TriGS Debugger
A Tool for Debugging Active Database Behavior

G. Kappel, G. Kramler, W. Retschitzegger

Institute of Applied Computer Science, Department of Information Systems (IFS)
Altenbergerstraße 69, A-4040 Linz, University of Linz, AUSTRIA
Tel: +43-732-2468-883; Fax: +43-732-2468-9511
email: {gerti, gerhard, werner}@ifs.uni-linz.ac.at

Abstract. Active database systems have been developed since several years and represent a powerful means to respond automatically to events that are taking place inside or outside the database. However, one of the main stumbling blocks for their widespread use is the lack of proper tools for the verification of active database behavior. This paper copes with this need by presenting TriGS Debugger, a tool which supports mechanisms for predicting, understanding and manipulating active database behavior. First, TriGS Debugger provides a holistic view of both active and passive behavior by visualizing their interdependencies, thus facilitating pre-execution analysis. Active behavior is represented in terms of Event/Condition/Action rules, passive behavior is represented in terms of classes and their methods. Second, post-execution analysis is supported by tracing and graphically representing active behavior not only in terms of primitive events and serially executed rules but also by considering composite events and rules, which are executed in parallel. Special emphasize is drawn on the complexity of the resulting trace data by allowing to customize the visualization using a filter mechanism as well as by providing the possibility to mine behavior patterns out of the trace data and to visualize them in an aggregated fashion. Third, TriGS Debugger allows to interactively examine and manipulate the active behavior at runtime. In particular, mechanisms are provided to set breakpoints, to replay single events or event sequences, to (de)activate selected events and rules, and to modify rule properties and the rule code itself.

Keywords. Active databases, event/condition/action rules, rule debugging, trace visualization

1 Introduction

Active database systems have been developed since several years. Basic active facilities in terms of Event/Condition/Action rules (ECA rules) have already found their way into commercial database systems [ACT96], [Coll99], [Kapp98b]. Although active facilities are suitable for a wide range of different tasks, they are not straightforward to use when developing active database applications [Pato99]. The main reasons are as follows. First, the very special nature of active behavior, which is controlled dynamically by events rather than statically by a flow of control as it is the case for traditional applications based on passive database behavior. Second, while each single rule is easy to understand, complexity arises from the interdependencies among rules and between active behavior and passive behavior. Finally, the inherent complexity of active behavior is increased by concepts such as composite events, cascaded rule execution, and parallel rule execution. The actual behavior of a set of rules responsible for a certain active behavior is very hard to understand without proper tool support. The special characteristics of active behavior, however, prevent the straightforward employment of traditional debuggers realized for application development based on passive database behavior. Therefore, specific approaches for the verification of active behavior have been investigated.

First of all, there are approaches supporting *static rule analysis* to determine certain qualities of a set of rules, like termination, confluence, and observable determinism [Aike92], [Aike95], [Bail98], [Bara99], [Etzi95], [Frat97], [Lee99], [Mont99], [Vadu97]. A major drawback of these approaches is that expressive rule languages which are not formally defined are hard to analyze, leading to imprecise results. Furthermore, on one hand it is not always obvious what action should be taken when a potential source of nontermination or nonconfluence is detected and on the other hand, the fact that a set of rules exhibits terminating and confluent behavior does not necessarily imply that it is correct. Due to these drawbacks, static rule analysis has no major influence on the development of active applications [Pato99]. Most existing systems take a complementary approach in that they record the active behavior at run-time, and visualize rule behavior afterwards [Behr94], [Bena95], [Bern99], [Chak95], [Coup97], [Diaz93], [Fors95], [Jahn96], [Thom96], [Vadu98]. In order to explain rule behavior, the recorded event detections and rule executions are visualized by means of event/rule graphs [Diaz93], [Fors95],

[Thom96], [Vadu98] event graphs [Chak95], [Coup97], animated replay [Chak95], [Fors95], and even 3D graphics [Coup97]. Besides mere recording and viewing, some systems let the rule developer control the active behavior by means of breakpoints and step-by-step execution, enabling the inspection of database states at any time during rule execution.

However, existing systems often do not cope with the interdependencies between passive behavior and active behavior. They still lack proper debugging support for important aspects of rule behavior, like the composite event detection process [Bern99], and parallel executed rules, which are not considered at all. Finally, the information overload induced by complex trace data is often not handled properly.

TriGS Debugger copes with these drawbacks and the special nature of active database behavior in three different ways. First, pre-execution analysis is allowed on the basis of a holistic view of active behavior and passive behavior. Second, post-execution analysis is supported by a graphical representation of active behavior which includes the detection of composite events and rules which are executed in parallel. In order to deal with the huge amount and complexity of the recorded trace data, TriGS Debugger allows to customize the visualization by using a filter mechanism as well as by providing the possibility to mine behavior patterns out of the trace data and to visualize them in an aggregated fashion. Third, TriGS Debugger allows to interactively examine and manipulate the active behavior at run-time. In particular, mechanisms are provided to set breakpoints, to replay single events or event sequences, to (de)activate selected events and rules, and to modify rule properties and the rule code itself.

The remainder of this paper is organized as follows. The next section provides an overview of the active object-oriented database system TriGS representing the basis for TriGS Debugger. In Section 3, the functionality of TriGS Debugger is presented from a rule developer's point of view. A discussion on the architecture and implementation of TriGS Debugger can be found in the section following. The paper concludes with some lessons learned from user experiences and points to future research.

2 Overview of TriGS

TriGS Debugger is realized on top of *TriGS (Triggersystem for GemStone)* [Kapp94a], [Kapp98b] representing an active extension of the object-oriented database system GemStone [GemS00]. The two main components of TriGS are *TriGS Engine* comprising the basic concepts employed in TriGS for specifying and executing active behavior and *TriGS Developer*, an environment supporting the graphical development of active database applications. In the following, TriGS Engine and TriGS Developer are described as far as necessary for understanding the forthcoming sections.

TriGS Engine. Like most active systems, TriGS is designed according to the ECA paradigm. Rules and their components are implemented as *first-class objects* allowing both the definition and modification of rules during run time. Fig. 1 shows the basic structure of TriGS rules using the Backus-Naur Form (BNF). The symbols ::= [ ] are meta symbols belonging to the BNF formalism. Angle brackets denote non-terminal symbols.

```
<rule_definition> ::=
        DEFINE RULE <rule_name> AS
        ON <EselC>                              /* Condition event selector */
                IF <bool_expr> THEN             /* Condition part */
        [[WAIT UNTIL] ON <EselA>]               /* Action event selector */
                EXECUTE [INSTEAD] <action>      /* Action part */
        [WITH PRIORITY <number>]
        [TRANSACTION MODES(C:{serial|parallel},
                           A:{serial|parallel})]
        END RULE <rule_name>.
```

Fig. 1. BNF of an ECA Rule in TriGS

The event part of a rule is represented by a *condition event selector ($Esel_C$)* and an optional *action event selector ($Esel_A$)* determining the events (e.g., a machine breakdown) which are able to trigger the rule's condition and action, respectively [Kapp94b]. Triggering a rule's condition (i.e., an event corresponding to the $Esel_C$ is signaled) implies that the condition has to be evaluated. If the condition evaluates to true, and an event corresponding to the $Esel_A$ is also signaled, the rule's action is executed. If the $Esel_A$ is not specified, the action is executed immediately after the condition has been evaluated to true. By default, the transaction signaling the condition triggering event is not blocked while the triggered rule is waiting for the action triggering event to occur. Blocking can be specified by the keyword `WAIT UNTIL`.

