An Active Rule Language for ROCK & ROLL

Andrew Dinn, Norman W. Paton†, M. Howard Williams and Alvaro A.A. Fernandes††

Department of Computing and Electrical Engineering,
Heriot-Watt University, Riccarton, Edinburgh, UK
email: <andrew,howard>@cee.hw.ac.uk
Department of Computer Science†,
University of Manchester, Oxford Road, Manchester, UK
email: norm@cs.man.ac.uk
Department of Mathematical Sciences††,
Goldsmiths College, New Cross, London SE14 6NW
email: a.fernandes@gold.ac.uk

Abstract. This paper presents an active rule language for the ROCK & ROLL deductive object-oriented database system. A characteristic feature of ROCK & ROLL is that it blends imperative and deductive programming styles so that both can be used together in support of passive database applications. The aim in developing an active extension is to allow declarative expression of aspects of active behaviour wherever possible, without imposing prohibitive restrictions on the power of the resulting system. The proposal which results is more powerful than most earlier declarative active rule systems, in both its language and execution model, without resorting to the wholly procedural approach supported by most proposals for active object-oriented databases. The paper indicates where retaining declarative features yields greatest benefits, but also where difficulties are encountered which lead to compromises.

1 Introduction

The ROCK & ROLL deductive-object-oriented database (DOOD) system [4, 3] supports an expressive object-oriented data model, a deductive language in which to write queries and rules, and an imperative database programming language. As such, it provides a comprehensive collection of passive facilities for defining database functionality. However, in the same way as the imperative and deductive mechanisms in ROCK & ROLL are complementary, an active extension can itself be seen as adding to the range of facilities available to a programmer. For example, there is no built-in mechanism for integrity checking in ROCK & ROLL, and while certain checks can be implemented using methods in the context of strict encapsulation, it is sometimes the case that the checking of integrity should not be done immediately an update is executed, but rather later, when any temporary inconsistencies should have been overcome. Essentially, active capabilities in a system such as ROCK & ROLL allow non-intrusive monitoring of activities in the database, with a view to enforcing integrity, maintaining derived information, keeping users informed of activities or refreshing displays (for an overview of active databases, see [25]).

Research into active databases leading to the development of complete systems has generally fallen into one of two camps:

- Active relational databases [24, 26], which are based on the query language of the underlying database system, and thus support declarative expression of conditions and straightforward facilities for performing updates. Such systems generally have simple event specification languages and execution models.
- Active object-oriented databases [12, 17, 11, 6], which are based on the programming language of the underlying database system, and thus support procedural mechanisms for the expression of conditions and for performing updates. Such systems generally have sophisticated event specification languages and execution models.

However, the trend that is emerging in standards, for example SQL-3 and ODMG [8], is towards systems which support both declarative query facilities and powerful programming capabilities. This raises the question as to how procedural and declarative mechanisms can be integrated effectively in active systems, without the introduction of arbitrary restrictions and inconsistent rule structures as are found in early commercial implementations of active facilities [19]. The experience reported in this paper for ROCK & ROLL indicates that while blending declarative and procedural mechanisms is not without its challenges, coherent and powerful rule systems can be developed.

This paper is structured as follows: section 2 outlines recent research on active database systems; section 3 summarises the passive capabilities of ROCK & ROLL; section 4 presents the language extensions of the active ROCK & ROLL system; section 5 describes the corresponding execution model; section 6 gives an example of an active application in ROCK & ROLL; section 7 outlines some distinctive features of the active extension to ROCK & ROLL; and section 8 presents some conclusions.

2 Related Work

2.1 Active Database Systems

Limited space prevents a comprehensive description of the characteristics of active database systems; for a classification and brief survey, see [21]. The distinguishing feature of an active database system is that it is able to respond automatically to situations that arise inside or outside the database itself. The active behaviour of a database is generally described using rules, which most commonly have three components, an *event*, a *condition* and an *action*. A rule with such components is known as an event-condition-action rule, or *ECA-rule*. Such a rule lies dormant until an occurrence of the event that it is monitoring, when the rule is said to be *triggered*. The condition of a triggered rule is subsequently evaluated, and if true, then the action of the rule is scheduled for execution.

The structural and behavioural characteristics of active database systems can be classified according to a number of dimensions, as outlined informally in [21]. For the purposes of this paper we classify the aspects to be described into two areas: *knowledge model* and *execution model*.

The knowledge model represents the syntactic view of active rules as seen by the rule programmer. This has three main facets:

Event language: a notation in which the situations that trigger a rule can be specified. Events are classified as *primitive*, in which case no components are distinguished (e.g. the *deletion* of an object), or *composite*, in which case a number of primitive occurrences are linked using the operations of an event algebra (e.g. the *salary* of a *person* has been updated and the person is given an increased *holiday* allowance within the same transaction).

Condition language: a notation used to enquire about the context in which the rule has been triggered, to ascertain if it is appropriate for the action to be carried out.

Action language: a notation used to specify the effect that the rule must have when it is executed.

The execution model describes how rules interact in the context of the whole database system. It has the following aspects:

Coupling mode: the temporal and causal relationship between triggering and execution. For example, it is possible that the action of a rule is executed as soon as possible after the evaluation of the condition of the rule (immediate), or at a later point, such as at end of transaction (deferred).

Transition granularity: the nature of the binding between event occurrences and rule activations. It is possible that an individual event occurrence will trigger a rule, or that a collection of occurrences of an event will together trigger a rule.

Rule priorities: the mechanism for indicating to the system the order in which rules are selected for processing when many are triggered at the same time.

2.2 Event Specification Languages

The event and the condition of an ECA rule between them describe the *situation* that is being monitored. The overall expressiveness of situation monitoring thus depends upon the capabilities of the event and condition description languages. In general, increasing the power of the event specification language means that more of the situation can be described using the event part of a rule, and that conditions are both evaluated less frequently and perform more straightforward tests. However, this is achieved at some cost – rich event description languages often require that considerable amounts of information be recorded about partially detected events, and thus impose space and time overheads on executing transactions.

An event specification language enables declarative queries to be posed about what is happening in the database. As such, event specifications can be seen as queries over the history of operations that have been applied to the database, although implementations do not generally implement event detection in this way for performance reasons. Rather, the only information that is recorded is on specific activities of interest to the application.

The best known languages for describing composite events are supported by ODE [18], SAMOS [16] and Sentinel [11]. These languages provide facilities, for example, for detecting:

- *Conjunctions* of events (the event E_1 and the event E_2 have taken place within some time interval).
- *Sequences* of events (the event E_1 followed the event E_2 within some time interval).
- *Repetition* of an event (the event E occurred multiple times within some time interval).

The event detection language for ROCK & ROLL supports a range of facilities that are present in the languages mentioned above. However, it has several noteworthy characteristics: the syntax of the event detection language is comparable to that of the condition expression language, and the semantics of both languages are defined in terms of the semantics of logic programs; literals and unification within an event description are used to filter out irrelevant events early. Matching within event expressions is also supported in PFL [23], although in that system the considerable power of the event expression language could cause performance problems.

2.3 Architectures

There are two principal architectures for active database systems – *layered* and *integrated*. Of these, ROCK & ROLL has an integrated active rule system, in that event detection, scheduling, optimisation, etc are built into the kernel.

3 ROCK & ROLL

This section summarises the features of ROCK & ROLL that are relevant to the description of the active rule language that follows. The approach to the development of a DOOD adopted in ROCK & ROLL involves the derivation and subsequent integration of two languages, one a logic query language (ROLL), and the other an imperative programming language (ROCK), from a single object-oriented data model (OM). The architecture is presented in figure 1. This architecture remains largely untouched by the active extensions. The languages and the underlying model remain the same, although language facilities are now also used from within active rules which monitor the execution of programs and changes to databases described using the model. The following sections present the individual components of ROCK & ROLL and their integration.

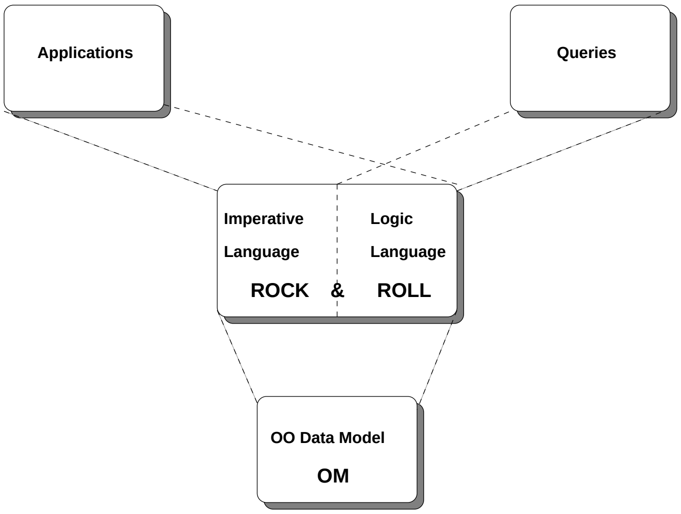

Fig. 1. Relationship between the principal components of ROCK & ROLL.

3.1 OM

The OM data model underpins the two languages ROCK and ROLL; it supports a range of modelling constructs familiar from semantic data models for describing sets, aggregates, inheritance, lists, etc. To illustrate some of the features of OM, the example power supply application from [10] will be used.

Information has to be stored on the elements that appear in the power supply network, and on the way in which these elements are connected together. The elements in the network are represented as shown in figure 2, using the types: `point`, which is an abstract type defining the property `location`, which is shared by the subtypes of `point`; `plant`, which is a source of power for the network; `user`, which is a sink drawing power from the network; and `node` which is the location of a connection in the network.

The connections between the elements in the network are represented as shown in figure 3, using the types: `link`, which identifies the `point`s that are connected; `wire`, which describes the current carrying component of the connection; and `tube` which describes any container which is used to wrap the `wire`s.

Note that this is rather a straightforward model which uses only a few of the features of OM; for examples of the use of ROCK & ROLL in a geographic application see [1].

