed by the spawning transaction which is waiting for the resource, a hidden deadlock could occur. Since the underlying Exodus storage system cannot handle this situation and recognize a rule transaction, REACH solves the problem by timing out the rule transaction. Detached sequential rules can be executed as strong transactions.

14.3.3 Events: Scoping, Composition, and Parameter Passing

Special attention was given to the correct scoping of events and to the composition of events relative to transaction boundaries. Rules triggered by a single method event can be executed in any coupling mode. Rules fired by a purely temporal event may be executed only as independent detached

rules. If all the events that participate in a composite event originate in a single transaction, all coupling modes are acceptable, but for the runtime handling of parameters mentioned above, immediate coupling mode is disallowed. If the events that make up a composite event originate in multiple transactions, neither immediate nor deferred couplings are allowed since no identification of the spawning transaction is possible. The other four coupling modes are legal with the restriction that all transactions from which primitive events originate must commit in the detached parallel and sequential cases, and all must abort in the detached exclusive mode.

Events are consumed according to one of two consumption policies: chronologically or most recent. In chronologic consumption, the first possible event of a type participates in a composition. Under a most recent policy it is the latest occurrence of an event that participates in a composition. Which policy is used depends on the semantics of an application.

One of the more difficult problems is the correct passing of parameters, particularly the state of objects that must be acted upon. The detectors that detect the events also are responsible for passing the appropriate parameters. These include the object reference to be acted upon, timestamp, and the arguments that make up the signature of the method. Correct parameter passing depends on the transaction execution model.

The condition and action parts of a rule are mapped as ordinary functions that may have parameters of various types. In the case of a method event the parameters are, in addition to the timestamp, a reference to the object on which the method was invoked, all the arguments of the method invocation, and in the case of an after-event, the return value. Pointer variables are always valid for the case of immediate execution. For deferred and detached execution, the pointers may not be valid at the time the condition or action is executed, and dereferencing of the pointer may cause a runtime error. Therefore, in any non-immediate case, the referenced values must be saved. This leads to a copy semantic for parameters that goes beyond the copying of single values. This copy semantic must also be used for event composition since the events that participate in the composition may be raised separately and the composite event is raised at a different time than the component events.

The goal in parameter passing is to support as many data types as possible (ideally all the C++ data types), produce as little overhead as possible at runtime, and provide a simple interface to the rule compiler. These three goals are contradictory and require a compromise. The compromise implemented in REACH consists in supporting simple values (integers, enumerated types, floats, and single characters), pointers to simple values, strings, and persistent objects. Parameters are stored in an object called a non-immediate parameter object that consists of a parameter bag and a pointer array indicating the position of the parameter in the bag.

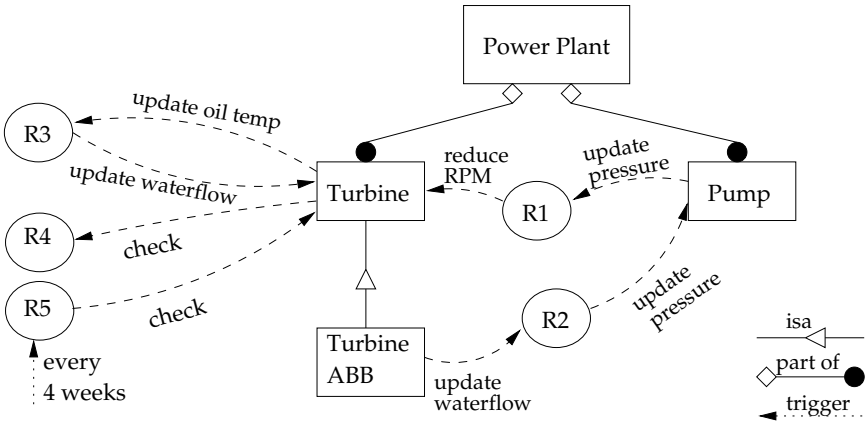

FIGURE 14.3. An OMT+ diagram.