In TriGS, any message sent to an object may signal a *pre- and/or post-message event*. In addition, TriGS supports *time events*, *explicit events*, and *composite events*. Composite events consist of *component events* which may be primitive or composite and which are combined by different *event operators* such as conjunction,

sequence and disjunction (for a more detailed description see [Rets98]). The event starting the detection of a composite event is called *initiating event*, the event terminating the detection is called *terminating event*. TriGS allows to detect not only composite events whose component events are all signaled within a single transaction or within different transactions. It is even possible that component events span different database sessions each comprising one or more transactions [Kapp96a], [Kapp96b]. For each event, a *guard*, i.e., a predicate over the event's parameters, may be specified, which further restricts the events able to trigger a condition or action, respectively. The condition part of a rule is specified by a Boolean expression, possibly based on the result of a database query (e.g., are there some scheduled jobs on the damaged machine?). The action part is specified again in terms of messages (e.g., display all jobs scheduled on the damaged machine and reschedule them on another machine). Considering rules incorporating message events, the keyword `INSTEAD` allows to specify that the action should be executed instead of the method corresponding to the message triggering the condition evaluation. The execution order of multiple triggered rules, i.e., conditions and actions of different rules which are triggered at the same time, is controlled by means of *priorities*.

The *transaction mode* is specified separately for conditions and actions respectively. It defines, in which transaction *rule processing* comprising *rule scheduling* and *rule execution* takes place. Rule scheduling includes the detection of composite events and the determination of triggered conditions and actions, respectively. Rule execution refers to condition evaluation and action execution. In case of a *serial mode*, rule processing is done as part of the database session's transaction which has signaled the triggering event, in case of a *parallel mode* rule processing is done within transactions of separate database sessions. In particular, rule scheduling is done within a dedicated database session running in a separate thread of control called *Parallel Rule Scheduler*. Rule execution is made efficient by means of several *Parallel Rule Processors* each running within a separate thread [Kapp98a]. The number of these threads is dynamically controlled and depends on the utilization of the corresponding rule processors. Note that rules incorporating composite events whose component events are signaled by different database sessions can have a parallel transaction mode only. This is also the case for rules which are triggered by events being signaled outside of a database session, e.g., time events.

TriGS Developer. TriGS Developer, the graphical development environment, is realized as Visual Works application [Visu00] and provides a set of tools in order to support the whole development cycle of an active database application [Kapp00] (cf. Fig. 2).

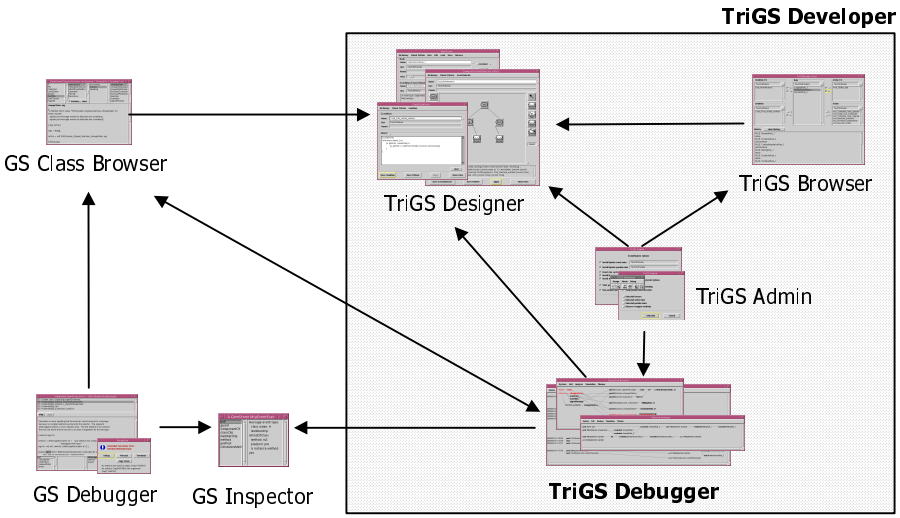

Fig. 2. Components of TriGS Developer

Rules and their components can be graphically defined and manipulated by means of different tools provided by *TriGS Designer*, such as Composite Event Editor, Condition Editor, and Action Editor. In order to facilitate the retrieval and thus the reuse of already existing rules, *TriGS Browser* allows to search for rules and their components in various ways. *TriGS Admin* is employed for setting up the run-time environment for applications requiring active behavior. Finally, the verification of active behavior is facilitated by *TriGS Debugger* which will be explained in more detail in the forthcoming sections. It has to be emphasized that the tools of TriGS Developer are *seamlessly integrated* among each other, as well as with some standard GemStone (GS) tools comprising GS Inspector, GS Debugger, and GS Class Browser. In particular, it is not only possible to navigate between the tools as depicted by the arrows in Fig. 2, but also to preserve the context during navigation. This implies that, e.g., a composite event selected in TriGS Browser is automatically loaded when TriGS Designer is opened.

3 Functionality of TriGS Debugger

This section is dedicated to an in-depth description of the functionality of TriGS Debugger.

3.1 Providing a Holistic View on Active and Passive Behavior

In TriGS, the passive behavior in terms of classes and their methods and the active behavior in terms of rules are developed using different tools, namely GS Class Browser and TriGS Designer, respectively. Although developed separately, interdependencies exist in that, e.g., rules may be triggered by method calls in case of message events. To enable a *holistic view*, which is important for pre-execution analysis, TriGS Debugger provides a *Structure Browser* (cf. Fig. 3). The Structure Browser visualizes both the elements of the passive behavior, i.e., *methods* together with the corresponding *class hierarchy* on the left side of the window, and the elements of the active behavior in terms of *rules* and their *event selectors* no matter being primitive or composite on the right side.