3.2 ROCK

ROCK is an imperative object-oriented database programming language based upon OM (i.e. OM describes the structural characteristics of objects that can be

```
type point
    properties:
      public:
        location: string;
end-type

type plant:
    specialises: point;
    properties:
      public:
        power_output: float;
end-type

type user:
    specialises: point;
    properties:
      public:
        power_reqt: float;
end-type

type node:
    specialises: point;
    properties:
      public:
        power_loss: float;
end-type
```

Fig. 2. Elements in the power supply network

created and manipulated by ROCK). As such, ROCK can be used to implement complete applications over OM databases, as together ROCK and OM constitute an object-oriented database system. ROCK programs can either be free standing applications, or can be attached to OM types as methods. To illustrate some ROCK concepts, the following code fragment increases the `max_voltage` of every `wire` with a current `max_voltage` of less than `10.0` by `10%`:

```
foreach w in wire do begin
    if (get_max_voltage@w < 10.0) then
        put_max_voltage(get_max_voltage@w * 1.1)@w;
end
```

This fragment illustrates: iteration over the instances of a type using `foreach`; retrieval of the stored properties of an object using a system-generated method of the form `get_`*attributename* invoked using the message sending operator `@`; and updating of the value of a stored property of an object using a system-generated method of the form `put_`*attributename*.

ROCK can be considered to be a conventional imperative programming language operating in an object-oriented context. Standard facilities for object-

```
type link:
    properties:
      public:
        from: point,
        to: point;
end-type

type wire:
    specialises: link;
    properties:
      public:
        max_voltage: float,
        max_power: float;
        voltage:float,
        power_carried: float;
end-type

type tube:
    specialises: link;
    properties:
      public:
        protected: bool;
end-type
```

Fig. 3. Connections in the power supply network

oriented programming are supported, including encapsulation, overriding and dynamic binding. ROCK programs defined as methods on persistent types are stored in the database alongside the data to which they relate.

3.3 ROLL

ROLL is a logic query language based upon OM (i.e. OM describes the structural characteristics of objects that can be explored by ROLL queries and rules). ROLL is a conventional deductive database language, in that its semantics are defined by a mapping onto Datalog [9] in the context of an axiom set which describes the structural features of the OM data model [14]. ROLL is free from extensions to support updates or control, which are present in many other deductive database languages (e.g.[22]), as such facilities are provided by ROCK.

ROLL can be used to write queries or methods. As an example, the following ROLL query will retrieve the set of `user` objects that are indirectly connected to the `plant` with `location` equal to `Sizewell`.

```
[ALL User | get_to@Link == User:user,
            get_from@Link == Plant:plant,
            get_location@Plant == "Sizewell"]
```

The part of the query to the left of the `|` is the projection expression, which indicates that what is of interest is the set of bindings obtained for the variable `User` when the query to the right of the `|` is evaluated. The query looks for bindings for the variables `Link`, `Plant` and `User`, such that the `Link` joins the `Plant` to the `User` and the `location` of the `Plant` is `Sizewell`. The types of variables in ROLL queries are inferred by a type inference system, and thus need not be given explicitly except to resolve ambiguities. In the above example, the variable `User` is explicitly associated with the type `user` and the variable `Plant` is explicitly associated with the type `plant`. The subgoals to the right of the query can be listed in any order, and an optimiser plans an efficient execution strategy [13]. ROLL can also be used to define methods, expressed as deductive rules, as described in [1, 3, 13].

3.4 ROCK & ROLL

The languages ROCK and ROLL can be used together in the development of a single program. The following facilities are available for their combined use:

- ROLL queries can be embedded in ROCK programs. For example, the example query from the previous section can be embedded in ROCK thus:

  ```
  var us := [ALL User | get_to@Link == User:user,
                        get_from@Link == Plant:plant,
                        get_location@Plant == "Sizewell"]
  ```

  The type of the ROCK variable `us` is inferred by the type inference system from the type of the logic variable `User`. ROLL queries can also access ROCK variables, which are named in ROLL with the prefix `!`.
- ROCK methods can be invoked from ROLL queries and rules, although with the restriction that the ROCK method must not have any side-effects. The check for potential side-effects is carried out at compile time.

The level of seamlessness offered by the integration of ROCK and ROLL is discussed in [4].

4 Knowledge Model

This section describes the languages available to the user for describing events, conditions and actions in ROCK & ROLL. Essentially, rules are constructed using the syntax:

$$Rule ::= \texttt{RULE}\ RuleId\ \texttt{WHEN}\ EventSpec\ \texttt{IF}\ CondSpec\ \texttt{DO}\ Action\ \texttt{END\_RULE}$$

The remainder of this section, along with section 5 will fill in the details of how rules are defined.

4.1 Event Language

ROCK & ROLL supports a variety of primitive event types and provides an event algebra for describing composite events.

$$Event ::= PrimEvent \mid\mid CompEvent$$

Primitive Events: Primitive events are able to monitor a wide range of structure operations and behaviour invocations, as well as more global activity, such as the start or the end of a session.

$$\begin{aligned}PrimEvent ::= {} & MsgEvent \mid\mid CreateEvent \mid\mid DeleteEvent \mid\mid \\ & SessStartEvent \mid\mid SessCommitEvent \mid\mid SessAbortEvent\end{aligned}$$

In ROCK & ROLL, structure operations are implemented using a message sending syntax to invoke system generated methods, and thus share the same syntax as user defined operations. Object creation and deletion occurs via special built-in operators which are also shared by the event syntax.

Message events must specify a method name, recipient and arguments, and are described using a syntax comparable to that used for method invocation:

$$\begin{aligned}MsgEvent &::= Msg\ \texttt{SENT} \mid\mid Msg \mid\mid Msg ==> Param \\ Msg &::= \mathrm{Id}\ OptParams\ @\ Rcpt \\ OptParams &::= \epsilon \mid\mid (\ Params\ ) \\ Params &::= Param \mid\mid Param\ ,\ Params\end{aligned}$$

Message events must also specify whether the event occurs when the message is sent or after it has been processed. In the latter case it may also be desirable to know the return value from the method (where there is one). The keyword `SENT` can be appended to identify that the event occurs when the message is *sent* i.e. before executing the method. If the return value from a completed message is required in the event specification then the message send can be followed by the symbol `==>` and a parameter for the returned value.

The parameters of a message event can be either variables which *bind* message arguments, constant values which *match* message arguments or anonymous variables (written as _), which unify with arguments that play no other role in the event detection process. Note that there is no way to specify an instance of a user defined type as a constant value so the recipient can only be supplied as a variable (or anonymous variable).

$$\begin{aligned}Param &::= Var \mid\mid Val \mid\mid \mathrm{AnonVar} \\ Rcpt &::= Var \mid\mid \mathrm{AnonVar} \\ Var &::= \mathrm{Id} \mid\mid \mathrm{Id} : \mathrm{Id}\end{aligned}$$

As an example of a message event, `put_max_voltage(0)@W:wire` detects when an attempt is made to assign `0` as the maximum voltage of `wire W`.

Composite Events: Composite events allow both conjunctions and disjunctions of events. Conjunctions may be either sequenced or unsequenced. In the present implementation, the interval over which the components of a composite event are accumulated is a single session.

$$\begin{aligned}CompEvent ::= {} & OptPolicy\ Event\ \texttt{WITH}\ Event \mid\mid \\ & OptPolicy\ Event\ \texttt{THEN}\ Event \mid\mid \\ & Event\ \texttt{OR}\ Event \mid\mid \\ & \texttt{REPEATED}\ \mathrm{Count}\ \texttt{TIMES}\ Event\end{aligned}$$

The `WITH` keyword implies unordered conjunction whereas `THEN` implies sequenced conjunction.

As an example of a composite event:

```
put_max_voltage(Max)@W:wire WITH put_min_voltage(Min)@W
```

detects the setting of the maximum and the minimum value for the voltage of a wire within a single session. Note that unification of variable W between the two events ensures that the composite event is only triggered by two messages sent to the same wire. As a further example of a composite event, the following event specification detects when the maximum voltage of a wire is changed more than once within a session:

```
REPEATED 2 TIMES put_max_voltage(Max)@W:wire
```

In [11] it is shown that different *consumption modes* for composite events can be discerned, which indicates how component events are grouped together to form composite events (there is not space here to elaborate on the details). In ROCK & ROLL, three consumption modes are supported for conjunctive events, namely recent, chronicle and continuous. The keyword `LATEST` (the default) specifies recent consumption, `EARLIEST` specifies chronicle consumption and `ALL` specifies continuous consumption.

$$OptPolicy ::= \epsilon \mid\mid \texttt{LATEST} \mid\mid \texttt{EARLIEST} \mid\mid \texttt{ALL}$$

Event Detection and ROLL: As indicated in the introduction, one of the aims of this paper is to show what costs and benefits result from the integration of declarative language features with an active database system. Benefits which derive from the use of a declarative condition language include consistency of style with the event language, conciseness, and amenability to optimisation and analysis (see section 7). There are difficulties, however, with detection of events raised within a declarative language component. In ROCK & ROLL the problem arises when operations invoked from ROLL queries or methods are being monitored by events.

There are two basic problems. Firstly, the number of times a method is invoked and the parameters with which it is invoked depend on the evaluation strategy chosen by the optimiser. Secondly, certain operations appearing in the source of the program may be replaced by other operations when the program is compiled.

Consider the query in section 3.4 when a rule is monitoring the message event `get_location@P:plant ==> L`. The query might be executed naively by scanning class `user` traversing `links` to the relevant `plants` and testing the value of attribute `location`. Alternatively, class `plant` might be scanned first or, in the presence of an index on `location`, the relevant `plants` generated directly bypassing any access to attribute `location`.

The approach adopted in ROCK & ROLL is to raise an event for every set of bindings *generated* for a method invocation with unbound arguments or every set of bindings *validated* for a method invocation whose arguments are all bound. So, in the example, whatever plan the optimiser chooses, an event is raised for each plant/location pair generated during evaluation. If a plant is considered which has undefined location or location other than the required value `Sizewell` then no event is raised.

4.2 Condition Language

Rule conditions are essentially arbitrary ROLL expressions, although the pseudo-goal `TRUE` is also supported, effectively allowing optional conditions.

$$Condition ::= \texttt{TRUE} \;||\; GoalCond$$

Alternative conditions can be expressed using a disjunction operator, so long as the condition as a whole is in Disjunctive Normal Form (DNF).

$$GoalCond ::= GoalList \;||\; GoalList \;\texttt{OR}\; GoalCond$$

Each disjunct is a list of ROLL goals, built-in, system-generated or user-defined, as appears in the right hand side of a normal query expression. Rule conditions can also access information about the event which has triggered the rule, as discussed in section 5.2. Conditions are thus expressed using the declarative language facilities of the ROCK & ROLL system, which makes them amenable to query optimisation, as discussed in section 7.3.

