Unbundling Active Functionality

Stella Gatziu[1], **Arne Koschel**[2], **Günter von Bültzingsloewen**[3], **Hans Fritschi**[1]

[1]Department of Computer Science, University of Zurich {gatziu, fritsch}@ifi.unizh.ch
[2]Forschungszentrum Informatik (FZI), Karlsruhe {koschel@fzi.de}
[3]Swiss Bank Corporation, {guenter.von_bueltzingsloewen@mhs.swissbank.com}†

Abstract *New application areas or new technical innovations expect from database management systems more and more new functionality. However, adding functions to the DBMS as an integral part of them, tends to create monoliths that are difficult to design, implement, validate, maintain and adapt. Such monoliths can be avoided if one configures DBMS according to the actually needed functionality. In order to identify the basic functional components for the configuration the current monoliths should be broken up into smaller units, or in other words they could be "unbundled". In this paper we apply unbundling to active database systems. This results in a new form of active mechanisms where active functionality is no longer an integral part of the DBMS functionality. This allows the use of active capabilities with any arbitrary DBMS and in broader contexts. Furthermore, it allows the adaption of the active functionality to the application profile. Such aspects are crucial for a wide use of active functionality in real (database or not) applications.*

1 Introduction

Active functionality, as it is offered today by active database management systems (active DBMS), is the ability to react not only after an explicit request from the application or the user, but also when a specific situation of interest has occurred in the DBS or its environment. The basic notion of active DBMS are *Event/Condition/Action-Rules* (ECA-Rules), meaning WHEN a certain event occurs and IF a given condition holds THEN a certain action is executed.

In order to offer active functionality, new components (so-called *activity components*) implementing tasks like rule management and execution are required in addition to the known components of a DBMS.

Today, activity components are developed for a particular DBMS. By implementing the activity components as an integral part of a DBMS, active functionality is rather tightly coupled to the DBMS behind it. Thus, the active DBMS becomes part of large monolithic pieces of software. Such software is difficult to implement, validate, maintain and adapt. Even where active DBMSs have a layered system architecture with the activity components residing "on top" of a conventional DBMS the active functionality remains tightly interconnected and as such bound to the particular target DBMS.

In either architectural approach, a substantial effort is required to adopt active functionality for various DBMSs. It is close to impossible to develop activity mechanisms that can be ported from one DBMS to the other. Today, few commercial DBMS offer restricted active mechanisms while the more expressive ones are only found in research prototypes.

Moreover, active functionality tightly coupled to a concrete environment (a particular DBMS environment) hampers its adaption to changes in the information technology scenery. Just consider the present change of information systems in environments with several (existing and newly developed) heterogeneous and distributed information sources. Active mechanisms, e.g., complex situation monitoring or cooperation and coordination should take into account heterogeneity and distribution.

A further weakness of the tightly coupling of active and conventional database mechanisms is that active functionality is not usable on its own, i.e., without the "added" DBMS features. However, active functionality is also needed in applications which require either no database functionality at all or just some like persistence. Thus, active functionality should be offered not only as part of the functionality of a DBMS, but also as a separate service which can be combined with other services like a persistence service. In this way, users could develop "lean" solutions without any overhead due to unneeded components.

In this paper, we investigate the provision of active database mechanisms as an individual service. In other words, we *unbundle* active functionality from the DBMS. Thus, we follow a general direction that database research is currently about to take, namely to provide individual database management services that can be used and combined in a variety of ways and in a variety of environments [1, 2, 4, 11, 24].

We first discuss the advantages of unbundling active functionality. We then show how the unbundling process may take place. We start with a domain analysis of active DBMS-style active functionality. The main task is the identification of components and the cooperation between them. This leads to several (architectural) configurations. Each of these configurations is sufficient for one (or maybe

†. This author's work was performed during his stay at FZI Karlsruhe.

several) specific application categories. It is the job of the developer to choose just the "right" configuration for his application, hopefully without incurring much overhead for not needed components. In this paper we restrict ourselves to one possible configuration as an example. A number of several possible architectural configurations, an overview of problems arising from unbundling the active functionality and first experiences with using unbundling in concrete projects are discussed in [13].

