

ACTIVE RULE ANALYSIS AND OPTIMISATION IN THE ROCK & ROLL
DEDUCTIVE OBJECT-ORIENTED DATABASE[†]ANDREW DINN¹, NORMAN W. PATON² and M. HOWARD WILLIAMS¹¹Department of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh, UK²Department of Computer Science, University of Manchester, Oxford Road, Manchester, UK*(Received 12 December 1996; in final revised form 22 March 1999)*

Abstract — Active database systems provide facilities that monitor and respond to changes of relevance to the database. Active behaviour is normally described using Event Condition Action rules (ECA-rules), and a number of systems have been developed, based upon different data models, that support such rules. However, experience using active databases shows that while such systems are powerful and potentially useful in many applications, they are hard to program and liable to exhibit poor performance at runtime. This document addresses both of these issues by examining both analysis and optimisation of active rules in the context of a powerful active database system. It is shown how rule analysis methods developed for straightforward active rule languages for relational data models can be extended to take account of rich event description languages and more powerful execution models. It is also shown how the results of analyses can be exploited by rule optimisers, and that multiple query optimisation can be applied in a range of circumstances to eliminate duplicated work during rule processing. Copyright ©1999 Elsevier Science Ltd

Key words: Active Database, Rule Analysis, Rule Optimisation, Object-Oriented Databases, Deductive Object-Oriented Databases

1. INTRODUCTION

An active database system is a database system enhanced with mechanisms that support automatic reactions to situations of relevance to a data intensive application. Active behaviour is generally expressed by way of rules, which describe events to be monitored, conditions that examine the context in which an event has taken place, and actions that indicate the responses that have to be made to particular events. There have been many proposals for systems that support such event-condition-action (ECA) rules, and the fundamental characteristics of these systems are now well understood [17, 35, 26, 25]. However, experience in the use of such systems indicates that certain difficulties arise when the technology is put into practice [29]; central to these problems are issues relating to comprehension and performance of active applications.

Active rules can be seen as providing low-level but powerful mechanisms for describing certain behavioural aspects of an application. Users of active database systems have become aware that the behaviour exhibited by a set of rules is not always straightforward to anticipate or understand, and a number of researchers have made proposals for the static analysis [37, 34, 20, 3, 2, 1] and runtime observation [13, 8, 11] of rule behaviour. Work on static analysis, however, has generally been conducted in the context of rule languages that are less powerful than are found in many prototype active databases, and it is not immediately obvious how easily certain results can be extended for use with such systems. This paper examines the applicability of earlier work on rule analysis for use in a powerful active rule language, and shows how results can be adapted and extended for use in an active extension of the ROCK & ROLL deductive object-oriented database system [7, 6].

In addition to the problems in designing and debugging rule bases, [29] also identifies performance as a key concern of potential users of active database systems. One of the reasons why applications that exploit active rules have given poorer performance than anticipated is that rules that have been designed and implemented to be correct as free standing units may repeat certain database accesses when triggered together in a transaction. Effective optimization of active rule

[†]Recommended by Klaus R. Dittrich

bases should be able to identify and share retrieval tasks that are common to multiple simultaneously triggered rules, by exploiting multiple query optimization techniques (e.g. [28]). However, exploitation of such techniques in active database systems has been blocked to date, as it is only possible to perform multiple query optimization in a context where static rule analysis can guarantee that optimizations will not change the meaning of a set of rules. This paper shows how certain rule analyses can be used to identify contexts in which multiple query optimization can be performed, and describes how this technique can be exploited in the active rule language of ROCK & ROLL.

This document is structured as follows: Section 2 presents the principal characteristics of active database systems, introduces some terminology used in later sections, and describes earlier work on the analysis of active rule bases; Section 3 outlines the features of the ROCK & ROLL deductive object-oriented database, and describes the active rule language associated with it; Section 4 discusses rule analysis of ROCK & ROLL; Section 5 shows how results in the area of rule analysis can be applied to active rule optimization; and Section 6 presents some conclusions.

2. CONTEXT

2.1. Active Databases

This paper assumes some measure of familiarity with active databases, along the lines of that presented in [17, 35, 25, 26]. However, as terminology is not always used consistently in the literature, this subsection introduces a few terms that are used later in the paper.

Active database systems monitor events, which may be primitive (e.g. an update to a database object, the invocation of an operation) or composite (e.g. a number of specific operations take place in a specified order within a transaction). When an event has been *detected* as having taken place, the rules that are monitoring the event are *triggered*. If a rule is triggered and its condition evaluates to true, then its action is executed, and the rule is said to have *fired*.

There is considerable variety both in the languages that are used to describe active behaviour and in the semantics of rules at runtime. Further important features of active rule system semantics are introduced as they are used in the paper.

2.2. Rule Analysis in Active Databases

This section summarises the results of earlier research on the static analysis of active rules. Static rule analysis involves identifying certain properties of a rule base at rule definition time, with a view to warning rule programmers of undesirable behaviour patterns that may be exhibited by the rule base. The two properties that are most often identified as meriting static detection are [1]:

Termination: Is rule processing guaranteed to terminate after any set of changes to the database in any state?

Confluence: Is rule processing guaranteed to reach a unique final state for every valid and complete rule execution sequence starting at any legal database state?

The identification of potential non-termination in a set of rules warns programmers at compile time of the possibility of infinite loops at run time, while the identification of potential non-confluence warns programmers that a rule base may exhibit nondeterministic behaviour.

This section addresses two aspects of rule analysis: the examination of properties of graphs which show how sets of rules interact with each other [1, 2], and the study of how individual rule actions and conditions relate to each other [4].

2.2.1. Analysing Rules using Trigger and Activation Graphs

Formal definitions of the structures and proofs of the properties introduced in this section are given in [1, 2].

A *Triggering Graph* [1] is a directed graph $\{V, E\}$, where each node in V corresponds to a rule in the rule base, and each edge $r_i \rightarrow r_j$ in E means that the action of rule r_i generates events that trigger r_j .

If there are no cycles in a triggering graph for the rule set R , then processing involving the rules in R is guaranteed to terminate.

The triggering graph, however, fails to take into account the details of the interaction between the conditions and the actions of potentially non-terminating rules. Consider, for example, the following rule:

```
R1: ON : update to attribute A of T
     IF : new value for A > 10
     DO : set A of T to 10
```

The triggering graph for the rule base involving R1 contains a cycle, as the action of R1 updates the attribute A of T, which in turn triggers R1. However, non-terminating behaviour will not result, as the action of R1 assigns to A a value for which the condition of R1 will never be true. It is to overcome this limitation in triggering graphs that activation graphs have been introduced.

An *Activation Graph* [2] is a directed graph $\{V, E\}$, where each node in V corresponds to a rule in the rule base, each edge $r_i \rightarrow r_j$ ($i \neq j$) in E means that the action of rule r_i may change the truth value of the condition of r_j from *false* to *true*, and each edge $r_i \rightarrow r_i$ means that the condition of r_i may be *true* after the execution of its own action.

Detection of potential non-termination can be made less conservative by considering both triggering graphs and activation graphs together; the position is essentially that a rule set can only exhibit non-terminating behaviour when there are cycles in both the triggering graph and the activation graph that have at least one rule in common [2].

The activation graph for rule R1 given above contains no cycle, as its condition cannot be true after the execution of its action. Thus, even though the triggering graph contains a cycle, execution of the rule does terminate.

Analysis of confluence is built on the notion of rules that commute. A pair of rules r_i and r_j commute if, for any database state, executing r_i followed by r_j produces the same final database state as executing r_j followed by r_i .

The rules r_i and r_j commute if: the execution of rule r_i cannot trigger rule r_j , the action of r_i cannot affect the truth of the condition of r_j (by changing it from true to false, or vice versa), the action of r_i cannot change the effect of executing the action of r_j , and if the above conditions also hold with i and j reversed.

A rule set R is guaranteed to be confluent if, for every pair of rules r_i and r_j from R , the following algorithm (from [1], where an explanation and correctness proof are given) yields sets R_1 and R_2 , such that every rule in R_1 commutes with every rule in R_2 .

```
R1 ← {ri}
R2 ← {rj}
```

repeat until unchanged:

```
R1 ← R1 ∪ {r ∈ R | r ∈ Triggers(r1) for some r1 ∈ R1
               and r mayFireBefore r2 for some r2 ∈ R2
               and r ≠ rj }
R2 ← R2 ∪ {r ∈ R | r ∈ Triggers(r2) for some r2 ∈ R2
               and r mayFireBefore r1 for some r1 ∈ R1
               and r ≠ ri }
```

The function *Triggers* returns the set of rules that may be triggered by its argument. [1] bases the *mayFireBefore* predicate for Starburst rules on the rule priority ordering defined by the rule programmer. More complex rule systems require a more sophisticated definition of this predicate.

The approaches described in this section are not the whole story, however, as the definitions are dependent on the analysis of pairs of rules, to determine how, if at all, these rules may interact with each other. This latter issue is considered in the following subsection.

2.2.2. *Analysing Conditions and Actions*

The work on graph analyses in the previous subsection has been conducted in association with two pieces of research, one more conservative but of more general applicability than the other, which seek to establish ways in which one rule may affect another.

The first of these, from [1], is based on a set of operations that characterise the notion of triggering in the Starburst [36] active rule system. The following are representative of the operations over rules in a rule set R , but abstract over Starburst-specific features of the operations.

- $TriggeredBy(Rule) \rightarrow \{Operation\}$: Given a *Rule*, return the set of *Operations* that trigger the rule. This information is in the event description of the rule.
- $Performs(Rule) \rightarrow \{Operation\}$: Given a *Rule*, return the set of *Operations* that the rule may perform. This information is in the condition and the action of the rule.
- $Triggers(Rule) \rightarrow \{Rule\}$: Given a *Rule* r in rule base R , return the set of *Rules* that may be triggered by r . $Triggers(r) = \{r' \in R \mid Performs(r) \cap TriggeredBy(r') \neq \emptyset\}$.

Note that while the sort of simple syntactic analysis described above supports a plausible, if conservative, notion of when triggering may occur, it is not possible to build an activation graph based upon similar techniques. This is because anything but the most conservative definitions of the operations $ActivatedBy(Rule) \rightarrow \{Operation\}$ and $DeactivatedBy(Rule) \rightarrow \{Operation\}$ requires detailed examination of the conditions and actions of the rules involved. Such analyses are necessarily specific to particular condition and action languages.