4.3 Action Language

Action specifications are operations to be performed either in addition to or in place of those mentioned in the event specification. Alternatives are only appropriate when a message event forms part of the rule event specification.

$$Action ::= Acts \;||\; \texttt{INSTEAD}\; Acts \;||\; Acts\; \texttt{RETURN}\; Expr \;||\; \texttt{INSTEAD}\; Acts\; \texttt{RETURN}\; Expr$$

A list of actions (*Acts*) is a sequence of semi-colon separated ROCK & ROLL expressions or statements, which may also access information originating from the event or condition, as described in section 5.2.

$$Acts ::= \mathrm{ROCKOp} \;||\; \mathrm{ROCKOp}\; ;\; Acts$$

Thus actions are expressed using the procedural language facilities of the passive ROCK & ROLL system, which allows direct execution of stored operations, updates, or user interaction commands.

4.4 Control Language

Language facilities are also provided to allow the user to control the way in which rules are executed. As these relate to the execution model, they are considered in section 5.

5 Execution Model

This section describes the execution model of the ROCK & ROLL active rule system, along with the syntactic constructs that relate to execution model features.

5.1 Coupling Modes

The coupling mode of a rule indicates when the event, condition and action of the rule are evaluated relative to each other. The coupling mode is described by a statement given separately from the rule definition:

$Coupling$ ::= `COUPLING` $RuleId$ `CONDITION` $Mode$ `ACTION` $Mode$ `END_COUPLING`
$Mode$ ::= `IMMEDIATE` || `DEFERRED`

The coupling mode is expressed separately for the condition and the action of the rule, and thus indicates when the condition is evaluated relative to the event and when the action is evaluated relative to the condition. For example, it would be possible to check a condition immediately after the event that the rule is monitoring, but to defer execution of the action to the end of the session.

As ROCK & ROLL has a very straightforward transaction model which is derived from that of the EXODUS system on which it is built [7] (a session is treated as a single transaction), deferred processing is initiated either in response to the top-level (i.e. not a program statement) command `ASSERT`, or, by default, at the end of a session.

Rules with `INSTEAD` actions always require immediate processing and rules which specify a set level transition granularity (see below) always require deferred mode processing. Furthermore, rules which perform actions on objects arising from a delete event must be processed immediately since it is not possible to perform such operations after the deletion has occurred. This is part of a wider difficulty which arises for all object-oriented models with explicit deletion, as information used as input to a rule may no longer be current when the rule is executed. This is handled in ROCK & ROLL by untriggering all rule activations that have deleted objects as parameters, but would be much more difficult to support in object-oriented databases with lower level data models.

An additional problem exists in supporting `IMMEDIATE` rule processing where rules are triggered by operations invoked from ROLL queries or rules. Not only is the order and number of events raised dependent upon the way in which the optimiser has planned the execution strategy, no updates can be performed during the execution of a piece of ROLL, which must operate over a static database. Thus when events are raised during the execution of a fragment of ROLL, rules defined for `IMMEDIATE` processing are deferred until ROLL code is no longer being processed, and are then scheduled for execution; a similar approach is adopted in PFL [23].

5.2 Transition Granularity

Deferred rules need to specify whether the condition is to be executed repeatedly, once each time the specified event occurs, or once only, even if the specified event occurs several times. As well as defining whether these occurrences should be processed individually or as a group, it is also necessary to specify how bindings for variables in the event are to be passed as parameters to the condition or action, and how bindings obtained from the condition are to be supplied to the action.

Projections from events and conditions use a comprehension syntax similar to that employed by ROLL queries. The right hand side contains a description of the relevant events or conditions. The left hand side contains a projection of the relevant bindings. To allow bindings from the event or condition to be referred to in the condition/action it is possible to bind the projected value(s) to a variable. Subsequent mention of the variable in the condition/action provides access to the relevant value(s).

$$
\begin{aligned}
EventSpec &::= \text{Var} <== [\, EProj \mid Event \,] \mid\mid \\
&\qquad [\, EGran \mid Event \,] \\
CondSpec &::= \text{Var} <== [\, CProj \mid Condition \,] \mid\mid \\
&\qquad [\; \mid Condition \,]
\end{aligned}
$$

If the rule is fired *each* time the event is raised then the projection can retrieve a single value or a single aggregation of values from the event occurrence for subsequent use in condition or action processing. If the rule is only fired once in response to multiple raisings of the event then the projection can take one of two forms. It can pick a single value or a single aggregation of values from *any* of the event occurrences which raised the event, or it can retrieve bindings from *all* event occurrences supplying the collection of values to the condition and action as an association.

$$EProj ::= \texttt{EACH}\ Proj \mid\mid \texttt{ANY}\ Proj \mid\mid \texttt{ALL}\ Proj$$

An event projection which does not retrieve bindings specifies what to do when an event is raised more than once. The rule may be fired in response to *each* raising of the event or once only in response to *any* event occurrence.

$$EGran ::= \texttt{EACH} \mid\mid \texttt{ANY}$$

As an example of an event specification, the following raises an event for every wire that is assigned both a maximum and a minimum voltage during a session, and projects out the object identifier of the wire:

```
awire <== [EACH W |put_max_voltage(Max)@W:wire WITH put_min_voltage(Min)@W]
```

The following event specification raises a single event whenever one or more wires have had their maximum and minimum voltages set during a session, and projects out the set of wires modified in this way (this supports a form of nett-effect processing):

```
wires <== [ALL W |put_max_voltage(Max)@W:wire WITH put_min_voltage(Min)@W]
```

A condition specification includes a ROLL query expression specifying the condition to be evaluated. If the corresponding event specification produces a binding for the event variable Var then the bound value may be employed as an input in the query expression by mentioning Var prefixed with `!` in the query body.

The projection part of the query uses the conventional query syntax producing either a boolean result, a single value or aggregation of values obtained from *any* solution to the expression on the right hand side of the query or a set of values or aggregations retrieved from *all* solutions to the expression. In all cases the action will only be executed if the query expression can be solved using the current state of the database.

$$CProj ::= \texttt{EACH}\ Proj\ ||\texttt{ANY}\ Proj\ ||\ \texttt{ALL}\ Proj$$
$$CProj ::= \epsilon$$

If an `ANY` query is used then the action is run once should there be *any* solution to the query, with the bindings picked arbitrarily from one such solution. If an `ALL` query is used then the action is run once if the query has solutions, with bindings being collected from *all* solutions as an association. If an `EACH` query is used then the action is run once for each solution to the query with the relevant bindings projected out of the solution and made available to the action.

The syntax for specifying projected variables in event and condition specifications is the usual ROLL query projection syntax, in which aggregates are specified using angled brackets:

$$Proj ::= Id\ ||\ \langle\ Id\ ,\ Ids\ \rangle$$

Examples of condition specifications are given in section 6.

Rule Priorities: Where multiple rules or rule components (conditions/actions) are scheduled for execution at the same time, a priority mechanism can be used to indicate to the scheduler the order in which different items of processing should take place. In ROCK & ROLL a simple numerical scheme is used:

$$Priority ::= \texttt{PRIORITY}\ RuleId\ RealConst\ \texttt{END\_PRIORITY}$$

6 Example Application

This section continues the power supply example from [10], showing how some constraints can be expressed using the ROCK & ROLL rule language.

Example 1: *All wires must be inside a tube.* The problem arises here when there exist some wires for which there is no corresponding tube. The condition to test for this situation exploits ROLL methods defined on `link` which associate a link with nodes that it links:

```
links(Point1,Point2)@Link :-
    get_from@Link == Point1,
    get_to@Link == Point2.

tube_linked(To)@From :-
    links(To, From)@T:tube.
```

The constraint can only be broken when the end points of a `wire` or a `tube` are changed, or when a tube is deleted. Each case has to be treated separately since in the first case there is a specific wire to check, whereas in the second all wires need checking. The constraint is enforced by creating a new tube to hold the wires which have no tube.

The rule dealing with modification of wires is as follows:

```
RULE AllWiresInTube_1
WHEN Wires <== [ALL W | put_to(T)@W:wire OR put_from(F)@W]
IF Ends <== [ALL <To, From> | W1 in !Wires,
                              links(To, From)@W1:wire,
                              ~tube_linked(To)@From:tube]
DO
foreach pair in Ends do
begin
    var t := new tube();
    put_to(get_to@pair)@t;
    put_from(get_from@pair)@t;
end
END_RULE

COUPLING AllWiresInTube_1 CONDITION Deferred ACTION Immediate END_COUPLING
```

The rule for deletion or modification of a tube is similar but requires exploration of the extension of class wire:

```
RULE AllWiresInTube_2
WHEN [ANY | delete T:tube OR put_to(To)@T:tube OR put_from(From)@T:tube]
IF Ends <== [ALL <To, From> | links(To, From)@W:wire,
                              ~tube_linked(To)@From:tube]
DO
foreach pair in Ends do
begin
    var t := new tube();
    put_to(get_to@pair)@t;
    put_from(get_from@pair)@t;
end
END_RULE

COUPLING AllWiresInTube_2 CONDITION Deferred ACTION Immediate END_COUPLING
```

Note that in this rule any event instance will do to trigger the rule since there is no parameter required from the event, and one evaluation of the condition can identify all cases where the constraint has been broken.

Example 2: *If a tube contains any high voltage wires then it must be protected.* This constraint can only be violated when the end point of a tube or wire is changed, or when the voltage of a wire is modified. It can be fixed by making the tube protected. The cases involving a wire must be handled separately from those involving a tube.

The rule for tubes needs to check the wires in the tube for one with voltage greater than 5000 volts.

```
RULE HighVoltageWires_1
WHEN Tube <== [EACH T | put_to@T:tube OR put_from@T:tube]
IF [ | get_protected@!Tube == false, links(To, From)@!Tube,
       links(To,From)@W, get_voltage@W > 5000]
DO
```

```
begin
    put_protected(true)@Tube;
end
END_RULE

COUPLING HighVoltageWires_1 CONDITION Deferred ACTION Immediate END_COUPLING
```

The condition is merely a boolean since the action is only parameterised by the event variable. The action is executed once per event binding irrespective of the number of wires found by the condition (so long as at least one wire is found).