The rest of this paper is organized as follows: In chapter 2 we discuss the rationale for unbundling in general while unbundling in the context of active DBMS is discussed in chapter 3. The unbundling process and a possible architecture for unbundled active functionality are described in chapter 4 and 5. Chapter 6 concludes the paper.

2 Unbundling in General

Unbundling is the activity to break up monolithic software systems into smaller units with a fair degree of autonomy. Each unit provides a specific service of the software system, which should not only be useful in combination with the further functionality of the system but also separate from it. In contrast to monolithic systems, the communication and the cooperation between the units is not hidden within the system but can clearly be identified from outside.

Considering database management systems as a form of traditionally large and monolithic software system, it makes sense to build them out from a number of cooperating components [1, 2, 4, 11, 24].

The major motivation for unbundling DBMS comes from the ever increasing demands on the functionality of DBMS because of (new) application areas (like data warehousing), new architectural forms (like distribution and heterogeneity) and technical innovations (like middleware). If one augments the DBMS as we know them today by the new functionalities one obtains ever more unwieldy systems if the integration is succeeded at all with. Nor are we sure that the same situation may arise in the future again. Therefore, unbundling the functions and configuring the components according to needs into a specifically tailored system seems the only reasonable answer.

3 Unbundling Active Functionality

One prominent example of new functionality required for DBMS is the (re)active capability as it is offered today by active DBMS. Unbundling active functionality from a DBMS means first to separate the active part from the active DBMS and then eventually to break up this active part into units which provide specific services like:

- *the definition of rules*: Active DBMS offer constructs for the definition of ECA-rules, which specify the part of the application semantics representing the active behavior. The expressiveness and the variety of the constructs, especially those used for the specification of events, is crucial for the wide use of active DBMS.
- *the management of rules*: The information about rules, events, conditions and actions is managed by the DBMS so that it could be accessed efficiently whenever necessary.
- *the detection of events*: One main feature of active DBMS is the ability to detect efficiently all events requiring a reaction.
- *the execution of rules*: Upon an event detection the active DBMS reacts with the execution of a rule. How this execution takes place is determined by several predefined or user-defined guidelines which form the execution strategy of the active DBMS.

Unbundling active functionality from a concrete DBMS brings a number of new opportunities which are critical for a wide use of active functionality in real (database or not) applications:

Use of active capabilities with arbitrary DBMS. As mentioned in the introduction, today, active mechanisms are always tightly coupled to a specific DBMS. Typically, active functionality is rather restricted and provided only by few DBMS. There are many applications with active behavior where a DBMS is used, which fulfills almost all application requirements (e.g., performance, programming environment), but offers none or restricted active functionality.

The separation of active mechanisms from the DBMS is a primary requisite to implement them as much as possible independent from a specific DBMS. Indeed there are many activity components like the event detector for complex events, which are independent from a specific DBMS and can be used with minor adaptations with arbitrary DBMS. For a concrete environment, only the DBMS-dependent components (like the detector for database events) must be implemented. This eases the implementation of active mechanisms and should contribute to their wider utilization.

Use of active capabilities in broader contexts separate from a DBMS. A range of applications like workflow management, health care monitoring, etc. require active functionality. However, they do not necessarily require a whole database functionality but only parts like persistence or query facilities. Unfortunately the tight integration of active and database mechanisms in one system forces users to accept the entire database functionality that comes with the active functionality.

On the other hand, a separation of active and conventional database mechanisms would allow the use of active capabilities without any overhead due to unneeded components.

This fits well into an environment of combinable services (like CORBA or OLE/DCOM). These distributed object systems specify a set of generic base services for objects, such as services for persistence, transactions, or concurrency control. It would only be natural to add to this infrastructure a higher-level service of active functionality. Applications could then combine active functionality with other services.

Use of active functionality in heterogeneous environments. Today's, database applications often refer to several (existing and newly developed) heterogeneous and distributed information sources. The need for active functionality in heterogeneous, distributed environments is obvious in the area of data warehousing [26] and has also been reported in [5, 19, 23].

The impact of heterogeneity on active DBMS is primarily the richer set of information resources that must be monitored. This means that relevant information may not only reside in a database system but also in files, on web pages, etc. Furthermore, checking complex situations like an environmental pollution, a business opportunity, or an emerging traffic congestion usually requires more complex computations than supported by database conditions. Typically, some specific analysis tools have to be invoked. Thus, such analysis and processing tools may become important resources as well [5, 6].