The research reported in [4] shows how the operations $ActivatedBy$ and $DeactivatedBy$ can be supported, where conditions and actions are described using relational algebra. In this context, a rule condition consisting of a relational algebra condition is considered to be *true* whenever it returns a non-empty result. The results of an action are then described in terms of a relational algebra expression that characterises the changes made to the database. An algorithm is presented that propagates the description of the action of one rule through the condition of another.

2.2.3. *Applying Existing Results on Rule Analysis*

The results summarised in Sections 2.2.1 and 2.2.2 represent the state of the art in static rule analysis. However, there are a number of limitations that reduce the ease with which they can be exploited and the range of systems within which they can be applied.

- The execution models considered are generally both conservative and largely implicit; it is not made clear what effect features such as transition granularities and coupling modes, which vary from system to system, have on the results of rule analyses.
- The rule systems that provide the context for the analysis work have little or no role for explicit event detection: the notion of net effect, as supported in Starburst, minimises the role of events in ECA rule systems, and the analyses of [4] assume that condition-action rules are being used.
- The construction of activation graphs in [2] depends upon the ability to guarantee that a rule will deactivate its own condition. This is a much stronger requirement than elsewhere in the analyses, where conservative results are sufficient, and all that is required is to know if a rule *may* trigger, activate or deactivate another. In certain rule systems in which rules are triggered in response to accumulating changes in the database (e.g. [36, 19]), consideration of a rule automatically renders its condition false. However, this is not the standard semantics for active rules in object-oriented databases (e.g. [14, 9],) and is by no means always a feature

of relational systems (e.g. [31, 23]). This means that in many systems it will not be possible to construct the activation graph of [2] automatically. Where it is not possible to determine if rules deactivate themselves automatically, the static analyses of [2] offer no benefits over those of [1].

- The analyses of [4] are dependent upon the use of a declarative language to express rule actions. While it may be common or appropriate to restrict the condition language of an ECA rule system to be the declarative query language of the underlying database, active rule systems generally provide more extensive and less declarative facilities for describing rule actions.

A number of researchers have sought to build upon the results described in Sections 2.2.1 and 2.2.2. [21] develops the trigger graph based analysis method of [1], applying it to a rule system developed over an object-oriented logical data model, which differs most notably from Starburst in that it employs immediate coupling for condition and action evaluation and tuple level transition granularity. Neither of these features requires significant modification to the method of trigger analysis. [21] profit by the logical formalism used in their rule language to obtain *refined trigger graphs*, based on a stricter analysis of the *Triggers* relation. Method invocations in rule actions are *unified* with event specifications, allowing pruning of the trigger graph where the call and the event specification fail to unify, say because corresponding arguments have incompatible bindings. Furthermore, analysis of condition and action code is employed to identify constraints that apply to method arguments at the point of invocation. These may be compared with constraints imposed by the event or condition specification of the triggered rule allowing pruning of the graph where the constraints are incompatible.

Although this refinement succeeds in providing a stricter analysis than [1] it does so by switching from one specific case (deferred, set granularity rules) to another (immediate, tuple granularity rules). The event and action languages have restricted functionality, and the constraint analysis used to prune the trigger graph depends on the fact that rules are fired immediately to ensure that nothing changes between method invocation and event triggering. Experience using these analysis techniques is reported in [32].

It is reported in [33] how termination analysis can be carried out in active database systems with composite events. The aim of the work of [33] is similar to that of the work reported in this paper, namely to conduct rule analysis in powerful active rule systems, although the scope of this paper is wider.

In summary, there has been a reasonable amount of work on active rule analysis, including a number of seminal papers that have provided a comprehensive foundation for the field. This paper builds on and extends the earlier work, by indicating how it can be adapted for application in a representative, powerful active object-oriented database system.

2.3. Optimization

Previous work on optimization of active rule systems has tended to concentrate on condition-action rules, which (implicitly) monitor primitive structure events and exploit straightforward execution models. Such work has generally focused on algorithms that may need to exploit conventional optimization techniques (e.g. the TREAT algorithm of [24] needs a join optimizer), or that transform programs to exploit intermediate results during rule processing (e.g. [5, 30]).

There has been surprisingly little work on optimization in ECA-rule systems. Perhaps the work that is closest to that described here is that of [12], where it is described how rule analysis in NAOS can be used to detect opportunities for parallel evaluation of rules. [12] generalizes the confluence analysis defined in [1], providing a wider notion of compatibility between rules than commutativity, used as the basis of a parallel execution strategy. This generalization copes with immediate and deferred coupling modes and complex condition and action languages. It does not attempt to deal with event algebras and mixed transition granularities and it relies on the use of phased execution cycles for deferred rule execution to simplify confluence analysis. However, in common with the work presented in this paper, it does consider together rule analysis and optimisation.

The work described in this paper incorporates more powerful event specifications and execution model features into both analysis and optimisation, and looks at multiple query optimisation rather than parallelism as a means of providing better performance.

3. ACTIVE RULES IN ROCK & ROLL

This section introduces the ROCK & ROLL active database system which is used later in the paper as the context for work on rule analysis and optimization. ROCK & ROLL is an object-oriented database system that also supports a general purpose active rule language and execution model. In particular, it allows the use in combination of features more commonly found in isolation in other rule systems, such as deferred and immediate coupling modes, set and tuple transition granularity, simple and composite event specifications and a variety of event consumption policies. ROCK & ROLL can thus be seen as representative of work on active object databases, and as a challenging context in which to seek to perform rule analysis and optimization.

3.1. ROCK & ROLL

ROCK & ROLL is a deductive objective-oriented database system (DOOD) which has been extended to include an active rule language, ROCK & ROLL Active Programming System (RAP) [15]. ROCK & ROLL has 3 major components, an object manager (OM) which implements an object-oriented data model, an imperative object-oriented programming language, ROCK, and a deductive object-oriented programming language, ROLL (see Figure 1). Both languages operate over the same data model and are intercallable with minimal impedance mismatch [7].

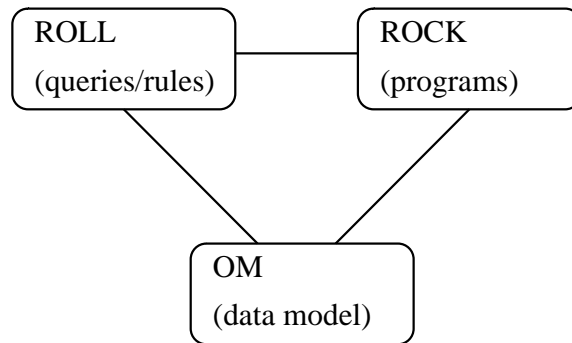


Fig. 1: ROCK & ROLL Architecture

The object manager supports primitive types (strings, numbers and booleans), which are manipulated using a predefined set of built-in operations, and user-defined types. User types may have attributes and/or a construction. Attributes are properties associated with instances of the type whose value type may be primitive or user-defined. Three kinds of construction are supported: sets, lists and aggregates.

The example schema in Figure 2 defines an abstract class `part` with two subclasses, `base_part` and `composite_part`. `part` includes some generic attributes inherited by both subtypes. Instances of `composite_part` have two extra attributes and are constructed as a set of `parts`, allowing a recursive parts hierarchy to be assembled by installing parts as members of a composite.

Built-in operations are provided that allow access or updates to all aspects of an object's structure. For example, the method `get_lifetime` is automatically added to allow access to the assembly `lifetime` attribute of the class `part`; the `is_in` method may be used to test whether a `composite_part` contains a `part` as member. Types may also declare an interface that describes methods defined either in ROCK or in ROLL. A separate class definition provides the implementation of these methods. `base_part` and `composite_part` both implement new methods in ROCK; `composite_part` also provides ROCK methods to compute cost and quality values for the whole assembly. Finally, `composite_part` implements a ROLL method `contains(part)` that can be used

```

type part:
  properties:
    public:
      name : string,
      partid : int,
      lifetime : int,
      quality : real,
      cost : real;
end_type

type base_part:
  specialises: part;
  ROCK:
    new (n:string, l:int,
         q:real, c:real, id:int);
end_type

type composite_part:
  specialises: part;
  properties:
    public:
      assem_cost : real,
      assem_quality: real;
  public {part};
  ROCK:
    update_quality();
    update_cost();
    new (n:string, l:int,
         q:real, c:real, id:int);
  ROLL:
    contains(part);
end_type

```

Fig. 2: Example ROCK & ROLL Type Declarations

to identify which parts are included (directly or indirectly) as components of a `composite_part` in the hierarchy.

ROCK is used to implement functional or procedural methods and program blocks. It provides imperative control structures, operators for creating and deleting objects and built-in operations for accessing and updating object structure and performing input/output.

ROLL is a first order Horn-clause logic language, used to implement deductive methods and queries. In terms of deductive language facilities, ROLL is essentially `datalog \neg` , except that it acts over a richer data model and the clause base is organised as methods on object classes. Examples of methods written in both ROCK and ROLL are given in Figure 3

ROCK code can invoke ROLL using a syntax that resembles a list comprehension (see Figure 4). The goal set to the right of the vertical bar (`|`) contains sub-goals to be solved. The projection to the left of the `|` specifies which values should be retrieved from the solutions and returned to the caller. An empty projection means that `true` or `false` is returned depending on whether there are any solutions. An ANY query returns either a single value or a single tuple of values, corresponding to an arbitrarily selected solution; the choice between a value or a tuple depends upon whether one or several variables are mentioned after the ANY keyword. An ALL query returns a set of values or tuples containing all solutions. If no solution exists then an empty set is returned. ROLL methods may also invoke ROCK methods provided that they are side-effect free functions, a property that is detected by the compiler.