The rule for wires is as follows:

```
RULE HighVoltageWires_2
WHEN Wires <== [ALL W | put_to@W:wire OR put_from@W:wire OR
                        put_voltage(V)@W]
IF Tube <== [EACH T | W1 in !Wires, get_voltage@W1 > 5000,
                      links(To,From)@W1, links(To, From)@T,
                      get_protected@T == false]
DO
begin
    put_protected(true)@Tube;
end
END_RULE

COUPLING HighVoltageWires_2 CONDITION Deferred ACTION Immediate END_COUPLING
```

7 Distinctive Features

This section outlines some distinctive features relating to the development of the ROCK & ROLL active rule system, and some significant consequences which derive from the way the system has been designed.

7.1 Specification

This paper has taken the approach of describing the languages used by the user to define active applications in ROCK & ROLL in an informal manner. The complete system has, however, been formally specified. The semantics of the condition and action languages are the semantics of the passive languages ROCK and ROLL, as described in [5] and [14] respectively. The active rule extension has two additional components which must be described, namely:

1. *Event Definition Language:* This has been described by way of a mapping onto deductive rules over an event history described using the event calculus of [20]. As an example, the event for the rule:

   ```
   RULE R1
   WHEN aWire <= [EACH W | put_max_voltage(0)@W]
   IF ...

   COUPLING R1 CONDITION Immediate ...
   ```

gives rise to the following event rules:

```
starts(EId, trigger('R1', EId, W)) ←
        event(EId),
        happened(EId, completed(put_max_voltage, W, wire, 0)).

ends(EId, trigger('R1', _, W)) ←
        event(EId),
        happened(EId, completed(delete, W, wire)).

ends(EId, trigger('R1', EId, _)) ←
        event(EId),
        happened(EId, fired('R1', EId)).
```

The first rule indicates that it becomes true that the rule $R1$ is triggered when a *put_max_voltage* method has been sent to a wire W with argument *0*. The rule can then be untriggered in one of two ways: as a result of the wire W being deleted, or as a result of the rule being fired by the rule processor, in which case the fact that $R1$ has been $fired$ is explicitly asserted. This description is necessarily rather superficial; the approach adopted is described fully in [15].

2. *Execution Model:* This has been defined formally using an operational style, which describes the algorithms and data structures used to record triggered rules and to schedule rule components for execution. The approach is described in [15].

Developing a formal specification of the active rule system has proved very useful to the development process, particularly in making explicit issues which require decisions to be made by designers at an early stage in the development process.

7.2 Analysis

The analysis of active rule bases involves the automatic detection of certain properties which are often felt to be desirable in rule bases. The two features most commonly searched for are *termination* (is rule processing guaranteed to terminate after any set of changes to the database in any state?) and *confluence* (is rule processing guaranteed to reach a unique final state for every valid and complete rule execution sequence starting at any database state?). Probably the most complete and broadly applicable framework for rule analysis is provided by [2], where there are two principal aspects in the development of a rule analyser:

1. *Trigger Graph Construction:* This involves identifying which pairs of rules are able to trigger each other. This analysis has often been done in a very conservative manner, which means that terminating or confluent rule bases are not always detected as such. In ROCK & ROLL, the presence of literals and unification in the event language combined with the constraint based techniques applied in the optimiser (for conditions described in [13]) makes it possible to build less conservative trigger graphs than in other systems.

This is an important benefit which derives from the retention of a declarative condition language.
2. *Trigger Graph Analysis:* This builds upon the triggering relationships mentioned above, and searches for patterns which guarantee terminating and confluent rule bases; the algorithms of [2] are applicable to ROCK & ROLL.

7.3 Optimisation

Active rule optimisation has received surprisingly little attention, but is important to the overall performance and uptake of active database systems. There are two principal aspects:

1. *Optimising Single Rules:* In ROCK & ROLL, the condition of each rule is optimised using the ROLL optimiser of [13], which often allows highly constrained searches to be carried out, given knowledge about literals and bindings extracted from events.
2. *Optimising Multiple Rules:* It is common for multiple rules to be triggered at the same time, which potentially allows multiple query optimisation to be applied to their conditions. In practice, multiple query optimisation can only be carried out when the order in which the rules are chosen for processing is known to have no effect on the results of evaluating their conditions. However, this property is exactly what is tested for when rule analysis searches for confluent rule sets, and the rule optimiser for ROCK & ROLL will exploit knowledge about confluence to support multiple query optimisation.

8 Conclusions

This paper has described the active rule language for the ROCK & ROLL DOOD system. The work is novel in a number of respects:

1. It is shown how a DOOD system can be extended with active features, indicating where existing functionality can be reused, and where new languages and features are required.
2. It is shown how a comprehension based syntax for event detection and condition expression can be employed to support a range of coupling modes and transition granularities.
3. It is shown how the filtering capabilities of events can be increased by the introduction of literals and unification, and how more flexible parameter passing can be obtained between events, conditions and actions.
4. It is shown how declarative features can be used for describing events and conditions (with benefits for analysis and optimisation), while comprehensive imperative programming facilities are provided for action execution. It is anticipated that future commercial active systems will have a similar balance of declarative and procedural features.
5. It has been indicated how the approach described can be formally specified, and how it lends itself to comprehensive analysis and optimisation of rules.

Availability: The ROCK & ROLL system is available over the WWW from http://www.cee.hw.ac.uk/Databases. The active rule language is currently being implemented, and is scheduled for inclusion in the public release from September 1996.

Acknowledgement: This research is supported by the UK EPSRC (grant GR/H43847) and the EU Human Capital and Mobility Network ACT-NET.

References

1. A.I. Abdelmoty, N.W. Paton, M.H. Williams, A.A.A. Fernandes, M.L. Barja, and A. Dinn. Geographic Data Handling in a Deductive Object-Oriented Database. In D. Karagiannis, editor, *Proc. 5th Int. Conf. on Databases and Expert Systems Applications (DEXA)*, pages 445–454. Springer-Verlag, 1994.
2. A. Aiken, J.M. Hellerstein, and J. Widom. Static Analysis Techniqies for Predicting the Behaviour of Active Database Rules. *ACM TODS*, 20(1):3–41, 1995.
3. 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*, 20:185–211, 1995.
4. 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)*, pages 463–474. Morgan-Kaufmann, 1994.
5. M.L. Barja, N.W. Paton, and M.H. Williams. Semantics Based Implementation of a Deductive Object-Oriented Database Programming Language. *J. Programming Languages*, 2(2):93–108, 1994.
6. H. Branding, A. Buchmann, T. Kudrass, and J. Zimmermann. Rules in an Open System: The REACH Rule System. In N.W. Paton and M.H. Williams, editors, *Rules in Database Systems*, pages 111–126. Springer-Verlag, 1994.
7. M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenberg. The EXODUS Extensible DBMS Project: An Overview. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Databases*, CA 94303-9953, 1990. Morgan Kaufman Publishers, Inc.
8. R.G.G. Cattell. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, 1993.
9. S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer-Verlag, Berlin, 1990.
10. S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *16th Intl. Conf. in Very Large Data Bases, Brisbane*, pages 567–577. Morgan Kaufman, 1990.
11. 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.
12. 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*, pages 317–326. Morgan Kaufmann, 1991.
13. 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*, pages 143–160. Springer-Verlag, 1995.

14. A.A.A. Fernandes, M.H. Williams, and N.W. Paton. A Logical Query Language for an Object-Oriented Data Model. In N.W. Paton and M.H. Williams, editors, *Proc. 1st Int. Workshop on Rules in Database Systems*, pages 234–250. Springer-Verlag, 1994.
15. A.A.A. Fernandes, M.H. Williams, and N.W. Paton. A Logic-Based Integration of Active and Deductive Databases, 1996. to be published in *New Generation Computing*.
16. S. Gatziu and K.R. Dittrich. Events in an active object-oriented database. In N.W. Paton and M.H. Williams, editors, *Proc. 1st Int. Workshop on Rules in Database Systems*, pages 23–39. Springer-Verlag, 1994.
17. S. Gatziu, A. Geppert, and K. Dittrich. Integrating active concepts into an object-oriented database system. In P. Kanellakis and J.W. Schmidt, editors, *Proc.* 3^{rd} *Workshop on Database Programming Languages.* Morgan-Kaufmann, 1991.
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*, pages 327–336. Morgan Kaufmann, 1991.
19. G. Koch and K. Loney. *ORACLE: The Complete Reference (3rd Edition).* Osborne McGraw-Hill, 1995.
20. R. Kowalski. Database updates in the event calculus. *Journal of Logic Programming*, 12:121–146, 1992.
21. N.W. Paton, O. Diaz, M.H. Williams, J. Campin, A. Dinn, and A. Jaime. Dimensions of active behaviour. In N.W. Paton and M.H. Williams, editors, *Proc. 1st Int. Workshop on Rules In Database Systems*, pages 40–57. Springer-Verlag, 1994.
22. R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL-Control, Relations and Logic. In Li-Yan Yuan, editor, *Proceedings of the 18th International Conference on Very Large Databases*, pages 239–250. Morgan Kaufman, 1992.
23. S. Reddi, A. Poulovassilis, and C. Small. Extending a Functional DBPL With ECA-Rules. In T. Sellis, editor, *Proc. 2nd Int. Wshp. on Rules in Database Systems*, pages 101–115. Springer-Verlag, 1995.
24. M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in database systems. In *Proc. ACM SIGMOD*, pages 281–290, 1990.
25. J. Widom and S. Ceri. *Active Database Systems.* Morgan Kaufmann, 1996.
26. J. Widom, R.J. Cochrane, and B.G. Lindsay. Implementing Set-Oriented Production Rules as an Extension to Starburst. In R. Camps G.M. Lohman, A. Sernadas, editor, *17th Intl. Conf. on Very Large Data Bases, Barcelona*, pages 275–286. Morgan Kaufmann (ISBN 1-55860-150-3), 1991.