Passive behavior is visualized by means of a *class hierarchy tree*. The nodes of this tree are formed by a class (cf. ① in Fig. 3) together with those methods of that class which are referenced by event selectors (cf. ②). The edges between the classes denote inheritance relationships (cf. ③). Let's stick to an example, which is from a simple production scheduling and control system [Kapp95]. Considering our universe of discourse, there are two classes shown at the top level of the class hierarchy namely `Machine` and `Buffer`. Buffers are able to store operations (`add:`) which have to be scheduled and executed on a Machine. Machines can change their state (`changeState:`), maintain themselves (`maintain:`), and signal a warning (`signalWarning:`). `Machine` is further specialized into the object class `MillingMachine` containing a method used for scheduling operations (`schedule:`). Finally, `MillingMachine` is specialized into `MMWithoutBuffer` containing an overridden version of the method `changeState:`.

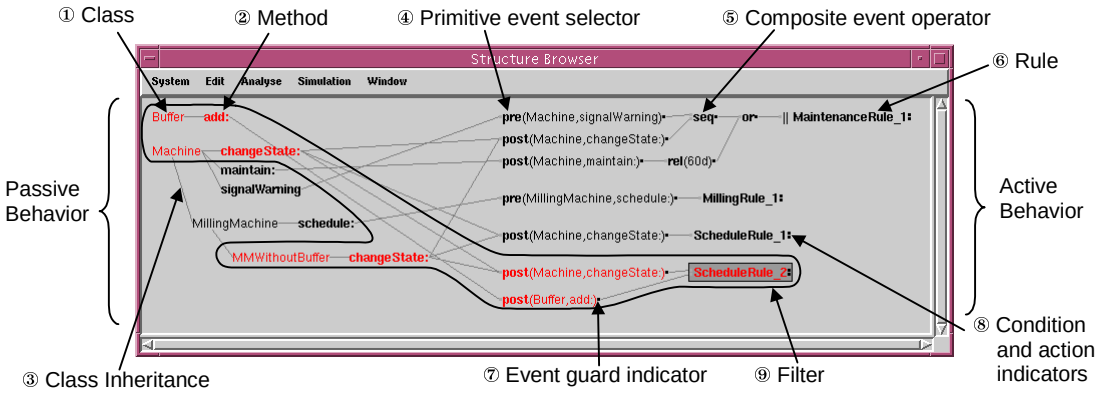

Fig. 3. Structure Browser

Visualization of active behavior comprises *event selector/rule trees*. Leaf nodes of that tree represent primitive event selectors (cf. ④). Inner nodes depict the event operators of composite event selectors (cf. the sequence operator in ⑤). Finally, root nodes represent rules (cf. ⑥). Note that the ordering of event selector nodes adheres to syntactical structures in that, e.g., the first (upper) child node of a rule denotes the $Esel_C$, and the second child node denotes the optional $Esel_A$. The symbol "||" preceding the name of a rule indicates that the rule, i.e., its condition and/or action has to be processed in parallel. In order to reduce information overload, guards, conditions, and actions of a rule are not explicitly shown within the Structure Browser. Rather, their existence is indicated by means of small black boxes only (cf. ⑦ and ⑧). To get more details on the active behavior or the passive behavior, or even for modification purposes, both the GS Class Browser and TriGS Designer may be opened directly from within the Structure Browser, whereby the selected class, method, event selector or rule is loaded automatically into the tool opened. In turn, the GS Class Browser has been adapted, in that a Structure Browser can be opened to display the event selectors and rules related to a particular class or method.

The interdependencies between passive behavior and active behavior are depicted by edges between methods and corresponding message event selectors. One and the same event selector is visualized as often as it is incorporated in different composite event selectors and/or rules. For example, the message event selector `post(Machine, changeState:)` is used as component event selector of `MaintenanceRule_1`, as $Esel_C$/$Esel_A$ of `ScheduleRule_1` and as $Esel_C$ of `ScheduleRule_2`. Due to inheritance, rules are inherited along the class hierarchy [Kapp94a]. However, to reduce complexity, edges from the subclasses of the class containing the respective method to the event selector(s) are not visualized, except the method has been overridden in a

subclass. For example, the event selector `post(Machine, changeState:)` is associated with `changeState:` in both `Machine` and `MMWithoutBuffer`, since `changeState:` has been overridden in `MMWithoutBuffer`.

Finally, the Structure Browser provides a filtering mechanism to cope with the fact that, when many classes and rules are visualized at once the display becomes cluttered. Different filters allow to focus on the passive and active behavior related to selected classes and/or rules. For example, one may select a single rule and a filter will provide the composite and primitive event selectors of that rule, together with the participating methods and classes (cf. ⑨). The result of a filter can be visualized in two ways. First, it is possible to exclusively show the filter result and hide the rest. Second, the filter result may be visualized in a context preserving way by means of highlighting using different colors. (Note that due to the black and white printout frames are used instead in Fig. 3.)

Example. Regarding the active behavior of our running example, the rule named `ScheduleRule_1` is responsible for scheduling the next operation which is fetched out of the machine's buffer, each time the machine becomes available (`post(Machine, changeState:)`). This rule is inherited by all subclasses of `Machine`, i.e., `MillingMachine`, and `MMWithoutBuffer`. `ScheduleRule_2` supplements `ScheduleRule_1` in the case that the buffer is empty which is checked by the condition incorporating the $Esel_C$ `post(Machine, changeState:)`. In that case it waits for a new operation by means of the event selector for action execution (`post(Buffer, add:)`). `MillingRule_1` checks every time a new operation is scheduled on a certain milling machine (`pre(MillingMachine, schedule:)`) whether the machine has still enough capacity to schedule the operation. If not, scheduling on this machine is aborted. `MaintenanceRule_1` is responsible for initiating maintenance of a machine either after the regular maintenance interval expires, which is every 60 days (`rel(60d)`), or as soon as the machine signals a warning (`pre(Machine, signalWarning:)`), but not before the machine changes its state to "available" (`post(Machine, changeState:)`).

3.2 Visualizing Trace Data

Visualizing trace data is an important feature of a rule debugger allowing post-execution analysis of the actual active behavior. For this, TriGS Debugger provides a *History Browser* (cf. Fig. 4) which may be opened at any time during or after the run of an active application to get a graphical snapshot of its behavior until that point in time. In particular, the History Browser contains the following information:

- **Temporal Context**. Since active behavior occurs over time, a *timeline* (cf. ① in Fig. 4) is used to organize trace information in a *temporal context*, from top to bottom. The timeline is displayed at the left side of the window with a granularity of one second, meaning that one timestamp subsumes all trace data which occurs within one second. Note that only those timestamps are shown for which trace data exists.
- **Event Detection**. Event detection of both primitive and composite events is visualized as *event graph*. This event graph is different from the event selector tree as depicted by the Structure Browser in that it represents events not at type level but rather at instance level. Since each event may be component of multiple composite events, a directed acyclic graph is formed. Leaf nodes on the left side of the graph denote detected primitive events, while other inner nodes depict the detection of composite events (cf. ②). A small black box right to an event or an event operator node indicates that a rule has been triggered by this event, and in case of a composite event that it has been detected completely (cf. ③). Note that our event graph is different from the ones used in [Chak95], [Coup97], and [Bern99], in that composite event detection is visualized by preserving the temporal context of the occurrence of events.
- **Rule Execution**. The visualization of rule execution is decomposed into condition evaluation and action execution, because these two phases of rule execution may be triggered at different points in time by different events, namely condition event and action event respectively. Visualization of a condition evaluation comprises the begin of evaluation (cf. `evaluating` in Fig. 4) together with the name of the rule, possibly cascaded rule executions, and the end of evaluation indicated by the result of evaluation, i.e., true, false, an object, or nil. The relationship between condition triggering event and begin of evaluation is depicted by an edge. Another edge is connecting begin of evaluation and end of evaluation. Action execution is visualized accordingly, with the difference that the relationship to both condition event and action event is shown by means of an edge. The symbol "||" preceeding the begin of evaluation/execution denotes that the condition/action was performed by a Parallel Rule Processor outside the session which signaled the triggering event.
- **Database Sessions**. Database sessions during which events have been signaled or rules have been processed appear in the History Browser as parallel vertical tracks (cf. ④). According to the separation of serial rule processing and parallel rule processing in TriGS as discussed in Section 2, all trace data concerning a serial

rule is shown within the track of the database session where the triggering event was signaled. This could be either an *Application Session* or a *Parallel Rule Processor Session*. Since events triggering parallel rules are processed within a dedicated database session, there is an own track labeled *Parallel Rule Scheduler* displayed right to the Application Session where all trace data of event detection for parallel rules is shown. In order to visualize the origin of primitive events processed by the Parallel Rule Scheduler, these events are displayed twice, both within the track of the event signaling session and within the Parallel Rule Scheduler track. Within the event signaling session, the primitive events signaled for parallel processing are preceded by the symbol "||" and depicted in italics to distinguish them from events signaled for serial processing. As discussed in Section 2, parallel rules are executed by parallel rule processors in separate database sessions. The tracks of Parallel Rule Processor sessions are displayed right to the Parallel Rule Scheduler track.

- **Transaction Boundaries**. Transaction boundaries within database sessions are represented by primitive events corresponding to commit/abort operations. Since committing a transaction may fail due to serialization conflicts, success or failure of a commit operation is indicated as well (cf. `transaction commit = true` in Fig. 4).

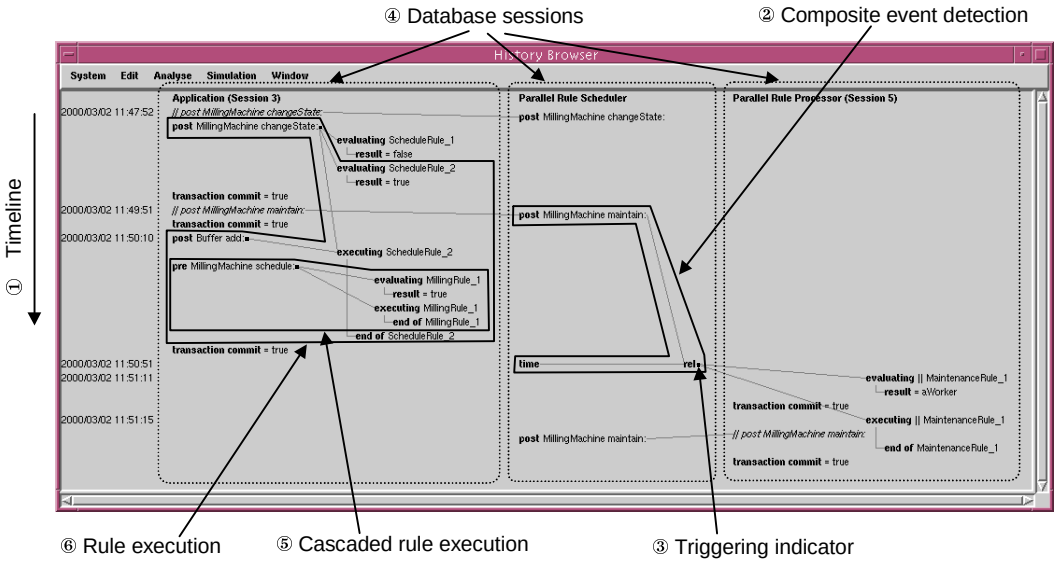

Fig. 4. History Browser

Further details are available by means of the integrated GS Inspector allowing the inspection and modification of trace data and related database objects, like condition results and action results as well as event parameters. However, one has to be aware that the Inspector shows the current state of an object, which may have changed since the time when, e.g., the object was used to evaluate the condition of a rule.

Example. Considering our running example, the trace data shown in the History Browser of Fig. 4 starts at timestamp `2000/03/02, 11:47:52` with a message event (`post MillingMachine changeState:`) signaled by Application Session 3. Since it should trigger a parallel rule, it is transferred to the Parallel Rule Scheduler track. However, as there is no black box right to this event, it still awaits further processing with respect to the end of the history at `11:51:15`. The same event is signaled for serial rule processing, now triggering condition evaluation of both `ScheduleRule_1` (the result being false) and `ScheduleRule_2` (the result being true) as well as being the component event of the $Esel_A$ of `ScheduleRule_2`. After `ScheduleRule_2` has been evaluated, the transaction of Application Session 3 committs successfully (`transaction commit = true`). The next event `post MillingMachine maintain:` is again responsible for triggering a parallel rule, however the composite event detection process is only initiated at that time (cf. ②). In the following transaction the event `post Buffer add:` is signaled triggering the action of `SchedulingRule_2`. During action execution, rule cascading takes place (cf. ⑤) in that another event `pre MillingMachine schedule:` is signaled and immediately triggers condition evaluation and action execution of `MillingRule_1`, before the action of `SchedulingRule_2` can be completed (cf. ⑥). Some time later the Parallel Rule Scheduler signals a time event, which terminates a composite event incorporating a relative event operator, and thus triggers condition and action of `MaintenanceRule_1`. Since `MaintenanceRule_1` has been specified as parallel, evaluation and execution takes place within parallel transactions, and as shown in Fig. 4 both performed by Parallel Rule Processor Session 5.

3.3 Customizing and Analyzing Trace Data

As one can imagine when looking at the exemplary screenshot of Fig. 4, the graphical representation of trace data comprising a huge amount of events and rules can get fairly complex. Especially composite events, which may have to be traced over a long period of time and originating from various different sessions contribute to the complexity of visualization. To reduce the information overload, TriGS Debugger provides two complementary mechanisms allowing to customize the visualization of trace data, namely a *filtering mechanism* and a *pattern analyzer*.