3.2. Active Rule System Knowledge Model

RAP (ROCK & ROLL Active Programming System) is an active rule language developed as an extension to ROCK & ROLL. Activation graphs are not used because of the complexity of the action language and the difficulty of demonstrating in such a context that an action will always falsify a condition. RAP supports Event-Condition-Action rules defined using a WHEN-IF-DO syntax. Designing an active rule system requires specifying languages to be used for event, condition and action specifications and defining the precise details of the execution model employed for selecting and scheduling rules for execution. In RAP, ROLL is the condition language, and ROCK is the action language. To complete the language specification requires the definition of an event specification language (which is described in the remainder of this subsection), and an execution model, which is described in Subsection 3.3.

```

class composite_part
public:
  update_quality()
  begin
    var min_q:real;

    min_q := get_assem_quality()@self

    foreach p in self do
    begin
      var part_q := get_quality()@p;
      if (part_q < min_q) then
      begin
        min_q := part_q;
      end
    end
    end

    put_quality(min_q)@self;
  end

  contains(part)
  begin
    contains(P)@Self :-
      P is_in Self;

    contains(P)@Self :-
      contains(P)@Sub:composite_part,
      Sub is_in Self;
  end
  ...
end

```

Fig. 3: Example Methods. The ROCK Method `update_quality` Computes and Stores the Quality of a `composite_part` as the Quality of Its Lowest Quality Component. The ROLL Method `contains` is the Transitive Closure of the `is_in` Collection Membership Relationship. The Symbol `@` is the Message Sending Operator.

```

begin
  var p:composite_part;
  var children:{part};
  var parents:{composite_part};

  p := [ ANY P | get_partid@P == 2001 ];
  children := [ ALL M | contains(M)@!p ];
  parents := [ ALL CP | get_partid@P == 2002, contains(P)@CP ];
  ...
end

```

Fig. 4: Example Queries. The Type of the Logic Variable `P` in the First Embedded ROLL Query is Inferred to be `composite_part`, since the Result is Used to Assign Variable `p` of Type `composite_part`. The Second Query Retrieves All Parts that are Contained in Part 2001. The `!` Indicates that the Value of ROCK Variable `p` Should be Used as a Binding for the Recipient Argument in the `contains` Goal. The Third Query Retrieves All Composite Parts that Contain the Part with id 2002.

3.2.1. Primitive Events

An event specification describes actions undertaken in the database system or by user programs. Most rules are triggered in response to initiation or completion of operations, either user method invocations or built-in operations that read or update the structure of objects or create or delete instances of a given class. Message event definitions are goals written using a ROLL-like syntax which identifies an operation of interest to the rule program and any bindings or type constraints that must be met by the arguments to the operation. For example, `P is_in CP:composite_part` matches any call to the `is_in` operation which retrieves a member from a set instance of type `composite_part`. The user message event definition `put_assem_quality(99.0)@CP SENT` matches any operation which attempts to update the assembly quality of a composite part with quality 99.0. The `SENT` keyword is used to detect the event before the message is sent rather than after. This allows an alternative, `DO INSTEAD` action to be substituted in place of the `put` operation. Another option is to retrieve the result of an operation or maybe even to return an alternative result. For example, the event definition `new part ==> P` binds `P` to the object returned by the `new` operation.

3.2.2. Event Algebra

An event algebra can be used to identify combinations of events, allowing rules to employ very specific descriptions of the circumstances in which they should be triggered. The algebra provides the following operators:

```
[ALL | EARLIEST | LATEST] event AND event [WITHIN t]
[ALL | EARLIEST | LATEST] event THEN event [WITHIN t]
event OR event
event REPEATED n TIMES [WITHIN t]
event WITHOUT event [WITHIN t]
```

The `AND` and `THEN` operators specify an unordered or ordered conjunction of events within a single ROCK & ROLL program. If the two component events are detected before the program ends (within the relevant number of seconds of each other if a time limit is provided) then the composite event is triggered. The `OR` keyword specifies a disjunctive event. It is triggered whenever one or other of the components is triggered. The `REPEATED` keyword detects repeated occurrences of events that all match the same component specification (possibly limited to occurring within a given time interval). The `WITHOUT` keyword provides a controlled means of detecting the *absence* of an event. The composite event is triggered if the first event occurs and the program run completes (or the given number of seconds elapses) with no subsequent occurrence of the second event.

The keywords `ALL`, `EARLIEST` and `LATEST` specify a consumption policy for conjunctive events [10]. This deals with the situation where one component event is triggered more than once before the other component is triggered. With an `ALL` policy each occurrence of the first event is combined with the second event occurrence to trigger the conjunction. With a policy of `EARLIEST` (or `LATEST`) only the earliest (respectively latest) occurrence of the first event is combined with the occurrence of the second event to trigger for the conjunction.

3.2.3. Unification and Matching

As explained in discussion of the example `put_assem_quality` event above, when a constant value is supplied as an argument in an event specification it only selects events whose actual argument matches the constant supplied in the specification. Repeated mention of variables in event specifications is handled in a similar way. Only events or combinations of events that share the same binding for the repeated variable are selected by the specification. So, for example, the specification `add_member(CP)@CP:composite_part` will be matched whenever a composite part is installed as a member of itself. Any attempt to bind the first occurrence of `CP` will lead to unification of the same value with the second occurrence. If the recipient and argument are not the same object then the composite event will not be detected.

3.3. Active Rule System Execution Model

The event language provides a means of identifying circumstances that should raise an event, possibly leading to firing of a rule. However, there are many different ways of implementing the firing process. The execution model describes how rules are selected and scheduled for firing.

3.3.1. Event Projections and Transition Granularity

As well as specifying the circumstances in which the rule is triggered, the event specification also controls how many times the rule is fired in response to multiple event occurrences and which event parameters should be retrieved from the event for use in condition evaluation or action execution. A complete event specification is written using a syntax similar to a query with an event specification on the right hand side and a projection on the left hand side of the vertical bar. The projection specifies how frequently the rule is fired in response to multiple triggers and which bindings from the event are to be collected. The following rule has a TUPLE transition granularity i.e. it is fired each time a `put_quality` message is sent, once for each base part updated.

```
RULE quality_repair_1
WHEN [ EACH | put_quality(_)@P:base_part]
IF . . .
DO . . .
```

If the keyword `ANY` were used in place of `EACH` then the rule would only be fired once (usually at the end of the program run) even if there were multiple calls to method `put_quality` with multiple bindings for the recipient.

The rule may also employ a binding specification as follows:

```
RULE quality_repair_1
WHEN p <== [ EACH P | put_quality(_)@P:base_part]
IF . . .
DO . . .
```

The rule still has a TUPLE level granularity, and thus is fired every time a call to `put_quality` occurs, the variable `p` being bound to the recipient of the message. `p` may be passed as an input to the query in the `IF` part or may be referred to in the `DO` part of the rule. If an `ANY P` projection were employed then the rule would be fired once with a binding chosen arbitrarily from one of the occurrences of the `put_quality` event.

Another option is to use SET level granularity i.e. have the rule fired once in response to ALL firings. Without any bindings and projection this is just the same as using the `ANY` keyword. However, with a binding specification it differs.

```
RULE quality_repair_1
WHEN ps <== [ ALL P | put_quality(_)@P:base_part]
IF . . .
DO . . .
```

In this case all triggers for the event are collected and installed into a set which is bound to variable `ps` and may be referenced from the condition or action. The rule is run once only at the end of a program run or at a rule assertion point in response to all such firings.

3.3.2. Coupling Modes

Coupling modes determine *when* a rule is fired i.e. the timing of when the condition and action of the rule are executed. In `ROCK & ROLL`, two coupling modes are supported: `IMMEDIATE` and `DEFERRED`. An `IMMEDIATE` condition is evaluated directly after a rule is triggered. Evaluation of a `DEFERRED` condition is queued until either the end of the current program run or a user defined assertion point, whichever arrives first, at which point it is evaluated. An `IMMEDIATE` action

is executed directly after the corresponding condition is evaluated (assuming it is satisfied). A DEFERRED action is queued until the end of the current program or an assert point is reached. When rules with DEFERRED coupling modes are triggered during assert point processing they are added to the queue of deferred conditions or actions for the current assert point.

3.3.3. Priorities and Scheduling

IMMEDIATE conditions and actions are executed in a depth first order. If evaluation of a condition or action triggers another rule with an IMMEDIATE condition or action then the second rule is fired before completing processing of the original rule. It is possible that an event may trigger several rules simultaneously. Rules can be prioritised to ensure that execution follows a determined order. In the absence of any priority ordering simultaneously triggered rules are selected for firing in arbitrary order.

Rules with DEFERRED conditions and actions are queued for processing at an assert point or at program end. The deferred queue is ordered using rule priorities so that higher priority rules are executed first. For rules of the same priority, those which are deferred awaiting condition evaluation are run in preference to those which are deferred awaiting action execution. Where this does not define a complete ordering an arbitrary selection is made by the rule scheduler.

An example set of rules for the parts program is given in Figure 5. The rule `no_cycles` detects any attempt to link the parts hierarchy in a cycle and intercepts the operation. It has higher priority than the quality and cost repair rules since it must run before they do. The latter respond to a change in quality or cost of a component part or a modification of the parts hierarchy by updating the cost or quality of all composite parts that contain the component. This triggers the update rules recursively and the recursion traverses up to the top of the hierarchy until all composite parts have been updated. If a cycle was allowed in the hierarchy recursive triggering would never terminate.

4. RULE ANALYSIS IN RAP

This section describes rule analysis in RAP, presenting results on both termination and confluence.

4.1. Termination

Termination analysis in RAP relies on the result detailed in [1] that a rule program can only exhibit non-terminating rule triggering and firing if there is a cycle in the rule triggering graph. However, this basic result needs to be applied in a context in which event definitions support unification, rules can be triggered by composite as well as primitive events, and rule conditions as well as rule actions can raise events. Consideration of termination analysis in RAP starts by looking at the effect of these features on the construction of a triggering graph, as defined in Section 2.2.1.

4.1.1. Triggering Graphs and Event Parameters

As described in Section 3.2.3, event definitions in RAP can be defined so that they match quite precisely the call being made in an application. For example, `put_name("handle")@P:part` only matches calls to `put_name` that have the constant string "handle" as argument. In addition, there may be type filters that increase the specificity of event definitions. For example, the event specification `put_name("handle")@P:base_part` does not match with `part` objects that are `composite_parts`.