14.4 REACH Environment and Tools

Application developers will use active DBMS technology only if the active DBMS's offer a clean interface and a set of support tools. Since REACH was conceived as a testbed for application development, emphasis was placed on integrating a set of tools [ZBB+96] into the REACH platform. The three main objectives were to provide

- a well-understood design methodology for rules in an active DBMS,
- a user-friendly interface, and
- a toolset to manage and maintain the rules and events.

14.4.1 Modeling Rules with OMT+

Rather than inventing yet another design methodology, we considered it more appropriate to extend a well-known and widely used methodology with a rule component. Therefore, OMT [RBP+91] was extended with the concept of ECA rules, resulting in OMT+ [ZBB+96]. This methodology guarantees an integral approach to modelling objects and rules while minimizing the effort required on the part of the database and application designers to learn new concepts. Figure 14.3 shows an OMT+ diagram.

A rule in an active OODBMS can be viewed as an n-ary relationship between the objects in the event part and those in the action part. The condition part is a constraint. Hence, OMT+ introduces only the notion of a rule (represented as an oval) and a triggering relationship between the classes. The input slots of a rule represent the (composite) events that trigger the rule, and the output slots refer to the effects resulting from the

rule execution. These output slots refer to methods invoked for other object classes, and the name of the method appears next to the edge connecting the rule to the object that is invoked. Every rule must have at least one input slot, but there may be rules without an output slot if the rule does not invoke a method. If the input to a rule is a method event, it will originate in another class represented by a box. However, events may also be temporal events or system events, in which case the triggering event is not connected to any application class.

14.4.2 Rule Language REAL

OMT+ serves as the starting point for rule definition in REAL, the rule language of REACH. REAL treats a rule as an atomic unit with the respective clauses for event, condition, and action. Below, we illustrate a rule from a demonstration scenario of a power plant that was implemented in REACH:

```
#include "Reactor.hh"
rule /powerplant/reactor/r1 {
   prio    5
   decl    River*   river;
           int      level;
           Reactor* reactor named "BlockA";
   event   after river->updateWaterLevel(level);
   cond    imm   level < 3 &&
                 river->getTEMP() > 24.5 &&
                 reactor->getHeatOutput() > 1000000;
   action  imm   reactor->reducePlannedPower(0.05);
};
```

The object for which the method was invoked and all the arguments of the method call are passed. Therefore, these parameters are declared in the `decl` clause, and their types have to be provided in C++ include files. For objects that are not directly referenced in the event clause to be accessible in the condition and/or action part of the rule (e.g., `reactor` in our example), they must be uniquely identifiable. REAL requires these objects to be root objects (with a unique name) in the database system. The example above also shows the declaration of the priority of the rule and the individual coupling modes between event and condition and condition and action.

REAL offers a structured name space modeled along the lines of the UNIX directory structure. This is an advantage when dealing with applications that may have hundreds of rules. In such a case, it becomes difficult to assign manageable, self-explaining names to the rules, and it is even harder to find the rules. In a structured name space, the rules that belong to the

same context can be stored in the same rule directory that is reachable through a hierarchical path. Since rules may not always be attached to exactly one directory, REACH offers the possibility of creating UNIX-like links to create multiple paths to a rule.

14.4.3 Administration Tools

REACH provides a command line interface for managing rule directories. These commands are styled in analogy with their counterparts in UNIX, and include commands for rule manipulation, such as copying, enabling, disabling, changing protection mode, etc.; commands for rule inspection, such as listing, displaying, or plotting; commands for handling of include files and paths, such as appending or removing files and paths; and commands for handling events.

A rather useful tool is the graphical rule browser that displays the whole directory tree or it can be qualified to display only parts of the directory tree. Through clicking on the corresponding rule name, this can be displayed in detail.

In the same graphical manner, a static analysis of the triggering graph can be displayed. The triggering graph is a pessimistic approach that considers potential cycles of rule firings based on the methods invoked in the action part of a rule and the events that trigger the rules defined in the system. REACH exploits the fact that all the necessary information is available explicitly in the event detectors which know which rules are fired by an event. By extracting the method calls from the action part and the corresponding pointers from event detector to the fired rules, one can construct the triggering graph.

