The Active Database Management System Manifesto: A Rulebase of ADBMS Features

A Joint Report by the ACT-NET Consortium[1]

Abstract. Active database systems have been a hot research topic for quite some years now. However, while "active functionality" has been claimed for many systems, and notions such as "active objects" or "events" are used in many research areas (even beyond database technology), it is not yet clear which functionality a database management system must support in order to be legitimately considered as an active system. In this paper, we attempt to clarify the notion of "active database management system" as well as the functionality it has to support. We thereby distinguish mandatory features that are needed to qualify as an active database system, and desired features which are nice to have. Finally, we perform a classification of applications of active database systems and identify the requirements for an active database management system in order to be applicable in these application areas.

Preamble

Database systems have been accepted as a vital part of the information system infrastructure in industry, commerce, public administration and in society in general. They can be considered a mature technology whose characteristics have been well-covered in past manifestos on, e.g., RDBMS and ooDBMS. All manifestos, textbooks and user views alike have accepted as a fact that database systems are monolithic systems that provide across a uniform interface an extremely wide range of services. However, such a view raises more and more doubts at a time when new applications as well as technological innovations continuously seem to impose further functional requirements on database systems, thus slowly turning them into dinosaurs, while enterprises in industry and administration break up their monolithic organization into smaller units with a fair degree of autonomy, and thus replace tight integration by looser forms of communication and cooperation.

One may argue that informatics has an obligation to follow the decentralization and concentration on core competences of organizations, by likewise breaking up its traditionally monolithic information systems into smaller units which operate on looser forms of communication and cooperation. The process seems applicable to DBMS as well and is variously referred to as the "unbundling" of DBMS functions. Witness specialized architectural developments such as middleware, data dictionaries, mediators, transaction monitors. If successful, DBMS with a prescribed range of services could then be constructed by "(re)bundling" components with the appropriate functional specializations.

This manifesto concerns itself with bundling into DBMS one particular functionality. Case-by-case cooperation among autonomous and semi-autonomous, loosely-coupled system components must be based on some mechanism that allows these components to become aware of what is happening around them. This is particularly true if database systems and (high-level) telecommunication technologies enter into a close liaison due to global networking, global information services, and the distribution of applications. Even if some of the needed awareness may be due to classical communication mechanisms, in the final analysis they must all be based on the notion of "event" which is either raised (pro-activity) or responded to (re-activity). Events, and responses to them, will therefore become a pradigm (perhaps the basic paradigm) of future information systems.

Since we consider this a general phenomen of information systems, DBMS technology has no choice but to focus its efforts on the event-response paradigm. In other words, all future DBMS — be they relational, object-oriented, or otherwise — must exhibit some active functionality. Indeed, there have been developments in the past in which first experiences have been gained with the event-response mechanism, even though this occurred in the narrower focus of events raised by internal changes to the database proper. These systems — referred to as Active Database Management Systems (ADBMSs) — have reached a degree of maturity that suggests, and justifies a summary of its characteristics in a new manifesto. They should thus be able to act as a benchmark from which new developments in the general direction mentioned above could evolve.

1. ACT-NET is a network in the "Human Capital and Mobility" programme funded by the commission of the EU (contract no. CHRX-CT93-0089). The current participants in ACTNET are the Universities of Aberdeen, Athens, Darmstadt, Heriot-Watt (Edinburgh), Karlsruhe, Linköping, Manchester, San Sebastian, Skövde, Versailles, and Zürich. ACT-NET is coordinated by Alex Buchmann (TU Darmstadt) who can be contacted at buchmann@dvs1.informatik.th-darmstadt.de. Information on ACT-NET is also available in the WWW at http://www.informatik.th-darmstadt.de/DVS1/act-net.html.
This report has been edited by Klaus Dittrich, Stella Gatziu, and Andreas Geppert (University of Zurich) who can be contacted at {dittrich|gatziu|geppert}@ifi.unizh.ch.

1 Introduction

ADBMSs [e.g., 5, 16] have over the past years become a very attractive database research topic. Many different systems and models have been proposed which claim to be "active". However, it is still unclear what the term "active database management system" really means. Under which condition can you legitimately claim that your system is "active"?

Authors often define the term ADBMS along the lines of "being able to react automatically to situations in the database and beyond" or "allowing the specification and implementation of reactive behavior". Some of these definitions are not very precise, and often there is no broad agreement on how to explain the term "active". This may be due to the fact that the proposals for systems and architectures have preceded the formal and conceptual groundwork.