Event parameters and type filters are relevant to the construction of triggering graphs because a rule can only trigger another if the call in the triggering rule may match the specification in the event definition. This gives rise to the notion of the unification of a method call in a rule body with a method specification in an event definition. As method calls in a program and the specification of primitive method events in an event specification contain the same information,

```

RULE no_cycles
  WHEN pair <== [ EACH <P, CP> | insert(P)@CP:composite_part SENT ]
  IF [ | get_part@!pair == P,
        get_composite_part@!pair == CP,
        contains(CP)@P ]
  DO INSTEAD NOTHING

  CONDITION IMMEDIATE ACTION IMMEDIATE
  PRIORITY 5.0
END_RULE

RULE quality_repair_1
  WHEN cpart <== [ EACH CP | insert(_)@CP:composite_part OR
                    remove(_)@CP OR
                    put_assem_quality(_)@CP ]

  IF true
  DO update_quality@cp;

  CONDITION DEFERRED ACTION IMMEDIATE
  PRIORITY 4.0
END_RULE

RULE quality_repair_2
  WHEN apart <== [ EACH P | put_quality(_)@P:part ]
  IF cparts <== [ all C | !apart is_in C:composite_part ]
  DO foreach cp in cparts do
    update_quality@cp;

  CONDITION DEFERRED ACTION IMMEDIATE
  PRIORITY 3.0
END_RULE

```

Fig. 5: Example Rule Base for Parts Database

they are presented in what follows as values of type *Call*. A method call in a program matches with a method event specification in an event definition if both have the same selector, the same number of arguments, the arguments are of compatible types, and, if the values of the arguments are known, then they are the same.

$$\text{unify}(c1 : \text{Call} \times c2 : \text{Call}) \longrightarrow \text{Boolean}$$

$$\text{selector}(c1) = \text{selector}(c2) \wedge \text{arity}(c1) = \text{arity}(c2) \wedge$$

$$(\forall i \in 1 \dots \text{arity}(c1) : \text{unify_arg}(\text{arg}(c1, i), \text{arg}(c2, i)))$$

$$\text{unify_arg}(a1 : \text{Argument} \times a2 : \text{Argument}) \longrightarrow \text{Boolean}$$

$$\text{if } (\text{type}(a1) \leq \text{type}(a2) \vee \text{type}(a2) \leq \text{type}(a1)) \text{ then}$$

$$\quad \text{if } (\text{unbound}(a1) \vee \text{unbound}(a2)) \text{ then true}$$

$$\quad \text{else } (\text{value}(a1) = \text{value}(a2))$$

$$\text{else false}$$

4.1.2. Triggering Graphs and Composite Events

In rule systems where rules monitor only primitive events, an edge in the triggering graph indicates that one rule can generate an event that can in turn trigger another. However, where a rule r has a composite event, it may be that no other single rule in the rule base can trigger r , but that a subset of the rules may together be able to trigger r [33].

This means that in searching for nonterminating rule sets, it may be necessary to consider the cumulative effect of a set of calls from one or more rules on the event of a rule. This notion is captured as follows[†]:

$$\begin{aligned}
 \text{unify_event}(e : \text{Event} \times \{cs : \text{Call}\}) &\longrightarrow \text{Boolean} \\
 \exists c \in cs : \text{unify}(e, c) & \quad \text{if } \text{primitive_event}(e) \\
 \text{unify_event}(e1, cs) \wedge \text{unify_event}(e2, cs) & \quad \text{if } e = (e1 \text{ AND } e2) \\
 \text{unify_event}(e1, cs) \wedge \text{unify_event}(e2, cs) & \quad \text{if } e = (e1 \text{ THEN } e2) \\
 \text{unify_event}(e1, cs) \vee \text{unify_event}(e2, cs) & \quad \text{if } e = (e1 \text{ OR } e2) \\
 \text{unify_event}(e1, cs) & \quad \text{if } e = (e1 \text{ REPEATED } n \text{ TIMES})
 \end{aligned}$$

4.1.3. Triggering During Condition Evaluation

A further feature of RAP that must be taken into account is that condition evaluation may itself lead to rule triggering. RAP allows events to be raised by user method invocations from either ROCK or ROLL, and also in response to invocations of built-in operations implementing attribute fetches (e.g., `get_name@P == "handle"`) or collection membership tests (e.g., `P is_in C:composite_part`). It is by no means uncommon for database systems to allow conditions to trigger events [31, 18, 27]. So, when identifying the set of operations that may be performed by a particular rule, both the condition and action call graphs must be analysed. An obvious, minor generalisation of the *Performs* predicate from Section 2.2.2 copes with this extra complexity.

- $\text{PerformsC}(\text{Rule}) \longrightarrow \{\text{Call}\}$: Given a *Rule*, return the set of *Calls* that the condition of the rule may perform.
- $\text{PerformsA}(\text{Rule}) \longrightarrow \{\text{Call}\}$: Given a *Rule*, return the set of *Calls* that the action of the rule may perform.
- $\text{Performs}(\text{Rule}) \longrightarrow \{\text{Call}\}$:
 $\text{Performs}(\text{Rule}) = \text{PerformsC}(\text{Rule}) \cup \text{PerformsA}(\text{Rule})$
- $\text{Performs}(\{\text{Rule}\}) \longrightarrow \{\text{Call}\}$:
 $\text{Performs}(rs : \{\text{Rule}\}) = \{c | r \in rs, c \in \text{Performs}(r)\}$

Relating event definitions to rule conditions and actions requires access to the calls that appear in an event definition:

- $\text{TriggeredBy}(\text{Rule}) \longrightarrow \{\text{Call}\}$: Given a *Rule*, return the set of *Calls* that are monitored by the event of the rule.

4.1.4. Nontermination Algorithm

The definition of triggering graph given in Section 2.2.1 assumes that the triggering relationship exists between a single triggering rule and a single triggered rule. However, where rules monitor composite events, a rule may become triggered as a result of the activities of a collection of other rules. This means that the notion of a straightforward cycle in the triggering graph no longer fully captures the nature of the potential interactions between rules in a language with composite events.

In languages with composite events, the rules that may participate in nonterminating rule executions are those that belong to groups of rules that may trigger each other indefinitely. In what follows, such groups of rules are identified by first computing the equivalent of a triggering graph in which the edges represent the *may trigger* relationship, and then pruning this graph of the rules that are not *fully triggered*. The complete rule base is referred to as *Rules*.

[†]The fact that AND and THEN are treated in the same way makes the analysis more conservative, but significantly more straightforward, than when separate arrangements are made to account for THEN. A further extension could check for incompatible types or values in event definitions involving shared variables.

$$\begin{aligned}
\text{MayTrigger}(r_1 : \text{Rule}) &\longrightarrow \{\text{Rule}\} \\
&\{r_2 \in \text{Rules} \mid \\
&\quad \exists c_1 \in \text{Performs}(r_1), \exists c_2 \in \text{TriggeredBy}(r_2) : \\
&\quad \text{unify}(c_1, c_2)\}
\end{aligned}$$

Given a rule r , the collection of rules that may be involved in a cycle of infinite rule triggering with r are those that r may trigger directly or indirectly. This is referred to as the set of rules that are *Reachable* from r :

$$\begin{aligned}
\text{Reachable}(r : \text{Rule}) &\longrightarrow \{\text{Rule}\} \\
rs &= \{r\} \\
&\text{repeat until unchanged} \\
&\quad rs = rs \cup \{r_2 \mid r_1 \in rs \wedge r_2 \in \text{MayTrigger}(r_1)\} \\
&\text{return } rs
\end{aligned}$$

The collection of rules that are *Reachable* from r must now be reduced, to account for the fact that some of the rules may have composite events, not all components of which may be triggered by rules *Reachable* from r . This process yields the collection of *potentially* non-terminating rules with which r is involved:

$$\begin{aligned}
\text{NonterminatingGroup}(r : \text{Rule}) &\longrightarrow \{\text{Rule}\} \\
rs &= \text{Reachable}(r) \\
&\text{repeat until unchanged} \\
&\quad rs = rs - \{r_1 \mid r_1 \in rs \wedge \text{not unify_event}(\text{event}(r_1), \text{Performs}(rs))\} \\
&\text{return } rs
\end{aligned}$$

4.1.5. Summary

This section has indicated how to conduct rule termination analysis in RAP, taking into account event parameters, composite events, and the fact that conditions can raise events. The algorithms presented lean heavily on the analysis of event specifications in the context of triggering graphs. Experimental work reported in [32], which performs significantly less sophisticated event unification than that presented above, suggests that termination analysis based on event unification is highly effective in practice.

4.2. Confluence

Confluence analysis in RAP is also based on the method described in [1]. As with termination analysis, changes need to be made to allow for the extra complexities of the RAP knowledge and execution models.

4.2.1. Commutativity, Coupling Modes and Transition Granularity

Confluence analysis in [1] is built over a method for establishing commutativity of pairs of rules in the Starburst active rule language. The Starburst commutativity algorithm considers operations performed by each rule's condition and action, identifying the rule's read-set, the subset of the database schema, the extent of which may be read during condition evaluation or action execution, and its write set, the subset of the database schema the extent of which may be written by the rule's action. It identifies whether there is a write-read clash between the rules, i.e. the write set of one rule overlaps the read set of the other, or a write-write clash, i.e., the write sets of the rules overlap. If any such clash occurs then the order in which the rules execute affects the final outcome. A pair of rules ordered by priority can only have one outcome and there is no clash for them; any rule set that contains only such rules is vacuously confluent. Also, a rule commutes with itself in the sense that successive firings can only occur in successive cycles of rule execution, and so have a determinate order and hence a determinate outcome.

These results depend strongly on the fact that in Starburst conditions are DEFERRED, action execution is IMMEDIATE, and the transition granularity is SET. The coupling modes ensure that firing of a rule condition and execution of the corresponding action is an atomic operation. No rule firings can occur during condition evaluation or action execution. This allows identification of write-read and write-write clashes by comparing rule conditions and actions. If IMMEDIATE condition modes are employed, then firing of one rule may involve executing conditions and actions for immediately triggered rules. As a result, the read and write set of a rule must also include the read and write sets of any condition or action code that can be executed as a result of such immediate triggering.

The SET transition granularity ensures that in any given firing cycle only one firing of a rule occurs, even if the rule is triggered multiple times. If a deferred condition or deferred action rule employs TUPLE granularity then multiple firings for the same rule may execute in different orders during a given rule execution cycle. If the rule's read and write sets overlap then these different firing orders may possibly have different final outcomes. Most rule priority mechanisms normally only apply to triggerings of different rules, i.e., they do not order multiple firings for the same rule. Even if a determinate ordering is adopted, say first come first served, analysing the firing order and deriving consequences for confluence of the rule program may prove intractable, especially in the context of declarative languages. As a result, any rule with TUPLE granularity and DEFERRED condition or action mode must be considered a potential source of non-confluent execution.

Note that this problem does not *normally* arise for IMMEDIATE mode rules with TUPLE granularity, since each trigger for the rule is normally processed immediately, before any other triggers for the same rule can arise. Although the rule may recursively trigger itself, it can only do so in a determinate order. However, there are some special circumstances in which multiple instances of an IMMEDIATE mode rule may be triggered at the same time.

