D W Q

Foundations of **D**ata **W**arehouse **Q**uality

National Technical University of Athens (NTUA)
Informatik V & Lehr- und Forschungsgebiet Theoretische Informatik (RWTH)
Institute National de Recherche en Informatique et en Automatique (INRIA)
Deutsche Forschungszentrum für künstliche Intelligenz (DFKI)
University of Rome «La Sapienza» (Uniroma)
Istituto per la Ricerca Scientifica e Tecnologica (IRST)

M. Bouzeghoub, F. Fabret, F. Llirbat,

M. Matulovic and E. Simon

ACTIVE-DESIGN: A Generic Toolkit for Deriving Specific Rule Execution Models

Proc. of the Third International Workshop, Rules in Database Systems 97,

Springer Verlag,Lecture Notes in Computer Science 1312,

Skovde, Sweden, June 1997.

DWQ : ESPRIT Long Term Research Project, No 22469
Contact Person : Prof. Yannis Vassiliou, National Technical University of Athens,
15773 Zographou, GREECE Tel +30-1-772-2526 FAX: +30-1-772-2527, e-mail: yv@cs.ntua.gr

ACTIVE-DESIGN: A Generic Toolkit for deriving specific rule execution models*

Mokrane Bouzeghoub[1] and Françoise Fabret[2] and François Llirbat[2] and Maja Matulovic[2] and Eric Simon[2]

[1] University of Versailles, 78035 France,
Mokrane.Bouzeghoub@prism.uvsq.fr
[2] INRIA, Rocquencourt,78153 France,
FirstName.LastName@inria.fr

Abstract. Active rules or triggers are widely accepted as powerful mechanisms to implement applications or systems behaviour. Several rule execution models were proposed as extended functionalities for different database systems. However, these models lack in flexibility and adaptability to specific database systems or specific application requirements. In this paper, we propose a generic framework which provides a set of basic functions which can be used to implement any execution model. This framework, called Active-Design toolkit, can be exploited in many situations where applications need specific model or different models for different subsets of rules.

1 Introduction

Active systems extend traditional database systems by enabling the automatic execution of rules when certain events occur. This mechanism is widely accepted as a uniform and powerful way to implement general system extensions (e.g., integrity checking, maintenance of materialized views, management of replicated data, etc.), and business rules in a wide variety of applications. An active database system is generally considered as a passive database system with three extra components (see e.g., the Manifesto in [DGG95]):

- A rule specification language - Rules are specified as an ECA triple: an event expression (E) that triggers the rule, a condition (C) to be evaluated if the rule is triggered, and an action (A) to execute if the condition evaluates to true.
- An event detector - It detects the occurrences of events and signal them, together with possible parameters, to a rule execution engine.
- A rule execution engine - It invokes and synchronizes the execution of rule conditions and actions with respect to the events that have been signaled.

* This work is partially supported by the European project EP22469 "Foundations on Data Warehouse Quality (DWQ)"

The execution model underneath an active database system depends on many parameters (also called dimensions), which describe the operational semantics of rule execution (see e.g., [WC96, FT95]). These parameters specify for instance how the events that have occured are used to determine if a rule is triggered, at which point in time a triggered rule should be executed, or how a rule should be selected among several other conflicting rules.

Usually, an active application makes particular assumptions on the dimensions of the execution model which seem the most appropriate for the active rules it intends to use. The problem that sometimes arises in practice is that these assumptions are not consistent with the execution model supported by the active database system on which the application has to be developed [KDS97]. To illustrate this, consider a relational active database system that provides SQL triggers[3]. Suppose that your application needs to use active rules that (i) are triggered at the end of transactions, (ii) consider globally all the events that have occurred since the beginning of the transaction, and (iii) schedule the rules according to some priority-based policy. Unfortunately, these dimensions are not compatible with the restricted SQL triggers currently implemented by existing products. The application developer will be forced to implement an active rule monitoring layer, let us call it Active Monitor, on top of the database system. The SQL triggers provided by the active database system will merely be used to interrupt the execution of user transactions after each SQL statement, and pass the control to the Active Monitor. In particular, at each interruption, the Active Monitor will have to record the event that has just occurred into specific database relations.