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

An Active Rule Language for ROCK & ROLL

Andrew Dinn, Norman W. Paton†, M. Howard Williams and Alvaro A.A. Fernandes††

Department of Computing and Electrical Engineering,
Heriot-Watt University, Riccarton, Edinburgh, UK
email: <andrew,howard>@cee.hw.ac.uk
Department of Computer Science†,
University of Manchester, Oxford Road, Manchester, UK
email: norm@cs.man.ac.uk
Department of Mathematical Sciences††,
Goldsmiths College, New Cross, London SE14 6NW
email: a.fernandes@gold.ac.uk

Abstract. This paper presents an active rule language for the ROCK & ROLL deductive object-oriented database system. A characteristic feature of ROCK & ROLL is that it blends imperative and deductive programming styles so that both can be used together in support of passive database applications. The aim in developing an active extension is to allow declarative expression of aspects of active behaviour wherever possible, without imposing prohibitive restrictions on the power of the resulting system. The proposal which results is more powerful than most earlier declarative active rule systems, in both its language and execution model, without resorting to the wholly procedural approach supported by most proposals for active object-oriented databases. The paper indicates where retaining declarative features yields greatest benefits, but also where difficulties are encountered which lead to compromises.

1 Introduction

The ROCK & ROLL deductive-object-oriented database (DOOD) system [4, 3] supports an expressive object-oriented data model, a deductive language in which to write queries and rules, and an imperative database programming language. As such, it provides a comprehensive collection of passive facilities for defining database functionality. However, in the same way as the imperative and deductive mechanisms in ROCK & ROLL are complementary, an active extension can itself be seen as adding to the range of facilities available to a programmer. For example, there is no built-in mechanism for integrity checking in ROCK & ROLL, and while certain checks can be implemented using methods in the context of strict encapsulation, it is sometimes the case that the checking of integrity should not be done immediately an update is executed, but rather later, when any temporary inconsistencies should have been overcome. Essentially, active capabilities in a system such as ROCK & ROLL allow non-intrusive monitoring of activities in the database, with a view to enforcing integrity, maintaining derived information, keeping users informed of activities or refreshing displays (for an overview of active databases, see [25]).

Research into active databases leading to the development of complete systems has generally fallen into one of two camps:

- Active relational databases [24, 26], which are based on the query language of the underlying database system, and thus support declarative expression of conditions and straightforward facilities for performing updates. Such systems generally have simple event specification languages and execution models.
- Active object-oriented databases [12, 17, 11, 6], which are based on the programming language of the underlying database system, and thus support procedural mechanisms for the expression of conditions and for performing updates. Such systems generally have sophisticated event specification languages and execution models.

However, the trend that is emerging in standards, for example SQL-3 and ODMG [8], is towards systems which support both declarative query facilities and powerful programming capabilities. This raises the question as to how procedural and declarative mechanisms can be integrated effectively in active systems, without the introduction of arbitrary restrictions and inconsistent rule structures as are found in early commercial implementations of active facilities [19]. The experience reported in this paper for ROCK & ROLL indicates that while blending declarative and procedural mechanisms is not without its challenges, coherent and powerful rule systems can be developed.

This paper is structured as follows: section 2 outlines recent research on active database systems; section 3 summarises the passive capabilities of ROCK & ROLL; section 4 presents the language extensions of the active ROCK & ROLL system; section 5 describes the corresponding execution model; section 6 gives an example of an active application in ROCK & ROLL; section 7 outlines some distinctive features of the active extension to ROCK & ROLL; and section 8 presents some conclusions.

2 Related Work

2.1 Active Database Systems

Limited space prevents a comprehensive description of the characteristics of active database systems; for a classification and brief survey, see [21]. The distinguishing feature of an active database system is that it is able to respond automatically to situations that arise inside or outside the database itself. The active behaviour of a database is generally described using rules, which most commonly have three components, an *event*, a *condition* and an *action*. A rule with such components is known as an event-condition-action rule, or *ECA-rule*. Such a rule lies dormant until an occurrence of the event that it is monitoring, when the rule is said to be *triggered*. The condition of a triggered rule is subsequently evaluated, and if true, then the action of the rule is scheduled for execution.

The structural and behavioural characteristics of active database systems can be classified according to a number of dimensions, as outlined informally in [21]. For the purposes of this paper we classify the aspects to be described into two areas: *knowledge model* and *execution model*.

The knowledge model represents the syntactic view of active rules as seen by the rule programmer. This has three main facets:

Event language: a notation in which the situations that trigger a rule can be specified. Events are classified as *primitive*, in which case no components are distinguished (e.g. the *deletion* of an object), or *composite*, in which case a number of primitive occurrences are linked using the operations of an event algebra (e.g. the *salary* of a *person* has been updated and the person is given an increased *holiday* allowance within the same transaction).

Condition language: a notation used to enquire about the context in which the rule has been triggered, to ascertain if it is appropriate for the action to be carried out.

Action language: a notation used to specify the effect that the rule must have when it is executed.

The execution model describes how rules interact in the context of the whole database system. It has the following aspects:

Coupling mode: the temporal and causal relationship between triggering and execution. For example, it is possible that the action of a rule is executed as soon as possible after the evaluation of the condition of the rule (immediate), or at a later point, such as at end of transaction (deferred).

Transition granularity: the nature of the binding between event occurrences and rule activations. It is possible that an individual event occurrence will trigger a rule, or that a collection of occurrences of an event will together trigger a rule.

Rule priorities: the mechanism for indicating to the system the order in which rules are selected for processing when many are triggered at the same time.

2.2 Event Specification Languages

The event and the condition of an ECA rule between them describe the *situation* that is being monitored. The overall expressiveness of situation monitoring thus depends upon the capabilities of the event and condition description languages. In general, increasing the power of the event specification language means that more of the situation can be described using the event part of a rule, and that conditions are both evaluated less frequently and perform more straightforward tests. However, this is achieved at some cost – rich event description languages often require that considerable amounts of information be recorded about partially detected events, and thus impose space and time overheads on executing transactions.

An event specification language enables declarative queries to be posed about what is happening in the database. As such, event specifications can be seen as queries over the history of operations that have been applied to the database, although implementations do not generally implement event detection in this way for performance reasons. Rather, the only information that is recorded is on specific activities of interest to the application.

The best known languages for describing composite events are supported by ODE [18], SAMOS [16] and Sentinel [11]. These languages provide facilities, for example, for detecting:

- *Conjunctions* of events (the event E_1 and the event E_2 have taken place within some time interval).
- *Sequences* of events (the event E_1 followed the event E_2 within some time interval).
- *Repetition* of an event (the event E occurred multiple times within some time interval).

The event detection language for ROCK & ROLL supports a range of facilities that are present in the languages mentioned above. However, it has several noteworthy characteristics: the syntax of the event detection language is comparable to that of the condition expression language, and the semantics of both languages are defined in terms of the semantics of logic programs; literals and unification within an event description are used to filter out irrelevant events early. Matching within event expressions is also supported in PFL [23], although in that system the considerable power of the event expression language could cause performance problems.

2.3 Architectures

There are two principal architectures for active database systems – *layered* and *integrated*. Of these, ROCK & ROLL has an integrated active rule system, in that event detection, scheduling, optimisation, etc are built into the kernel.

3 ROCK & ROLL

This section summarises the features of ROCK & ROLL that are relevant to the description of the active rule language that follows. The approach to the development of a DOOD adopted in ROCK & ROLL involves the derivation and subsequent integration of two languages, one a logic query language (ROLL), and the other an imperative programming language (ROCK), from a single object-oriented data model (OM). The architecture is presented in figure 1. This architecture remains largely untouched by the active extensions. The languages and the underlying model remain the same, although language facilities are now also used from within active rules which monitor the execution of programs and changes to databases described using the model. The following sections present the individual components of ROCK & ROLL and their integration.

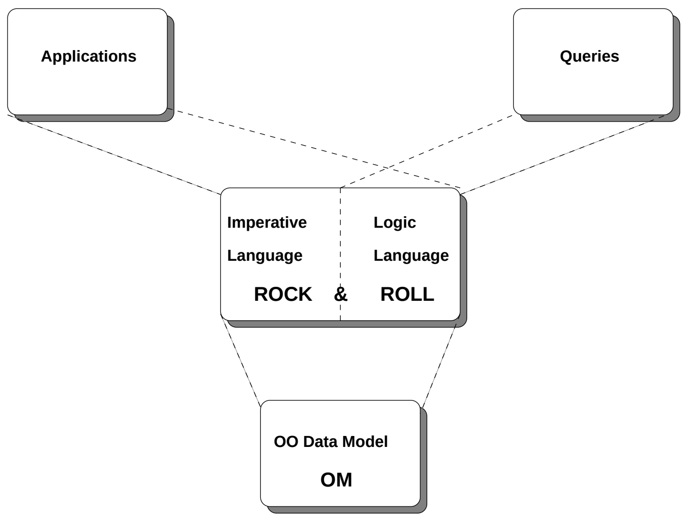

Fig. 1. Relationship between the principal components of ROCK & ROLL.

3.1 OM

The OM data model underpins the two languages ROCK and ROLL; it supports a range of modelling constructs familiar from semantic data models for describing sets, aggregates, inheritance, lists, etc. To illustrate some of the features of OM, the example power supply application from [10] will be used.

Information has to be stored on the elements that appear in the power supply network, and on the way in which these elements are connected together. The elements in the network are represented as shown in figure 2, using the types: `point`, which is an abstract type defining the property `location`, which is shared by the subtypes of `point`; `plant`, which is a source of power for the network; `user`, which is a sink drawing power from the network; and `node` which is the location of a connection in the network.

The connections between the elements in the network are represented as shown in figure 3, using the types: `link`, which identifies the `point`s that are connected; `wire`, which describes the current carrying component of the connection; and `tube` which describes any container which is used to wrap the `wire`s.

Note that this is rather a straightforward model which uses only a few of the features of OM; for examples of the use of ROCK & ROLL in a geographic application see [1].

