# University of Florida

Computer and Information Sciences

# Architectures and Monitoring Techniques for Active Databases : An Evaluation

**Sharma Chakravarthy**

email: sharma@snapper.cis.ufl.edu

**UF-CIS-TR-92-041**
**(Submited for publication)**

(This work was (in part) supported by the Office of Naval Technology and the Navy Command, Control and Ocean Surveillance Center RDT&E Division and in part by the NSF grant IRI-9011216. Also, part of this work was carried out when the author was at Xerox Advanced Information technology, Cambridge, MA and was supported by the DARPA and Rome Air Development Center under contract No. F30602-C-0029.)


**Department of Computer and Information Sciences**
**Computer Science Engineering Building**
**University of Florida**
**Gainesville, Florida 32611**

**Abstract**

The need for active capability for non-traditional applications and its concomitant benefits are well-established. Although the event-based technique for monitoring conditions (leading to the integrated architecture) is the most versatile of all the techniques, from a practical viewpoint there is a need for enhancing pre-existing non-active DBMSs to support active capability. The set of techniques that can be used for providing this add-on active capability (leading to the layered architecture) imposes limitations on the active capability that can be supported. Insights into the details of techniques as well as their impact on the architecture entails a better design that meets the active database objectives.

This paper identifies a repertoire of techniques for condition monitoring and discusses their suitability to different architectures. This paper argues that from a pragmatic viewpoint, both *layered* and *integrated* approaches to support active capability need to be pursued. Then it compares *polling* and *event-based or asynchronous* monitoring techniques using an implementation of flavors on Symbolics. The focus of this comparison is on: techniques, performance, influence of implementation strategies on performance, and identification of opportunities for optimization.

**Index Terms:** Active databases, Polling, Performance evaluation, Implementation strategies, Layered approach, Object-oriented design

# 1 Introduction

The concept of monitoring conditions, which forms the basis for supporting active capability, is not entirely new. ON conditions in programming languages and early DBMSs [Oll78], and signals in operating systems [Bac88, Geh84, Hug79, LMKQ89] have been used early on. Later, Artificial Intelligence (AI) systems have used *daemons* for asynchronous execution of procedures attached to frame slots [CRM80, WB79]. Furthermore, multi-paradigm systems, such as LOOPS [BS83] and KEE [Int85] have incorporated active values as a new technique that generalizes asynchronous rule processing. However, most of these systems do not support typical database functionality, such as data sharing, consistency, and multi-user execution.

A large class of non-traditional applications, such as process control, threat assessment and analysis, air traffic control, Computer Integrated Manufacturing, and cooperative problem solving, need to react (often subject to timing-constraints) to a variety of conditions that are defined over the database state and events that change the state of the database. Hence, there is a critical need for providing a generic capability which: is at a higher level of abstraction (than ON statements and daemons), has well-defined semantics, is efficient, and is tailored to the specific needs of Database Management Systems (DBMSs).

In contrast, traditional DBMSs are *passive*. They only respond to external requests in the form of transactions/applications to change their state. Hence, the current trend is to augment a conventional DBMS with active capability. The effect of condition monitoring can also be achieved either by encoding condition evaluation as part of the application program (equivalent to posing external queries) or by polling the database (periodically) to detect whether any of the conditions have

become true. Encoding condition evaluation within the application program not only transfers the burden – of determining the conditions, formulating queries for these conditions, and the timing of their evaluation – to the application programmer but also interferes with application development. Furthermore, optimization of such conditions is extremely difficult. Polling seems to waste resources and transfers the onus of determining the frequency of polling to the user/application developer. Polling frequency is likely to be dependent on a variety of parameters such as the frequency of update, timeliness (the time window within which the condition needs to be detected) etc.

Active databases typically accept a declarative (non-imperative is perhaps more accurate) specification of situation-action rules (or event-condition-action or ECA rules) and manage their execution. Each rule consists of an event, a condition, and an action; when an event occurs, the condition is checked and if it evaluates to true the action is executed. Support for these rules enable a database system to react to changes in its state and perform specified actions (e.g., alerting application programs, notifying users, restoring the consistency of the database, or denying access) asynchronously and without user/application intervention.

Most of the work on active database systems [C+89, DBAB+88, RCBB89, SR86, SHP87, DB87, DKM86, KDM88, WCB91, WF90, GJ91, GJS92, Han89, Int90a, Int90b, Anw92, CM91, CHS93, GrD93, DPG91] is aimed at supporting some form of rule processing capability (e.g., alerters, triggers, situation-action rules) and techniques for their management and optimization (e.g., lazy, eager, overlapped execution). Active capability is viewed as a unifying mechanism for supporting a number of DBMS functionality, such as integrity/security enforcement, maintenance of materialized (e.g., view) data, constraint management, and rule-based inferencing.

Although a number of active database issues such as, expressive event specification language [CM91, GJS92, HJ91], integration of ECA rules into an object-oriented database [Anw92, GJ91, GrD93] and others [Tan92, C+93, CN90] are being pursued vigorously, to the best of our knowledge, there is no work on the discussion of architectural alternatives, techniques for condition monitoring, and evaluation and comparison of the suitability of these techniques. In this paper, we identify a suite of techniques for condition monitoring and analyze their appropriateness for the proposed architectures. We also compare two of the techniques identified from the viewpoint of their impact on the architecture. Towards this end, we designed and implemented active capability for an object-oriented data model. Specifically, the thrust of this work is towards answering basic questions, such as the ones listed below, rather than an in-depth performance evaluation.

> Is asynchronous condition monitoring always better than polling?,
> Should polling be retained as a viable alternative to be used by the system?,
> What issues need to be considered when an existing DBMS is made active?, and
> What techniques are meaningful for a given architecture?

The *prototype* object-oriented DBMS[1] was designed and implemented on a Symbolics machine using Symbolics Common Lisp and flavors. This prototype was modularly extended to include active objects for supporting automatic condition monitoring in addition to other useful DBMS functions. Needless to say that transaction processing and concurrency control were not included in the prototype.

[1] The prototype has the functionality of a subset of PROBE – *a passive*, extensible DBMS developed at CCA [D+85, DMB+87].

The remainder of this paper is structured as follows. In section 2 we briefly overview a subset of the active database issues that are relevant to this paper. Section 3 identifies several techniques useful for supporting active capability along with an analysis and proposes alternative active database architectures. Section 4 describes the implementation of an object-oriented DBMS using Common Lisp. Section 5 discusses the design and implementation of active objects and the functionality of the resulting prototype. Section 6 describes the evaluation of the approaches and contrasts the expected behavior with the observed behavior and provides an analysis of the observed results. Section 7 contains conclusions and future work.

## 2 Active Database Issues

There seems to be consensus among the researchers in the database community that support for ECA (event-condition-action) rules is at the core of active capability. However, the semantics of rule execution in the context of databases requires that several additional issues be considered [WF90, Cha89, C$^+$89]. Here we include only those issues that are relevant to the rest of the discussion in the paper:

- Rule execution points: In HiPAC [HLM88], three coupling modes for rule execution were introduced to support application needs. Their semantics with respect to triggering transactions is defined as follows: in the *immediate coupling* mode a rule is executed at the point where the event occurs, in the *deferred coupling* mode a rule is executed at the end of the transaction prior to its commit, and in *detached coupling* mode a rule is executed as an independent transaction. A causally dependent variation of the detached mode was introduced in which the independent rule transaction is not committed unless the triggering transaction commits. These modes can be specified on a finer granularity (i.e., independently between event and condition as well as between condition and action).
- Rule scheduling: When several rules are triggered at the same time, there has to be a policy for their order of execution. Either a conflict resolution strategy can be used to totally order the rules or traditional serializability theory can be applied to execute rules concurrently. Starburst [WF90] uses the former approach whereas HiPAC [HLM88] uses the latter.
- Nested rules: When rule actions can trigger other rules, there is potential for nested (or even cyclic) rule execution. Again scheduling strategies (depth-first, breadth-first etc.) for these rules need to be outlined.
- Rule management: If rules are to be shared by applications (like any other shared data), then modification of rules should be possible. This entails subjecting rules to the same concurrency control mechanism used for any other shared data. Otherwise, rules have to be treated as meta-data whose manipulation is deemed different from shared data.