ROLL execution is required to be side-effect free. This ensures that the language has a simple clear semantics, and that every query terminates. It also allows different evaluation and optimization strategies to evaluate goals in different orders. If triggers for a rule occur during ROLL query evaluation, they cannot be processed while ROLL evaluation is still in progress, since this might involve execution of rule actions that have side effects. As a result, triggers are queued up for processing after the query has completed. Since the exact order of evaluation is determined by the strategy adopted by the evaluator and optimizer, this implies that the order in which the triggers are processed is indeterminate. Note that this complication is shared by all active rule systems with declarative condition languages that allow events to be raised by structure accesses or operation invocations (e.g. POSTGRES [31], PFL [27]).

Multiple occurrences of an immediate rule can also be triggered concurrently where the rule's event is a conjunction of the form A THEN B or A AND B. If the consumption mode for the conjunction is ALL, then repeated triggers for one event, say A will be collected and combined with the first occurrence of the other event, event B. So, if event A is triggered twice and then event B occurs there will be two combinations of A THEN B or A AND B resulting in two concurrent triggerings of the rule. These two special cases must also be treated as potential sources of non-confluent execution.

4.2.2. Computing Read and Write Sets for Isolated Rules

In RAP, read sets and write sets can be characterised by the structural components of the schema that are accessed or updated by a rule: class extents, attributes, aggregate components, sets, lists, etc. Each ROCK or ROLL built-in operation may read or write structure associated with particular classes. For example, the method `put_name` on class `part` writes the attribute of type `name` associated with class `part`. The read sets of user methods and rule conditions/actions can be built up recursively from those of the built-in methods they may potentially invoke. This involves traversing the internal representation of ROCK & ROLL programs, identifying the primitive structure changing operations they invoke. The following definitions indicate how structure changes are identified from the calls made by ROCK & ROLL programs.

To define read and write sets more formally, it is necessary to provide some auxiliary definitions:

$UsesC$ identifies the set of structural relations on which evaluation of the condition of a rule depends. $UsesA$ performs the same role for rule actions, and $Uses$ combines the results of $UsesC$ and $UsesA$.

$$UsesC(r : Rule) \longrightarrow \{StructureRelation\}$$

$$\bigcup_{c \in PerformsC(r)} depends(c)$$

$$UsesA(r : Rule) \longrightarrow \{StructureRelation\}$$

$$\bigcup_{c \in PerformsA(r)} depends(c)$$

$$Uses(r : Rule) \longrightarrow \{StructureRelation\}$$

$$UsesC(r) \cup UsesA(r)$$

The function $depends$ identifies elements of the data model that a ROCK & ROLL method or built-in operation reads.

$$depends(Call) \longrightarrow \{StructureRelation\}$$

$\{ \langle attribute, t, t' \rangle \}$	<i>if c is an attribute access</i>	$get.t@X : t'$
$\{ \langle component, t, t' \rangle \}$	<i>if c is a component access</i>	$get.t@X : t'$
$\{ \langle member, t \rangle \}$	<i>if c is a set access</i>	$X \text{ is_in } Y : t$
$\{ \langle index, t \rangle \}$	<i>if c is a list access</i>	$\left\{ \begin{array}{l} X \text{ is_in } Y : t \\ Y : t[I] \end{array} \right.$
$\{ \langle member, t \rangle, \langle extent, t \rangle \}$	<i>if c is a set test</i>	$\left\{ \begin{array}{l} empty(Y : t) \\ upper(Y : t) \\ lower(Y : t) \end{array} \right.$
$\{ \langle index, t \rangle, \langle extent, t \rangle \}$	<i>if c is a list test</i>	$\left\{ \begin{array}{l} empty(Y : t) \\ upper(Y : t) \\ lower(Y : t) \end{array} \right.$
$\{ \langle extent, t \rangle \}$	<i>if c is an iteration over the extent of class t</i>	
$\{ \langle member, t \rangle \}$	<i>if c is an iteration over the members of a collection type t</i>	
\emptyset	<i>otherwise</i>	

$SideEffects$ identifies the set of structural relations that the action of rule r can modify.

$$SideEffects(r : Rule) \longrightarrow \{StructureRelation\}$$

$$\bigcup_{c \in PerformsA(r)} modifies(c)$$

The function $modifies$ identifies elements of the data model that a method or built-in operation reads:

$$modifies(c : Call) \longrightarrow \{StructureRelation\}$$

$\{ \langle attribute, t, t' \rangle \}$	<i>if c is an attribute update</i>	$put.t(Y)@X : t'$
$\{ \langle component, t, t' \rangle \}$	<i>if c is a component update</i>	$put.t(Y)@X : t'$
$\{ \langle member, t \rangle \}$	<i>if c is a set update</i>	$\left\{ \begin{array}{l} Y : t ++ X \\ Y : t -- X \end{array} \right.$
$\{ \langle index, t \rangle \}$	<i>if c is a list update</i>	$\left\{ \begin{array}{l} Y : t ++ X \\ Y : t -- X \\ Y : t[I] := X \\ delete_at(I)@Y : t \end{array} \right.$
$\{ \langle extent, t \rangle \}$	<i>if c is a new operation</i>	$\left\{ \begin{array}{l} new t \\ new t(a_1, \dots, a_N) \end{array} \right.$
$modifies_delete(t)$	<i>if c is a delete operation</i>	$delete X : t$
\emptyset	<i>otherwise</i>	

The definition of *modifies* requires a few further definitions to capture the consequences of the deletion of an object:

$$\begin{aligned}
\text{modifies_delete}(t : \text{Type}) &\longrightarrow \{\text{StructureRelation}\} \\
&\{\langle \text{extent}, t \rangle\} \cup \{\langle \text{attribute}, t, t_i \rangle \mid t_i \text{ is an attribute of } t\} \cup \text{modifies_construction}(t) \\
\\
\text{modifies_construction}(t : \text{Type}) &\longrightarrow \{\text{StructureRelation}\} \\
&\{\langle \text{member}, t \rangle\} && \text{if } t \text{ is a set} \\
&\{\langle \text{index}, t \rangle\} && \text{if } t \text{ is a list} \\
&\{\langle \text{component}, t_i \rangle \mid t_i \text{ is a component of } t\} && \text{if } t \text{ is an aggregate} \\
&\emptyset && \text{otherwise}
\end{aligned}$$

4.2.3. Computing Read and Write Sets for Groups of Rules

Computing the read set of a rule r_k first requires computing the total set of rules that may be triggered during evaluation of the rule's condition or action, either directly or indirectly. This requires computing the transitive closure of *MayTrigger* from Section 4.1.4. Clearly all immediate rule triggerings must be included, since execution of the condition and action of r_k will not complete until all immediate rules have also been fired. It is also necessary to include some deferred rule triggers. If, say, rule r_l has a priority less than or equal to that of r_k , then any deferred rule triggered by r_k with priority greater than or equal to that of r_l will execute before r_l is fired. In assessing the commutativity of r_k and r_l it is necessary to include the read and write sets of these intervening rules in the read set of r_k . So, the algorithm for computing read sets includes both immediately triggered rules and also deferred rules whose priority is greater than or equal to some given value.

The set of triggered rules is computed as two subsets, *AllTriggersC*(r, p) comprising rules triggered from the condition of r and *AllTriggersA*(r, p) comprising rules triggered from the action of r ; in each case, the rules returned must run with a priority at least as great as p^\dagger .

$$\begin{aligned}
\text{AllTriggersC}(r : \text{Rule} \times p : \text{Priority}) &\longrightarrow \{\text{Rule}\} \\
RS &= \{r_t \in \text{TriggersC}(r) \mid \text{ConditionMode}(r_t) = \text{immediate} \vee \text{Priority}(r_t) \geq p\} \\
&\text{repeat until unchanged} \\
RS &= RS \cup \\
&\{r_t \mid r' \in RS, r_t \in \text{TriggersC}(r') \wedge \\
&\quad (\text{ConditionMode}(r_t) = \text{immediate} \vee \text{Priority}(r_t) \geq p)\} \cup \\
&\{r_t \mid r' \in RS, r_t \in \text{TriggersA}(r') \wedge \\
&\quad ((\text{ActionMode}(r') = \text{immediate} \vee \text{Priority}(r') \geq p) \wedge \\
&\quad (\text{ConditionMode}(r_t) = \text{immediate} \vee \text{Priority}(r_t) \geq p))\} \\
&\text{return } RS
\end{aligned}$$

$$\begin{aligned}
\text{AllTriggersA}(r : \text{Rule} \times p : \text{Priority}) &\longrightarrow \{\text{Rule}\} \\
RS &= \{r_t \in \text{TriggersA}(r) \mid \\
&\quad (\text{ConditionMode}(r_t) = \text{immediate} \wedge \text{ActionMode}(r_t) = \text{immediate}) \vee \\
&\quad \text{Priority}(r_t) \geq p\} \\
&\text{repeat until unchanged} \\
RS &= RS \cup \\
&\{r_t \mid r' \in RS, r_t \in \text{TriggersC}(r') \wedge \\
&\quad (\text{ConditionMode}(r_t) = \text{immediate} \vee \text{Priority}(r_t) \geq p)\} \cup \\
&\{r_t \mid r' \in RS, r_t \in \text{TriggersA}(r') \wedge \\
&\quad ((\text{ActionMode}(r') = \text{immediate} \vee \text{Priority}(r') \geq p) \wedge \\
&\quad (\text{ConditionMode}(r_t) = \text{immediate} \vee \text{Priority}(r_t) \geq p))\} \\
&\text{return } RS
\end{aligned}$$

[†]In rule analysis, p is always the priority of r ; the two parameters are kept separate because of analyses conducted for use in rule optimization in Section 5.

An invariant property of the rule sets defined by this algorithm is that their members either have immediate coupling modes or have priority greater than or equal to the lower bound, p . The initial values for the sets are the subsets of the rules triggered by the condition and action, respectively, which satisfy these constraints. The subsequent parts of the definitions ensure that further rules are only added if they are triggered by an existing member of the set and also satisfy the constraints. Note also that the second set intension in the second part of each definition ensures that rules triggered by an action are only included when the action is executed immediately or else the priority of the action exceeds the lower bound. If a rule in the set has a deferred action with priority less than p then the action will be executed after rules with priority greater than or equal to p , so triggers resulting from this action will not interfere with rules with priority p or higher.