In this paper, we describe what, in our opinion, the characteristics are that a database management system must exhibit in order to be legitimately called "active". In doing so, our yardstick is an ADBMS that is practically useful for real-life applications (instead of a mega-system justifying all the fancy research concepts ever invented).

Two problems have been encountered while trying to define essential characteristics:

1. Many different ADBMSs and systems with some sort of "active functionality" have been defined so that simply taking the commonalities among these systems as the definition of an ADBMS would easily result in a collection of general and meaningless statements.
2. For a deeper understanding of ADBMSs and maybe different classes thereof, it is useful to look at their intended application domains and the facilities they require. Otherwise, the implications of specific system characteristics are difficult to grasp. For instance, is it mandatory for an ADBMS to support composite events? The answer is that for some "reactive" applications composite events are needed, for others they may be convenient to have, and again for others they are not needed at all. The same holds for coupling modes — not all conceivable coupling modes are required for simple applications.

Concerning the second observation, note that the development of ADBMSs is motivated by the application areas in mind, probably even more so than has been the case for object-oriented DBMSs. We therefore think that one cannot simply abstract from the application domains when defining ADBMSs. Instead, we propose to specify some general features required for all ADBMSs and to define more specific features depending on the application domains the system in question is appropriate for (similar to the classification of object-oriented DBMSs [9]).

The remainder of this paper is organized as follows. The next section introduces the terminology and basic concepts of ADBMSs, while section 3 describes required characteristics of ADBMSs. Section 4 introduces a classification of ADBMSs based on the requirements of the types of intended application domains.

2 Terminology

Before we introduce the characteristics of an ADBMS, we establish a common terminology.

A *database management system* (DBMS) is a software system for reliably and efficiently creating, maintaining, and operating large, integrated, multi-user databases. It implements storage and retrieval of data, secondary storage management for large sets of objects (including access paths, clustering, etc.), concurrency control, and recovery. The collection of data including secondary information is stored in the *database*. The structure of the database is defined by the *database schema*. Schemas are specified by using the data definition language (DDL), and access to the database is by means of a data manipulation language (DML). Both together represent a so-called *data model*. Finally, a *database system* (DBS) is a DBMS together with a concrete database.

In a nutshell, ADBMSs extend "passive" DBMS with the possibility to specify *reactive behavior*. Below, we introduce the concepts relating to the specification and implementation of this additional DBMS-functionality.

*ECA-rule*s (event-condition-action rules) consist of events, conditions and actions. The meaning of such a rule is: "when an event occurs, check the condition and if it holds, execute the action". Other terms are used synonymously in active database systems, for example *triggers*.

Once a set of rules has been defined, the active database system monitors the relevant events. Whenever it *detects* the occurrence of a relevant event it notifies the component responsible for rule execution of it. We call this notification the *signalling* of the event. Subsequently, all rules which are defined to respond to this event are *triggered* (or *fired*) and must be *executed*. Rule execution incorporates condition evaluation and action execution. First, the condition is evaluated and if it is satisfied, the ADBMS executes the action.

In analogy to a data definition language which supports the modeling of data structures (and behavior in the case of object-oriented DBMS), an ADBMS provides a *rule definition language* as a means to specify ECA-rules. The language consists of constructors for the definition of rules, events, conditions, actions and execution constraints. The *rulebase* of an ADBS contains meta information on defined ECA-rules.

In addition to rule specification and event detection, an ADBMS has to support rule execution. It thus needs an *execution model*. An execution model determines when rules are executed, and which properties rule execution has.

In general, events occur within transactions, and rules are executed within transactions. The execution model thus has to suit the transaction model of the DBMS. If in a transaction an event occurs that triggers a rule, then this transaction is called the *triggering transaction*. If the rule is executed in one or more transactions, these are called *triggered transactions*.

The execution model defines the relationships between triggering and triggered transactions in terms of commit and abort dependencies, and the semantics of rule execution with respect to concurrency control and recovery. The most common framework for specifying these relationships are nested transactions [13]. With respect to dependencies, a triggered transaction is either a subtransaction of the triggering transaction, or it is independent from the triggering transaction. In the latter case, rule execution means spawning a new transaction tree. With respect to execution semantics, the execution model determines the time when a triggered transaction is executed with respect to the triggering transaction or event. The triggered transaction can be started immediately after the event has been signalled, or at the end of the triggering transaction (i.e., directly before commit).

3 Characteristics of ADBMSs

In this section we describe the features any full-fledged ADBMS must/should have, regardless of its application area.