Several examples of similar situations are described in [KDS97], concerning experiences of development of active database applications that implement ad-hoc system extensions or business rules. Usually, the major drawback for application developers is that developing and maintaining an Active Monitor is a quite difficult task that requires a lot of skill and training.

1.1 Related work

The usual answer given to this problem in the research literature is to design an active database system that offers a powerful parametrized rule execution model, which can presumably accommodate the requirements of a large class of application's semantics. This approach, is emphasized by the Chimera environment [WC96], Acto [MFLS96], and Naos [Cou96]. In these systems, the user specifying the rules can choose the appropriate values of the execution model's parameters that correspond to the desired excution semantics for the rules. In NAOS, the parametrized execution model is implemented within the O2 database system. Thus, every application pays the price for active functionalities that are implemented by the database system in order to support the parametrized execution

[3] By SQL triggers, we refer to the kind of SQL3-like triggers, which are supported by most today relational products.

model, but that will possibly be bypassed, and hence not used by the application. On the positive side, the implementation of the execution model can a priori be better optimized because it makes use of the database system internals. On the contrary, Chimera and ACTO follow an approach whereby the target database system is a "light-weight" active database system, which, for instance, only supports SQL triggers. The Chimera and ACTO systems behave more like "active application generators" for the target database system. More concretely, the user defines rules through a specific language or interface and then the system (that is, Chimera or ACTO) generate the appropriate code, including the definition of triggers in the target database system, that is needed to monitor the execution of the user-defined rules.

However, the three above approaches still suffer from the following drawbacks. First, it is not possible for the application developer to accomodate other parameters values than the ones hard-wired in those systems. For instance, if one desires that the action of a rule be executed as an independant transaction, then if the system does not incorporate this possibility, the application developer will not be able to add it. Second, the user cannot customize, for optimization purposes, the implementation of the rules she defined. For instance, all these systems make use of so-called "delta structures" that are used to manage the history of events that occured within a transaction. However, the application developer is not able to modify the format and the implementation of these delta structures. Last, for the above "application generator" approaches, the code that is generated always incoporate a fixed engine that is not minimal with respect to the application needs.

1.2 A Toolkit Approach

In this paper, we propose an alternative approach, whereby the implementation of an active application follows three distinct phases.

The first phase consists of using an extensible Toolbox to build an Active Monitor, on top of a DBMS. Like with Chimera and ACTO, this DBMS is supposed to be a "light-weight" active system. The minimal requirements to this system are to support the detection of primitive database events (e.g., as modern relational systems do), and enable a trigger's action to execute an arbitrary user-defined program in a given language. The Toolbox is specific to the target DBMS because: (i) it makes use of the capabilities of the target DBMS (detection of events, management of temporary relations, creation of stored procedures, etc), and (ii) it must be implemented in a language that can be invoked from within the action of the trigger supported by the DBMS. The Toolbox consists of a set of reusable building blocks that can be combined to implement a specific rule execution model. The design of the Toolbox is the major contribution of this paper.

During the second phase, the user defines the active rules. Each rule definition entails the specification of an event part that can be recognized by the

Active Monitor, a condition consisting of a query executable by the DBMS, and an action that contains statements executable by the DBMS. The condition and action of a rule may also include and possible specific events that can be recognized by the Active Monitor.

Finally, during the third phase, the user-defined rules lead to the generation of several components. DBMS triggers are generated to detect the events that contribute to the triggering of the user-defined rules, and invoke the Active Monitor. Specific procedures that implement the conditions and actions of rules are generated. They will be directly invoked by the Active Monitor. Finally, the necessary information about the user-defined rules is loaded into the internal data structures of the Active Monitor.

In this paper, we focus on the design of the Toolkit used to produce the code of an Active Monitor. Apart from this introduction, this paper is structured as follows. In Section 2, we give a formal model of a generic Active Monitor, which can describe the execution of rules under any execution model. In Section 3, we derive a generic functional architecture for an Active Monitor, and show how to design the building blocks that are necessary to generate such a functional architecture. We also detail the interfaces of the building blocks. Section 4 concludes.