Hence, the means of the interaction between a service offering active functionality and the heterogeneous environment (e.g., the information sources) becomes much more diverse. What is desirable then is a uniform view that hides the diversity of the cooperation forms and allows to choose a particular form according to actual needs. This seems possible after a clear identification and isolation of the interfaces, which clearly should be the result of unbundling active functionality.

Tailoring active functionality to the application profile. In the more general environment sketched above active functionality plays more roles than what has been used in active DBMS.

Supporting advanced active functionality able to fulfill the requirements of every application area is not a solution, because there are many cases where the system is overloaded with functionality which is not necessarily needed. Active mechanisms should be tailored to the application profile. Consequently, collecting all active functionality that could conceivably be used by some application into one system is not the answer. Instead, the unbundling of active mechanisms from the DBMS should be continued into the unbundling of these capabilities themselves. For example, active functionality could be split into separate functions like composite event detection or rule execution. This functions can be selected individually and configured and interconnected independently.

Availability of uniform active functionality. The major advantage of separating active functionality from the DBMS is its use in a variety of ways and in a variety of environments as mentioned above. However, this presumes that offered active functionality is of general purpose. More specifically the rule definition language must subsume a major part of the state-of-the-art approaches and have a clear semantics.

Offering active functionality as a service allows a high distribution of these general active mechanisms. Thus, it also helps to overcome the variety of existing rule languages with different semantics and enables the common use of standards for active mechanisms.

4 The Unbundling Process

The unbundling process consists of two major steps (a) the domain analysis where we identify the covered functionality in terms of the services provided and (b) the description of the unbundled system, based on an architectural model, in form of a specific architecture [11].

In case of unbundling active functionality the domain analysis consists of a review of existing work. This is feasible because after around 10 years of research in active DBMS the basic concepts and functionalities for active behavior and especially the semantics behind them are well-understood and established in these systems [8]. This has recently lead to a consensus on those base concepts, which spawned several comprehensive overviews on the concepts and dimensions of active functionality in active DBMS [3, 18, 25].

In order to describe the architecture of an unbundled system we adopt a rather general architectural model from [10, 20]. Our architectural model consists of *components*, *connectors* (between components) and a policy restricting the ways how components and connections can be arranged/interacted. An architecture of a particular unbundled system is then defined by a number of (more or less) autonomous processing components, so-called *services* (or agents), which cooperate with each other. The knowledge about the regulations which govern the cooperative behavior of the components is captured by connectors. Connectors provide both, services interconnecting the components (e.g. object invocation, multicast messaging) and services for supervision and control of the activities performed by the components. Of course, unbundling can proceed recursively, i.e., a component or a connector can in turn be subdivided into a set of components cooperating via connectors.

Facilities of a component or possible constraints which must be fulfilled in a concrete (architectural) configuration can be specified with constructs proposed by a *specification model*.

5 An Unbundling Step as an Example

Starting from a monolithic active DBMS we show as an example a possible unbundling step. In a monolithic active database environment, the only autonomous components are a number of clients and the active database manager (Figure 1)[†].

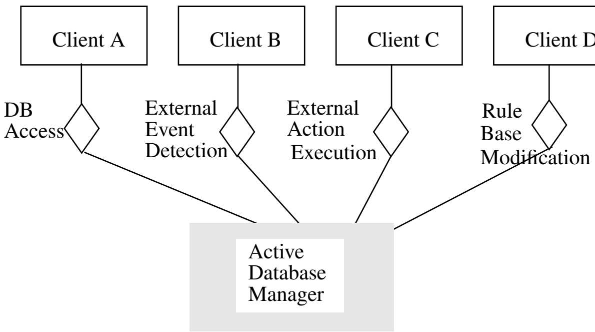

Figure 1: The monolithic active DBMS

Four different connectors regulate the interaction among these components:

- *Database Access*. This connector allows clients to establish a session with the database manager, to start and end transactions, and to access or modify data items (e.g. relations, objects) within the database. Thereby, the database manager synchronizes concurrent client accesses. In case of a passive DBMS this is the only connector.
- *Rule Base Modification.* Clients may modify the rule base e.g., by adding/removing/updating rules.
- *External Event Detection and External Action Execution*. To trigger a rule, clients (e.g., the system clock or an application program) may raise events, so-called *external* or *abstract* events, indicating some specific occurrence of interest. Furthermore, a rule may trigger an action which is to be executed by some client. The protocols enforced by the external action execution connector depend on the rule execution model.