3.1 ECA-Rule Definition Features

Feature 1: **An ADBMS is a DBMS.**
All the concepts required for a passive DBMS are required for an ADBMS as well ("passive" modeling facilities, query language, multi-user access, recovery, etc.). That means, if a user ignores all the active functionalities, an ADBMS can be used in exactly the same way as a passive DBMS.

Feature 2: **An ADBMS has an ECA-rule model.**
An ADBMS extends a passive DBMS in that it supports *reactive behavior.* Reactive behavior must be specifiable/definable by the user. The means to define rules are called the *ECA-rule model (*together with the data definition facilities they are sometimes also called the *knowledge model* [7]). Seen the other way, the DBMS-interface (e.g., the data definition language) is extended or complemented by operations for defining rules.

The use of ECA-rules implies the following three features:

Feature 2.a: **An ADBMS has to provide means for defining event types.**
An event type (synonymously: event description, pattern, definition) describes situations to which a reaction must be shown. An event type can be *primitive* or *composite*. Primitive event types define elementary occurrences that are of interest; possible primitive event types are method invocation, data item modification, transaction operation, abstract, and time event types. In general, we require that —wherever meaningful — *before* and *after* events can be defined. In case of database operations, for instance, a *before event* is signalled directly before the operation is actually executed. An *after event* is signalled directly after the operation has been performed. The event types supported should at least subsume the DML-operations and transaction statements. This means that, e.g., , the update of a specific relation can be defined as an event of interest.

Composite event types are defined as combinations of other primitive or composite events using a set of *event constructors* such as disjunction, conjunction, sequence, etc.

Event occurrences are the instances of event types. Such an occurrence is conceived as a pair (*<event type>, <timestamp>*) where *<event type>* denotes the event type of the occurrence, and *<timestamp>* represents the point in time when the occurrence actually took place. The timestamp of composite event occurrences is determined based on information about their component events, e.g., the point of time when the last component occurred.

In general, the point in time when an event occurs is a parameter of the event occurrence. Further parameters are possible, such as the transaction in which an event occurs, or the name of the user who has started this transaction.

In the case of composite events, *event restrictions* specify conditions the components must fulfill in order to form a (legitimate) composition. One way to specify such restrictions are predicates on the *event parameters* (such as the timestamp), but more general predicates are possible. For instance, it may be required for a composite event that all of its components occurred within the same transaction, or all refer to operations on the same data item.

Feature 2.b: **An ADBMS has to provide means for defining conditions.**

A *condition* formulates in which state the relevant part of the database has to be in order to execute the action, i.e. it shows *what* must be checked. It is checked after the rule has been triggered. A condition may be either a predicate on the database state like the “where”- part in an SQL-statement, or a database query with an empty or non-empty result. The condition is satisfied if either the result evaluates to true or is non-empty.

Feature 2.c: **An ADBMS has to provide means for defining actions.**

An *action* formulates the reaction to an event and is executed after the rule has been triggered and its condition determined to hold. An action may contain data modification or data retrieval operations, transaction operations like commit or abort, call of arbitrary procedures/methods, etc. It should at least be possible to use DML-commands in actions, including transaction commands (e.g., to abort the triggering transaction).

Feature 3: **An ADBMS must support rule management and rulebase evolution.**

The following three functionalities are essential with respect to rule management.

Feature 3.a: **An ADBMS has to support rulebase management.**

The set of rules defined at a given point in time forms the rulebase. The rulebase should be managed by the ADBMS. In other words, definitions of ECA-rules are a part of the DBMS meta information and the database. The ADBMS should store information about which rules currently exist (and how they are defined). This stored information on ECA-rules should be visible to users and applications.

Preferably, the ADBMS represents meta-information on rules in terms of its underlying data model as a part of the catalogue. In this case, users and applications can retrieve information about rules from the catalogue in the same way as information about the “passive” schema can be retrieved (i.e., relations, classes, etc.).

Feature 3.b: **An ADBMS has to support rulebase evolution.**

The rulebase must be changeable over time: it is neither sufficient to support only a fixed set of ECA-rules, nor is it appropriate to support reactive behavior only as ECA-rules that are hard-wired into the DBMS code. An ADBMS must therefore allow new ECA-rules to be defined and old ones to be deleted. It should also be possible to modify event, condition, or action definitions of existing rules.

Feature 3.c: **An ADBMS has to support enabling and disabling of rules.**

Rules can be disabled and enabled. Disabling a rule means that the rule definition remains in the rulebase, but that it will not fire upon subsequent occurrences of its event. Enable is the inverse operation to disable: enabling a disabled rule means that the rule will from now on fire again upon occurrences of its event.