2 Modeling a Generic Active Monitor

2.1 Preliminaries

An event is described by its name and a sequence of parameters. In this paper, we consider primitive and complex events: A primitive event may be a user's notification (explicit event) or a database operation i.e., an insertion, deletion, updating, selection on a relational table, or any method invocation in an object database system. A complex event is a combination of primitive events (using logical, arithmetic and/or temporal operators) An event whose parameters are instantiated is an event instance. An active rule r is defined by a triggering event, r_e, a condition and an action programs. A rule instance of r results from the triggering of r due to an occurrence of an instance of r_e: it is represented by a triple $<r, r\_i, e\_i>$ where e_i denotes the instance of r_e that triggered r and r_i is the instance identifier. Executing a rule instance $<r, r\_i, e\_i>$ consists in computing its condition and performing its action when the condition is evaluated to true. Condition and action programs may use the e_i value as input parameter.

Throughout this paper, we assume that the rules are triggered by events occurring during the execution of a user program which may be an application program or a flat transaction. The execution of this program may be interrupted for executing the rules. We also assume that there is no external event, i.e., the occurrences of triggering events are due to the execution of the operations occurring in the user program or in the corresponding triggered rules.

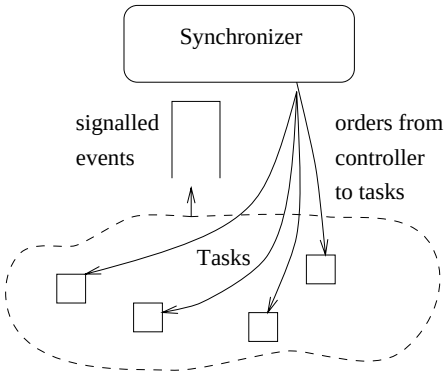

Fig. 1. Execution system

2.2 The model

We represent the execution of a set of rules by means of the notions of tasks, task synchronizer, event history and task history. Communications beetwen the tasks and the synchronizer are represented by messages. There is no inter-task communication. Every time a task sends a message, it is inactive until the synchronizer sends a response. The synchronizer handles the messages in the order in which they were sent by the tasks, in response to the messages, it may create tasks or send commands to the waiting tasks. A task represents a rule instance or an initial program triggering the rules. At every step in the execution process, the current set of tasks consists of the active tasks plus the tasks waiting for a command from the synchronizer. The event history and the task history respectively contain all the messages sent by the tasks and the commands sent by the synchronizer from the beginning of the transaction.

2.3 Tasks : State diagram and messages

We describe a rule instance task by the state transition diagram depicted in Figure 2 with labelled transitions. Grey ovals, dashed ovals, and white ovals respectively represent inactive states (i.e. states where T is waiting for a command sent by the synchronizer), final states of T, and active states (i.e. states where T performs computations and, possibly, send messages to the synchronizer). There are four inactive states : *triggered*, *evaluated*, *interrupted*, and *wait*. A transition from state S to state S', noted (S, S'), with a label of the form "R:m" has the following meaning: "on receive command m from synchronizer" T executes command m and enters in state S' (remark that such situation occurs only if S is an inactive state). On the opposite, a label of the form "S:m" may only occur if S is an active state, the meaning is: T sends message m to the synchronizer and enters in state S' (remark that S' is necessarily an inactive state).