3.3.1 Filtering Mechanism

The filtering mechanism of TriGS Debugger allows to focus the History Browser's view on particularly interesting details of the trace data, while preserving their order of occurrence by providing several *context independent* and *context dependent* filters. Whereas context dependent filters allow to focus on causes and effects of selected parts of trace data, context independent filters allow to focus on the active behavior without selecting any particular trace data. Analogous to the Structure Browser, the filter results may be visualized either exclusively in a new window, or by simply highlighting them. Context independent filters supported by TriGS Debugger are:

- **Session.** With the session filter, it is possible to show/hide every track of the History Browser.
- **Time.** The time filter allows to focus on particular time intervals, specified either by begin and end time, or relative to the trace end, e.g. "the last 10 minutes".
- **Names of Event Selectors and Rules.** The view on a certain trace can be restricted to events conforming to certain event selectors, and/or having triggered evaluations and executions of certain rules.

The initial focus set by context independent filters can be modified step-by-step in an interactive manner by means of the following context dependent filters:

- **Cause.** The cause filter applied to a condition evaluation or an action execution shows the triggering event. Applying the cause filter to a primitive event results in either the rule execution during which it has been signaled, or the Application Session which signaled the event.
- **Direct/Indirect Effects.** The effect filter allows to focus either on direct effects only or on both direct and indirect effects of events and rule executions. Concerning a *primitive event*, the direct effects are the composite events terminated by that event, and the conditions/actions triggered, whereas the indirect effects are in addition all initiated composite events and all conditions/actions triggered by that initiated composite events. Concerning a *rule execution*, the direct effects are the primitive and composite events detected during execution as well as the first level of cascaded rule executions. The indirect effects of a rule execution include all levels of cascaded rule executions, as well as the events detected during these cascaded executions.
- **Time.** In order to focus on the temporal context of an event or of a rule execution, the time filter may be applied resulting in all events and rule executions within some time interval before and after the selected event/rule execution.

Fig. 5 is based on the same trace data as shown in Fig. 4, with filters being applied, thus making the trace data more readable. In particular, first a context independent filter has been applied showing only the events signaled from Application Session 3 (cf. ①), and second, based on the result of the first filter, a context dependent filter has been used to highlight the direct and indirect effects of the selected event (cf. ②).

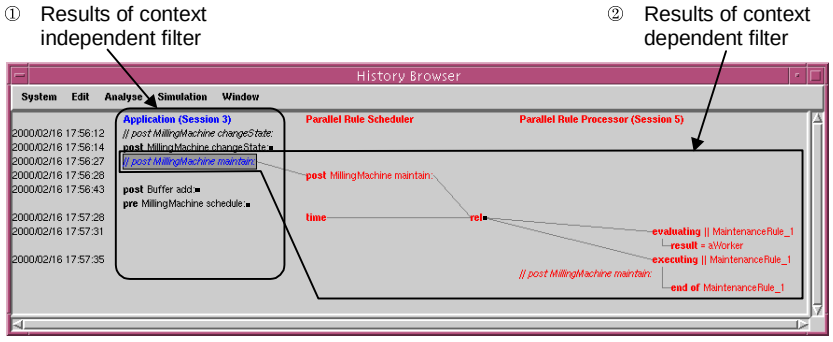

Fig. 5. Filtering Trace Data Within the History Browser

3.3.2 Pattern Analyzer

Besides filtering mechanisms, TriGS Debugger provides a *Pattern Analyzer*. In contrast to the History Browser, which shows each detected event and each executed rule in order of their occurrence, the Pattern Analyzer mines

patterns of recurring active behavior and shows these patterns once, in an aggregated fashion. With this, a compact view on trace data is provided. Causal dependencies between rules and their components are visualized at one place, without having to browse through the whole history of trace data.

In order to mine patterns, the trace is analyzed step by step and similar trace data is aggregated into *equivalence classes*, whereby not only the equivalence of events and rule executions themselves is considered, but also the equivalence of relationships among them in terms of causal dependencies. Therefore, the *equivalence relation* is defined recursively, as follows:

- Two *primitive events* are considered equivalent if (1) they conform to the same primitive event selector, and (2) their causes, i.e., either the rule executions or the applications having signaled them, are in the same equivalence class.
- Equivalence of (partially detected) *composite events* is defined by the equivalence of their composition trees consisting of primitive component events (leafs) and event operators (nodes). Two such trees are equivalent if the root nodes are equivalent. Two nodes are in turn equivalent if (1) the event operator is the same, and (2) the child nodes/leafs are in the same equivalence class.
- Two *condition evaluations* are equivalent if (1) they belong to the same rule, and (2) the triggering events are in the same equivalence class.
- Two *action executions* are equivalent if (1) they belong to the same rule, and (2) the corresponding condition evaluations are in the same equivalence class, and (3) the action triggering events – if defined – are in the same equivalence class.

Patterns of behavior are visualized as a *directed acyclic graph*, with root nodes on the left representing equivalence classes of active behavior with outside cause, e.g., equivalence classes of primitive events raised by an application, and child nodes denoting equivalence classes of resulting active behavior. Note that this abstract view on rule behavior is similar to so-called *triggering graphs* as used in static rule analysis [Aike92]. Unlike static rule analysis, which can be done before run-time, trace data patterns are not available until the application has been started. However, since trace data patterns represent the real active behavior of an application they have the advantage that *quantitative information* can be provided for each element of a pattern in terms of number of occurrences and execution time, comprising minimum, maximum and average values. *Qualitative information* in terms of, e.g., rule cycles is not yet automatically provided but rather subject to future work (cf. Section 5).

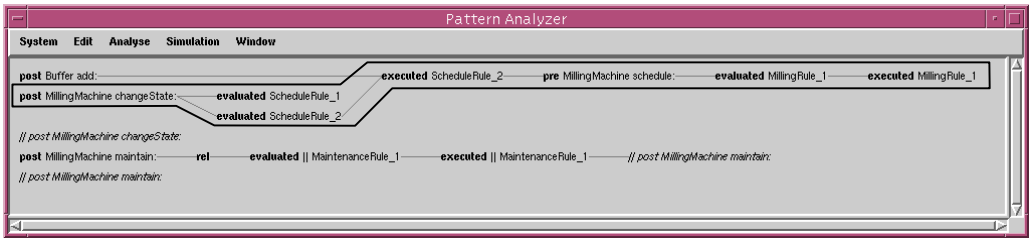

Fig. 6. Pattern Analyzer

Fig. 6 shows the Pattern Analyzer's view of the trace data as shown in Fig. 4. One can see at one place, e.g., all the direct and indirect consequences of the event `post(MillingMachine,changeState)`. Another insight is that `MillingRule_1` has been processed completely within the execution of `ScheduleRule_2`.

3.4 Interactive Rule Debugging

Beside the visualization and analyzing capabilities, TriGS Debugger allows to interactively control, examine and manipulate the active behavior by means of breakpoints, a replay and simulation mechanism and again by taking advantage of the other components of TriGS Developer and the standard GS tools.