Variations of ECA-rules. Although the general and preferred form of active rules are full ECA-rules, special cases may occur:

- Omission of condition part (EA-rules): Whereas the event part is mandatory, the condition part might be omitted. We refer to this situation as event-action rules. In this case, however, it should be possible to specify conditions as parts of actions.
- Implicit events (CA-rules): In some cases it may be useful to let a compiler or the ADBMS itself generate the event definition. In this case, we say that the event is defined implicitly. The user then specifies conditions and actions, and the ADBMS determines the event automatically (e.g., consider a consistency enforcement mechanism where only constraints and repairs are specified, but the system internally uses events signalled upon modification of data items to determine when the consistency constraint has to be checked). Nevertheless, the ADBMS also has to offer the possibility for the user to define events explicitly at the ADBMS-interface. If explicitly definable events are not provided, ECA-rules are nothing but an internal implementation mechanism for tasks that could also be implemented “passively”, and there is no general support for reactive behavior.

3.2 ECA-Rule Execution Features

Feature 4: **An ADBMS has an execution model.**

Feature 4.a: **An ADBMS must detect event occurrences (situations).**

Ideally, an ADBMS detects event occurrences of all sorts automatically. Less ideally, there are some occurrences that are not detected by the ADBMS and, hence, have to be explicitly signalled by the user/application. If in the extreme application programmers or users are responsible for the correct signalling of *all* sorts of events, this system is just a syntactic variant of a passive DBMS (although users may in addition to other things also have the right to signal events).

Feature 4.b: **An ADBMS must support binding modes.**

Event occurrences are typically *bound* to data items (i.e., an event is bound to the data item for which the event occurred). The kind of binding determines the *granularity* of events as well as the data items that conditions and actions of rules associated to the event can refer to. Different kinds of bindings have been proposed [14]: *instance-oriented* binding means that the particular instances (e.g., tuples or single objects) for which an event occurred can be referred to by conditions and actions. If the same event is bound to several instances within a collection, then the associated rule is executed for each of the instances separately. If the binding is *set-oriented*, then an event occurrence is associated with a set of instances for which the particular event occurred (e.g., all modified tuples of a relation). In this case, an associated rule is executed only once for all the instances the event occurrence is bound to. Conditions and actions can then refer to this set of instances. In both cases, if the binding is *prior*, then the condition can also refer to the state of instances prior to the event occurrence.

Ideally, the ADBMS supports the choice of the binding mode on a per rule basis. Alternatively, the binding mode is fixed and hard-wired in the system.

Feature 4.c: **An ADBMS must be able to evaluate conditions.**

An ADBMS must be able to evaluate conditions subsequent to event detection. For practical cases it is necessary to pass information from events to conditions. If an event has occurred for a specific object or a set of tuples in a relation, it must be possible to refer to this information in the condition.

Feature 4.d: **An ADBMS must be able to execute actions.**

An ADBMS must be able to execute actions upon event detection and after the condition is known to hold. It must be possible to pass information from the event and condition to the action (e.g., information on the object for which the condition held). It should be possible to execute actions as part of the triggering transaction, and as such the action execution should be subject to concurrency control and recovery.

Feature 5: **An ADBMS must offer different coupling modes.**

The two relationships between triggering and triggered transactions mentioned in the terminology section (the point in time when the triggered transction starts with respect to the triggering one as well as dependency of the triggered transaction on the triggering one) are usually specified through *coupling modes*. The originally proposed ones [12] are:

- *immediate*: the triggered transaction is executed directly after the event has been signalled.
- *deferred*: the triggered transaction is executed at the end of the triggering transaction, but before it commits.
- *decoupled*: the triggered transaction is started as a separate transaction.

In the first two cases, triggered transactions are essentially subtransactions of the triggering transaction. In the `decoupled` case, the triggered transaction can be started directly after event signalling, but it is independent from the triggering transaction. Especially, the commit and abort dependencies as present in the `immediate` and `deferred` cases do not exist in the decoupled case. A decoupled transaction can commit or abort regardless how the triggering transaction terminates. Additionally, the triggered transaction runs concurrently to the triggering one.

Further coupling modes have been proposed in the literature. In [8], the decoupled case is subdivided into causally dependent and independent transactions. In the first case, the triggered transaction can commit only if the triggering transaction also commits, and is serialized after the triggering transaction. The causally dependent coupling mode has been refined in [3]. Two new subcases of this coupling modes are proposed: the *sequential causally dependent* mode means that the triggered transaction can only start after the triggering one has committed. In the *exclusive causally dependent* mode, the triggered transaction may commit only if the triggering one has failed.