3.2 ROCK

ROCK is an imperative object-oriented database programming language based upon OM (i.e. OM describes the structural characteristics of objects that can be

```
type point
    properties:
      public:
        location: string;
end-type

type plant:
    specialises: point;
    properties:
      public:
        power_output: float;
end-type

type user:
    specialises: point;
    properties:
      public:
        power_reqt: float;
end-type

type node:
    specialises: point;
    properties:
      public:
        power_loss: float;
end-type
```

Fig. 2. Elements in the power supply network

created and manipulated by ROCK). As such, ROCK can be used to implement complete applications over OM databases, as together ROCK and OM constitute an object-oriented database system. ROCK programs can either be free standing applications, or can be attached to OM types as methods. To illustrate some ROCK concepts, the following code fragment increases the `max_voltage` of every `wire` with a current `max_voltage` of less than `10.0` by `10%`:

```
foreach w in wire do begin
    if (get_max_voltage@w < 10.0) then
        put_max_voltage(get_max_voltage@w * 1.1)@w;
end
```

This fragment illustrates: iteration over the instances of a type using `foreach`; retrieval of the stored properties of an object using a system-generated method of the form `get_`*attributename* invoked using the message sending operator `@`; and updating of the value of a stored property of an object using a system-generated method of the form `put_`*attributename*.

ROCK can be considered to be a conventional imperative programming language operating in an object-oriented context. Standard facilities for object-

```
type link:
    properties:
      public:
        from: point,
        to: point;
end-type

type wire:
    specialises: link;
    properties:
      public:
        max_voltage: float,
        max_power: float;
        voltage:float,
        power_carried: float;
end-type

type tube:
    specialises: link;
    properties:
      public:
        protected: bool;
end-type
```

Fig. 3. Connections in the power supply network

oriented programming are supported, including encapsulation, overriding and dynamic binding. ROCK programs defined as methods on persistent types are stored in the database alongside the data to which they relate.

3.3 ROLL

ROLL is a logic query language based upon OM (i.e. OM describes the structural characteristics of objects that can be explored by ROLL queries and rules). ROLL is a conventional deductive database language, in that its semantics are defined by a mapping onto Datalog [9] in the context of an axiom set which describes the structural features of the OM data model [14]. ROLL is free from extensions to support updates or control, which are present in many other deductive database languages (e.g.[22]), as such facilities are provided by ROCK.

ROLL can be used to write queries or methods. As an example, the following ROLL query will retrieve the set of `user` objects that are indirectly connected to the `plant` with `location` equal to `Sizewell`.

```
[ALL User | get_to@Link == User:user,
            get_from@Link == Plant:plant,
            get_location@Plant == "Sizewell"]
```

The part of the query to the left of the `|` is the projection expression, which indicates that what is of interest is the set of bindings obtained for the variable `User` when the query to the right of the `|` is evaluated. The query looks for bindings for the variables `Link`, `Plant` and `User`, such that the `Link` joins the `Plant` to the `User` and the `location` of the `Plant` is `Sizewell`. The types of variables in ROLL queries are inferred by a type inference system, and thus need not be given explicitly except to resolve ambiguities. In the above example, the variable `User` is explicitly associated with the type `user` and the variable `Plant` is explicitly associated with the type `plant`. The subgoals to the right of the query can be listed in any order, and an optimiser plans an efficient execution strategy [13]. ROLL can also be used to define methods, expressed as deductive rules, as described in [1, 3, 13].

3.4 ROCK & ROLL

The languages ROCK and ROLL can be used together in the development of a single program. The following facilities are available for their combined use:

- ROLL queries can be embedded in ROCK programs. For example, the example query from the previous section can be embedded in ROCK thus:

  ```
  var us := [ALL User | get_to@Link == User:user,
                        get_from@Link == Plant:plant,
                        get_location@Plant == "Sizewell"]
  ```

  The type of the ROCK variable `us` is inferred by the type inference system from the type of the logic variable `User`. ROLL queries can also access ROCK variables, which are named in ROLL with the prefix `!`.
- ROCK methods can be invoked from ROLL queries and rules, although with the restriction that the ROCK method must not have any side-effects. The check for potential side-effects is carried out at compile time.

The level of seamlessness offered by the integration of ROCK and ROLL is discussed in [4].

4 Knowledge Model

This section describes the languages available to the user for describing events, conditions and actions in ROCK & ROLL. Essentially, rules are constructed using the syntax:

Rule ::= `RULE` *RuleId* `WHEN` *EventSpec* `IF` *CondSpec* `DO` *Action* `END_RULE`

The remainder of this section, along with section 5 will fill in the details of how rules are defined.

4.1 Event Language

ROCK & ROLL supports a variety of primitive event types and provides an event algebra for describing composite events.

$$Event ::= PrimEvent \mid\mid CompEvent$$

Primitive Events: Primitive events are able to monitor a wide range of structure operations and behaviour invocations, as well as more global activity, such as the start or the end of a session.

$$\begin{aligned}PrimEvent ::= {} & MsgEvent \mid\mid CreateEvent \mid\mid DeleteEvent \mid\mid \\ & SessStartEvent \mid\mid SessCommitEvent \mid\mid SessAbortEvent\end{aligned}$$

In ROCK & ROLL, structure operations are implemented using a message sending syntax to invoke system generated methods, and thus share the same syntax as user defined operations. Object creation and deletion occurs via special built-in operators which are also shared by the event syntax.

Message events must specify a method name, recipient and arguments, and are described using a syntax comparable to that used for method invocation:

$$\begin{array}{ll} MsgEvent & ::= Msg\ \texttt{SENT} \mid\mid Msg \mid\mid Msg \texttt{ ==> } Param \\ Msg & ::= \mathrm{Id}\ OptParams\ @\ Rcpt \\ OptParams & ::= \epsilon \mid\mid (\ Params\ ) \\ Params & ::= Param \mid\mid Param\ ,\ Params \end{array}$$

Message events must also specify whether the event occurs when the message is sent or after it has been processed. In the latter case it may also be desirable to know the return value from the method (where there is one). The keyword `SENT` can be appended to identify that the event occurs when the message is *sent* i.e. before executing the method. If the return value from a completed message is required in the event specification then the message send can be followed by the symbol `==>` and a parameter for the returned value.

The parameters of a message event can be either variables which *bind* message arguments, constant values which *match* message arguments or anonymous variables (written as _), which unify with arguments that play no other role in the event detection process. Note that there is no way to specify an instance of a user defined type as a constant value so the recipient can only be supplied as a variable (or anonymous variable).

$$\begin{array}{ll} Param & ::= Var \mid\mid Val \mid\mid \mathrm{AnonVar} \\ Rcpt & ::= Var \mid\mid \mathrm{AnonVar} \\ Var & ::= \mathrm{Id} \mid\mid \mathrm{Id} : \mathrm{Id} \end{array}$$

As an example of a message event, `put_max_voltage(0)@W:wire` detects when an attempt is made to assign `0` as the maximum voltage of `wire W`.

Composite Events: Composite events allow both conjunctions and disjunctions of events. Conjunctions may be either sequenced or unsequenced. In the present implementation, the interval over which the components of a composite event are accumulated is a single session.

$$\begin{aligned}CompEvent ::= {} & OptPolicy\ Event\ \texttt{WITH}\ Event \mid\mid \\ & OptPolicy\ Event\ \texttt{THEN}\ Event \mid\mid \\ & Event\ \texttt{OR}\ Event \mid\mid \\ & \texttt{REPEATED}\ \mathrm{Count}\ \texttt{TIMES}\ Event\end{aligned}$$

The `WITH` keyword implies unordered conjunction whereas `THEN` implies sequenced conjunction.

As an example of a composite event:

```
put_max_voltage(Max)@W:wire WITH put_min_voltage(Min)@W
```

detects the setting of the maximum and the minimum value for the voltage of a wire within a single session. Note that unification of variable W between the two events ensures that the composite event is only triggered by two messages sent to the same wire. As a further example of a composite event, the following event specification detects when the maximum voltage of a wire is changed more than once within a session:

```
REPEATED 2 TIMES put_max_voltage(Max)@W:wire
```

In [11] it is shown that different *consumption modes* for composite events can be discerned, which indicates how component events are grouped together to form composite events (there is not space here to elaborate on the details). In ROCK & ROLL, three consumption modes are supported for conjunctive events, namely recent, chronicle and continuous. The keyword `LATEST` (the default) specifies recent consumption, `EARLIEST` specifies chronicle consumption and `ALL` specifies continuous consumption.

$$OptPolicy ::= \epsilon \mid\mid \texttt{LATEST} \mid\mid \texttt{EARLIEST} \mid\mid \texttt{ALL}$$

Event Detection and ROLL: As indicated in the introduction, one of the aims of this paper is to show what costs and benefits result from the integration of declarative language features with an active database system. Benefits which derive from the use of a declarative condition language include consistency of style with the event language, conciseness, and amenability to optimisation and analysis (see section 7). There are difficulties, however, with detection of events raised within a declarative language component. In ROCK & ROLL the problem arises when operations invoked from ROLL queries or methods are being monitored by events.

There are two basic problems. Firstly, the number of times a method is invoked and the parameters with which it is invoked depend on the evaluation strategy chosen by the optimiser. Secondly, certain operations appearing in the source of the program may be replaced by other operations when the program is compiled.

Consider the query in section 3.4 when a rule is monitoring the message event `get_location@P:plant ==> L`. The query might be executed naively by scanning class `user` traversing `links` to the relevant `plants` and testing the value of attribute `location`. Alternatively, class `plant` might be scanned first or, in the presence of an index on `location`, the relevant `plants` generated directly bypassing any access to attribute `location`.

The approach adopted in ROCK & ROLL is to raise an event for every set of bindings *generated* for a method invocation with unbound arguments or every set of bindings *validated* for a method invocation whose arguments are all bound. So, in the example, whatever plan the optimiser chooses, an event is raised for each plant/location pair generated during evaluation. If a plant is considered which has undefined location or location other than the required value `Sizewell` then no event is raised.

4.2 Condition Language

Rule conditions are essentially arbitrary ROLL expressions, although the pseudo-goal `TRUE` is also supported, effectively allowing optional conditions.

$$\mathit{Condition} ::= \texttt{TRUE} \mid\mid \mathit{GoalCond}$$

Alternative conditions can be expressed using a disjunction operator, so long as the condition as a whole is in Disjunctive Normal Form (DNF).

$$\mathit{GoalCond} ::= \mathit{GoalList} \mid\mid \mathit{GoalList}\ \texttt{OR}\ \mathit{GoalCond}$$

Each disjunct is a list of ROLL goals, built-in, system-generated or user-defined, as appears in the right hand side of a normal query expression. Rule conditions can also access information about the event which has triggered the rule, as discussed in section 5.2. Conditions are thus expressed using the declarative language facilities of the ROCK & ROLL system, which makes them amenable to query optimisation, as discussed in section 7.3.