A first unbundling step is the separation of the active mechanisms from the DBMS. This enables the use of active functionality in different environments, e.g., with various existing commercial database systems. From a software engineering perspective, active capabilities have to be developed only once and can be re-used in different contexts.

In a second unbundling step we isolate an *event service* and a *rule service.* This allows the flexible configuration of active functionality for specific application profiles. Active mechanisms are decomposed into two components which can be selected individually and configured and interconnected independently. For example, in case of event detection, which is supported by the event service, different applications may require different types of primitive and composite events, different event consumption modes, and different durations over which events are kept as a history. In case of rule execution, which is supported by the rule service, they may demand different execution guidelines. Furthermore, the event detection and rule execution can now be used each on their own, independently from the complete active functionality. For example, we can provide separate event monitoring and production rule facilities.

The corresponding architecture is illustrated in Figure 2. Beyond the two components responsible for the two new services, four new connectors, the *Database Event Detection*, the *Condition Evaluation*, the *Action Execution* and the *Event Signalling* are introduced.

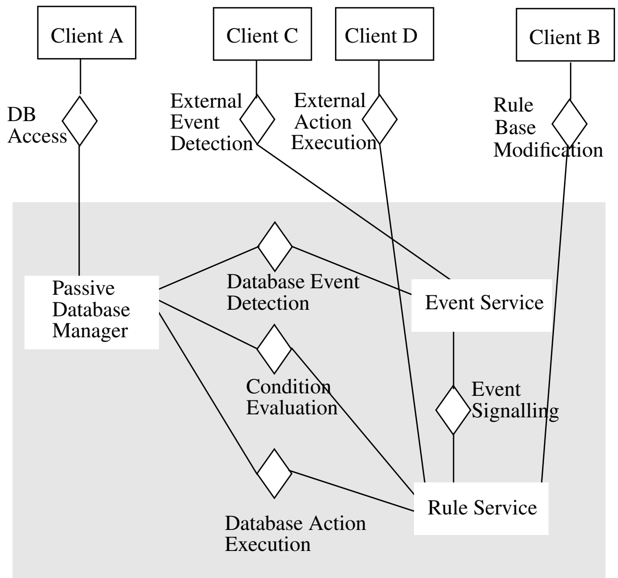

Figure 2: Offering an event and a rule service

Event Service The Event Service records events, maintains a (persistent) event history (which consists of all event occurrences which have not yet been consumed for a rule execution or composite event detection) and detects composite events. It is informed about the occurrence of primitive events via the event detection connectors. It provides event information to condition evaluation and action

†. Architectures consisting of components and connectors can be described by means of ER-style diagrams, denoting components as rectangular boxes and connectors as diamond-shaped relationships.

execution via condition evaluation and action execution connectors.

Rule Service The Rule Service is responsible for the maintenance of the rule base and implements the rule execution cycle.

Database Event Detection This connector determines what kind of events the Passive Database Manager produces and the kinds of events the Event Service subscribes to. Database events are raised in the context of some database access. An access may raise several events (e.g., a set-oriented insert raises several tuple insert events), and one event may depend on several accesses (e.g., the net effect of all updates during a transaction). Depending on the event semantics and rule execution guidelines, the Database Event Detection connector may enforce that the Event Service performs certain steps before the Database Manager can proceed with processing the access that raised the event. The Database Event Detection connector is also invoked during rule base modification.

Condition Evaluation and Database Action Execution In order to evaluate conditions and to execute database actions, the Event and the Rule Service must cooperate with the Database Manager. The Rule Service defines, according to the rule execution model, the point in time when evaluation of the database condition has to be initiated and when an action has to be executed. Since the Database Manager is responsible for the actual condition evaluation and action execution, these two connectors require database access capabilities as provided by the DB Access connector. Furthermore, they require a transaction service allowing database accesses to be performed, e.g., as part of a transaction performed on behalf of some client. If the Database Manager requires that access commands are statically compiled before they are executed (as opposed to dynamic invocation), the corresponding compilation is also performed via the Condition Evaluation or Database Action Execution connector during a rule base modification.