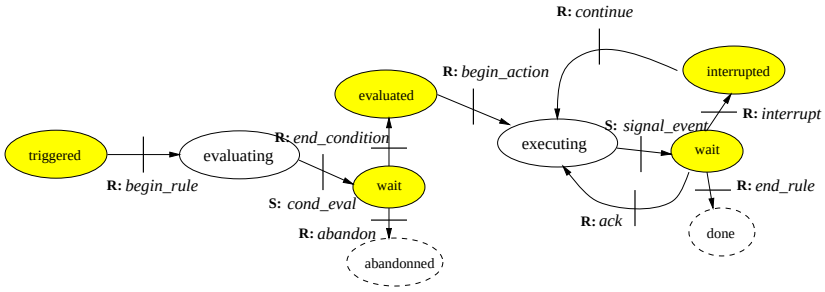

Fig. 2. Rule instance execution state diagram

State *triggered* is the initial state of T where the task is waiting for the command begin_rule. On receive this command, T enters in active state *evaluating* where it computes the condition, sends the message cond_eval including the reporting of the result of the computation, then T enters in the inactive state *wait* where it does nothing. At this state, the command received by T depends on the result of the condition evaluation: if the condition has the value false, T receives the command abandon and enters in state *abandon* where it, possibly, executes protocols ending the task. On the opposite, if the condition holds, T enters into the *evaluated* state, and waits for the begin_action command. On receive this command, T enters in the *executing* state where it executes the action program. During this execution, T may send messages to the synchronizer (for example to signal database operations occurred during the execution). After each message, T enters in the inactive state *wait* where it does nothing. In response to these messages, the synchronizer may send an interrupt command that leads T in the inactive state *interrupted* or an ack command (for acknowledgement) that leads T to continue the program, or an end_rule command. This last command responds to a message signalling the end of the program execution; it leads T in the final state *done*.

The state transition diagram of the task representing the initial program is given in Figure 3. This task may be *executing*, signalling events, or be *interrupted* for executing rules.

2.4 Event and Task Histories

The execution of an initial program that triggers active rules can be traced using two histories: The *Event History (EH)* and the *Task History (TH)*.

The *Event History (EH)* contains the messages sent by the tasks, that is the event instances that occur during the execution of the initial program and of the rules. Each event instance is described by a triplet $<E, args, ts>$ where:

1. E is the name of the associated event,

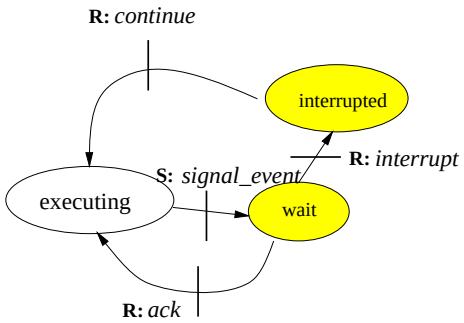

Fig. 3. transaction execution state diagram

2. *args* contains the event parameter values associated with the event,
3. *ts* is a timestamp that indicates the time when the event occured (i.e. the message is sent).

The *Task history (TH)* records the scheduling of rule instances during the program execution. It is a sequence of tuples $<r$, ri, O, $ts>$ that reflects the commands sent to the tasks during execution of the transaction.

1. r is either a rule identifier, or the initial program,
2. ri is an identifier of an instance of r (i.e. a task),
3. O is a command name,
4. ts is a timestamp that indicates the time when the command is taken into account by the task (i.e. the time when the task changes its state).

Moreover, TH contains a tuple of the form $<r$, ri, $init$, $ts>$ per rule instance ri, where ts indicates the time when the task was created.

2.5 Synchronizer: Built-in functions

The synchronizer may be seen as a process which is awaken by the messages sent by the tasks. It is in charge of two works: creating new tasks, and sending commands to the inactive tasks. Creating new tasks raises several problems: when creating new tasks? what rules are triggered at a certain point in the time? what instance(s) have to be created for each triggered rule? Sending commands to the inactive tasks requires to select the task(s) to activate: what are the selection criteria? Answering these questions fully determines the semantics of the rule execution.

When creating new tasks? : function check_synchro
Function check_synchro checks the histories and derives a *synchronization point* that may be a *processing* point, a *scheduling* point or a *null* point. This function

may be specified by using two boolean functions Processing and Scheduling:

$$\begin{array}{l} check\_synchro() = \text{processing point iff Processing(HT, HE)} \\ check\_synchro() = \text{scheduling point iff Scheduling(HT, HE)} \\ check\_synchro() = \text{null point} \quad \text{iff } \neg(\text{Processing(HT, HE)} \\ \qquad\qquad\qquad\qquad\qquad\qquad \vee \text{Scheduling(HT, HE))} \end{array}$$

The synchronizer uses the *synchronization* point returned by check_synchro to take the following decision: it creates new tasks if the point is a *processing* point, it sends an ack to the task having sent the last message if the point is a *null* point, and otherwise it sends a command to an inactive task.

What are the triggered rules, and the new instances?: function trigger Function *trigger* computes a set of rule instances: It checks the histories in order to compute the triggered rules. Then it computes the instance(s) associated with each triggered rule. To do that, function *trigger*, uses three functions: *is_triggered*, *synthesis*, and *compute_interval*.