## 3 Active Database Architectures

In this section, we identify and describe various techniques for condition monitoring which lead to different architectures for supporting active capability. We further analyze and highlight the need, advantages, and limitations of architectures as well as condition monitoring techniques.

### 3.1 Condition Monitoring Techniques

A number of techniques can be employed for accomplishing the (re)active capability in a database. Below, we identify some of them and discuss their advantages and disadvantages with respect to the issues outlined in the previous section.

- **Busy waiting:** Typically used in single user or single CPU systems without multi-programming or multi-tasking. In these systems, it is not possible to use the CPU for any other purpose. This technique is used only in custom-crafted process control applications where the time taken to execute the loop in which readings are collected is carefully adjusted to the frequency with which data has to be gathered. Also, in some of the process control applications, the busy-waiting period is utilized for performing computations (e.g., refresh display).
- **Encoding or embedding situation monitoring in applications:** In this approach, the application developer includes condition monitoring code at appropriate places in the application code. The main advantage of this technique is that it does not require any changes to the underlying system as all the condition monitoring is done by the application developer as part of the application. A secondary advantage of this approach is that various coupling modes proposed in HiPAC [HLM88] (except the causally dependent mode) can be easily implemented. The main problems with this approach are: i) the application developer has to be aware of the conditions to be monitored and code it as part of the application, ii) the condition monitoring code is not maintained or managed by the system and hence cannot be shared among applications, iii) software maintenance of application programs is extremely difficult as there is no modularity and no reusability of code. Although a library approach for condition monitoring is possible in principal, at least in the host language context it is extremely difficult to accomplish. Data Base Programming Language (DBPL) environments will be more amenable to this approach.
- **Polling:** In this technique, each condition is checked with a pre-determined polling frequency and if the condition evaluates to true then appropriate actions are executed. However, the frequency of polling (or the polling interval) needs to be determined for each rule or classes of rules. The polling interval needs to take into account the size of the relation (as one may have to check all the tuples for a condition), the complexity of the condition to be evaluated, and the frequency of update. In general, it would be a burden on the part of the user (or even the Data Base Administrator or the Data Base Customizer) to explicitly indicate the polling frequency. This technique does not lend itself to coupling mode enforcement and nested execution of rules. Also timeliness of checking is tied to the polling interval and hence this technique may not be suitable for monitoring rules with time-constraints. In this paper, we present some conclusions on the feasibility of this approach as an alternative that the system can use for implementing situation monitoring.
- **Aperiodic checking of situations:** This is a variation of polling in which the system or a component of the system checks whether the conditions are true aperiodically by evaluating queries. The event which triggers this evaluation has to be specified explicitly or determined by the system. This approach will allow conditions to be specified once but the checking has to be explicitly invoked. Again, timeliness of checking is not guaranteed and hence this technique is not suitable for monitoring rules with time-constraints.
- **Asynchronous monitoring:** This is similar to interrupt-driven processing where the system is responsible for the detection of events, evaluation of conditions, and execution of actions.

| Basic Techniques | Architecture | Advantages | Disadvantages |
|---|---|---|---|
| busy waiting,<br>encode applications | Application-based | no changes to the<br>underlying DBMS | application managed<br>condition monitoring,<br>rules cannot be shared,<br>no modularity |
| busy waiting,<br>polling,<br>aperiodic checking | Layered | developed as add-on | no nested rules,<br>limited coupling modes |
| polling,<br>asynchronous monitoring | Integrated | versatile,<br>enhanced DBMS<br>functionality | enhancement needed<br>at the kernel level |

Table 1: Techniques-Architecture summary

This approach has the potential for timely detection of events and execution of actions as well as opportunities for optimizing conditions and actions that are specified to the system declaratively. This is the most versatile of the techniques outlined here. However, this technique requires extensions to the core of the DBMS and a re-examination of the DBMS architecture to accommodate active capability. This technique lends itself very well for addressing various active database issues outlined in the previous section.

### 3.2 Architectures

Table 1 depicts three distinct architectures that result from using one or more of the techniques listed above.

In the application-based architecture the burden of monitoring is on the user and can use either busy waiting or encode condition checking as part of applications. No generality is lost as the user can control when to check the conditions (either at the point of state change or later). Certain coupling modes (for example, causally dependent) cannot be supported in this approach. The main drawbacks are that the same condition checking code is distributed among application programs violating modularity and software engineering principles, leading to difficulties in software maintenance.

Figure 1 shows the layered architecture. A number of condition monitoring techniques described earlier are applicable here. The underlying DBMS is augmented with a layer that is responsible for providing active capability. The architecture shown permits access to the augmented system either through a user interface tool that transforms user active database design to underlying system constructs [Tan92] or through a stand-alone interface. Although full active capability cannot be obtained in this approach, a number of techniques can be used and some optimizations can be performed by the situation monitor layer. For instance, the layer can decide whether to rewrite a transaction to include the condition monitoring code (similar to the application-based architecture but the rewrite is done by the situation monitor layer) or use either the polling or aperiodic checking approach depending upon the meta-data used by the system. All applications that require active capability have to interact with the system through this interface; otherwise, active capability will