Two additional tools that have proven quite useful deserve mention: the trace mode and the event history browser. The static information provided by the triggering graph is not very helpful in understanding the dynamics of applications when rules are fired. The trace mode allows the application designer to step through the execution of rules for debugging purposes. However, since tracing is quite expensive, two modes were implemented with macros: a tracing and debugging mode for developing applications, and an optimized mode without tracing that runs up to twenty times faster. The event history browser is a useful support in tracking errors or unexpected system behavior and allows the system administrator or application developer to retrace the history of events and the rules that were fired by them.

14.5 Summary of Features

The features of the REACH system can be organized according to the parameters outlined in Chapter 1, and are shown in the following tables:

Knowledge Model:

Event	Source	Structure Operation, Transaction, Clock, Behavior Invocation
	Granularity	Member, Subset, Set
	Type	Primitive, Composite
	Role	Mandatory
Condition	Role	Optional
	Context	Bind_E, DB_E, DB_C, DB_A
Action	Options	Structure Operation, Behavior Invocation, Abort, Do Instead, Inform
	Context	Bind_E, DB_E, Bind_C, DB_A

Execution Model:

Condition-Mode	Immediate, Deferred, Detached
Action-Mode	Immediate, Deferred, Detached
Transition granularity	Tuple
Net-effect policy	No
Cycle policy	Recursive, Iterative
Priorities	Numerical
Scheduling	All Sequential
Error handling	Abort, Contingency

Management Model:

Description	Programming Language (OO)
Operations	Activate, Deactivate
Adaptability	Runtime (limited)
Data Model	Object-Oriented

14.6 Conclusion

A prototype of REACH with a demonstration application showing the control of a small subsection of a powerplant was demonstrated during SIGMOD95, CeBIT96, and EDBT96. The development of even such a small application illustrated many of the difficulties that will be faced until active

database systems mature. However, the small application scenario clearly showed that the active database paradigm is well-suited to handling such diverse applications requirements as those posed by plant control and workflow management. Both kinds of applications could be modeled within the same system and could coexist. This indicates that the expressive power of the events, the rule language, and the execution model are adequate.

The passing of parameters is a critical area both for the semantics as well as the performance of REACH. Performance gains for non-immediate rules and event composition are expected through further tuning of the parameter-passing mechanism.

During the development phase of the application, the trace mode and the event history were particularly useful. It was also during this phase that extensive testing of the rule compiler took place and many modifications had to be made.

The use of OpenOODB gave us an initial head start since we could integrate the REACH functionality well into the overall concept. It was also a great help to have a precompiler available that could be modified instead of having to write one from scratch. It turned out to be less than ideal as far as the concurrency control mechanism. Particularly problematic were the Open OODB write-back mechanism of every object that was fetched in a transaction, the duplication of shared objects if they are not root objects, and the restriction to open only one database.

The basic ideas developed in REACH were used to expand the functionality of the commercial middleware Persistence. This system offers a C++ interface to relational databases. Active database functionality was integrated into Persistence to provide an active object-oriented mediator system for consistency enforcement in heterogeneous legacy systems.

Currently efforts are underway to implement the REACH ideas on top of ObjectStore. We are convinced that the design principles described in this chapter are the right approach. At the same time, we are humbled by the detailed work that is still needed to make active OODBMS's into robust tools capable of exploiting the full potential of the technology.

14.7 Acknowledgments

A large project such as the REACH active OODBMS could not be carried out without the help of many people. Special thanks are due to A. Deutsch, who modified the precompiler, wrote the first event detector, and spent many hours testing and debugging. Many students provided valuable input and contributed to portions of REACH in their Diplom-thesis: G. Arens, C. Türker, J. Marschner, W. Dürholt, T. Simon, and T. Kröhl. Other members of the Group who contributed in early discussions are H. Branding and T. Kudraß. J. Blakeley, S. Ford, C. Thompson, and D. Wells, all formerly at

Texas Instruments, were instrumental in helping us to obtain OpenOODB. Last but not least, the input from members of the ACT-NET research network on active databases, funded by the European Union, was always appreciated.