4.3 Action Language

Action specifications are operations to be performed either in addition to or in place of those mentioned in the event specification. Alternatives are only appropriate when a message event forms part of the rule event specification.

$$\mathit{Action} ::= \mathit{Acts} \mid\mid \texttt{INSTEAD}\ \mathit{Acts} \mid\mid \mathit{Acts}\ \texttt{RETURN}\ \mathit{Expr} \mid\mid \texttt{INSTEAD}\ \mathit{Acts}\ \texttt{RETURN}\ \mathit{Expr}$$

A list of actions (*Acts*) is a sequence of semi-colon separated ROCK & ROLL expressions or statements, which may also access information originating from the event or condition, as described in section 5.2.

$$\mathit{Acts} ::= \mathrm{ROCKOp} \mid\mid \mathrm{ROCKOp}\ ;\ \mathit{Acts}$$

Thus actions are expressed using the procedural language facilities of the passive ROCK & ROLL system, which allows direct execution of stored operations, updates, or user interaction commands.

4.4 Control Language

Language facilities are also provided to allow the user to control the way in which rules are executed. As these relate to the execution model, they are considered in section 5.

5 Execution Model

This section describes the execution model of the ROCK & ROLL active rule system, along with the syntactic constructs that relate to execution model features.

5.1 Coupling Modes

The coupling mode of a rule indicates when the event, condition and action of the rule are evaluated relative to each other. The coupling mode is described by a statement given separately from the rule definition:

$Coupling$ ::= `COUPLING` $RuleId$ `CONDITION` $Mode$ `ACTION` $Mode$ `END_COUPLING`
$Mode$ ::= `IMMEDIATE` || `DEFERRED`

The coupling mode is expressed separately for the condition and the action of the rule, and thus indicates when the condition is evaluated relative to the event and when the action is evaluated relative to the condition. For example, it would be possible to check a condition immediately after the event that the rule is monitoring, but to defer execution of the action to the end of the session.

As ROCK & ROLL has a very straightforward transaction model which is derived from that of the EXODUS system on which it is built [7] (a session is treated as a single transaction), deferred processing is initiated either in response to the top-level (i.e. not a program statement) command `ASSERT`, or, by default, at the end of a session.

Rules with `INSTEAD` actions always require immediate processing and rules which specify a set level transition granularity (see below) always require deferred mode processing. Furthermore, rules which perform actions on objects arising from a delete event must be processed immediately since it is not possible to perform such operations after the deletion has occurred. This is part of a wider difficulty which arises for all object-oriented models with explicit deletion, as information used as input to a rule may no longer be current when the rule is executed. This is handled in ROCK & ROLL by untriggering all rule activations that have deleted objects as parameters, but would be much more difficult to support in object-oriented databases with lower level data models.

An additional problem exists in supporting `IMMEDIATE` rule processing where rules are triggered by operations invoked from ROLL queries or rules. Not only is the order and number of events raised dependent upon the way in which the optimiser has planned the execution strategy, no updates can be performed during the execution of a piece of ROLL, which must operate over a static database. Thus when events are raised during the execution of a fragment of ROLL, rules defined for `IMMEDIATE` processing are deferred until ROLL code is no longer being processed, and are then scheduled for execution; a similar approach is adopted in PFL [23].

5.2 Transition Granularity

Deferred rules need to specify whether the condition is to be executed repeatedly, once each time the specified event occurs, or once only, even if the specified event occurs several times. As well as defining whether these occurrences should be processed individually or as a group, it is also necessary to specify how bindings for variables in the event are to be passed as parameters to the condition or action, and how bindings obtained from the condition are to be supplied to the action.

Projections from events and conditions use a comprehension syntax similar to that employed by ROLL queries. The right hand side contains a description of the relevant events or conditions. The left hand side contains a projection of the relevant bindings. To allow bindings from the event or condition to be referred to in the condition/action it is possible to bind the projected value(s) to a variable. Subsequent mention of the variable in the condition/action provides access to the relevant value(s).

$$\begin{array}{lll} \mathit{EventSpec} & ::= & \mathrm{Var} <== [\, \mathit{EProj} \mid \mathit{Event}\,] \;||\; \\ & & [\, \mathit{EGran} \mid \mathit{Event}\,] \\ \mathit{CondSpec} & ::= & \mathrm{Var} <== [\, \mathit{CProj} \mid \mathit{Condition}\,] \;||\; \\ & & [\;\mid \mathit{Condition}\,] \end{array}$$

If the rule is fired *each* time the event is raised then the projection can retrieve a single value or a single aggregation of values from the event occurrence for subsequent use in condition or action processing. If the rule is only fired once in response to multiple raisings of the event then the projection can take one of two forms. It can pick a single value or a single aggregation of values from *any* of the event occurrences which raised the event, or it can retrieve bindings from *all* event occurrences supplying the collection of values to the condition and action as an association.

$$\mathit{EProj} ::= \mathtt{EACH}\ \mathit{Proj} \;||\; \mathtt{ANY}\ \mathit{Proj} \;||\; \mathtt{ALL}\ \mathit{Proj}$$

An event projection which does not retrieve bindings specifies what to do when an event is raised more than once. The rule may be fired in response to *each* raising of the event or once only in response to *any* event occurrence.

$$\mathit{EGran} ::= \mathtt{EACH} \;||\; \mathtt{ANY}$$

As an example of an event specification, the following raises an event for every wire that is assigned both a maximum and a minimum voltage during a session, and projects out the object identifier of the wire:

```
awire <== [EACH W |put_max_voltage(Max)@W:wire WITH put_min_voltage(Min)@W]
```

The following event specification raises a single event whenever one or more wires have had their maximum and minimum voltages set during a session, and projects out the set of wires modified in this way (this supports a form of nett-effect processing):

```
wires <== [ALL W |put_max_voltage(Max)@W:wire WITH put_min_voltage(Min)@W]
```

A condition specification includes a ROLL query expression specifying the condition to be evaluated. If the corresponding event specification produces a binding for the event variable Var then the bound value may be employed as an input in the query expression by mentioning Var prefixed with `!` in the query body.

The projection part of the query uses the conventional query syntax producing either a boolean result, a single value or aggregation of values obtained from *any* solution to the expression on the right hand side of the query or a set of values or aggregations retrieved from *all* solutions to the expression. In all cases the action will only be executed if the query expression can be solved using the current state of the database.

$$CProj ::= \texttt{EACH}\ Proj \,||\, \texttt{ANY}\ Proj \,||\, \texttt{ALL}\ Proj$$
$$CProj ::= \epsilon$$

If an **ANY** query is used then the action is run once should there be *any* solution to the query, with the bindings picked arbitrarily from one such solution. If an **ALL** query is used then the action is run once if the query has solutions, with bindings being collected from *all* solutions as an association. If an **EACH** query is used then the action is run once for each solution to the query with the relevant bindings projected out of the solution and made available to the action.

The syntax for specifying projected variables in event and condition specifications is the usual ROLL query projection syntax, in which aggregates are specified using angled brackets:

$$Proj ::= Id \,||\, \langle\ Id\ ,\ Ids\ \rangle$$

Examples of condition specifications are given in section 6.

Rule Priorities: Where multiple rules or rule components (conditions/actions) are scheduled for execution at the same time, a priority mechanism can be used to indicate to the scheduler the order in which different items of processing should take place. In ROCK & ROLL a simple numerical scheme is used:

$$Priority ::= \texttt{PRIORITY}\ RuleId\ RealConst\ \texttt{END\_PRIORITY}$$

6 Example Application

This section continues the power supply example from [10], showing how some constraints can be expressed using the ROCK & ROLL rule language.

Example 1: *All wires must be inside a tube.* The problem arises here when there exist some wires for which there is no corresponding tube. The condition to test for this situation exploits ROLL methods defined on `link` which associate a link with nodes that it links:

```
links(Point1,Point2)@Link :-
    get_from@Link == Point1,
    get_to@Link == Point2.

tube_linked(To)@From :-
    links(To, From)@T:tube.
```

The constraint can only be broken when the end points of a `wire` or a `tube` are changed, or when a tube is deleted. Each case has to be treated separately since in the first case there is a specific wire to check, whereas in the second all wires need checking. The constraint is enforced by creating a new tube to hold the wires which have no tube.

The rule dealing with modification of wires is as follows:

```
RULE AllWiresInTube_1
WHEN Wires <== [ALL W | put_to(T)@W:wire OR put_from(F)@W]
IF Ends <== [ALL <To, From> | W1 in !Wires,
                              links(To, From)@W1:wire,
                              ~tube_linked(To)@From:tube]
DO
foreach pair in Ends do
begin
    var t := new tube();
    put_to(get_to@pair)@t;
    put_from(get_from@pair)@t;
end
END_RULE

COUPLING AllWiresInTube_1 CONDITION Deferred ACTION Immediate END_COUPLING
```

The rule for deletion or modification of a tube is similar but requires exploration of the extension of class wire:

```
RULE AllWiresInTube_2
WHEN [ANY | delete T:tube OR put_to(To)@T:tube OR put_from(From)@T:tube]
IF Ends <== [ALL <To, From> | links(To, From)@W:wire,
                              ~tube_linked(To)@From:tube]
DO
foreach pair in Ends do
begin
    var t := new tube();
    put_to(get_to@pair)@t;
    put_from(get_from@pair)@t;
end
END_RULE

COUPLING AllWiresInTube_2 CONDITION Deferred ACTION Immediate END_COUPLING
```

Note that in this rule any event instance will do to trigger the rule since there is no parameter required from the event, and one evaluation of the condition can identify all cases where the constraint has been broken.

Example 2: *If a tube contains any high voltage wires then it must be protected.* This constraint can only be violated when the end point of a tube or wire is changed, or when the voltage of a wire is modified. It can be fixed by making the tube protected. The cases involving a wire must be handled separately from those involving a tube.