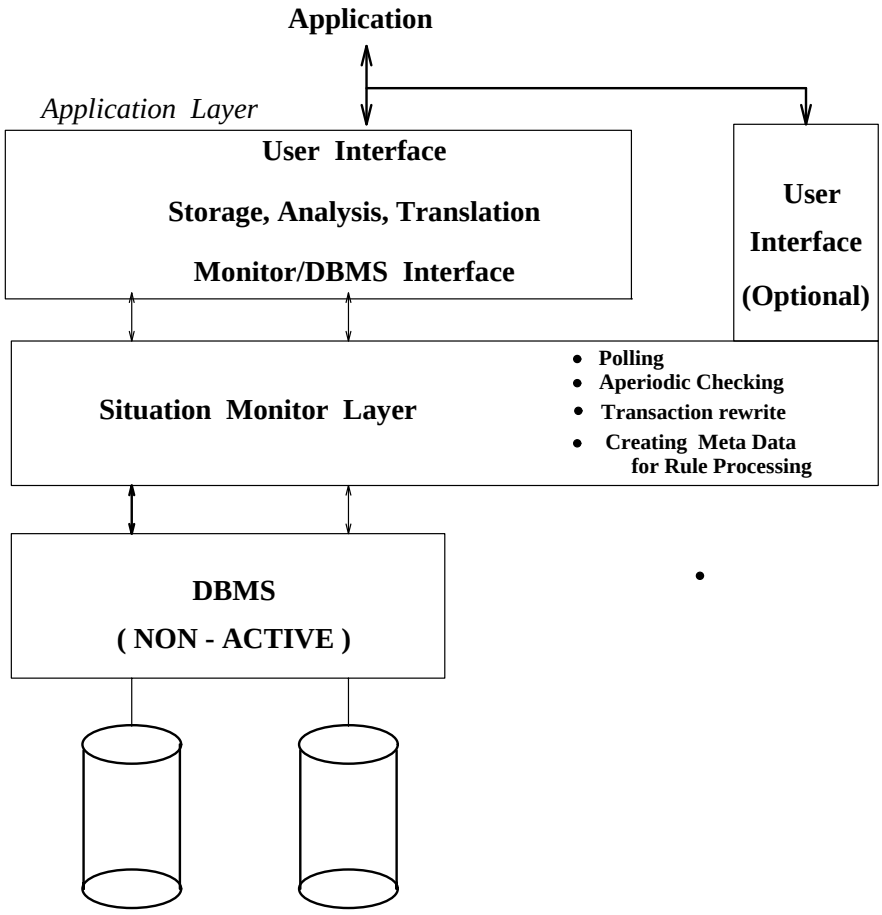


Figure 1: Layered Active Database Architecture

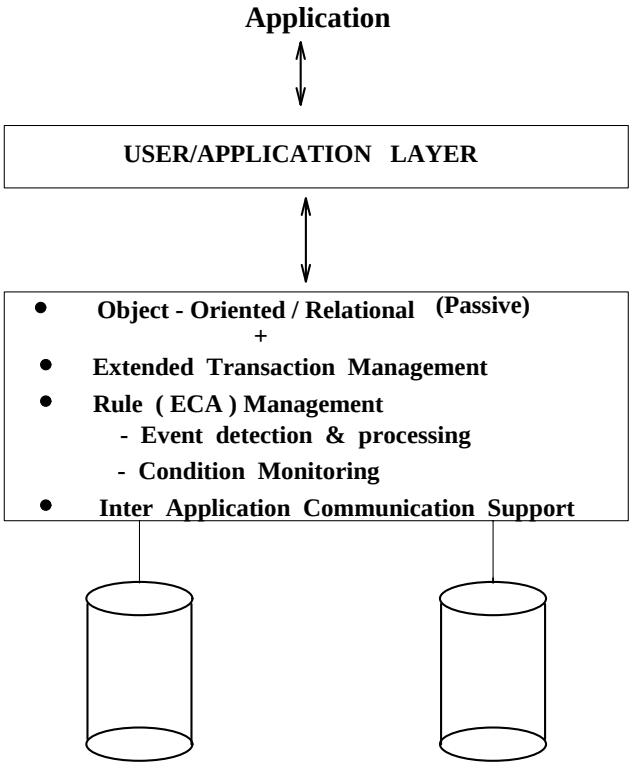


Figure 2: Integrated Active Database Architecture

not be available. Alert [S+91] proposed a layered architecture.

Layered Architecture requires that a layer be built (on top of a passive DBMS such as ORACLE or INGRES) through which all the ECA rule specifications are routed. The layer is responsible for monitoring the situations and executing appropriate rules which also means that all transactions are routed through the layer (although eventually processed by the underlying system's transaction manager). There may be some limitations on the class of ECA rules that can be supported using this approach. For example, immediate mode coupling may not be possible as the layer may not be able to suspend a transaction that is being executed by the underlying DBMS. Also, explicit and other temporal events cannot be supported in this approach without resorting to polling. A number of optimizations can be incorporated when the cardinality and frequency of database operations are known. It may even be possible to cache some data in the situation monitor layer for condition monitoring purposes.

Not surprisingly, most of the research developmental efforts on active databases have opted for the integrated architecture shown in Figure 2. In this architecture, the kernel functionality of a passive DBMS is enhanced to include event detection, condition monitoring, and an extended transaction management to support concurrent rule processing. An integrated approach does not necessarily have to support full active capability as evidenced by commercial systems such as Sybase [DB87], Interbase [Int90a, Int90b] and others. All the active database issues outlined earlier are being addressed in various research efforts in the context of an integrated architecture. Currently, Ode [GJS92, GJ91], Sentinel [CHS93, Anw92], SAMOS [GrD93] are some of the efforts that are integrating active capability into an object-oriented database.

### 3.3 Analysis

Of all the architectures discussed above, the integrated architecture is the most versatile and flexible which does not have any of the problems associated with the other two architectures. However, there are compelling reasons for pursuing the layered architecture:

1. To provide a migration path between currently used non-active DBMSs that are **not** likely to be replaced in the near future with systems that have some active capability, and
2. To support the integration of a number of pre-existing DBMSs with differing (or no) active capability.

From a practical viewpoint, both 1) and 2) are extremely important. It is unlikely that currently used non-active systems will be replaced by active systems in the near future. However, the need for active functionality is clear and the layered architecture can play a significant role in providing a useful migration path.

Currently, there are no tools or interfaces available that support either the design of active databases or even rule specification at the application level and translate them to ECA rules. Irrespective of the architecture used, there is a need for an interface that will accept trigger/rule specifications from the user and translate them into a form accepted by the layered architecture (shown in Figure 1) or by the integrated active system (shown in Figure 2). In the case of the layered approach, it is possible to combine both the situation monitor layer and the application level trigger specification interface into a single one. The application shown in Figure 1 is a tool that supports design interface at one end (extended ER diagram, for example) and generates ECA rules either for the layered or for the integrated architecture. Such a system is being implemented in [Tan92, NTC93].

Figure 3 shows the utility of the architectures proposed in this paper. In order to support federated/heterogeneous active DBMSs, it is critical that the underlying systems support some active capability [2]. The approach proposed in this paper will help address some of the problems faced by a global transaction manager [SRK92] in a multidatabase environment. Also, federated active databases is likely to be useful for enforcing interdatabase dependencies [SLE91] as the situation monitor layer (or the active capability of the integrated architecture) can be used to reveal some of the information in the underlying system to help make global decisions. For example, if each system can inform the global transaction manager before the commit of a transaction, then the global transaction manager can use that information for global serializability. Currently, we are researching the issues related to multidatabase constraints using this framework.

## 4 Design of a Prototype Active OODBMS

For the purposes of this effort, a subset of PROBE[3], excluding spatial/temporal objects and recursion, was designed and implemented on Symbolics using Common Lisp and its object-oriented

[2] otherwise this has to be built as the part of the mediator layer; however, as the layer is likely to have differences depending upon the underlying system, it is best done as part of the system rather than as part of the mediator layer.

[3] PROBE [MD86, RHDM86, D+87], supports a rich data model incorporating spatial/temporal objects as well as recursion at the kernel level. The Probe Data Model (PDM) provides *objects* and *functions* as basic constructs. An *object* is used to represent a real world object. *Functions* can be applied to objects to obtain properties of objects, invoke operations on objects, and describe relationships among objects. Objects and functions are manipulated

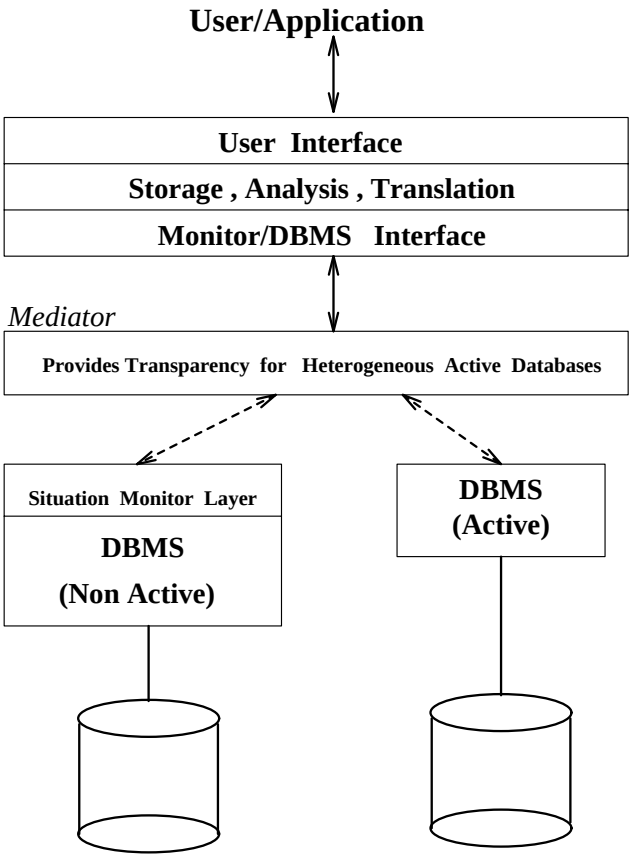


Figure 3: Heterogeneous Active Database Architecture

Figure 4: Object hierarchy used in the prototype

extension, flavors. In this version, stored and virtual relations are supported. A generalization of relational algebra is used as the query language.