Immediate and *decoupled* coupling mode are the bottom line. Further coupling modes are optional yet a prerequisite for certain application areas (see section 4).

Feature 6: **An ADBMS must implement consumption modes.**

If composite events are supported by the ADBMS, *event consumption* must be implemented, too. Event consumption determines which component events are considered for a composite event, and how event parameters of the composite event are computed from its components. Different application classes may require different consumption modes, such as "recent", "chronicle", "continuous", and "cumulative" [4]. Either an ADBMS follows a fixed strategy for event consumption (whereby the bottom line is given by the “chronicle” mode), or it offers a choice out of a collection of consumption modes.

Feature 7: **An ADBMS must manage the event history.**

The *event history* or *history* consists of all occurrences of the defined event types (including components of composite events). Such an event history exists for any operational active database system. The history begins at the point in time when the first event is detected. The history may last over many sessions and over several transactions. Thus, a persistent event history is required whenever it should be possible to signal a composite event based on events that have occurred during different application sessions or transactions.

The notion of event history also defines the *lifetime* of event occurrences. Minimally, the event history stores only those occurrences that can still be used for event composition. In this case, no occurrence is persistent if composite events are *not* supported, or only those occurrences persist that might contribute to composite event occurrences sometime in the future. One step further, it can be specified by the rule designer on a per event type basis how long event occurrences should persist. In this case, the event history can be used for further tasks such as monitoring and tracing (see below).

Feature 8: **An ADBMS must implement conflict resolution.**

It can happen that multiple triggered transactions are to be executed at a given point in time. The ADBMS must then perform *conflict resolution*, i.e., either determine a serial order in which these triggered transactions are executed or control their concurrent execution in some way. Since conflict resolution typically depends on the semantics of the rules (which in general is only known by the user), the rule specifier must have the opportunity to define how conflicts are resolved, e.g., by means of priorities. If the user, however, does not want to define conflict resolution, he/she is not obliged to do so, and the ADBMS will either determine some order or execute rules non-deterministically. *Priorities* are one possibility for conflict resolution in ADBMSs [1].

3.3 ADBMS-Usability and Application Features

Feature 9: **An ADBMS should support a programming environment.**

It goes without saying that an ADBMS must be *usable*. The bottom line for usability is the availability of a rule definition language (which of course, may be part of the DDL; see Feature 2 above). In order to assist the user in beneficially using the ADBMS, a number of tools should be provided:

- a rule browser,
- a rule designer,
- a rulebase analyzer,
- a debugger,
- a maintenance tool,
- a trace facility, and
- performance tuning tools (see below).

These tools may be separate tools dedicated for an ADBMS-programming environment, or may be extensions of already existing DBMS- or CASE-tools. Note further that it is not intended to require that all tools are separate systems, we are simply interested in their functionality.

A *rule browser* allows inspection of the set of currently existing rules. The rulebase extends the catalogue (or data dictionary) in passive DBMSs, since it contains meta-information on defined ECA-rules. Clearly, when defining ECA-rules the possibility to survey which rules have already been defined is essential.

The ADBMS should offer a *design tool* that assists users in defining new rules. Such a tool support is crucial when the reactive behavior as required by the universe of discourse has to be systematically mapped into ECA-rules. This support is possible in two not necessarily mutually exclusive ways: either a general design tool also covers ECA-rule design, or the reactive behavior is specified using dedicated high-level languages (e.g., a constraint definition language).

A *rulebase analyzer* is a tool that allows certain properties of the currently existing rulebase to be checked. Examples of such desired properties of rule sets are termination, confluence, and observably deterministic behavior [2]. If the ADBMS supports cascaded rule execution, it is important to ensure that rule execution terminates under all circumstances. Together with the other properties, termination ensures that the current rulebase is safe and correct with respect to the intended reactive behavior. In general, these properties cannot be proven automatically, but an analyzer might assist a DBA in proving them or at least in detecting inconsistencies.

A *debugger* is a tool that allows the controlled execution of rules (and applications) and helps to check whether the rulebase implements the required reactive behavior adequately. Thus, in contrast to proving properties with an analyzer, a debugger supports test-modify cycles.

A *maintenance tool* for an ADBMS supports the user in performing rulebase evolution. In addition to the rule definition facility, it supports deletion and modification of existing rules. This leads to a more integrated environment which must combine the previous tools to check whether these modifications change the behaviour of the rulebase or not.