1. *compute_interval* specifies what subset of elements contained in the histories must be considered for computing the triggered rules, *compute_interval* may be specified by means of a formula over the histories.

$$compute\_interval: \quad () \longrightarrow \quad \text{set of event instances}$$

2. *synthesis* takes a set of event instances and derives a set of event instances, it may be specified by a formula over *compute_interval*.
3. *is_triggered* takes a rule r, tests if r is triggered with respect to the set of event instances returned by functions *compute_interval* and *synthesis*, and returns the set of rule instances for r.

$$\begin{array}{l} < r, r\_i, e\_i > \in is\_triggered \text{ iff } \exists e = < E, args > \quad s.t \\ \quad \text{(r is set\_oriented and } args = \{a \mid < E, a > \in synthesis(compute\_interval)\}) \\ \qquad\qquad\qquad\qquad or \\ \quad \text{(r is instance\_oriented and } < E, a > \in synthesis(compute\_interval)) \end{array}$$

Finally, function *trigger* is specified as :

$$< r, r\_i, e\_i > \in \quad trigger() \quad \Rightarrow < r, r\_i, e\_i > \in \quad is\_triggered$$

Example 1. In Starburst, the rules are set-oriented, *compute_interval* contains all the event occurrences that have arisen since the last time r was executed and *synthesis* specifies the standard *net-effect*. A possible specification of *compute_interval* could be:

$$\begin{array}{l} compute\_interval = \{< e, args, ts > \in EH \mid \forall < r', r'\_id, begin\_rule, ts' > \in TH \\ \quad (r = r' \quad and \quad command = begin\_action) \Rightarrow ts > ts'\} \end{array}$$

What tasks to activate?: function choose Function choose takes a set of inactive tasks and selects a set of tasks to activate. It may be specified by a logical formula over the task history. For example, the SQL triggers are executed in depth first search. Every time an event arises, the rules triggered by this event are executed. The specification of the function *choose* may be :

$$< r, r\_i, e\_i > \in choose() \text{ iff } < r, r\_id, init, ts > \in HT \text{ and } \forall < r', r'\_id, any, ts' > \in TH, \ (ts' < ts) \text{ or } (ts' = ts \text{ 'and } (r = r' \text{ or r has priority over r'}))$$

2.6 Synchronizer algorithm

Every time the synchronizer receives message e from task t, it uses function check_synchro to compute the associated synchronization point. If this point is a *null* point the synchronizer sends an *ack* command to t, else the procedure synchro_point implements the actions of the synchronizer.

```
On receive message e from t:
    let p = check_synchro();
    if p is not null
      then synchro_point(p, t, e);
      else send back ack to t
```

Fig. 4. Synchronization point computation

3 Active-design Toolkit

Our toolkit is based on a generic functional decomposition of an active monitor. Before describing the toolkit architecture we first present our functional view of an active monitor.

3.1 Functional decomposition of an Active Monitor

The functional organization of an Active Monitor is shown in Figure 6. It consists of a DBMS, an event manager, a task executor and an execution controller.

The DBMS: It executes the database operations occurring in the initial program and the rule condition and action programs. The DBMS may also provide a detection output interface which computes the event instances (if any) produced by the operations and signals them to the event manager.

```
synchro_point algorithm
input: a synchronization point p,a task t and a message e
      case e is
        cond_eval: let v denote the result of the condition evaluation reported in e
              if v = true then send end_condition to t;
            else send abandon to t;
        signal_event: let v denote the event signalled in the message;
              if v = end_action then send end_rule to t;
              else send interrupt to t;
      end case;
      if p is a processing point
       then for rt in trigger() do create_task(rt); end do
      end if
      let selectedSet = choose();
      for rs in selectedSet do
        case current state of rs is
         "triggered" : send begin_rule to rs;
         "evaluated": send begin_action to rs;
         "interrupted": send continue to rs;
         end case;
      end do
```

Fig. 5. Synchro-point algorithm

The Task executor (TE): In particular, it enforces execution commands sent by the execution controller. For this purpose, it uses the DBMS to perform database operations such as queries, update operations and transactional commands. It provides to the execution controller an input interface which consists of the transition functions of the task state transition diagram (see Figures 2 and 3).
The Task Executor also provides a detection output interface. Indeed, it may detect some specific events during the execution of rules and programs that cannot be detected by the DBMS. It signals them to the Event Manager. For example it may signal the end of a condition evaluation and/or the end of an action execution.