The algebra consists of relational operators (such as select, project, natural join etc.) with suitable extensions (apply-and-append, for example) to support the PROBE object classes. The system is composed of a hierarchy of objects and operations on objects. For example, relation is an object consisting of file objects, attribute objects and other instance variables. These objects are in turn built using other objects. Figure 4 shows the object hierarchy (including the objects introduced for active condition monitoring) used for supporting the relational model. Basic object of Figure 4 is inherited by all other objects (not shown to keep the diagram comprehensible).

A simple user interface that embeds algebra operators in LISP has been implemented. It allows the use of lists, strings, numbers, and conditional expressions using LISP syntax instead of objects and functions.

### 4.1 Active Objects

The conceptual ingredients of condition monitoring are: events, conditions, and actions. An event is an indicator of a happening (recognized by the system or signaled by an application program/user). For example, events such as an arithmetic overflow or a timer underflow are recognized by the hardware whereas events corresponding to updates (e.g., insert, delete, and modify) are detected

through an algebra that is a generalization of the relational algebra.

by the DBMS software. A condition is a predicate over the database state that evaluates to a boolean value. (In the presence of free variables, the predicate is false if there are no bindings to the free variables.) Actions are operations to be performed when an event occurs and the condition associated with the event evaluates to true. Actions can be arbitrary programs. Actions, conditions and events can be packaged in various ways.

Situation-action rules (henceforth rules) have been used to group a condition and its associated action. One or more rules can be associated with a specific event. When an event occurs, one or more conditions are evaluated and corresponding action(s) invoked.

Our design assumes that a set of primitive events (such as read, write, execution of a function, clock event) on objects can be detected by the underlying system (either by the operating system or the kernel of a DBMS). These events are independent of the model and the object classes in the system. Model and object specific events can be built from these events as we will demonstrate later. A condition and its associated action is packaged as a rule in our prototype and is implemented as instances of an object class (*active-object*).

We have introduced two new object classes - *active-object* and *activelist-object* which provide the basis for supporting condition monitoring in the prototype. The object class *active-object* encompasses the functionality of a rule including scope and context information. The object class *activelist-object* has been introduced to support multiple rules for a specific event. These object classes, along with the methods defined on them and the handlers for read and write events, provide all the necessary mechanisms for making the prototype DBMS *active*.

The *active-object* is a Common Lisp Flavor consisting of the following components:

*local* - to hold the value of an object,
*readfn* - a function to be executed on a read event,
*writefn* - a function to be executed on a write event,
*relation-id*, *tuple-id* - context variables (for the relational model[4]), and
*hist-vars* - history variables.

The *readfn* stores the rule (in the form of a LISP function that may include calls to database operations) to be executed on a read event. When an object with which an *active-object* is associated is read, the *readfn* of the active-object is executed passing the current value of *local*, the context, and history variables as parameters. The read function, usually, returns the value of the object as its final action. As part of the readfn execution, complex computations (including side-effects) can be performed. A NIL value for *readfn* is interpreted as a no-op and the value of *local* is immediately returned. Similarly, the *writefn* is a function to be executed when the object (with which the *active-object* is associated) is assigned a new value. The final action of a write function usually changes the value of *local*. Again, a NIL *writefn* value immediately assigns new value to *local*.

The *local* component of an *active-object* is a place holder for storing the value of the data object with which an *active-object* instance is associated. The value of *local* is used by the functions – *readfn* and *writefn*.

The *relation-* and *tuple- ids* are essentially variables providing the context in the form of the relation and the tuple identification, respectively. Storing references to the relation and the tuple provides context for the *active-object*'s read and write functions. The context information has

[4]The discussion is cast in the relational framework for comparison purposes, although the design is object-oriented.

several potential uses. First, the name, type, and the details of the object can be extracted from the tuple or the relation. Such accesses are often part of a triggered action. Second, the information in the tuple or relation may be used in computation performed by the read and/or write functions. For example, a function which computes the amount of time it would take an enemy ship object to reach a monitoring ship object must have access to the X and Y coordinates, speed, direction of the enemy ship object, and possibly other information about the object or other objects of the same type. Some of this information may be contained in the active object itself. The rest can be obtained from the tuple or relation containing the active object. Finally, it is useful to know the relation instance or the tuple instance containing the active object in order to pass it to a function called from within the rule. In our applications, we use the context information in all these ways.

The history variables can be used for a variety of purposes. Specifically, they can be used to store information (such as a previous value, the number of times the object is accessed or modified, an aggregate value computed using other objects or even a complex derived value) to be used for optimization and for optimizing rule evaluation.

The *activelist-object* is also a Common Lisp Flavor consisting of a list of *active-object* instances. For the sake of uniformity, an *activelist-object* is always used to make an object *active*.

## 4.2 Semantics of Active Objects

The semantics of an *active-object* can be easily inferred from the way *readfn* and *writefn* are invoked. The semantics of nested active objects is as follows. For read events, the *readfn*'s are executed inside out. In other words, the first operation is a read of *local* which may result in executing a nested *active-object*. The value of *local* generated for the innermost *local* is passed to the outer level *readfn* and so forth. Analogously, for write events, the *writefn*'s are executed outside in. Functions for both read and write events can be nested to arbitrary levels.

In the case of multiple rules, if rules are to be executed one at a time, then a 'conflict resolution' strategy is required to choose the appropriate rule. If multiple rules can be executed when an event occurs, they can be executed concurrently. In either case, ordering of the rules based on some criterion (e.g., priority) is useful. In the prototype, the read and write functions of an active object are executed in the order in which they appear in the *activelist-object*. The implementation of read function assumes that at least one of them will return the value of the object and the first non-NIL value returned is used as the value of the object. For multiple write functions, each of them is executed in the order specified. In this case, it is assumed that values are kept consistent by the functions. Other strategies for evaluating multiple rules are possible.

In addition to the *active-* and *activelist-object*, a set of methods are defined over the object classes introduced above. They include methods for: creating *active-object* instances, associating them with attributes and tuples of a relation, activating and deactivating them, and executing rules for read and write events.

## 4.3 Implementation of Event Handlers

In order to perform condition monitoring effectively and efficiently, events of interest (read and write on object instances in our case) need to be recognized. A variety of techniques can be used for recognizing the events of interest and an important criterion is to keep the overhead of the

detection of events to a minimum. New types of locks were introduced into the lock table for recognizing various types of events in the initial design of POSTGRES [S+86]. As it is necessary to have the locks for condition monitoring survive system failures, it seems appropriate to associate the mechanism for recognizing events with the data items (which survive system failures). This approach is also beneficial when the number of active objects are large as it distributes the locks among object instances rather than storing them in a single data structure.

The approach taken for the prototype already treats events as functions (read and write) or methods defined for a component of an object class. Hence, a component of an object class needs to be made active only at the object class level and will be inherited by all instances of the object class. It is clear that the access functions need to be intercepted in order to evaluate conditions and execute actions associated with the events.