The read set $ReadSet(r, p)$ of rule r is computed by combining the data read by r with the data read by rules in $AllTriggers(r, p)$.

$$\begin{aligned} ReadSet(r : Rule \times p : Priority) &\longrightarrow \{StructureRelation\} \\ &\{ u \mid r_t \in AllTriggers(r, p) \wedge u \in UsesC(r_t) \} \cup \\ &\{ u \mid r_t \in AllTriggers(r, p) \wedge u \in UsesA(r_t) \} \cup \\ &Uses(r) \end{aligned}$$

The write set $WriteSet(r, p)$ of rule r is computed in the same way as the read set, except that only the actions of rules can contribute to the write set since rule conditions have no side effects.

$$\begin{aligned} WriteSet(r : Rule \times p : Priority) &\longrightarrow \{StructureRelation\} \\ &\{ s \mid r_t \in AllTriggers(r, p) \wedge s \in SideEffects(r_t) \} \cup \\ &SideEffects(r) \end{aligned}$$

It will also be useful in Section 5.2, during discussion of query optimization, to be able to identify the read set of a rule's condition.

$$\begin{aligned} ReadSetC(r : Rule \times p : Priority) &\longrightarrow \{StructureRelation\} \\ &\{ u \mid r_t \in AllTriggersC(r, p) \wedge u \in UsesC(r_t) \} \cup \\ &\{ u \mid r_t \in AllTriggersC(r, p) \wedge u \in UsesA(r_t) \} \cup \\ &Uses(r) \end{aligned}$$

The condition cannot have a write set, per se, since ROLL query evaluation is side-effect free. However, rules that are triggered during condition evaluation may have side effects (execution of such rules is postponed until the end of condition evaluation). So, the write set of the condition may be non-empty.

$$\begin{aligned} WriteSetC(r : Rule \times p : Priority) &\longrightarrow \{StructureRelation\} \\ &\{ s \mid r_t \in AllTriggersC(r, p) \wedge s \in SideEffects(r_t) \} \end{aligned}$$

4.2.4. Commutativity and Confluence

A definition of commutativity can now be provided in terms of read and write sets for rules of the same priority (rules that have different priorities vacuously commute). If the rules are not ordered by priority then in order to commute, the rules must not update any data in common, and each rule must not produce a side effect on data read by the other rule. This is captured by the condition on the read and write sets.

$$\begin{aligned} commutative(r_i : Rule \times r_j : Rule) &\longrightarrow Boolean \\ (WriteSet(r_i, priority(r_i)) \cap WriteSet(r_j, priority(r_j))) &= \emptyset \wedge \\ ReadSet(r_i, priority(r_i)) \cap WriteSet(r_j, priority(r_j)) &= \emptyset \wedge \\ WriteSet(r_i, priority(r_i)) \cap ReadSet(r_j, priority(r_j)) &= \emptyset \end{aligned}$$

As explained in Section 4.2.1 above, a rule r that can have multiple concurrent triggers, either because it employs a conjunction event with consumption mode ALL or because it combines a DEFERRED condition or action mode with TUPLE granularity, *fails* to commute with itself unless $WriteSet(r, priority(r)) = \emptyset$. This is catered for in the above definition of commutative. Any other rule is self-commutative. This is a conservative criterion to adopt. For example, a TUPLE granularity rule might have an action that always updates an attribute to a constant value. Whatever the order in which the triggers are processed, the net result is always to update the attribute to the same value. A more sophisticated analysis would detect such cases.

Given this definition of commutativity, it is possible to define a confluent rule set as a set of rules that are either pairwise commutative or ordered by priority, and which are also self-commutative.

$$\begin{aligned} &confluent(R : \{Rule\} \rightarrow Boolean \\ &(\forall r_i, r_j \in R : \\ &\quad (priority(r_i) <> priority(r_j) \vee \\ &\quad commutative(r_i, r_j))) \end{aligned}$$

4.2.5. Summary

This section has indicated how rule confluence analysis can be conducted in a language such as RAP, making explicit the steps that have to be taken to account for features such as alternative coupling modes and different transition granularities.

5. ACTIVE RULE OPTIMISATION IN ROCK & ROLL

This section indicates how the query optimizer used in the underlying ROCK & ROLL system [16] can be adapted and extended for use with active rules. It concentrates on optimization of rule conditions, not because there are no opportunities for event or action optimization but because little attention has been paid to condition optimization in ECA rules, and it is likely to be a major performance bottleneck.

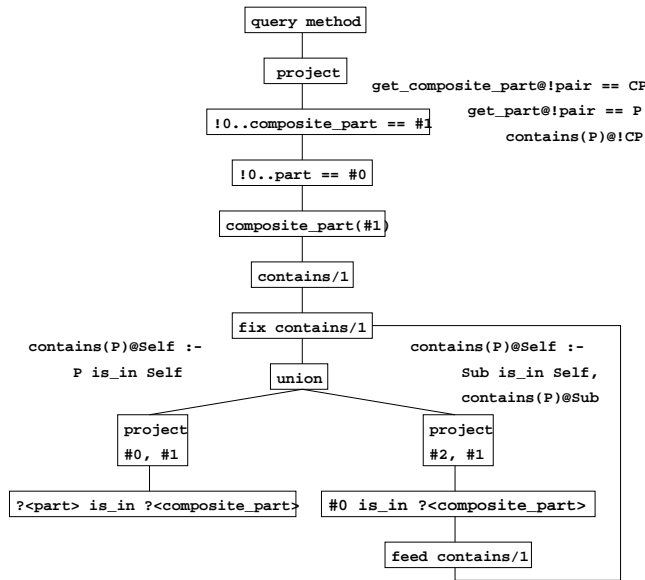
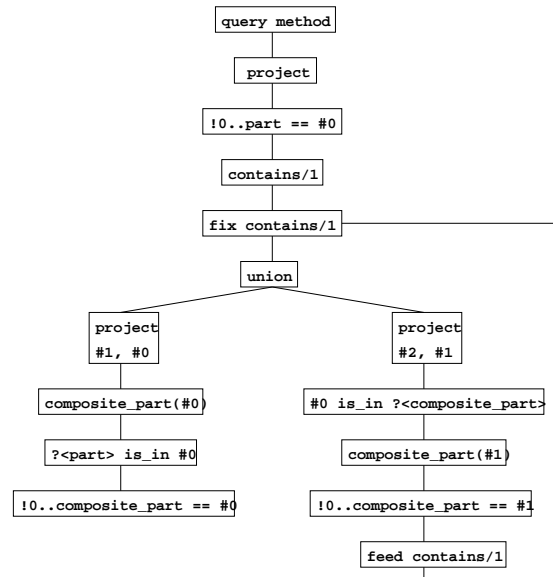
5.1. Optimising Single Rules

The ROLL compiler [16] includes a logical optimizer that generates a Processing Tree [22], a tree built from nodes representing joins, selections and projections that is used as a query plan. Processing Trees have also been supplemented with fixed point nodes which control computation of solutions to recursive methods. The optimizer works by rewriting the tree to a logically equivalent plan with better performance characteristics.

Figure 6 shows the processing tree resulting from compilation of the condition of rule `no_cycles` from Figure 5. The graph contains three sections, on the left, right and top. Each section has been labelled with the source lines from which it arises to make it easier to understand how the query plan relates to the original query.

The processing tree can be regarded as a dataflow graph. Nodes receive a stream of tuples from below, each element of which represents a partial solution to the query, the tuple entries corresponding to bindings for the variables in the method or query goal. The left hand section corresponds to the base clause in the definition of the method `contains`. The `is_in` constraint node generates pairs of `composite_part` and `part` instances where the `part` is a member of the `composite_part`. The `project` node passes on these pairs via the union node to a *fixed point* node which collects solutions for the `contains` method.

The right hand section starts with a *fixed feed* node, which is fed pairs of solutions to the `contains` method by the fixed point node. The `is_in` constraint node selects the `composite_part` from each solution and finds any composite parts of which it is a member, appending each such composite to the input tuple and outputting a triple to the project node. The project node picks out the nested `part` and the outermost `composite_part`, projecting them out as a solution from the recursive clause. Solutions from the left and right subtrees are merged by the union node to produce the full solution set for the `contains` method.

Fig. 6: Processing Tree for Condition of `no_cycles` RuleFig. 7: `no_cycles` Processing Tree after Optimization

The top section contains aggregation and type constraints that filter out redundant solutions. The first node rejects solutions where the contained `part` is not also a composite. The next two nodes test the first and second entries in each solution tuple, ensuring that they are equal to the respective components of the aggregation `pair` retrieved from the event specification. This unoptimized plan is highly inefficient since it generates all possible solutions to the `contains` goal only to reject most of them.

Optimization involves propagation of constraints down the tree from one section to the next. This can include propagation of binding constraints, type constraints and structural constraints. Figure 7 shows the `no_cycles` tree after optimization. The key actions of the optimizer have involved pushing constraints resulting from event parameters down into the processing tree. The type and aggregation constraints on argument 1 of the `contains` method have been identified as

invariant through the original call and recursive calls to `contains`. So, the optimizer propagates these constraints down into the constraint section of the left and right hand tree sections. The aggregation constraint on argument 0 is only valid for the initial call to `contains`. So, it cannot be propagated down into the lower tree sections.

5.1.1. Summary

The presence or absence of input parameters often has a significant effect on the amount of work that has to be done to compute the truth or otherwise of a condition, and it is important for any active rule system optimizer to have available some means of exploiting parameter information. This section has shown how the RAP compiler can use the existing functionality of the ROLL optimizer to achieve effective optimization of single rule conditions by propagating constraints from event parameters into the query execution plan.

5.2. Optimising Multiple Rules

This section shows how results from rule analysis can be applied to identify opportunities for multiple query optimization (MQO) in active database systems. The use of MQO is motivated by the observation that active rule programs often contain rules whose conditions contain the same or very similar goals. Evaluation of these conditions may involve repeated computation of essentially the same data. Multiple query optimization can only be applied to the conditions of a set of rules if:

1. The rules are triggered at the same time.
2. The order in which the rules are chosen for processing has no effect on the results of evaluating their conditions.

The first of these conditions is addressed in what follows on the assumption that multiple query optimization should be applied at rule compilation time, although in principle opportunities could be detected and exploited at rule execution time on the basis of the rules that happen to be triggered simultaneously. The detection and carrying out of multiple query optimization is quite expensive, and it is felt that net benefits are most likely to accrue when the optimized rules are executed multiple times after optimization. Clearly there is scope for future investigation of the trade-offs in this area.