First of all, TriGS Debugger allows to set breakpoints from within the History Browser and the Structure Browser. Breakpoints may be defined on all rule components namely *events*, *guards*, *conditions*, and *actions*. Whenever during rule processing TriGS Engine encounters a breakpoint, processing is stopped and a breakpoint window is opened (cf. Fig. 7).

The breakpoint window shows some details on the actual state of rule processing at the left side and offers different functionality at the right side, depending on the rule component the breakpoint has been set on. In case the breakpoint has been set on a condition, one may choose to (1) proceed with condition evaluation (cf. `Evaluate` in Fig. 7), (2) continue under control of the GS Debugger in order to step through the condition's code (cf. `Debug`), (3) skip the evaluation of the condition by setting the condition to either true or false (cf. `True`,

`False`), (4) terminate the evaluation (cf. `Terminate`). Concerning breakpoints on events, one can in addition ignore the raised event, or continue until the next event is signaled. It has to be emphasized that since the GS Debugger can be incorporated from within the Breakpoint Window, one can control and examine the passive behavior of the application and its active behavior simultaneously.

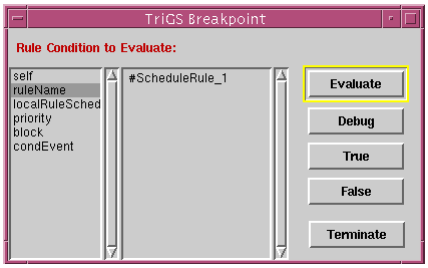

Fig. 7. Breakpoint Window

TriGS Debugger offers similar interactive features for modifying the active behavior at runtime as the GS Debugger. Exploring the interpretative nature of GS, it is possible to modify rule code at run time, even during a breakpoint halt. It is further possible to temporarily enable or disable certain events, guards, or rules and to modify rule priority settings.

The History Browser also supports a simple kind of event simulation, by allowing to replay events meaning that they are raised again. Either a single event or an arbitrary sequence of events can be selected from the event history in order to be replayed. By means of the GS Inspector, it is possible to modify event parameters before replaying, thus enabling the testing of guards and conditions. This way, an application scenario can be simulated, allowing to test a rule set without the need to run the application.

4 Architecture of TriGS Debugger

The overall architecture of TriGS Debugger can be divided into two parts, a *front-end* and a *back-end*. The front-end provides the user interface in terms of Structure Browser, History Browser, Pattern Analyzer and Breakpoint Window. The back-end comprises *TriGS Tracer*, responsible for trace generation as well as breakpoint detection and signaling and *TriGS Analyzer*, performing trace analysis and filtering. Note that the back-end is operational without any front-end tool activated, hence trace data can be generated even if no debugger window is opened. According to the three-tier architecture of GS, the front-end of TriGS Debugger is located on the client side, while the back-end is running on the object server together with the TriGS Engine (cf. Fig. 8). The database layer contains the passive behavior and the active behavior, together with trace data, and breakpoint data.

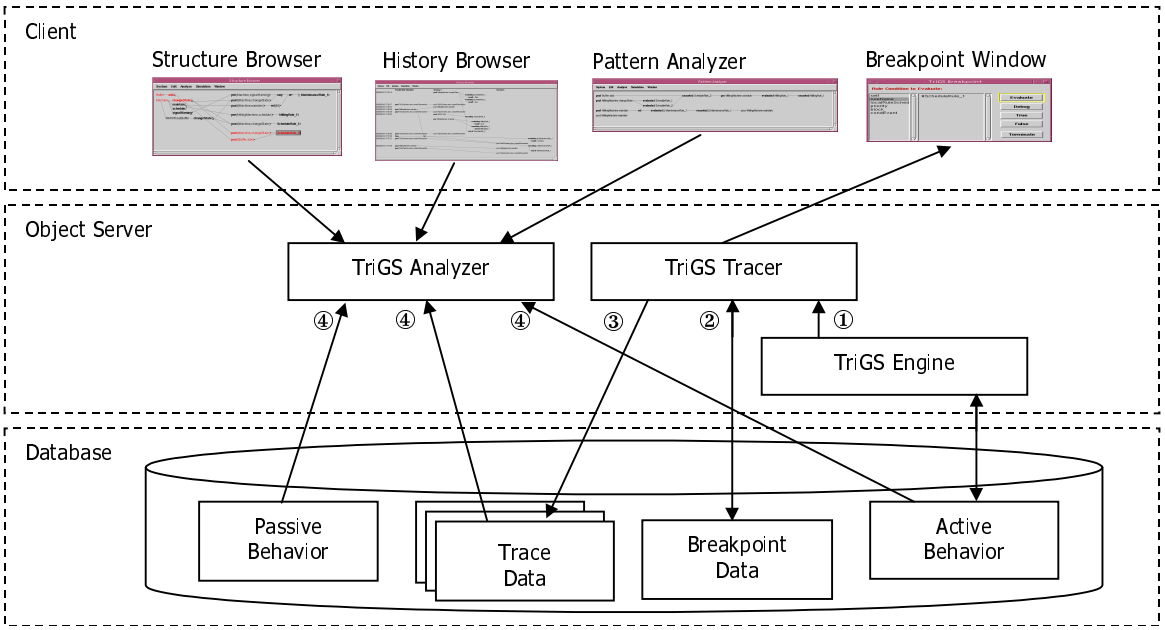

Fig. 8. Architecture of TriGS Debugger

TriGS Tracer. TriGS Tracer monitors TriGS Engine in order to generate trace data, and to detect breakpoints. Each step performed by TriGS Engine in order to process the active behavior, like the detection of

primitive and composite events, begin and end of condition evaluations and action executions, and the scheduling of rules, is signaled to the Tracer (cf. ① in Fig. 8). When the Tracer recognizes that a breakpoint is defined for such a step, it uses the GS error handling mechanism to raise a special exception signal (cf. ②). This signal is then routed to the client, and a breakpoint window opens. If no breakpoint was detected, or if execution continues from a breakpoint, the processing step signaled by TriGS Engine is added to the trace (cf. ③). Since trace data and application data are kept within the same database, the trace can include references to application objects, like the parameters of a message event. Furthermore, it is possible to run multiple active applications in parallel, each application contributing trace data. In order to avoid access conflicts, multiple traces are used, one trace for each database session of these applications (cf. below).

TriGS Analyzer. TriGS Analyzer performs the database queries and analysis methods necessary to generate a view on trace data, active behavior and passive behavior, which will then be displayed by front-end tools (cf. ④). This includes the integration of the separate session traces into one logical trace, and the application of filters. Since the trace can grow very large, it is very important that analysis and filtering is performed on the object server. Otherwise the trace would have to be replicated at the client, resulting in a huge communication and memory load.