Choice of the strategy for implementing event handlers was based upon:

1. Simplicity of the technique (without sacrificing the functionality of the resulting system),
2. Ease of implementation (one that would not require extensive modifications to existing code), and
3. Ease of maintenance (a single version of code which can be made active incrementally).

The following alternatives were considered:

1. Define our own access functions for components of object classes of interest,
2. Use error signaling mechanism to get control of the occurrence of an event,
3. Use before- and after-daemons to get control of the occurrence of an event, and
4. Use whoppers[5] combined methods to intercept the invocation of methods.

The first three options (listed above) were rejected as they did not satisfy our criteria. The first alternative involves extensive changes to the existing code in terms of introducing a user-defined access function for each component of an object we wanted to make active. Also, this option entails an implementation at the application level overriding the system level access functions. This approach would also have resulted in maintaining two versions of the system. Hence, this option violated the ease of implementation and ease of maintenance criteria.

The second option is mostly used for debugging and hence lacked the generality and simplicity required for our purpose. The third alternative permits one to extend the primary method of an object class using the before- and after- daemons. Briefly, before- and after- daemons permit adding code before and after the execution of a function or a method associated with a flavor (object class). This mechanism is very useful for combining independent methods of component flavors, but does not permit the binding of variables during the execution of a method. Also, this mechanism does not provide the flexibility of skipping the method for which before- and after-daemons are defined.

[5]There is another mechanism called *wrappers* which also permits adding code *around* the execution of any method. However, there are some fundamental differences between them. A wrapper is similar to a macro whereas a whopper is similar to a function; if a wrapper is modified, all combined methods using it must be recompiled whereas if a whopper is modified only the whopper needs to be recompiled; the body of a wrapper is expanded (by duplicating code) in all the combined methods in which it is involved whereas the body of a whopper is not expanded in multiple places. Whoppers, on account of the above differences, are slightly slower than wrappers.

```
;;; definition of stringobj object class (flavor)

(defflavor stringobj (string)
   (obj)
  :initable-instance-variables
  :readable-instance-variables
  :locatable-instance-variables
  :writable-instance-variables)

;;; a whopper around the read function of string component of stringobj

(defwhopper (stringobj-string stringobj) ()
  (if (typep string 'activelistobj)
      (p_triggerreadfns string)
      (continue-whopper)))

;;; a whopper around write function of string component of stringobj

(defwhopper ((setf stringobj-string) stringobj) (new-string)
  (if (typep string 'activelistobj)
      (p_triggerwritefns string new-string)
      (continue-whopper  new-string)))
```

Figure 5: A Sample whopper

The last alternative provided the type of control and at the level that satisfied our objectives. Whoppers provide a means for wrapping code *around* any method including system generated methods for an object class. Furthermore, it is a general mechanism that permits one to gain control prior to the execution of any method (both system defined as well user defined) and permits passing of parameters to the actual method itself. This mechanism is superior to other alternatives in providing a lexical scope within which the actual method itself can be invoked (optionally, of course) and local variables can be created and passed as parameters. In addition, whoppers delay the detection of triggers as far as possible thereby reducing the overhead incurred for condition monitoring.

Whoppers need to be defined as a separate function for each flavor whose components are active. One has to only load the whopper definitions in order to make objects active in this scheme and hence does not require any change to existing code. Furthermore, object instances can be made active incrementally. This incremental way of making objects active keeps the two versions of code used for comparison identical, except for the whopper code execution at run time. This makes our results and hence conclusions very reliable.

Figure 5 shows the definition of string-object and whoppers for read and write methods. Figure 6 shows the definition of the active object and how it is created. Note that the readfn and writefn are passed as parameters.

```
;;; -*- Mode: LISP; Package: USER; Base: 10; Syntax: Common-lisp -*-

;;; a flavor for active objects consisting of
;;; a value, history variables, context variables (tupid and relid),
;;; a readfn and a writefn.
;;; Currently the data members readfn and writefn (which are functions)
;;; are NOT assumed to be active objects, although theoretically,
;;; nothing prevents us from making them active.

(defflavor activeobj (value locals tupid relid readfn writefn)
   (obj)
  :initable-instance-variables
  :readable-instance-variables
  (:writable-instance-variables value locals tupid relid)
  (:locatable-instance-variables value))

(defvar *null-activeobj*
(make-instance 'activeobj :value nil
       :locals *null-obj*
       :tupid *null-obj*
       :relid *null-obj*
       :readfn nil
       :writefn nil
       :hist-vars nil))

;;;function for creating an active object

(defun createactive (val readfn writefn
     &key (locals *null-obj*) (tupid *null-obj*) (relid *null-obj*))
  (if (and
(or
  (null readfn) (p_typecheck *null-activeobj* readfn 'function 'createactive
readfn writefn locals tupid relid))
(or
  (null writefn) (p_typecheck *null-activeobj* writefn 'function 'createactive
readfn writefn locals tupid relid)))
      (make-instance 'activeobj :value val
     :locals locals
     :tupid tupid
     :relid relid
     :readfn readfn
     :writefn writefn
     :hist-vars nil)
      *null-activeobj*))
```

Figure 6: Active object class and its create method

### 4.4 Functionality of the Resulting System

Incorporation of active objects into a passive DBMS dramatically enhances the capabilities of the DBMS. We indicate below how active objects of the prototype support the following.

*Triggers/alerters*: The prototype supports triggers and alerters. For a relational DBMS, triggers/alerters can be associated with: an attribute instance (by associating an active object with an attribute instance), a tuple instance (by associating an active object with a tuple instance), an attribute name (by associating an active object with an attribute name to be associated with every instance of that attribute), and a relation instance (by associating an active object with a relation name to be associated with every tuple of that relation).

In order to support the last two, an *activelist-object* is maintained for each attribute of a relation which stores the active objects to be associated when a tuple is inserted into the relation. Similarly, active objects to be associated with the tuples of a relation are stored as part of the relation. This information is used when a tuple is inserted (deleted) into (from) a relation.

*Multiple and nested rules*: Multiple rules and nesting of rules (one active object invoking methods of other active objects including recursive invocation) are readily supported in our prototype. The mechanism used for nested rules is the same as the one used for supporting a rule in the first place and hence no special treatment is needed for handling nested rules.

*Forward/Backward Chaining*: The prototype supports both backward and forward chaining. The function associated with a read event can be used to support backward chaining (read on a virtual field or a virtual tuple) using the history variable to indicate whether the local value needs to be computed or not. The function associated with the write event simulates forward chaining and alerters. Embedded reads in rules cause deeper backward chaining. It is assumed that the functions defined as part of active objects are well-behaved (i.e., there are no self references generating infinite loops).

*Optimization*: The prototype can also be used to implement materialization, lazy, eager, and one-shot rule evaluation techniques. History variable mentioned earlier plays a key role in this regard. The computations encoded in *readfn* and *writefn* can be distributed differently and intermediate results can be stored in history variables. Currently, the functions are not analyzed/processed by the system and hence lazy and eager evaluations have to be encoded by the implementor of these functions judiciously. However, with a higher-level user interface these functions can be generated to exploit lazy, eager, and one-shot evaluation.

*Active Relational DBMS*: Data manipulation operators are built in terms of the basic events supported by the system. That is, relational level operations cause the corresponding basic events which in turn execute the functions associated with the events. For example, a modify operation on an active field value (or field values) of a relation will activate the write functions associated with the corresponding fields. Similarly, a retrieve on an active field (or field values) will activate the read function associated with the corresponding field (or field values), and a delete operation will activate the write functions associated with the corresponding field (or field values).