The Event Manager (EM): The Event Manager is called each time an event is detected. It receives events coming from the DBMS and/or the TE. It follows a generic behavior that is parametrized by the add_event and check_synchro functions: When EM receives an event instance e, it first adds this instance to the history by calling the function add_event(e), then it checks if a synchronization point is reached, using the function check_synchro which may consult the History by using various access functions. If a synchronization point p is reached, then EM transmits p to the Execution Controller.

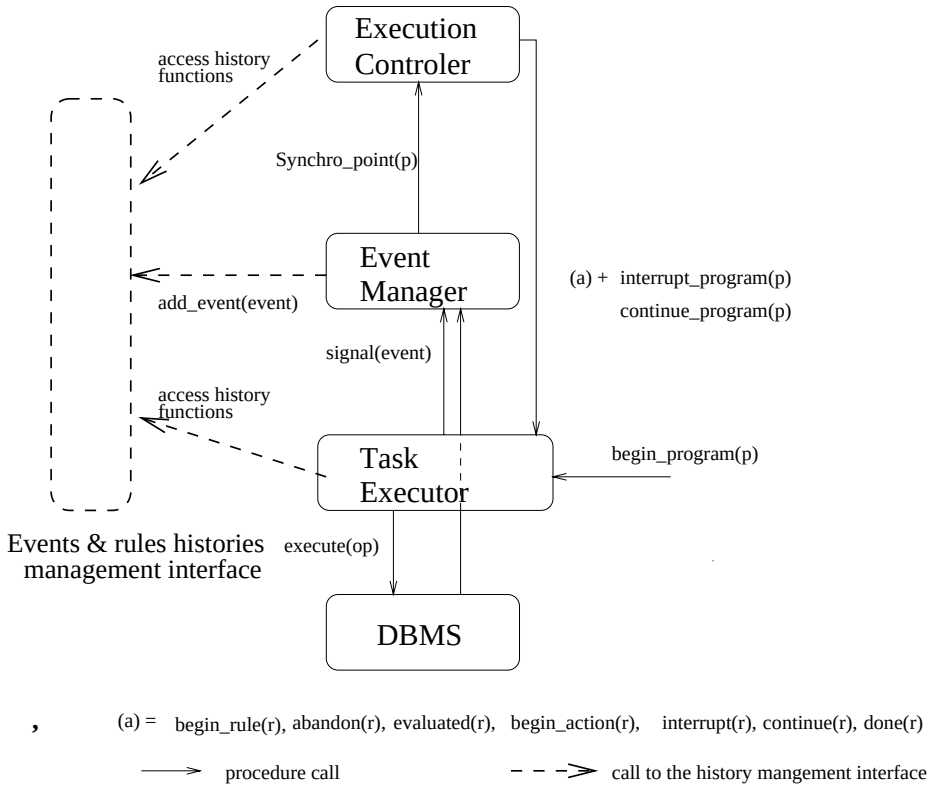

Fig. 6. Functional decomposition of an active application

The Execution Controller (EC): The Execution Controller (EC) is called each time a synchronization point is reached. EC reacts to synchronization points by sending commands to the Task Executor. It follows the generic behavior described in the procedure Synchro_point() (see Figure 5). Its behavior is parametrized by the trigger and choose functions and the history management access interface. It may use several Is_triggered, synthesis and compute-interval functions.

3.2 The toolkit architecture

The Active-design toolkit consists in a set of reusable building blocks, or modules that are combined to implement the RE, EM and EC components of an active monitor. Each module provides a *dynamic* and a *static* interface. The *dynamic* interface consists in procedures that are invoked by the generic code of the active monitor components during the execution of the rules. The *static* interface consists in procedures that are called when new rules are defined. They generate