5 Lessons Learned and Future Work

TriGS Debugger has been already employed in a project aiming at the development of a schema generator that translates a conceptual schema modeled by means of an AODB (Active Object/Behavior Diagram) editor [Lang97], [Lang98] into an executable TriGS schema [Ober00]. The active behavior generated thereby is highly complex and therefore difficult to understand, since it makes extensive use of both composite events and parallel rules. This section discusses the lessons learned in the course of this project and points to future work.

Conflicts Between Parallel Rules are Hard to Locate. As stated elsewhere [Pato99], the complexity of active behavior is not founded in single rules but in the interaction among rules. With parallel executing rules, this interaction is multiplied by the possibility of concurrent access to database resources, which in turn may lead to serialization conflicts and abort of the corresponding transactions. It has been shown that parallel rules were involved in the most non-trivial bugs which had to be found and removed during development of the schema generator. Therefore we believe that debugging tools for parallel executing rules are even more important than for serial ones. Although TriGS Debugger facilitates debugging of parallel rules in that parallel database sessions, transactions, rules executed within a transaction, and success or failure of transactions are visualized, the reason of a conflict is not explained by the current prototype. In this sense it would be desireable to show the conflicting transactions/rules and the objects involved by a debugger. Furthermore, techniques for debugging parallel programs should be adopted. For instance, the *trace driven event replay* technique as already supported by TriGS Debugger could be used for semi-automatic detection of *race conditions* [Grab95].

A History of Database States would Ease Debugging. Trace data gathered by TriGS Debugger represents the history of event detections and rule executions, but it does not include the history of related database objects. This leads to a problem already discussed in Section 3.2, namely that one can inspect the current state of an object only. However the object may have changed since the time when it has been used, e.g., to evaluate the condition of a rule. Breakpoints are a cumbersome means to solve this problem, since they force a rule developer to interactively handle each break during an application test. Instead, it would be beneficial if a rule debugger could access a snapshot of the database state as it was at the time when a guard or condition was evaluated or an action was executed.

Active Behavior Requires Active Visualization. In order to support a rule developer in observing active behavior, a debugging tool should notify the developer automatically of any new rules which have been triggered or events which have been signaled. This could be achieved by updating the view(s) on trace data any time when considerable trace data has been added, or within certain time intervals. In the current prototype, the views of History Browser and History Analyzer have to be updated explicitly in order to check for new events/rule executions.

Trace Analysis is a Promising Research Topic. We consider analysis of trace data a promising alternative to static rule analysis. Especially in an environment like TriGS, which is based on the untyped language GS Smalltalk, static analysis is very restricted. Instead, we will focus on deriving qualitative information on active behavior from trace data. For instance, as already noted in Section 3.3.2, we plan to enhance the Pattern Analyzer to provide information on rule execution cycles. In general, TriGS Debugger should detect (patterns of) active behavior which is violating certain properties like termination or determinism, and this should be highlighted/visualized in both the History Browser ‘s and Pattern Analyzer’s view. Another possible application of trace data analysis would be to save trace data of different application tests (of different application versions)

and compare them afterwards. Comparison might be in terms of execution order or encountered behavior patterns.

TriGS Debugger is not only Specific to TriGS. TriGS Debugger has been designed specifically for TriGS, without having its application to other active database systems in mind. As such, the implementation of features like interactive rule debugging relies heavily on specific properties of the underlying TriGS/GS system. Nevertheless, the basic ideas of TriGS Debugger, like providing a holistic view on active and passive behavior, the visualization and filtering of trace data, and pattern analysis of trace data, can be applied to debuggers for active (object-oriented) database systems different from TriGS as well.

References

[ACT96] ACT-NET Consortium, *The Active Database Management System Manifesto: A Rulebase of ADBMS Features*. SIGMOD Record, 25(3), 1996, pp. 40-49.

[Aike92] Aiken, A., Widom, J., Hellerstein, J., *Behavior of database production rules: Termination, confluence, and observable determinism*. SIGMOD Record, Vol. 21, 1992, pp. 59-68.

[Aike95] Aiken, A., Hellerstein, J. M., and Widom, J., *Static Analysis Techniqies for Predicting the Behaviour of Active Database Rules*. ACM Transactions on Database Systems (TODS), 20(1), 1995, pp. 3-41.

[Bail98] Bailey, J., Dong, G., and Ramamohanarao, K., *Decidability and Undecidability Results for the termination problem of active database rules*. In Proceedings of the 17th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, Seattle, Washington., 1998, pp. 264-273.

[Bara96] Baralis, E., Ceri, S., Fraternali, P., and Paraboschi, S., *Support Environment for Active Rule Design*. Journal of Intelligent Information Systems, 7(2), 1996, pp. 129-150.

[Bara99] Baralis, E., *Rule Analysis*. In Norman W. Paton (Ed.), Active Rules in Database Systems. Springer, New York, 1999, pp. 51-67.

[Behr94] Behrends, H., *Simulation-based Debugging of Active Databases*. In Proceedings of the 4th International Workshop on Research Issues in Data Engineering (RIDE) - Active Database Systems, Houston, Texas, IEEE Computer Society Press, 1994, pp. 172-180.

[Bena95] Benazet, E., Guehl, H., and Bouzeghoub, M. *VITAL: A Visual Tool for Analysis of Rules Behaviour in Active Databases*, In Sellis, T., (ed), Proceedings of the 2nd International Workshop on Rules in Database Systems, Springer LNCS Vol. 985, 1995, pp. 182-196.

[Bern99] Berndtsson, M., Mellin, J., Högberg, U., *Visualization of the Composite Event Detection Process*. In Paton, N. W. and Griffiths, T., (eds.), International Workshop on User Interfaces to Data Intensive Systems (UIDIS'99), IEEE Computer Society, 1999, pp. 118-127.

[Chak95] Chakravarthy, S., Tamizuddin, Z., Zhou, J., *A Visualization and Explanation Tool for Debugging ECA Rules in Active Databases*. In Sellis, T., (ed.), Proceedings of the 2nd International Workshop on Rules in Database Systems, Springer LNCS Vol. 985, 1995, pp. 197-212.

[Coll99] Collet, C., *NAOS*. In Active Rules in Database Systems, Monographs in Computer Science, Chapter 15, Springer, 1999, pp. 279--296.

[Coup97] Coupaye, T., Roncancio, C.L., Bruley, C., Larramona, J., *3D Visualization Of Rule Processing In Active Databases*. In Proceedings of the workshop on New paradigms in information visualization and manipulation, 1998, pp. 39-42.

[Diaz93] Diaz, O., Jaime, A., Paton, N., *DEAR: a DEbugger for Active Rules in an object-oriented context*. In Proceedings of the 1st International Workshop on Rules in Database Systems, Workshops in Computing, Springer, 1993, pp. 180-193.

[Etzi95] Etzion, O, *Reasoning About the Behaviour of Active Database Applications*. In Proceedings of the 2nd International Workshop on Rules in Database Systems, Springer LNCS Vol. 985, 1995, pp. 86-100.