# 5 Evaluation

In this section we compare monitoring of conditions using active objects and polling. We first describe the scenario used. We then analytically compare polling and active objects and predict their behavior. Finally, we conduct several simulations and analyze the observed behavior resulting in identifying potential optimization opportunities.

## 5.1 Scenario and Measurements

The scenario used for the purposes of comparison is a simulation of a simple command and control application. In this application, the database is populated with various kinds of platforms (such as airborne, naval, and submarine) which are converging towards a stationary platform. The stationary platform is monitoring the movement of all the platforms around it to determine the threat posed by other platforms based on the distance of a platform from itself. The application simulates updates to the position of the platforms in a way that moves the platforms towards the stationary platform at specified rates. When, during the simulation, the distance becomes less than a certain threshold value (which is distinct for each class of platforms and there are three thresholds for each class) the commander is alerted by displaying the appropriate alert code on the console.

The database consists of three relations - *friendly-platforms* - for the stationary platform (having its name, X, and Y coordinates), *hostile-platforms* - for the rest of the platforms (having their name, class, X and Y coordinates) and *thresholds* - for the thresholds (having the platform class and the threshold values - three for each class). Each tuple of *hostile-platform* relation is made active. Functions corresponding to write events obtain the appropriate threshold value based on the class of the platform and compare the distance computed with the threshold values to determine whether there is a change in the alert code. Graphic updates are performed by triggers for active values using the same functions used for updating graphics for polling.

For comparing the performance of polling and active objects, the following were measured at run time using the system clock on Symbolics.

*computations* – the number of times a rule is executed,

*comp-time* – the sum of times spent in executing rules, for a given number of updates,

*polling-time* – the amount of time spent in polling (only for polling), and

*total-time* – total time taken to run a fixed number of simulated updates.

The parameters input for each experiment are: strategy (polling or active), number of hostile platforms, number of updates, and polling interval. The *hostile-platforms* relation is populated with X and Y coordinate values generated randomly making sure that none of the platforms is within the threshold to start with. After setting up the initial graphics, specified number of updates are performed in a loop measuring the values indicated above. At the end of the experiment the values are displayed. A running average of the above values is also computed and displayed.

All measurements used for plotting graphs in this paper are the average of at least two simulations. We also eliminated the effect of paging objects from secondary storage by discarding initial simulations until the measurements stabilized. Hence most of the readings should reflect simulations

with the database resident in the main memory.[6]

## 5.2 Expected Behavior

Of the values measured, *computations* is a measure which is independent of the way in which tuples are accessed in a relation. For active objects the number of computations is the same as the number of updates (assuming that all tuples in the relation are active) and is independent of the size of the relation. For polling, it is directly proportional to size of the relation and inversely proportional to the time interval of polling. The crossover point of these curves (when number of tuples and the computations are plotted on the X- and Y- axis respectively) is a function of the size of the relation and the polling frequency.

Analytically, suppose N is the number of tuples in a relation, u is the update interval (of tuples) for the relation (average), and p is the polling interval. The polling interval is defined as the time between the beginning of one polling cycle to the beginning of the next polling cycle (a polling cycle corresponds to scanning the relation and evaluating the conditions associated with each tuple). The number of computations over a time period T for active objects is $\frac{T}{u}$. The number of computations for polling is $\frac{T}{p}$ *N. They are equal when p = N*u. For a fixed u, p increases directly with respect to N and hence the timeliness of detection (which is inversely proportional to p) is directly dependent on the size of the relation in the case of polling. If the same timeliness of detection is to be maintained as N increases, the amount of computation increases in every polling cycle which may give rise to thrashing (i.e., losing updates). The system will thrash if p is less than the time it takes to perform N computations, i.e., if $p \leq N*a$ , where a is the time it takes to perform an action (ignoring the access time of the tuple for the time being). The system can also thrash when conditions are monitored actively, if $u \leq a$. But the size of the relation does not have any impact in this case. Ideally, p should be large enough not only to perform all the computations in one polling cycle, but should also allow updates to occur after the polling cycle. If in each polling interval p, x units of time is left after performing N1 operations (N1 is the size of the relation) and if the size of the relation were to increase to N2, the new and old relation sizes are related (assuming p is same) by the equation N2 = N1 + O( $\frac{x}{a}$ ). Note that the two sizes are related additively (and not multiplicatively) and the computation performed for each tuple (a) is a significant factor. Preferably, the ratio x/a should be chosen to accommodate the expected size of the relation. As **a** increases, **x** needs to be increased, losing timeliness of detection as a result.

A similar analysis indicates that, for active objects, the *total-time* (or even the *comp-time*) should not be influenced by the size of a relation. However, our initial measurement of *total-time* (shown in Figure 7) clearly indicated the dependence of the operation on the size of the relation. It was not difficult to trace this anomaly to the implementation of our update operation which retrieved tuples sequentially instead of using a direct access.

The above observation implies the significance of physical database design in the presence of active values. The physical database design is influenced by the rules and their association with data

[6]Readings of time reported in this paper need to be qualified further. First, time measurements reported here are not comparable to DBMS benchmarks (in terms of number of transactions per second) as the implementation is a prototype on Symbolics. Furthermore, the conclusions drawn in this paper do *not* depend upon on absolute values of execution time, but instead depend upon relative values (for polling and active objects) and hence are valid in our opinion. The prototype primarily used for demonstrating the active capability *visually* uses graphics and manipulation of graphic icons (using expensive operations such as drawing and erasing line segments) extensively. In addition, time is measured with a granularity of 1/60th of a second and not in microseconds.

Figure 7: Total Time Vs. Number of Tuples

Figure 8: Total time vs. number of tuples (revised)

objects. A relation which may not need indexing when the database is passive may have to be indexed (on one or more attributes) when the same relation (or parts of it) is made active.

In response to the above observed behavior, we implemented a set of operators that were based on object identifiers (or cursors). Cursors can be used to access tuples directly from relations without having to scan the entire relation. Results obtained using the revised implementation are discussed below.

### 5.3 Comparison with Polling

*Simulation 1:* 200 database updates were executed. For polling, intervals of 2, 5, and 10 seconds were considered. In this experiment, we measured the *total-time* to run the simulation as the number of platforms in the database increases. In agreement with our analysis, the *total-time* for active values is flat (as shown in Figure 8) and does not seem to be influenced by the size of the relation. As expected, the *total-time* increases with the decrease of polling intervals as can be inferred from the graph. The curves for polling seem to diverge and the *total-time* spent on polling and active monitoring are more pronounced as the number of tuples increase and the polling intervals decrease. Note that the *total-time* used in this simulation *includes* the overhead incurred for the execution of active objects (whoppers in our case). The effect of the overhead does not appear to be significant for the number of updates and the size of the relation used in this simulation.

The measurements obtained from this simulation also indicate that active condition monitoring

Figure 9: Percent of total time vs. polling interval (revised)

is better than polling beyond the crossover point. In general, the crossover point shifts towards the Y-axis (non-linearly) as the polling interval decreases. This strengthened our initial conjecture that although active condition monitoring is often better than polling, the decision for choosing a strategy (polling or active objects) has to be based upon the various parameters of the application at hand.

*Simulation 2:* 200 database updates on 30 platforms were conducted. In the polling case, the polling interval is varied from 1 to 10 seconds. In this simulation, we measured the fraction of the *total-time* used for condition monitoring as the polling interval varied. Figure 9 shows the polling interval (in seconds) on the X-axis and the fraction (as a percentage) of the *total-time* spent on condition monitoring on the Y-axis.

In this experiment, the overhead for active condition monitoring *does not* come into the picture enabling us to compare the waste of resources in the case of polling as compared to active condition monitoring. It can be easily observed from the graph that as the polling interval decreases, the fraction of time spent on condition monitoring increases almost exponentially.