Finally, a *trace tool* is a facility that records event occurrences and rule executions, such that a DBA[2] is able to inspect which actions the ADBMS has triggered automatically. If such a tool is not supported, the ADBMS might perform actions that users never become aware of.

Feature 10: **An ADBMS should be tunable.**
An ADBMS must be useful in its application domain. Especially, the ADBMS solution must not show significant performance degradation as compared to equivalent solutions on top of a passive system. Besides recent experience reports [15] and benchmarking reports for object-oriented ADBMSs [11], there is little experience with current ADBMSs on how to measure their performance systematically. However, it is apparent that a practically useful ADBMS should offer the possibility to tune its rulebase (whereby of course the semantics of the rules should not change due to tuning!). A feasible approach might be the equivalent to the three-schema-architecture [10]: at the external level, user- or application-specific rules are specified (e.g., for consistency constraints), the conceptual level contains all rules relevant for the community of all users/applications, and the internal level specifies the implementation details. The internal level should then provide for the means for performance improvements, and all details concerning efficient rule execution are captured on this level.[3] This approach may lead to a strong coupling between the design tools and the ADBMS itself, providing a complete methodology for design and implementation of active rules.

4 A Classification of ADBMSs

The various characteristics discussed in section 3 still leave some choises and, hence, some latitude on how to integrate them into an ADBMS. In other words, there are various ways on how to "bundle" their functionalities into an ADBMS. If one attempted a classification into some standard bundles, the only conceivable way to do so seems to consider application classes, to determine their requirements, and then to identify classes of ADBMSs that are appropriate for these application classes. In this way, we determine appropriate and required realizations of the features described above (much like the description of dimensions and possible instantiations [14]).

We classify ADBMSs according to the role they are able to play in an information system, i.e., we discuss which functionality an ADBMS should exhibit in order to meet given requirements of certain types of applications. For that matter, we consider two dimensions:

- the role of the ADBMS in an information system (*monitoring* or *control*), and
- the degree of integration of the information system (*homogeneous* or *heterogeneous*).

Monitoring means that the ADBMS verifies requests for database operations against the database (or vice versa), and eventually performs simple actions (e.g., notification, transaction abort, update propagation). An ADBMS that controls the information system is in addition, able to trigger external functions, e.g., application programs. In this case, the ADBMS is able to control the behavior of the entire application environment (and not only the state of the database), and can do so possibly over a period of time spanning many sessions.

We call an information system "homogeneous" if all of its components are applications of the ADBMS in question, i.e., they share a common schema and common databases. Otherwise, we say that the application is "heterogeneous". Particularly, the ADBMS may have to control systems that are implemented on top of other platforms. Combining these dimensions leads to four classes of ADBMSs.

Below we investigate three of them in more detail; the combination monitoring/heterogeneity is not regarded as meaningful for the following reasons: first, in order to monitor valid requests/database states in a heterogeneous system (e.g., with foreign data sources), one would rather need active mechanisms in the external systems than in the monitoring ADBMS. Second, a monitoring ADBMS would typically not be able to reject requests and database states in external systems. Instead, "real" actions would be triggered, e.g., to reestablish consistency in external data sources, which in turn would then be classified as control.

4.1 ADBMSs for Monitoring in Homogeneous Information Systems

In the simplest case, an ADBMS is required to monitor database states in homogeneous information systems. Such an ADBMS recognizes certain user/application requests and verifies them against the database state (or verifies the database state against the most recent application requests). In this case, the user/application is responsible to enforce the semantic rules defining and constraining the behaviour of the entire information

2. DataBase Administrator
3. In other words, in analogy to passive DBMSs we require physical database design for ADBMSs.

system. Despite its ability to notify the user (e.g., printing messages on the console) and to abort transactions, the ADBMS has no control over the information system, i.e., it will not cause complex application programs to execute.

In this scenario, ADBMS technology is useful for implementing the "usual" DBMS tasks, e.g., (simple) consistency constraint enforcement, authorization, updates of materialized views, etc.). Note that some of these DBMS tasks might use the active functionality only internally as an implementation mechanism that might also be provided in a "passive" way. The benefit of active functionality is then the uniformity and minimality of implementation concepts, but not the support of functionality that would not be possible otherwise.

The kinds of events such an ADBMS can detect are given by the data model, it specifically does not need composite events. Thus, maintaining the event history is not necessary. For conditions, it is sufficient to query the database state and the data dictionary, and actions are DML-commands (including transaction abort).