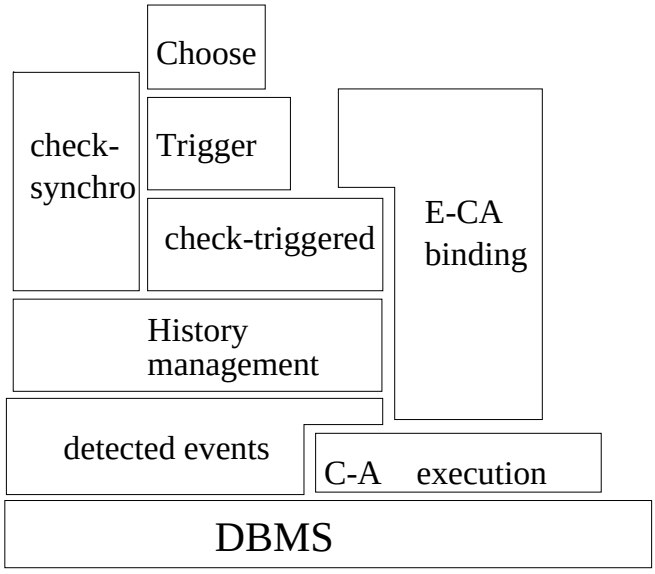

Fig. 7. Toolkit architecture

code and/or static information that will be used during the execution. Figure 7 shows the dependencies between the modules: Each module uses the interfaces provided by the module(s) right bellow. Other components of our toolkit architecture (not shown in Figure 7) are libraries of module implementations. There is one library per module. The designer defines the execution model semantics by selecting one module implementation from each library. For example, the library associated with the *choose* module may provide implementations of various rule execution policies (depth-first, iterative with priority, $\cdots$). The toolkit is extensible by adding new implementations to the libraries. In the following, we briefly describe the interfaces of each module.

C-A execution module This module ensures the execution of the condition and action parts of rules. It implements coupling modes between condition and action. The static interface consists of two functions add_condition(rulename, CDesc) and add_action(rulename, Adesc) that respectively allow to define a rule condition and a rule action. The type of the input parameters Cdesc and Adesc may vary according to the module implementations that are available in the library. The dynamic interface consists in procedures that execute the condition and/or action parts of the rules.

detected events module: This block is responsible for the detection of events. Its implementation depends on the active capabilities of the underlying DBMS and the implementation of the C-A execution module. Indeed, an event can be detected by using triggers or by rewriting the code of the programs and/or

rule action parts. The static interface consists of the new_event(D-E-D, S-E-D) procedure. The D-E-D parameter gives the description of the event that must be detected. The S-E-D parameter indicates the format in which the event must be transmitted to the history management module. There is no dynamic interface.

History management module: This module is responsible for building the event history. Its dynamic interface contains (i) the add_event(event) procedure that is used to store events in the history and, (ii)various history access functions. There is a huge variety of possible implementations. The simplest implementation stores the history in a simple sequential log of events and provides functions that allow to consult this log. More complicated implementations store events in delta relations and compute the net-effect of events incrementally. Such implementations enrich their interface with operations between deltas and functions that allow to select and consult deltas. We could also implement a module that constructs composite events. The static interface of such a module would consist in a composite event specification language.

Check_triggered module: This module has in charge to check if a rule is triggered. In which case, it computes the set of corresponding rule instances. The static interface consists in the following procedure: new_rule(rulename, EG, SYNT, Interval) The parameter EG indicates the execution granularity of the rule (tuple or set oriented), $SYNT$ is a specification of the synthesis function and $Intervall$ specifies the compute_interval function.
The dynamic interface provides the following functions:

- Is_triggered(rulename) returns a set of rule instance identifiers. This set corresponds to the set of triggered instances of rule *rulename*.
- Triggering_event(rulename, rule-instance-id) returns the value of the event associated to the triggered rule instance *rule-instance-id*.

E-C-A binding module: This module is responsible for the execution of rule instances. The static interface consists in the procedure new_rule(rulename, EP, CM). EP indicates if and how the triggering event has to be passed as input parameter to condition and/or action of the rule. CM indicates the transactional coupling mode. Several implementations of transactional coupling modes are feasible depending on the capabilities of the underlying DBMS.
The dynamic interface provides the following procedures:

1. begin_rule(rulename, r) where r is a rule instance identifier. This procedure prepares the condition evaluation context of the rule instance and starts its execution. Such context preparation may consist in creating a new process, initiating some variables and/or sending transactional commands to the DBMS (such as `start transaction` or `start sub-transaction`[4]).