The second of these conditions involves examining properties of the rules in the rule base. While it would be possible to identify global properties of the rulebase that would allow the conditions of any set of rules from the rulebase to be considered together for MQO, such conditions would be sufficiently stringent as to make them unlikely to be met by real rule bases. In practice, MQO is only likely to be beneficial with small groups of rules that are known to be triggered at the same time. This section thus uses the first condition above to identify sets of simultaneously triggered rules to which the second condition can be applied.

5.2.1. Rule Analysis for Multiple Query Optimization

A set of simultaneously triggered rules is a candidate for multiple query optimization if firing any of its member rules cannot affect the result of evaluating the other rules' condition. Where two rules are ordered by priority there is only one order in which they can execute, they are candidates for multiple query optimization if the higher priority rule's write set does not intersect with the read set of the lower priority rule's condition. If they are not ordered, then neither of the write sets of the rules may overlap with the read set of the other rule's condition. This notion is referred to here as *ca-independence*, and is defined as follows for a set of rules:

$$\begin{aligned}
&ca - independent(RS : \{Rule\}) \longrightarrow Boolean \\
&(\forall r_i, r_j \in RS : \\
&\quad (priority(r_i) > priority(r_j) \wedge \\
&\quad\quad ReadSetC(r_j, priority(r_j)) \cap WriteSet(r_i, priority(r_j)) = \emptyset) \vee \\
&\quad (priority(r_j) > priority(r_i) \wedge \\
&\quad\quad WriteSet(r_j, priority(r_j)) \cap ReadSetC(r_i, priority(r_i)) = \emptyset) \vee \\
&\quad (WriteSet(r_i, priority(r_i)) \cap ReadSetC(r_j, priority(r_j)) = \emptyset \wedge \\
&\quad\quad WriteSet(r_j, priority(r_j)) \cap ReadSetC(r_i, priority(r_i)) = \emptyset))
\end{aligned}$$

5.2.2. Identifying Candidate Rule Sets

The challenge for a rule optimizer is to identify collections of rules where it is both *valid* and *beneficial* to exploit multiple query optimization. Probably the most generally effective context in which to apply MQO is when several rules from a rulebase R with an immediate condition coupling mode are monitoring the same call.

$$\begin{aligned}
&optimizable(c : Call \times R : \{Rule\}) \rightarrow \{Rule\} \\
&\quad candidates = \{r_i \in R \mid \\
&\quad\quad ConditionMode(r_i) = immediate \wedge \\
&\quad\quad (\exists c_i \in TriggeredBy(r_i) : unify(c_i, c))\} \\
&\quad if *ca - independent*(candidates) then candidates else \emptyset
\end{aligned}$$

The above algorithm returns a set of rules for which it is possible to carry out multiple query optimization without the optimization process affecting the outcome of rule processing. The assessment as to whether or not it is beneficial to carry out multiple query optimization needs the judgement of a cost model.

For rules with an event-condition coupling mode of deferred, it may be possible to benefit from multiple query optimization across rules that are monitoring different events. This is because when rules are scheduled for condition evaluation at the end of a transaction or at a rule assertion point, these rules will have been triggered by a range of events that have taken place over a period of time, either as a result of operations performed by user programs or as a result of rule processing. This provides a larger search space for opportunities in which to perform multiple query optimization, but also requires that groups of less directly related rules are *ca-independent*.

For example, the following algorithm identifies sets of rules from the rulebase R that can be considered together for MQO:

$$\begin{aligned}
&optimizable(R : \{Rule\}) \longrightarrow \{\{Rule\}\} \\
&\quad \{drs \mid \\
&\quad\quad drs \subseteq \{r_d \in R \mid ConditionMode(r_d) = deferred\} \wedge \\
&\quad\quad ca - independent(drs) \wedge \\
&\quad\quad (\nexists r \in R - drs \mid \\
&\quad\quad\quad Priority(r) \geq \min\{Priority(r_d) \mid r_d \in drs\} \wedge \\
&\quad\quad\quad Priority(r) \leq \max\{Priority(r_d) \mid r_d \in drs\})\}
\end{aligned}$$

Given a rule base, this algorithm identifies sets of deferred rules that are guaranteed not to be interleaved at runtime with any other rules. The algorithm has the following components:

1. The set variable drs is bound in turn to all sets of deferred rules in the rulebase R .
2. The sets bound to drs are then filtered, so that it is known that:
 - (a) The rules in drs are *ca-independent*.
 - (b) No deferred rules triggered at a rule assertion point with the rules in drs can be interleaved with the rules in drs .

Making use of these results involves extensions to both the compilation and the runtime support modules of an active rule system. The compiler must be extended to search for optimization

opportunities using algorithms such as those given above, and the benefits likely to be derived by exploiting multiple query optimization assessed using the cost model of the query optimizer. The runtime rule scheduler must ensure that condition evaluation for rules that have been optimized together makes use of the optimised, shared query plans. The following subsection presents an example of how multiple query optimisation can be applied to the conditions of example rules from the parts databases.

5.2.3. Example Application of Multi-Query Optimization

As an example of the potential for such a technique, Figure 8 lists two rules that ensure that component parts never have a longer lifetime than the composite parts that contain them. Both rules are triggered by a common event, the creation of a link in the parts hierarchy which is effected by an insert into a composite part. Rule `min_life_1` is run with highest priority. It ensures that the inserted part has a lifetime no greater than parts above it in the hierarchy. Rule `min_life_2` lowers the lifetime of any children whose lifetime exceeds that of the inserted part. Note this has to be run after `min_life_1`. If it were done first then the original lifetime of the inserted part would be used to update the children and this might exceed the value installed by rule `min_life_1`.

```

RULE min_life_1
  WHEN new_part <== [ each P | insert(P)@CP:composite_part]
  IF parents <== [ ALL CP | contains (!new_part)@CP ]
  DO
  begin
    var maxlife := get_lifetime@new_part;
    foreach p in parents do
      if (maxlife > get_lifetime@p) then maxlife := get_lifetime@p
      put_lifetime(maxlife)@new_part;
    end
  CONDITION IMMEDIATE ACTION IMMEDIATE
  PRIORITY 2.0
END_RULE

RULE min_life_2
  WHEN new_part <== [ each P | insert(P)@CP:composite_part]
  IF children <== [ ALL P | contains (P)@!new_part ]
  DO
  begin
    var minlife := get_lifetime@new_part;
    foreach p in children do
      if (minlife < get_lifetime@p) then put_lifetime(minlife)@p
  CONDITION IMMEDIATE ACTION IMMEDIATE
  PRIORITY 1.0
  end
END_RULE

```

Fig. 8: Rules Used to Illustrate Multiple Query Optimization

These rules are good candidates for MQO. They can be triggered concurrently by a common event, an insertion of a part in the hierarchy, and both conditions share a common goal, a call to the recursive method `contains`. In addition, neither condition is affected by the operations performed in the action (the actions do not modify the parts hierarchy).

As the rules are ordered by priority they do not make the rule set non-confluent. However, they fail to commute since they have overlapping write sets (the attribute `lifetime` of class `part`).

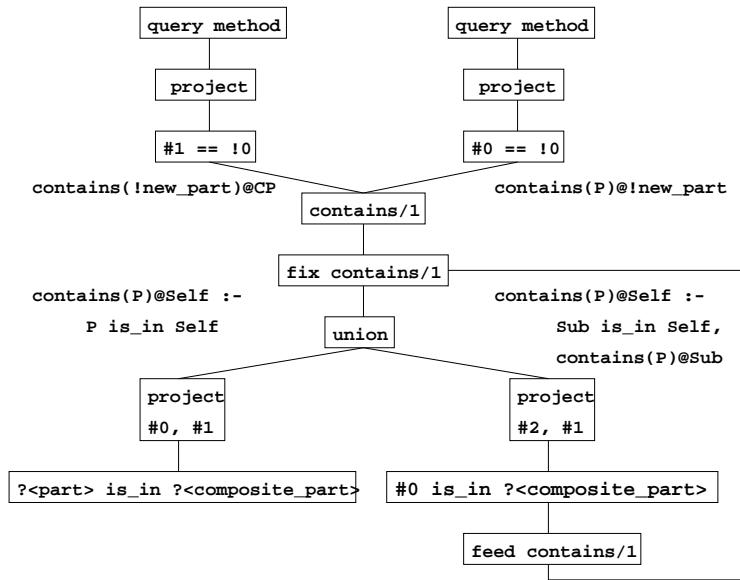


Fig. 9: Processing Tree for Multi-Query

However, they *are* ca-independent since their respective read sets (in both cases the extent of type `part` and the membership of type `composite_part`) do not intersect the other rule's write set.

Each rule is parameterised by a common binding which appears in the condition of the rule. The merged condition for the rules has to be compiled so that it uses this common binding during evaluation of the shared goals. Each time a trigger for rule `min_life_1` occurs there is also a corresponding trigger for rule `min_life_2`. When the `min_life_2` trigger is scheduled by the rule execution component the `min_life_1` trigger can be looked up and the relevant subset of the results from the merged condition cached with the trigger record. When the `min_life_2` trigger is scheduled the cached results can be used avoiding the need to execute the condition.

Compiling the merged condition requires splitting the body of the merged queries into common goals and rule-specific goals. In the example, the common goal is `contains(P)@CP` and the rule-specific goals are `P == !new_part` for rule `min_life_1` and `CP == !new_part` for rule `min_life_2`. A processing tree is constructed with two tree sections at the top, one for each rule condition (see Figure 9). This processing tree is optimized in the same way as a tree with one top tree section (the optimization algorithm employed works correctly irrespective of the number of root nodes in the tree). When the merged query is executed each of the two root nodes of the tree collects solutions relevant to the associated rule condition.

5.2.4. Summary

This section has indicated how results from rule analysis can be of benefit when performing rule optimization. It also demonstrates how to exploit MQO in ECA-rule systems using information obtained from rule analysis along with algorithms that identify collections of rules that may benefit from MQO.

What results from the MQO process described above is collections of merged rule conditions, represented as processing trees. At runtime, the scheduler must detect rules whose conditions have been merged, and evaluate the shared part of the processing tree only once in each triggering of the set of co-optimized rules.