In general, composite events etc. are not necessarily required, but may be beneficial in some situations. Depending on the concrete application in mind, different kinds of execution semantics may be necessary (the coupling mode `deferred` for consistency maintenance, `immediate` for authorization). For (advanced) consistency maintenance, deltas ("before values") are required.

Not all the functionality required for a given application class might be implementable in this way. However, we would claim that in most cases the aforementioned ADBMS characteristics are sufficient. See Table 1 for a summary of the ADBMS features required by this application class.

Feature	Instantiation
Events (2.a)	DML-operations, not necessarily composite
Conditions (2.b)	predicates on database state / queries
Actions (2.c)	DML-action, user notification
Rule management (3)	create/delete, enable/disable
Coupling modes (5)	at least immediate, deferred
Consumption modes (6)	chronicle
Execution	under local control

Table 1. ADBMSs for Monitoring in Homogeneous Information Systems

4.2 ADBMSs for Control in Homogeneous Information Systems

The second class is formed by ADBMSs that are capable of controlling not only the database, but also its environment (i.e., the applications). ADBMSs in this class are characterized through their ability to control "tightly integrated applications". All the application programs use the same schema, transaction model, DML, etc. Particularly, it is possible to run all triggered activities under the control of the local (ADBMS-) transaction manager.

The role of the ADBMS in the entire information system is then to enforce the "business rules" pertaining to the application domain. An example application domain for this type of ADBMS are stock trading information systems.

It is required that the ADBMS is able to encode (at least a substantial part of) the information about the application environment in the form of ECA-rules. The ADBMS is able to detect states or sequences thereof of the information system and to perform automatic reactions, including the automatic spawning of application programs. Applications are tightly integrated, and the active mechanisms are part of the homogeneous DBMS underlying the information system.

Everything that is provided by the first class of ADBMSs must be available in this kind of system, too. Additionally, in order to control the DBS environment, including the applications, more event types are necessary (e.g., time events). The ADBMS has to keep track of the relevant part of the event history, and must also be able to evaluate restrictions on this event history. Technically speaking, composite events, event restrictions, and monitoring intervals (or equivalents thereof) must be provided. Composite events are necessary in order to control and monitor complex sequences of situations in the DBS-environment. Composite event restrictions (such as referring to the triggering transaction —"same transaction"—) must also be provided. Likewise, a broad variety of rule execution semantics must be supported (i.e., when the rule is executed, and how its execution relates to the triggering transaction). Thus, the coupling modes `immediate`, `deferred`, and `decoupled` are the bottom line.

See Table 2 for a summary of the ADBMS features required by this application class.

Feature	Instantiation
Events (2.a)	DML-operations, external events, composite
Conditions (2.b)	boolean function, including predicates on database state / queries
Actions (2.c)	DML-action, user notification, external programs
Rule management (3)	create/delete, modification, enable/disable
Coupling modes (5)	at least `immediate`, `deferred`, `decoupled`
Consumption modes (6)	choice, including chronicle
Execution	under local control

4.3 ADBMSs for Control in Heterogeneous Information Systems

The third class is formed by ADBMSs that are capable of integrating possibly heterogeneous and autonomous systems. The active mechanism enables the ADBMS to perform control of such heterogeneous, loosely integrated component systems.

Example application domains where this kind of ADBMS functionality would be helpful are advanced workflow management systems, reactive behavior in heterogeneous DBSs, real-time plant control systems, and process-centered software development environments.

In addition to the capabilities of the second class described above, such an ADBMS has to be able to detect situations in other information systems (which themselves may be based on other DBMSs), thus affecting the event definition and detection facilities. It might also be necessary to detect events from external devices.

Most important, powerful rule execution mechanisms are necessary, since it might be the case that triggered actions cannot be executed under the control of the local transaction manager. The rule execution model must support complex relationships among application steps (e.g., compensation, ordering of application steps, dependencies).

Furthermore, if such an ADBMS is intended for real-time applications, it should support the specification of timing constraints for rule executions. The rule definition should also comprise contingency actions, which are executed whenever the timing constraint of a rule cannot be met. Clearly, an ADBMS for real-time applications should also possess the properties required for a "passive" real-time system.

Summarizing, in this type of ADBMS parts of the "middleware" can be moved into the DBMS, i.e., the active mechanism contributes to implementing the middleware in the ADBMS. Since the ADBMSs of this class are intended for loosely-coupled, possibly heterogeneous systems, their integration into software architectures that aim at mediation in such environments should be possible. Especially, it should be possible to integrate ADBMS-functionality into OMG's CORBA architecture [6]. Relevant services provided by the ADBMS would then refer to event definition, registration, notification, etc.