[4] If the targeted system provides the nested transactions model

2. end_condition((rulename, r) ends the condition evaluation context and prepares the action execution context. For example, it may enforce parameter passing between condition and action and/or execute transactional commands.
3. begin_action(rulename, r) starts the execution of the rule action. It may also first execute some transactional commands depending on transactional coupling modes.
4. interrupt(rulename, r) stops the execution of r action.
5. continue (rulename, r) continues the execution of r action.
6. abandon(rulename, r) and end_rule(rulename, r) close the execution context of r. For example, they may send some transactional commands depending on the transactional coupling modes.

Let us note that some interface functions may be not provided by some E-C-A binding modules implementations. For example, the *end_condition* and *begin_action* functions are not provided by implementations that only ensure the immediate C-A coupling mode.

Check-synchro module : This module has in charge to check if there is a synchronization point. The static interface consists in the new_synchropoint(ST,SDesc) procedure where ST indicates if the synchronization point is a rule processing or a rule scheduling point. The parameter $SDesc$ describes the function that has to be checked on the history in order to detect a synchronization point. The dynamic interface consists in the checksynchro() function that returns a synchropoint or the value NULL if no synchropoint is detected. the synchropoint.

Trigger module : The trigger module computes a set of triggered rule instances. Various implementations are possible. The usual implementation computes all the triggered rule instances. Another implementation may select specific rule instances. The static interface consists in the add_rule(rulename, AUX) function where AUX is auxilliary informations that are used to select rule instances (e.g., order priority). The definition of the AUX parameter may vary according to the module implementations. The dynamic interface is the trigger() function that returns a set of rule instances identifiers.

Choose module : The choose module computes a set of rule instances to be executed. Various implementations are possible according to the rule execution policies. The static interface consists in the add_rule(rulename, AUX) function where AUX is auxilliary informations that is used to select rule instances. The definition of the AUX parameter may vary according to the module implementations. For example, if the module implementation enforces an iterative execution with priority, the AUX information provides the priority of the rule. The dynamic interface is the choose() function that returns a set of rule instances identifiers.

4 Conclusion and future work

This paper has addressed the problem of engineering active applications by using a generic toolkit. The idea is to provide a set of basic functions which can be used to implement any of the semantic parameters which characterize a rule execution model. We have defined these functions and demonstrated their capability to implement known execution models. The main features of the toolkit based on these basic building blocks are (i) its *genericity* which allows to implement any rule execution model, (ii) its *flexibility* which allows its adaptation to specific application requirements, and (iii) its *extensibility* which allows addition of hight level components to facilitate application development.

Future work wil concern an effective implementation of the toolkit libraries and the organization of these libraries. We also envision the definition of some hight level components which allow the toolkit to be inserted in the context of middleware tools which facilitate rapid development.

To validate the Active-Design toolkit, we envision to use it for developping update propagation techniques within a datawarehouse architecture.

References

[Cou96] T. Coupaye. *Un Modèle d'exécution paramétrique pour base de données active.* Ph.d. thesis, Université Joseph Fourier - Grenoble 1, 1996.

[DGG95] K. R. Dittrich, S. Gatziu, and A. Geppert. The Active Databse Management System Manifesto : A Rulebase of ADBMS Features. In *Proc. 2-th International Workshop on Rules in Database Systems, RIDS'95*, volume 985, Athens, Greece, 1995. Lecture in Computer Science, Springer-Verlag.

[FT95] P. Fraternali and L. Tanca. A Structured Approach for the Definition of the Semantics of Active Databases. *ACM Transactions On Database Systems*, December 1995.

[KDS97] A. Kotz-Dittrich and E. Simon. Active database systems: Expectations, experiences, and beyond. In N. Paton and O. Diaz, editors, *Active Rules in Databases.* Springer Verlag, 1997.

[MFLS96] M. Matulovic, F. Fabret, F. Llirbat, and E. Simon. Un Système de Règles à la Sémantique Paramétrable. In *BDA'96*, pages 291–311. INRIA, 1996. Proc. 12-emes Journées Bases de Données Avancées, Geneve.

[WC96] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan-Kaufmann, San Francisco, California, 1996.

This article was processed using the LaTeX macro package with LLNCS style