Event Signalling is a connector between Event Service and Rule Service. The Rule Service subscribes to events which may trigger a rule. Events are signalled as soon as they are recorded in the event history.

Furthermore, the Event and the Rule Service may also make direct use of the DB Access connector in order to store rules persistently, to search the rule base, and to retrieve rules (not shown in the figure).

We briefly illustrate here how the architecture introduced can be extended to cope with distribution and heterogeneity. As mentioned in chapter 3 heterogeneity is reflected on the interaction between a service offering active functionality and the environment (e.g., the information sources) which must be considered in a variety of ways. One typical interaction is the detection of primitive events. For this, we propose an abstract connector for primitive event detection which includes (and actually substitutes) the connectors External Event Detection and Database Event Detection of Figure 2. This connector hides the fact that several different event sources exist. Thus, the Event Service now has to issue event subscriptions only to this one connector. The connector can decide (by means of an internal component *Primitive Event Mediator*), which event source actually needs to be addressed, and issue the corresponding subscription. Thus, heterogeneity of event sources is hidden to the subscriber. Details about the appropriate architecture can be found in [13].

6 Conclusions

Despite their recent progress, active DBMS have rarely become an ingredient of applications. One major reason is the monolithic nature of existing active DBMS. Active functionality should be offered as a separate facility which could be combined with further facilities, e.g., certain DBMS functions, only to the needed extent. In other words, active functionality should be usable as just one component amongst others, but should still rely on the consensus-based functionalities and semantics as contained in active DBMS. As such it should support heterogeneous and distributed information systems.

To address these problems, we have discussed in this paper initial ideas towards a new form of active mechanisms where the active functionality is not a tight part of the DBMS functionality like in current approaches. Unbundling is the activity to break up traditionally monolithic software systems into smaller units. We apply the unbundling process to active database systems. In this paper, we presented a possible architecture where active mechanisms are separated from the DBMS and subdivided into an *event service* and a *rule service*.

There are several questions arising from the unbundling process and many problems must be solved. The most important one may be whether active DBMS-style ECA functionality can "survive" the whole unbundling process, giving the full semantics as in traditional active DBMS. Is an active DBMS the same as a passive DBMS plus a service offering active functionality? We think that in the general case this is not possible, at least not without modifications, since unbundling essentially means to give up the "closed world" assumption which traditionally underlies a DBMS.

Furthermore, much more experience is needed with the concrete investigation of the development of services offering active functionality outside the DBMS. Some work has been done in the FRAMBOISE project [9] which proposes a construction system for the development of so-called ECA-services which are decoupled from a particular

DBMS. [12] proposes TriggerMan, an asynchronous trigger processor which is designed to be able to gather updates from a wide variety of sources and to execute triggers asynchronously. In the C^2offein [14] approach, which is an extension of [5], a widely configurable service set for active functionality in CORBA-based heterogeneous, distributed systems is developed. The system is configurable with respect to the types and features of supported services, protocols between services, distribution parameters etc.

Acknowledgments

The work described here has been supported by the ACT-NET-Network. We thank Peter Lockemann and Klaus Dittrich for the careful reading of the paper and the fruitful comments.

References

1. Database Systems - Breaking Out of the Box. *ACM Workshop on Strategic Directions in Computing Research*, Report of Database Working Group, June, 1996 (to be published in ACM Computing Surveys)

2 M. Adler. Emerging Standards for Component Software. *IEEE Computer*, March, 1995.

3 The ACT-NET Consortium. The Active Database Management System Manifesto: A Rulebase of ADBMS Features. *ACM Sigmod Record, 25:3*, September 1996.

4 J. Blakeley. Data Access for The Masses through OLE DB. *Proc. of the Intl. Conf. of the ACM SIGMOD*, 1996.

5 G. von Bültzingsloewen, A. Koschel, R. Kramer. Active Information Delivery in a CORBA-based Distributed Information System. *Proc. of the 1st IFCIS Intl Conf. on Cooperative Information Systems*. Brussels, June 1996.

6 G. von Bültzingloewen, A. Koschel, R. Kramer. Accept Heterogeneity: An Event Monitoring Service for CORBA-based Heterogeneous Information Systems. Poster in *Proc. of the 2nd IFCIS Conf. on Cooperative Information Systems*. South Carolina, USA, June 1997.