14.8 References

[BBKZ92] A.P. Buchmann, H. Branding, T. Kudraß, and J. Zimmermann. Reach: A Real-Time Active and Heterogeneous Mediator System. *Bulletin of the TC on Database Engineering*, 15, December 1992.

[BBKZ93] H. Branding, A.P. Buchmann, T. Kudraß, and J. Zimmermann. Rules in an Open System: The Reach Rule System. In M. Williams N. Paton, editor, *Rules in Database Systems, Proc. 1st Intl. Workshop on Rules in Database Systems*, Edinburgh, 1993.

[BZBW95] A.P. Buchmann, J. Zimmermann, J. Blakeley, and D. Wells. Building an Integrated Active OODBMS: Requirements, Architecture, and Design Decisions. In *Proc. 11th Intl. Conference on Data Engineering*, Taipei, Taiwan, March 1995.

[CDRS86] M.J. Carey, D.J. DeWitt, J.E. Richardson, and E.J. Shekita. Object and File Management in the Exodus Extensible Database System. In *Proc. 12th VLDB*, Kyoto, Japan, August 1986.

[CKAK94] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts, and Detection. In *Proc. 20th Intl. Conf. on Very Large Databases*, Santiago, Chile, September 1994.

[DBB+88] U. Dayal, B. Blaustein, A.P. Buchmann, S. Chakravarthy, D. Goldhirsch, M. Hsu, R. Ladin, D. McCarthy, and A. Rosenthal. The Hipac Project: Combining Active Databases and Timing Constraints. *SIGMOD Record*, 17(1), March 1988.

[DBM88] U. Dayal, A.P. Buchmann, and D. McCarthy. Rules Are Objects Too: A Knowledge Model for an Active Object-Oriented Database System. In *2nd Intl. Workshop on Object-Oriented Database Systems*, Bad Münster am Stein, Germany, September 1988.

[GD93a] S. Gatziu and K.R. Dittrich. Eine Ereignissprache für das Aktive, Objektorientierte Datenbanksystem Samos. In *Proc. BTW*, Braunschweig, Germany, 1993.

[GD93b] S. Gatziu and K.R. Dittrich. Events in an Active Object-Oriented Database System. In M. Williams N. Paton, editor, *Rules in Database Systems, Proc. 1st Intl. Workshop on Rules in Database Systems*, Edinburgh, 1993.

[GD94] S. Gatziu and K.R. Dittrich. Detecting Composite Events in Active Database Systems Using Petri Nets. In *Proc. 4th Intl. Workshop on Research Issues in Data Engineering: Active Database Systems (RIDE-ADS)*, Houston, TX, February 1994.

[HLM88] M. Hsu, R. Ladin, and D. McCarthy. An Execution Model For Active Database Management Systems. In *Proc. 3rd Intl. Conf. on Data and Knowledge Bases*, Jerusalem, June 1988.

[Mar95] J. Marschner. Non-Standard Transaktionsmanagement in inem Aktiven Objektorientierten Datenbanksystem. Master's thesis, Dept. of Computer Science, Technical University Darmstadt, 1995.

[Mos85] E. Moss. *Nested Transactions*. MIT Press, Cambridge, MA, 1985.

[RBP+91] J. Rumbaugh, M. Blaha, W. Permerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design*. Prentice-Hall, 1991.

[WBT92] D.L. Wells, J.A. Blakeley, and C.W. Thompson. Architecture of an Open Object-Oriented Database Management System. *IEEE Computer*, 25(10), October 1992.

[ZBB+96] J. Zimmermann, H. Branding, A.P. Buchmann, A. Deutsch, and A. Geppert. Design, Implementation and Management of Rules in an Active Database System. In *Proc. DEXA*, September 1996.