The rule for tubes needs to check the wires in the tube for one with voltage greater than 5000 volts.

```
RULE HighVoltageWires_1
WHEN Tube <== [EACH T | put_to@T:tube OR put_from@T:tube]
IF [ | get_protected@!Tube == false, links(To, From)@!Tube,
       links(To,From)@W, get_voltage@W > 5000]
DO
```

```
begin
    put_protected(true)@Tube;
end
END_RULE

COUPLING HighVoltageWires_1 CONDITION Deferred ACTION Immediate END_COUPLING
```

The condition is merely a boolean since the action is only parameterised by the event variable. The action is executed once per event binding irrespective of the number of wires found by the condition (so long as at least one wire is found).

The rule for wires is as follows:

```
RULE HighVoltageWires_2
WHEN Wires <== [ALL W | put_to@W:wire OR put_from@W:wire OR
                        put_voltage(V)@W]
IF Tube <== [EACH T | W1 in !Wires, get_voltage@W1 > 5000,
                      links(To,From)@W1, links(To, From)@T,
                      get_protected@T == false]
DO
begin
    put_protected(true)@Tube;
end
END_RULE

COUPLING HighVoltageWires_2 CONDITION Deferred ACTION Immediate END_COUPLING
```

7 Distinctive Features

This section outlines some distinctive features relating to the development of the ROCK & ROLL active rule system, and some significant consequences which derive from the way the system has been designed.

7.1 Specification

This paper has taken the approach of describing the languages used by the user to define active applications in ROCK & ROLL in an informal manner. The complete system has, however, been formally specified. The semantics of the condition and action languages are the semantics of the passive languages ROCK and ROLL, as described in [5] and [14] respectively. The active rule extension has two additional components which must be described, namely:

1. *Event Definition Language:* This has been described by way of a mapping onto deductive rules over an event history described using the event calculus of [20]. As an example, the event for the rule:

   ```
   RULE R1
   WHEN aWire <= [EACH W | put_max_voltage(0)@W]
   IF ...

   COUPLING R1 CONDITION Immediate ...
   ```

gives rise to the following event rules:

```
starts(EId, trigger('R1', EId, W)) ←
        event(EId),
        happened(EId, completed(put_max_voltage, W, wire, 0)).

ends(EId, trigger('R1', _, W)) ←
        event(EId),
        happened(EId, completed(delete, W, wire)).

ends(EId, trigger('R1', EId, _)) ←
        event(EId),
        happened(EId, fired('R1', EId)).
```

The first rule indicates that it becomes true that the rule $R1$ is triggered when a *put_max_voltage* method has been sent to a wire W with argument *0*. The rule can then be untriggered in one of two ways: as a result of the wire W being deleted, or as a result of the rule being fired by the rule processor, in which case the fact that $R1$ has been $fired$ is explicitly asserted. This description is necessarily rather superficial; the approach adopted is described fully in [15].

2. *Execution Model:* This has been defined formally using an operational style, which describes the algorithms and data structures used to record triggered rules and to schedule rule components for execution. The approach is described in [15].

Developing a formal specification of the active rule system has proved very useful to the development process, particularly in making explicit issues which require decisions to be made by designers at an early stage in the development process.

7.2 Analysis

The analysis of active rule bases involves the automatic detection of certain properties which are often felt to be desirable in rule bases. The two features most commonly searched for are *termination* (is rule processing guaranteed to terminate after any set of changes to the database in any state?) and *confluence* (is rule processing guaranteed to reach a unique final state for every valid and complete rule execution sequence starting at any database state?). Probably the most complete and broadly applicable framework for rule analysis is provided by [2], where there are two principal aspects in the development of a rule analyser:

1. *Trigger Graph Construction:* This involves identifying which pairs of rules are able to trigger each other. This analysis has often been done in a very conservative manner, which means that terminating or confluent rule bases are not always detected as such. In ROCK & ROLL, the presence of literals and unification in the event language combined with the constraint based techniques applied in the optimiser (for conditions described in [13]) makes it possible to build less conservative trigger graphs than in other systems.

This is an important benefit which derives from the retention of a declarative condition language.
2. *Trigger Graph Analysis:* This builds upon the triggering relationships mentioned above, and searches for patterns which guarantee terminating and confluent rule bases; the algorithms of [2] are applicable to ROCK & ROLL.

7.3 Optimisation

Active rule optimisation has received surprisingly little attention, but is important to the overall performance and uptake of active database systems. There are two principal aspects:

1. *Optimising Single Rules:* In ROCK & ROLL, the condition of each rule is optimised using the ROLL optimiser of [13], which often allows highly constrained searches to be carried out, given knowledge about literals and bindings extracted from events.
2. *Optimising Multiple Rules:* It is common for multiple rules to be triggered at the same time, which potentially allows multiple query optimisation to be applied to their conditions. In practice, multiple query optimisation can only be carried out when the order in which the rules are chosen for processing is known to have no effect on the results of evaluating their conditions. However, this property is exactly what is tested for when rule analysis searches for confluent rule sets, and the rule optimiser for ROCK & ROLL will exploit knowledge about confluence to support multiple query optimisation.

8 Conclusions

This paper has described the active rule language for the ROCK & ROLL DOOD system. The work is novel in a number of respects:

1. It is shown how a DOOD system can be extended with active features, indicating where existing functionality can be reused, and where new languages and features are required.
2. It is shown how a comprehension based syntax for event detection and condition expression can be employed to support a range of coupling modes and transition granularities.
3. It is shown how the filtering capabilities of events can be increased by the introduction of literals and unification, and how more flexible parameter passing can be obtained between events, conditions and actions.
4. It is shown how declarative features can be used for describing events and conditions (with benefits for analysis and optimisation), while comprehensive imperative programming facilities are provided for action execution. It is anticipated that future commercial active systems will have a similar balance of declarative and procedural features.
5. It has been indicated how the approach described can be formally specified, and how it lends itself to comprehensive analysis and optimisation of rules.

Availability: The ROCK & ROLL system is available over the WWW from http://www.cee.hw.ac.uk/Databases. The active rule language is currently being implemented, and is scheduled for inclusion in the public release from September 1996.

Acknowledgement: This research is supported by the UK EPSRC (grant GR/H43847) and the EU Human Capital and Mobility Network ACT-NET.

References

1. A.I. Abdelmoty, N.W. Paton, M.H. Williams, A.A.A. Fernandes, M.L. Barja, and A. Dinn. Geographic Data Handling in a Deductive Object-Oriented Database. In D. Karagiannis, editor, *Proc. 5th Int. Conf. on Databases and Expert Systems Applications (DEXA)*, pages 445–454. Springer-Verlag, 1994.
2. A. Aiken, J.M. Hellerstein, and J. Widom. Static Analysis Techniqies for Predicting the Behaviour of Active Database Rules. *ACM TODS*, 20(1):3–41, 1995.
3. 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*, 20:185–211, 1995.
4. 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)*, pages 463–474. Morgan-Kaufmann, 1994.
5. M.L. Barja, N.W. Paton, and M.H. Williams. Semantics Based Implementation of a Deductive Object-Oriented Database Programming Language. *J. Programming Languages*, 2(2):93–108, 1994.
6. H. Branding, A. Buchmann, T. Kudrass, and J. Zimmermann. Rules in an Open System: The REACH Rule System. In N.W. Paton and M.H. Williams, editors, *Rules in Database Systems*, pages 111–126. Springer-Verlag, 1994.
7. M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenberg. The EXODUS Extensible DBMS Project: An Overview. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Databases*, CA 94303-9953, 1990. Morgan Kaufman Publishers, Inc.
8. R.G.G. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, 1993.
9. S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer-Verlag, Berlin, 1990.
10. S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *16th Intl. Conf. in Very Large Data Bases, Brisbane*, pages 567–577. Morgan Kaufman, 1990.
11. 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.
12. 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*, pages 317–326. Morgan Kaufmann, 1991.
13. 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*, pages 143–160. Springer-Verlag, 1995.
14. A.A.A. Fernandes, M.H. Williams, and N.W. Paton. A Logical Query Language for an Object-Oriented Data Model. In N.W. Paton and M.H. Williams, editors, *Proc. 1st Int. Workshop on Rules in Database Systems*, pages 234–250. Springer-Verlag, 1994.
15. A.A.A. Fernandes, M.H. Williams, and N.W. Paton. A Logic-Based Integration of Active and Deductive Databases, 1996. to be published in *New Generation Computing*.
16. S. Gatziu and K.R. Dittrich. Events in an active object-oriented database. In N.W. Paton and M.H. Williams, editors, *Proc. 1st Int. Workshop on Rules in Database Systems*, pages 23–39. Springer-Verlag, 1994.
17. S. Gatziu, A. Geppert, and K. Dittrich. Integrating active concepts into an object-oriented database system. In P. Kanellakis and J.W. Schmidt, editors, *Proc.* 3^{rd} *Workshop on Database Programming Languages.* Morgan-Kaufmann, 1991.
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*, pages 327–336. Morgan Kaufmann, 1991.
19. G. Koch and K. Loney. *ORACLE: The Complete Reference (3rd Edition).* Osborne McGraw-Hill, 1995.
20. R. Kowalski. Database updates in the event calculus. *Journal of Logic Programming*, 12:121–146, 1992.
21. N.W. Paton, O. Diaz, M.H. Williams, J. Campin, A. Dinn, and A. Jaime. Dimensions of active behaviour. In N.W. Paton and M.H. Williams, editors, *Proc. 1st Int. Workshop on Rules In Database Systems*, pages 40–57. Springer-Verlag, 1994.
22. R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL-Control, Relations and Logic. In Li-Yan Yuan, editor, *Proceedings of the 18th International Conference on Very Large Databases*, pages 239–250. Morgan Kaufman, 1992.
23. S. Reddi, A. Poulovassilis, and C. Small. Extending a Functional DBPL With ECA-Rules. In T. Sellis, editor, *Proc. 2nd Int. Wshp. on Rules in Database Systems*, pages 101–115. Springer-Verlag, 1995.
24. M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in database systems. In *Proc. ACM SIGMOD*, pages 281–290, 1990.
25. J. Widom and S. Ceri. *Active Database Systems.* Morgan Kaufmann, 1996.
26. J. Widom, R.J. Cochrane, and B.G. Lindsay. Implementing Set-Oriented Production Rules as an Extension to Starburst. In R. Camps G.M. Lohman, A. Sernadas, editor, *17th Intl. Conf. on Very Large Data Bases, Barcelona*, pages 275–286. Morgan Kaufmann (ISBN 1-55860-150-3), 1991.

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