See Table 2 for a summary of the ADBMS features required by this application class.

Feature	Instantiation
Events (2.a)	DML-operations, external events, composite
Conditions (2.b)	boolean function, predicates on database state
Actions (2.c)	DML-action, user notification, external programs, contingency actions
Rule management (3)	create/delete, modification, enable/disable
Coupling modes (5)	`immediate`, `deferred`, `decoupled`, causal dependencies
Consumption modes (6)	choice, including chronicle
Execution	not fully under local control

Table 2. ADBMSs for Control in Heterogeneous Information Systems

5 Conclusion

In this paper, we have described what we regard the current confluence of perceptions of "active database management system". We identified the critical features, and attempted to identify "classes" of ADBMSs with respect to the application classes they are useful for.

Several systems already exist that exhibit most of these features, and commerical products come more or less close to what has been identified by this manifesto as requirements. As far as the conceptual state of the art is concerned, ADBMS technology thus seems quite mature. Nevertheless, a large bulk of work remains to be done, particularly to provide for systems that implement these features in a robust, efficient and well-defined way, and to gather practical experience such that users may employ ADBMS technology in a systematic manner for applications for which ADBMSs appear useful.

6 Acknowledgments

We gratefully acknowledge the funding of ACT-NET by the Commission of the European Union (contract no. CHRX-CT93-0089). The University of Zurich acknowledges the funding of its ACT-NET participation by the Federal Department for Education and Science (BBW, contract no. BBW Nr. 93.0313).

Furthermore, we are indebted to Martin Kersten for many detailed and knowledgeable comments on a prior version of this paper.

7 References

1 R. Agrawal, R.J. Cochrane, B. Lindsay: *On Maintaining Priorities in a Production Rule System*. Proc. 17th VLDB, Barcelona, Spain, September 1991.

2 A. Aiken, J. Widom, J.M. Hellerstein: *Behaviour of Database Production Rules: Termination, Confluence, and Observable Determinism*. Proc. SIGMOD, San Diego, June 1992.

3 H. Branding, A. Buchmann, T. Kudrass, J. Zimmermann: *Rules in an Open System: The REACH Rule System*. Proc. Rules in Database Systems. Workshops in Computing, 1994.

4 S. Chakravarthy, D. Mishra: *An Event Specification Language (Snoop) for Active Databases and Its Detection*. Technical Report UF-CIS TR-91-23, CIS Department, University of Florida, September 1991.

5 S. Chakravarthy (ed): *Special Issue on Active Databases*. Bulletin of the TC on Data Engineering 15:1-4, 1992.

6 *The Common Object Request Broker: Architecture and Specification*. OMG Document 91.8.1, August 1991. © Digital Equipment Corp., Hewlett-Packard Company, HyperDesk Corp., ObjectDesign Inc., SunSoft Inc.

7 U. Dayal: *Active Database Management Systems*. Proc. 3rd Intl. Conf. on Data and Knowledge Bases, Jerusalem, 1988.

8 U. Dayal, M. Hsu, R. Ladin: *A Transactional Model for Long-Running Activities*. Proc. 17th VLDB, Barcelona, Spain, September 1991.

9 K.R. Dittrich: *Object-Oriented Database Systems: The Notions and the Issues*. In K.R. Dittrich, U. Dayal, A.P. Buchmann (eds): On Object-Oriented Database Systems. Topics in Information Systems, Springer 1991.

10 R. Elmasri, S.B. Navathe: *Fundamentals of Database Systems*. Benjamin/Cummings Publishing, 1989.

11 A. Geppert, M. Berndtsson, D. Lieuwen, J. Zimmermann: *Performance Evaluation of Active Database Management Systems Using the BEAST Benchmark*. Technical Report 96.01, Department of Computer Science, University of Zurich, February 1996.

12 M. Hsu, R. Ladin, D. McCarthy: *An Execution Model for Active DBMS*. Proc. 3rd Intl. Conf. on Data and Knowledge Bases, Jerusalem, Israel, June 1988.

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

14 N.W. Paton, O. Diaz, M.H. Williams, J. Campin, A. Dinn, A. Jaime: *Dimensions of Active Behaviour*. Proc. Rules in Database Systems. Workshops in Computing, Springer-Verlag, 1994.

15 E. Simon, A. Kotz-Dittrich: *Promises and Realities of Active Database Systems*. Proc. 21st VLDB, Zurich, Switzerland, September 1995.

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