6. CONCLUSIONS

This paper has presented an approach to analysis and optimisation of ECA rules in the ROCK & ROLL deductive object-oriented database system, which is representative of active OODBs with

rich event specification languages and powerful execution models. It has been shown how earlier results on analysis can be extended for use with more powerful rule systems, and that when this work on analysis has been carried out, not only can users be informed of the anticipated behaviour of the rule base, but also the optimiser can use the results of the analyses to plan efficient evaluation strategies.

In rule analysis, the paper has:

1. Presented an approach to termination analysis that exploits event bindings and composite event operators to build conservative rule triggering graphs.
2. Presented an approach to confluence analysis that accounts for the impact of different coupling modes and transition granularities on rule confluence.
3. Demonstrated how dependencies between rule conditions and actions can be specified precisely in the context of a rich object-oriented data model and a powerful action language.

In rule optimisation, the paper has:

1. Indicated how event information can be used by condition optimisers to restrict searches during condition evaluation.
2. Shown how the results of rule analysis can be exploited by the optimiser to identify opportunities for applying multiple query optimisation to rule conditions. The contexts in which multiple query optimisation can be applied depend on the coupling modes of the rules involved, and algorithms have been presented that take account of these features when optimising rule bases.

Acknowledgements — This work is supported by grant GR/H43847 from the UK Engineering and Physical Sciences Research Council, whose support we are pleased to acknowledge. We are grateful to Alvaro Fernandes for his careful reading of the paper and the helpful comments that resulted from this. Our work on active databases also benefitted from input from the members of the ACT-NET European Active Rules Network.

REFERENCES

- [1] A. Aiken, J.M. Hellerstein, and J. Widom. Static analysis techniques for predicting the behaviour of active database rules. *ACM TODS*, **20**(1):3–41 (1995).
- [2] E. Baralis, S. Ceri, and S. Paraboschi. Improved rule analysis by means of triggering and activation graphs. In T. Sellis, editor, *Proc. 2nd Int. Wshp. on Rules In Database Systems (RIDS)*, Athens, pp. 165–181, Springer-Verlag (1995).
- [3] E. Baralis, S. Ceri, and J. Widom. Better termination analysis for active databases. In N.W. Paton and M.H. Williams, editors, *Proc. 1st Int. Workshop on Rules In Database Systems*, Edinburgh, pp. 163–179, Springer-Verlag (1994).
- [4] E. Baralis and J. Widom. An algebraic approach to rule analysis in expert database systems. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20th VLDB*, Santiago, pp. 475–486, Morgan-Kaufmann (1994).
- [5] E. Baralis and J. Widom. Using delta relations to optimize condition evaluation in active databases. In T. Sellis, editor, *Proc. 2nd Int. Wshp. on Rules In Database Systems (RIDS)*, Athens, pp. 292–308, Springer-Verlag (1995).
- [6] M.L. Barja, A.A.A. Fernandes, N.W. Paton, M.H. Williams, A. Dinn, and A.I. Abdelmoty. Design and implementation of ROCK & ROLL: a deductive object-oriented database system. *Information Systems*, **20**(3):185–211 (1995).
- [7] M.L. Barja, N.W. Paton, A.A.A. Fernandes, M.H. Williams, and A. Dinn. An effective deductive object-oriented database through language integration. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20th Int. Conf. on Very Large Data Bases (VLDB)*, Santiago, pp. 463–474, Morgan-Kaufmann (1994).
- [8] E. Benazet, H. Guehl, and M. Bouzeghoub. VITAL: A visual tool for analysis of rule behaviour in active databases. In T. Sellis, editor, *Proc. 2nd Int. Wshp. on Rules in Database Systems*, Athens, pp. 182–196, Springer-Verlag (1995).
- [9] S. Chakravarthy, E. Anwar, L. Maugis, and D. Mishra. Design of Sentinel: an object-oriented DBMS with event-based rules. *Information and Software Technology*, **36**(9):555–568 (1994).
- [10] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: semantics, contexts and detection. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20th Int. Conf. on Very Large Data Bases*, Santiago, pp. 606–617, Morgan-Kaufmann (1994).

- [11] S. Chakravarthy, Z. Tamizuddin, and J. Zhou. A visualisation and explanation tool for debugging eca rules in active databases. In T. Sellis, editor, *Proc. 2nd Int. Wshp. on Rules in Database Systems*, Athens, pp. 197–212, Springer-Verlag (1995).
- [12] C. Collet and J. Manchado. Optimization of active rules with parallelism. In M. Berndtsson and J. Hansson, editors, *Proc. Active and Real Time Database Systems (ARTDB)*, Skovde, pp. 82–103, Springer-Verlag (1995).
- [13] O. Diaz, A. Jaime, and N.W. Paton. DEAR: a DEbugger for Active Rules in an object-oriented context. In N.W. Paton and M.H. Williams, editors, *Proc. 1st Int. Wshp on Rules In Database Systems*, Edinburgh, pp. 180–193, Springer-Verlag (1994).
- [14] O. Diaz, N. Paton, and P.M.D. Gray. Rule management in object oriented databases: a uniform approach. In G.M. Lohman, A. Sernadas, and R. Camps, editors, *17th Intl. Conf. on Very Large Data Bases, Barcelona*, Barcelona, pp. 317–326, Morgan Kaufmann (1991).
- [15] A. Dinn, N.W. Paton, M.H. Williams, and A.A.A. Fernandes. An active rule language for ROCK & ROLL. In R. Morrison and J. Kennedy, editors, *Proc. 14th British National Conference on Databases*, Edinburgh, pp. 36–55, Springer-Verlag (1996).
- [16] A. Dinn, N.W. Paton, M.H. Williams, A.A.A. Fernandes, and M.L. Barja. The implementation of a deductive query language over an object-oriented database. In T.W. Ling, A.O. Mendelzon, and L. Vieille, editors, *Proc. 4th Intl. Conf. on Deductive Object-Oriented Databases*, Singapore, pp. 143–160, Springer-Verlag (1995).
- [17] K.R. Dittrich, S. Gatzui, and A. Geppert. The active database management system manifesto: a rulebase of ADBMS features. In T. Sellis, editor, *Rules In Database Systems: Proc. of the 2nd Int. Workshop*, Athens, pp. 3–17, Springer-Verlag (1995).
- [18] N.H. Gehani and H.V. Jagadish. ODE as an active database: constraints and triggers. In R. Camps G.M. Lohman, A. Sernadas, editor, *17th Intl. Conf. on Very Large Data Bases, Barcelona*, Barcelona, pp. 327–336, Morgan Kaufmann (1991).
- [19] E.N. Hanson. Rule condition testing and action execution in ariel. In *Proc. SIGMOD*, San Diego, pp. 49–58, ACM (1992).
- [20] A.P. Karadimce and S.D. Urban. Conditional term rewriting as a basis for analysis of active database rules. In J. Widom and S. Chakravarthy, editors, *Proc. IEEE RIDE-ADS Workshop on Active Database Systems*, Houston, pp. 156–162, IEEE Press (1994).
- [21] A.P. Karadimce and S.D. Urban. Refined triggering graphs: a logic-based approach to termination analysis in an active object-oriented database. In J. Widom and S. Chakravarthy, editors, *Proc. Data Engineering '96*, New Orleans, pp. 384–391, IEEE Press (1996).
- [22] W. Kim. A model of queries for object-oriented databases. In *Proceedings of 15th International Conference on Very Large Databases*, Amsterdam, pp. 423–432, Morgan-Kaufmann (1989).
- [23] G. Koch and K. Loney. *ORACLE: The Complete Reference (3rd Edition)*. Osborne McGraw-Hill (1995).
- [24] D.P. Miranker. TREAT: A better match algorithm for AI production systems. In *Proc. AAAI*, Seattle, pp. 42–47 (1987).
- [25] N.W. Paton, editor. *Active Rules In Databases Systems*. Springer-Verlag (1999).
- [26] N.W. Paton and O. Diaz. Active database systems. *ACM Computing Surveys*, to be published (1999).
- [27] S. Reddi, A. Poulouvassilis, and C. Small. Extending a functional DBPL with ECA-rules. In T. Sellis, editor, *Proc. 2nd Int. Wshp. on Rules in Database Systems*, Athens, pp. 101–115, Springer-Verlag (1995).
- [28] A. Rosenthal and U.S. Chakravarthy. Anatomy of a multiple query optimiser. In *Proc. 14th VLDB.*, Los Angeles, pp. 230–239 (1988).
- [29] E. Simon and A. Kotz-Dittrich. Promises and realities of active database systems. In U. Dayal, P. Gray, and S. Nishio, editors, *Proc. 21st Int. Conf. on Very Large Data Bases*, Zürich, pp. 642–653, Morgan-Kaufmann (1995).
- [30] M. Skold and T. Risch. Using partial differencing for efficient monitoring of deferred complex rule conditions. In S. Su, editor, *Proc. IEEE Data Engineering*, Taipei, pp. 392–401, IEEE Press (1995).
- [31] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in database systems. In *Proc. ACM SIGMOD*, Atlantic City, pp. 281–290 (1990).
- [32] M.K. Tschudi, S.D. Urban, S.W. Dietrich, and A.P. Kardimce. An implementation and evaluation of the refined triggering graph method for active rule termination. In A. Geppert and M. Berndtsson, editors, *Proc. 3rd. Int. Wshp. on Rules In Database Systems*, Skovde, pp. 133–148, Springer-Verlag (1997).
- [33] A. Vaduva, S. Gatzui, and K.R. Dittrich. Investigating termination in active database systems with expressive rule languages. In A. Geppert and M. Berndtsson, editors, *Proc. 3rd. Int. Wshp. on Rules In Database Systems*, Skovde, pp. 149–164, Springer-Verlag (1997).
- [34] L. van der Voort and A. Siebes. Enforcing confluence of rule execution. In N.W. Paton and M.H. Williams, editors, *Proc. 1st Int. Workshop on Rules In Database Systems*, Edinburgh, pp. 194–207, Springer-Verlag (1994).
- [35] J. Widom and S. Ceri. *Active Database Systems*. Morgan Kaufmann (1996).

- [36] J. Widom and S.J. Finkelstein. Set-oriented production rules in relational database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Atlantic City, pp. 259–270 (1990).
- [37] Y. Zhou and M. Hsu. A theory for rule triggering systems. In F. Bancilhon and et al., editors, *Proc. Extending Database Technology (EDBT)*, Venice, pp. 407–421, Springer-Verlag (1990).