The results of the above simulation clearly demonstrate two limiting aspects of polling, namely, waste of resources and the timeliness of detection. These are conflicting requirements and hence have to be balanced against application needs. In real-time (time critical) applications, the lack of timeliness may not be acceptable. On the other hand, the increased overhead of polling at very short time intervals may not be tolerable.

Polling, by its nature, involves excessive computation to support timely notification. However, even

Figure 10: Percent of total time vs. number of tuples (revised)

with a very small polling interval, the notification of a condition being met in the database is not ideal. On the other hand, the timeliness of notification approaches the ideal for active objects. There is no need to wait for the next polling cycle, however soon that might be.

*Simulation 3:* In this simulation, we measured the fraction of time spent on condition monitoring as the number of platforms in the database increases.

For active objects, the fraction of time spent on condition monitoring does not change significantly as the number of platforms in the database increases. For polling, on the other hand, the fraction of time spent on condition monitoring does increase as the number of platforms increases. This result reflects the fact that more work is done in polling a large number of objects, all other factors being equal.

This simulation strongly indicates a subtle aspect of polling. Note that the crossover points in Figure 10 occur much earlier than those in Figure 8 While the same fraction of the total time is spent on condition monitoring in active objects (as indicated by a flat line), for polling, the fraction increases with the increase in the number of tuples indicating that the condition is evaluated for all the tuples *whether they were changed or not.* This strongly suggests that there is potential for optimizing polling to reduce the overhead.

### 5.4 Analysis of Simulation Results

Below, we analyze the results of the above simulations and identify opportunities for improving the performance of both active objects and polling. The following analysis assumes that polling and active condition monitoring are being carried out on a relation where all the tuples are assumed to be active.

For a relation of size N, the amount of time spent during the time interval T using the polling technique is $\frac{T}{p}$ *(access time for $(N_{un} + N_{ch})$ tuples + $(N_u + N_{ch})$*a), where $\frac{T}{p}$ represents the number of polling cycles and the expression in the outermost parenthesis indicates the computing time during each polling cycle. $N_{un}$ and $N_{ch}$ represent the number of tuples that were not changed and the number of tuples that were actually changed, respectively, after the previous polling cycle. Clearly, N = $N_{un}$ + $N_{ch}$. For active condition monitoring the above can be expressed as (access time for $\frac{T}{u}$ tuples + $\frac{T}{u}$*a). Assuming that $T_u$*a and $T_p$ *($N_c$*a) are the same (i.e., the same amount of work is done for tuples that were really changed in both cases, though it is potentially less for polling as several changes may get grouped for each polling cycle), any optimization that can be incorporated for evaluating the conditions can be done for both techniques.

Simulations with revised implementation eliminated the dependency of the access time for $\frac{T}{u}$ tuples. However, for polling there are still additional overheads in the form of access time for $(N_{un} + N_{ch})$ tuples and $N_{un}$*a and this accounts for the observed results in Figures 8 and 10. When access time for N tuples was measured as *polling-time* (though not shown in any of the Figures), the access time increased with the number of tuples as a multiplicative factor of $\frac{T}{p}$.

The above analysis indicates that the polling technique used for condition monitoring can be improved in at least two ways: by reducing the access time in each polling cycle and by reducing or eliminating the condition evaluation on tuples that have not changed in a way to affect the outcome of the condition. These two improvements are further elaborated in the next section.

## 6 Summary and Future Directions

In this paper, we have proposed two distinct active database architectures based on available condition monitoring techniques. We have argued for using the layered architecture as an intermediary step that is beneficial in the short term, facilitates heterogeneous active DBMSs, and provides a better migration path for the long term. In order to facilitate the design of the layered architecture, we have compared polling and asynchronous monitoring techniques leading to the following insights.

The results presented in this paper confirm our initial hypothesis that active monitoring is, in general, better than polling when the relation being monitored is large or when timely response is important. However, our simulations strongly indicate that a naive implementation of active objects (or making a passive DBMS active without taking the physical redesign into consideration or adding active capability without analyzing the implementation of operations) will not provide the performance advantage that one would expect of active condition monitoring. In addition, there is clear indication that polling is better than active condition monitoring below the crossover point substantiating retention of polling as one of the strategies. Pragmatically, an active DBMS should intelligently choose from among a set of strategies to reduce the waste of resources and increase the timeliness of condition detection. A DBMS should also be capable of dynamically restructuring

the physical access to support active objects.

Typically, polling is implemented by scanning the entire relation during each polling cycle. The analysis of simulated results suggested opportunities for reducing the amount of work done during a polling cycle. First, the access time can be reduced by employing direct access techniques such as indexing for those tuples that have been modified from the previous polling cycle. Second, the time spent on condition evaluation can be reduced by storing some information as part of the tuples which can be used for deciding whether the condition should be evaluated at all. Though this optimization can be easily made part of condition evaluation, it will add an additional burden, of incorporating this into the code, on the implementor.

We are currently designing ECA rule support for object-oriented database management systems. We are developing a layered architecture for providing limited active capability. Further, we plan on comparing all the three approaches for condition monitoring (user-encoded, polling - with and without optimizations, and active objects) to ascertain conditions under which each of the approaches offer maximum performance advantages. We also propose to evaluate the above approaches for large databases involving secondary storage access (which is not addressed in the present discussion).

We have already identified a set of techniques for optimizing the evaluating situation-action rules for the Sentinel project [CHS93] and are investigating them in more detail. Our long term goal is to develop and consolidate a variety of techniques that are useful for condition monitoring and to develop a set of comprehensive criteria using which appropriate techniques can be chosen.

# 7 Acknowledgments

I want to thank Ms. Susan Nesson for her contribution (both design and implementation) to the comparison of polling and active condition monitoring technique. I want to thank Michael Brodie and Umeshwar Dayal for many fruitful discussions during the course of this work at CCA/XAIT. The implementation reported in this paper was carried out on a Symbolics machine given to CCA on loan by Symbolics Inc. I gratefully acknowledge the support Symbolics Inc. provided during the course of the work reported in this paper. I want to thank Ms. Eman Anwar for her useful comments and suggestions.

# References

[Anw92] E. Anwar. Supporting complex events and rules in an oodbms: A seamless approach. Master's thesis, Database Systems R&D Center, CIS Department, University of Florida, E470-CSE, Gainesville, FL 32611, November 1992.

[Bac88] Maurice J. Bach. *The Design of the Unix Operating System.* Prentice-Hall International, Inc., Englewood Cliffs, N.J., 1988.

[BS83] D. G. Bobrow and M. Stefik. *The Loops Manual.* Intelligent Systems Laboratory, Xerox Corporation, 1983.

[C+89] S. Chakravarthy et al. HiPAC: A Research Project in Active, Time-Constrained Database Management, Final Report. Technical Report XAIT-89-02, Xerox Advanced Information Technology, Cambridge, MA, Aug. 1989.

[C+93] S. Ceri et al. Constraint enforcement through production rules: Putting active databases to work. *To appear in IEEE Quarterly Bulletin on Data Engineering*, January 1993.

[Cha89] U. S. Chakravarthy. Rule management and Evaluation: An Active DBMS Perspective. *Special issue of ACM Sigmod Record on rule processing in databases*, 18(3):20–28, Sep. 1989.

[CHS93] S. Chakravarthy, E. Hanson, and S.Y.W. Su. Active Database Research at the University of Florida. *To appear in IEEE Quarterly Bulletin on Data Engineering*, January 1993.

[CM91] S. Chakravarthy and D. Mishra. An event specification language (snoop) for active databases and its detection. Technical Report UF-CIS TR-91-23, Database Systems R&D Center, CIS Department, University of Florida, E470-CSE, Gainesville, FL 32611, Sep. 1991.