[Fors95] Fors, T., *Visualization of Rule Behavior in Active Databases*. In Proceedings of the IFIP 2.6 3rd Working Conference on Visual Database Systems (VDB-3), 1995, pp. 215-231.

[Frat97] Fraternali, P., Teniente, E., and Urpi, T., *Validating Active Rules by Planning*. In Proceedings of the 3rd International Workshop on Rules in Database Systems, Springer LNCS Vol. 1312, 1997, pp. 181-196.

[GemS00] GemStone Systems Inc., http://www.gemstone.com/, 2000.

[Gepp97] Geppert, A. and Timbros, D., *Logging and Post-Mortem Analysis of Workflow Executions Based on Event Histories*. In Proceedings of the 3rd International Workshop on Rules in Database Systems, Springer LNCS Vol. 1312, 1997, pp. 67-82.

[Grab95] Grabner, S., Kranzlmüller, D., Volkert, J., *Debugging parallel programs using ATEMPT*, Proceedings of HPCN Europe 95 Conference, Milano, Italy, May, 1995.

[Jahn96] Jahne, A., Urban, S. D., and Dietrich, S. W., *PEARD: A Prototype Environment for Active Rule Debugging*. Journal of Intelligent Information Systems, 7(2), 1996, pp. 111-128.

[Kapp94a] Kappel, G., Rausch-Schott, S., Retschitzegger, W., Vieweg, S., *TriGS - Making a Passive Object-Oriented Database System Active*, Journal of Object-Oriented Programming (JOOP), 7(4), 1994, pp. 40-63.

[Kapp94b] Kappel, G., Rausch-Schott, S., Retschitzegger, W., *Beyond Coupling Modes - Implementing Active Concepts on Top of a Commercial ooDBMS*. In Proceedings of the International Symposium on Object-Oriented Methodologies and Systems (ISOOMS), Springer LNCS 858, Palermo, Italy, 1994, pp. 189-204.

[Kapp95] Kappel, G., Rausch-Schott, S., Retschitzegger, W., Tjoa, A M., Vieweg, S., Wagner, R., *Active Object-Oriented Database Systems For CIM Applications*. Information Management in Computer Integrated Manufacturing, A Comprehensive Guide to State-of-the-Art CIM Solutions, H. Adelsberger, J. Lazansky, V. Marik (eds.), Springer LNCS Vol. 973, 1995, pp. 96 - 131.

[Kapp96a] Kappel, G., Rausch-Schott, S., Retschitzegger, W., Sakkinen, M., 1996. Multi-Parent Subtransactions - Covering the Transactional Needs of Composite Events, *International Workshop on Advanced Transaction Models and Architectures (ATMA)*, Goa (India), 1996, pp. 269-282.

[Kapp96b] Kappel, G., Rausch-Schott, S., Retschitzegger, W., Sakkinen, M., *A Transaction Model For Handling Composite Events*. In Proc. of the Third International Workshop of the Moscow ACM SIGMOD Chapter on, Advances in Databases and Information Systems (ADBIS'96), MePhI, Moscow, September, 1996, pp.116-125.

[Kapp98a] Kappel, G., Rausch-Schott, S., Retschitzegger, W., A Tour on the TriGS Active Database System - Architecture *and Implementation*. In Proceedings of the 1998 ACM Symposium on Applied Computing (SAC'98), J. Carroll et al (eds.), Atlanta, USA, February/March, 1998, pp. 211-219.

[Kapp98b] Kappel, G., Retschitzegger, *The TriGS Active Object-Oriented Database System - An Overview*. ACM SIGMOD Record, Vol. 27, No. 3, September, 1998, pp. 36-41.

[Kapp00] Kappel, G., Rausch-Schott, S., Retschitzegger, W., Sakkinen, M., *Rule Patterns - Bottom-up Design of Active Object-Oriented Databases*. Accepted for publication in Communications of the ACM (CACM), 2000.

[Kran96] Kranzlmüller, D., Grabner, S., Volkert, J., *Event graph visualization for debugging large applications*, Proceedings of the SIGMETRICS symposium on Parallel and distributed tools, Philadelphia, PA USA, May, 1996, pp. 108-117.

[Lang97] Lang, P., Obermair, W., Schrefl, M., *Modeling Business Rules with Situation/Activation Diagrams*, In: A. Gray, P. Larson (eds.): Proceedings of 13th International Conference on Data Engineering (ICDE '97), Birmingham, U.K., IEEE Computer Society Press, April, 1997, pp. 455-464.

[Lang98] Lang, P., Obermair, W., Kraus, W., Thalhammer, T., *A Graphical Editor for the Conceptual Design of Business Rules*, In: Proceedings of the 14th International Conference on Data Engineering (ICDE '98), Orlando, Florida, IEEE Computer Society Press, February, 1998, pp. 599-609.

[Lee99] Lee, S. Y., Ling, T. W., *Unrolling cycle to decide trigger termination*. In Proceedings of the 25th International Conference on Very Large, Data Bases, 1999, pp. 483-493.

[Mont99] Danilo Montesi, Maria Bagnato and Cristina Dallera, *Termination Analysis in Active Databases*, Proceedings of the 1999 International Database Engineering and Applications Symposium (IDEAS'99), Montreal, Canada, August, 1999.

[Ober00] Obermair, W., Retschitzegger, W., Hirnschall, A., Kramler, G., Mosnik, G., *The AOODB Workbench: An Environment for the Design of Active Object-Oriented Databases*, Software Demonstration at the Int. Conference on Extending Database Technology (EDBT 2000), Konstanz, Germany, March, 2000.

[Pato99] Paton, N. W., Diaz, O., *Active Database Systems*. ACM Computing Surveys, 31(1), 1999, pp. 63-103.

[Rets98] W. Retschitzegger, *Composite Event Management in TriGS - Concepts and Implemention.* In Proceedings of the 9th International Conference on Database and Expert Systems Applications (DEXA'98), Vienna, August, 1998, pp. 1-15.

[Thom96] Thomas, I. S. and Jones, A. C., *The GOAD Active Database Event/Rule Tracer*. In Proceedings of the 7th International Conference on Database and Expert Systems Applications, volume 1134 of Lecture Notes in Computer Science, Springer, 1996, pp. 436-445.

[Vadu97] Vaduva, A., Gatziu, S., Dittrich, K. R., *Investigating Termination in Active Database Systems with Expressive Rule Languages*. In Proceedings of the 3rd International Workshop on Rules In Database Systems, Skovde, Sweden, 1997, pp. 149-164.

[Vadu98] Vaduva, A., *Rule Development for Active Database Systems*, PhD thesis, University of Zurich, 1998

[Visu00] Visual Works Smalltalk, ObjectShare Inc., http://www.objectshare.com/, 2000.