7 Computer Corporation of Amerika. *An architecture for database management standards*. NBS Spec. Pub. 1982.

8 U. Dayal. Ten Years of Activity in Active Database Systems: What Have We Accomplished? *Proc. of the Intl. Workshop in Active and Real-Time Database Systems*, Skövde 1995.

9 H. Fritschi, S. Gatziu, K.R. Dittrich. *FRAMBOISE -- An Approach to Construct Active Database Mechanisms.*Technical Report 97.04, Department of Computer Science, University of Zurich, April 1997 (http://www.ifi.unizh.ch/dbtg/).

10 D. Garlan, M. Shaw: An Introduction to Software Architecture. *Advances in Software Engineering and Knowledge Engineering*, Vol. I. V. Ambriola, G. Tortora (eds.), World Scientific Publishing Company.

11 A. Geppert, K.R. Dittrich. *Bundling: A new Construction Paradigm for Persistent Systems?*. Technical Report 97.08 Department of Computer Science, University of Zurich, July 1997.

12 E. Hanson, S. Khosla. An Introduction to the TriggerMan Asynchronous Trigger Processor. *Proc. of the 3rd Int. Workshop on Rules in Databases*, Skoevde, June 1997.

13 A. Koschel, S. Gatziu, G. von Bültzingsloewen, H. Fritschi. Applying the Unbundling Process on Active Database Systems. *Intl. Workshop on Issues and Applications of Database Technology (IADT)*, Berlin, July 1998 (to be published).

14 A. Koschel, R. Kramer, G. von Bültzingslöwen, T. Bleibel, P. Krumlinde, S. Schmuck C. Wein. Configurable Active Functionality for CORBA. *ECOOP'97 Workshop "CORBA: Implementation, Use and Evaluation"*, Jyväskulä, Finnland, June 1997.

15 P.C. Lockemann, K.R. Dittrich. Architektur von Datenbanksystemen. In P.C. Lockemann, J.W. Schmidt (eds) *Datenbankhandbuch*, Springer 1987.

16 Object Management Group. *CORBA services: Common Object Services Specification*. Object Management Group, Inc. (OMG), March 1995.

17 Object Management Group. The Common Object Request Broker: Architecture and Specification, Version 2.0. Object Management Group, Inc. (OMG), July 1995.

18 N.W. Paton, O. Diaz, M.H. Williams, J. Campin, A. Dinn, A. Jaime. Dimensions of Active Behaviour. *Proc. 1st Intl. Workshop on Rules in Database Systems.*Workshops in Computing, Springer-Verlag, 1994.

19 N. Pissinou and K. Vanapipat. Active Database Rules in Distributed Database Systems. *Journal of Computer Systems, 11:1*, January 1996.

20 M. Shaw, R. DeLine, D. V. Klein, T.L. Ross, D.M. Young, G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering* (TSE), 21:4, April 1995.

21 R. M. Soley. *Object Management Architecture Guide*. Object Management Group, Inc. (OMG). Revision 1.0, November 1990.

22 E. Simon, A. Kotz-Dittrich. Active Database Systems: Promices and Realities. *Proc. 21th Intl. Conf. on Very Large Data Bases* (VLDB), Zurich, Switzerland, September 1995.

23 S.Y.W. Su, H. Lam and J. Arroyo-Figueroa, T. Yu and Z. Yang. An Extensible Knowledge Base Management System for Supporting Rule-based Interoperability among Heterogeneous Systems. *Proc. of the Intl. Conf. on Information and Knowledge*, Baltimore, 1995.

24 D. Vaskevitch. Very Large Databases How Large? How Different? *Proc. 21th Intl. Conf. on Very Large Data Bases* (VLDB), Zurich, Switzerland, September 1995.

25 J. Widom, S. Ceri (eds). *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers, 1995.

26 J. Widom. Research Problems in Data Warehousing. *Proc. of the Intl. Conf. on Information and Knowledge*, Baltimore, 1995.

27 G. Zhou and R. Hull and R. King and J. Franchitti. Supporting Data Integration and Warehousing Using H2O. *Data Engineering, 18:2*, June 1995.