[CN90] U. S. Chakravarthy and S. Nesson. Making an Object-Oriented DBMS Active: Design, Implementation and Evaluation of a Prototype. In *Proc. of Int'l Conf. on Extended Database Technology (EDBT), Kobe, Japan*, pages 482–490, Apr. 1990.

[CRM80] Eugene Charniak, Christopher K. Riesbeck, and Drew V. McDermott. *Artificial Intelligence Programming.* Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey, 1980.

[D+85] U. Dayal et al. PROBE - A Research Project in Knowledge-Oriented Database Systems: Preliminary Analysis. Technical Report CCA-85-03, Computer Corporation of America, Jul. 1985.

[D+87] U. Dayal et al. PROBE - A Research Project in Knowledge-Oriented Database Systems: Final Report. Technical report, Computer Corporation of America, November 1987.

[DB87] M. Darnovsky and J. Bowman. *TRANSACT-SQL USER'S GUIDE.* Document 3231-2.1, Sybase Inc., 1987.

[DBAB+88] U. Dayal, B. Blaustein, S. Chakravarthy A. Buchmann, et al. The HiPAC project: Combining active databases and timing constraints. *Special Issue of Real Time Data Base Systems, SIGMOD Record*, 17(1):51–70, Mar. 1988.

[DKM86] K. R. Dittrich, A. M. Kotz, and J. A. Mulle. An Event/Trigger Mechanism to Enforce Complex Consistency Constraints in Design Databases. *SIGMOD Record*, 15(3):22–36, Sep. 1986.

[DMB+87] U. Dayal, F. Manola, A. Buchmann, U. Chakravarthy, D. Goldhirsch, S. Heiler, J. Orenstein, and A. Rosenthal. Simplifying Complex Objects: The PROBE Approach to Modelling and Querying Them. In *Proceedings German Database Conference (BTW)*, Springer-Verlag, 1987.

[DPG91] O. Diaz, N. Paton, and P. Gray. Rule Management in Object-Oriented Databases: A Unified Approach. In *Proceedings 17th International Conference on Very Large Data Bases*, Barcelona (Catalonia, Spain), Sept. 1991.

[Geh84] Narain Gehani. *Ada, An Adavanced Introduction including Reference Manual For The Ada Programming Language.* Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1984.

[GJ91] N. H. Gehani and H. V. Jagadish. Ode as an Active Database: Constraints and Triggers. In *Proceedings 17th International Conference on Very Large Data Bases*, pages 327–336, Barcelona (Catalonia, Spain), Sep. 1991.

[GJS92] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event Specification in an Object-Oriented Database. In *Proceedings International Conference on Management of Data*, pages 81–90, San Diego, CA, June 1992.

[GrD93] S. Gatziu and K. r. Dittrich. SAMOS: an Active, Object-Oriented Database System. *To appear in IEEE Quarterly Bulletin on Data Engineering*, January 1993.

[Han89] Eric N. Hanson. An Initial Report on the Design of Ariel: a DBMS with an integrated production rule system. *ACM SIGMOD RECORD*, 18(3):12–19, Sep. 1989.

[HJ91] R. Hull and D. Jacobs. Language constructs for programming active databases. In *Proceedings 17th International Conference on Very Large Data Bases*, pages 455–468, Barcelona (Catalonia, Spain), Sept. 1991.

[HLM88] M. Hsu, R. Ladin, and D. McCarthy. An Execution Model for Active Data Base Management Systems. In *Proceedings 3rd International Conference on Data and Knowledge Bases*, Jun. 1988.

[Hug79] Joan K. Hughes. *PL/I Structured Programming.* John Wiley & Sons, 1979.

[Int85] IntelliCorp., Mountain View. *KEE Software Development System User's Manual*, 1985.

[Int90a] Interbase Software Corporation, 209 Burlington Road, Bedford, MA 01730. *Data Definition Guide*, February 1990.

[Int90b] Interbase Software Corporation, 209 Burlington Road, Bedford, MA 01730. *DDL Reference*, February 1990.

[KDM88] A. M. Kotz, K. R. Dittrich, and J. A. Mulle. Supporting Semantic Rules by a Generalized Event/Trigger Mechanism. In *Proceedings International Conference on Extended Data Base Technology*, Venice, Mar. 1988.

[LMKQ89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD Unix Operating System.* Addison-Wesley Publishing Company, 1989.

[MD86] F. Manola and U. Dayal. PDM: An Object-Oriented Data Model. In *Proceedings 1st International Workshop on Object-Oriented Database Systems*, Sept. 1986.

[NTC93] S. B. Navathe, A. K. Tanaka, and S. Chakravarthy. Active Database Modeling and Design Tools: Issues, Approach, and Architecture. *To appear in IEEE Quarterly Bulletin on Data Engineering*, January 1993.

[Oll78] William T. Olle. *The Codasyl Approach to Data Base Management.* John Wiley & Sons, 1978.

[RCBB89] A. Rosenthal, U. S. Chakravarthy, B. Blaustein, and J. Blakeley. Situation Monitoring in Active Databases. In *Proc. of the 15th Int'l Conf. on Very Large Databases*, pages 455–464, Amsterdam, Aug. 1989.

[RHDM86] A. Rosenthal, S. Heiler, U. Dayal, and F. A. Manola. "traversal recursion: A practical approach to supporting recursive applications". In *Proceedings International Conference on Management of Data*, 1986.

[S+86] M. Stonebraker et al. A Rule Manager For Relational Database Systems. The POSTGRES Papers, Memo No. UCB/ERL M86/85, Electronics Research Lab, Univ. of California, Berkeley, CA, Nov. 1986.

[S+91] U. Schreier et al. Alert: An architecture for transforming a passive dbms into an active dbms. In *Proceedings 17th International Conference on Very Large Data Bases*, pages 469–478, Barcelona (Catalonia, Spain), Sept. 1991.

[SHP87] M. Stonebraker, M. Hanson, and S. Potamianos. A Rule manager for Relational database Systems. Technical report, Dept. of Electrical Engineering and Computer Science, Univ. of California, Berkeley, 1987.

[SLE91] A. Sheth, Y. Leu, and A. Elmagarmid. Maintaining consistency of interdependent data in multidatabases. Technical Report CSD-TR-91-016, Purdue University, Computer Sciences Department, West Lafayette, IN 47907, March 1991.

[SR86] M. Stonebraker and L. Rowe. The Design of POSTGRES. In *Proceedings of ACM-SIGMOD*, pages 340–355, 1986.

[SRK92] A. P. Sheth, M. R. Rusinkiewicz, and G. Karabatis. *Database Transaction Models for Advanced Applications*, chapter Using Polytransactions to Manage Interdependent Data, pages 555–581. Morgan Kaufmann Publishers, San mateo, CA, 1992.

[Tan92] A. Tanaka. *On Conceptual Design of Active Databases*. PhD thesis, Georgia Institute of Technology, College of Computing, December 1992.

[WB79] Patrick Henry Winston and Richard Henry Brown. *Artificial Intelligence: An MIT Perspective*. The MIT Press, Cambridge, Massachusetts, 1979.

[WCB91] J. Widom, R. J. Cochrane, and . Lindsay B, G. Implemented Set-Oriented Production Rules as an Extension of Starburst. In *Proceedings 17th International Conference on Very Large Data Bases*, pages 275–286, Barcelona (Catalonia, Spain), Sep. 1991.

[WF90] J. Widom and S. Finkelstein. Set-Oriented Production Rules in Relational Database Systems. In *Proc. of ACM-SIGMOD*, pages 259–270, May 1990.