Analysis of Functional Active Databases

James Bailey and Alexandra Poulovassilis

1 Introduction

Active databases are capable of reacting automatically to state changes without user intervention by supporting *event-condition-action* (ECA) rules of the form "on *event* if *condition* do *action*". ECA rules are also known as *triggers* and they are supported in some form by all the major DBMS products. Comprehensive accounts of the active database field can be found in [30, 26].

The potential benefits of active databases are numerous. Rather than application code having to explicitly query the database at pre-defined intervals in order to find out if particular events have occurred, ECA rules allow a timely response to critical events. They also allow reactive functionality to be defined and managed within the DBMS rather than being encoded in diverse programs. This enhances the modularity and maintainability of applications. Also, ECA rules are amenable to powerful analysis and optimisation techniques which cannot be applied if the same functionality is dispersed throughout the application code.

ECA rules provide a unifying approach to many advanced database features, including constraint enforcement, incremental view materialisation, provision of audit trails, and maintenance of database statistics. However, when several ECA rules have been defined within an active database, the execution dependencies between them can be complex and it can be difficult to predict their behaviour. It is recognised that the lack of sufficiently powerful analysis techniques for ECA rules is still an impediment to their wider deployment in database applications [22, 14].

As a contribution to this area, we have developed a framework for ECA rule analysis in functional active databases which is based on *abstract interpretation* [1, 17]. By *functional active database* we mean one whose ECA rule execution semantics have been specified in a purely functional metalanguage as a function that takes an initial database and an initial list of actions to be executed, and returns the resulting database when there are no more actions left to execute. We give an example of such a semantics in Section 2 of this chapter.

Abstract interpretation is a semantics-based method for static analysis of program behaviour. The basic idea is to infer and collect information about a program's run-time properties and its operational aspects. Once the program's semantics have been captured formally, a variety of analyses can be derived as abstractions of these semantics.

An important analysis question for ECA rules is that of termination, where a rule set is *terminating* if for any initial database and initial transaction each ECA rule will fire a finite number of times during the execution of the transaction.

This problem is undecidable even for relatively simple rule languages [5] and so work on termination analysis has focused on developing conservative techniques for detecting terminating rule sets.

One precursor to the work we describe here was the ECA sub-system of the PFL functional DBMS [29, 27]. The rule execution semantics of this ECA sub-system were specified in a functional metalanguage, namely PFL itself. This brought about several advantages: the formal semantics were a robust basis for implementing the ECA sub-system; they were executable and hence allowed us to experiment with the design of the ECA sub-system before investing effort in implementation; and, being specified in PFL itself, the semantics were relatively easily understood by PFL users. In this chapter we illustrate a further significant advantage of such functional specifications of ECA rule execution semantics, namely that they can be used as a basis for developing powerful analysis techniques for ECA rules, and moreover for proving the correctness of such techniques.

The other precursor to the work we describe here is earlier work on using abstract interpretation for termination analysis [3, 4]. This was for a simple variable-based language though, supporting variable assignment rather than database updates. Also, this earlier treatment was described somewhat informally and in an imperative language context.

Several other static analysis techniques have been developed for guaranteeing that a given set of ECA rules definitely terminates, and we will be reviewing some of them in Section 7. However, in many cases it is not clear whether a method developed for a particular rule language and rule execution semantics would be correct if applied to a different language or execution semantics. It may also not be clear whether there is a general strategy for proving the correctness of a method and understanding the trade-offs made in its design.

Abstract interpretation has proven to be a useful tool in the analysis of imperative, functional and logic programs [1, 17, 24, 25]. In this chapter we show how it can be used as a general, formal tool for developing techniques for analysing ECA rule behaviour.

We first specify in Section 2 our ECA rule execution semantics as a function that takes a database state and an initial list of actions to be executed, and returns the resulting database state when there are no more actions left to execute. We next specify in Section 3 an abstract version of these rule execution semantics, operating on abstract databases.

In our framework, a specific analysis technique is modelled by defining a specific abstraction for real databases, and we describe some abstractions in Section 5. The correctness of a particular abstraction is established by proving a set of generic requirements, which we give in Section 4. Our framework can be used both statically, i.e. at rule definition time, and dynamically, during rule execution. The formal nature of the framework allows a smooth adoption of previous work on query satisfiability, incremental evaluation techniques, and approximation techniques. In Section 6, we discuss how our framework can be used for analysing instance-oriented as well as set-oriented rules. We give an overview of other approaches to ECA rule analysis in Section 7. We finally describe how our framework can also be applied to analysing ECA rules on XML databases in Section 8.

2 The Rule Execution Semantics

Our ECA rule execution semantics are expressed as a function, *execSched*, which takes as input a *database* and a *schedule*. The schedule consists of a sequence of actions which are to be executed on the database. The execution of an action may cause events to occur. These may cause further rules to fire, modifying the schedule with new sub-sequences of actions. The execution continues until the schedule becomes empty.

In our semantics we assume that for each type of event E detectable by the database there are two system-defined database objects: *has_occurred_E* and *change_E*. The former is non-empty if event E occurred during the execution of the last action, and it contains information about the occurrences of event E e.g. the time at which they occurred and which transaction caused them to occur. *change_E* contains information about the changes that occurrences of event E made to user-defined database objects during the execution of the last action.

For example, in a typical active relational database there may be for each user-defined relation R a set of six system-defined relations:

- *has_occurred_insertion_R*, which would be non-empty if one or more INSERT statements on relation R occurred during the execution of the last action;
- *change_insertion_R*, which would contain the set of new tuples inserted into relation R during the execution of the last action;
- *has_occurred_deletion_R*, which would be non-empty if one or more DELETE statements on relation R occurred during the execution of the last action;
- *change_deletion_R*, which would contain the set of tuples deleted from R during the execution of the last action;
- *has_occurred_update_R*, which would be non-empty if one or more UPDATE statements on relation R occurred during the execution of the last action; and
- *change_update_R*, which would contain a set of pairs (*old_tuple*,*new_tuple*) for each tuple of R which was updated during the execution of the last action.

Similarly, in a typical active object-oriented database there may be classes *has_occurred_M* and *change_M* for each method M whose invocation is detectable as an event by the database. *has_occurred_M* would be non-empty if method M was invoked during the execution of the last action. *change_M* would contain information about changes made to user-defined database objects by invocations of method M during the last action.

The event part of an ECA rule is either *has_occurred_E* or *change_E*, for some event E. The identifiers *has_occurred_E* and *change_E* may also occur within the rule's condition and action parts.

The rule is said to be *triggered* if its event part is non-empty. Allowing either *has_occurred_E* or *change_E* to appear as rule events means that both "syntactic" and "semantic" triggering of rules are supported within our framework: syntactic triggering happens if the rule's event query is *has_occurred_E* and instances of event E occur; semantic triggering happens if the rule's event query is *change_E* and instances of event E occur and make changes to the database.

The condition part of the rule is a boolean-valued query. The rule's *event-condition query* is the conjunction of its event and condition parts, where an event part is deemed to be $True$ if it is non-empty and $False$ otherwise. The rule *fires* if it is triggered and its condition evaluates to $True$ i.e. if its overall event-condition query evaluates to $True$.

A rule has a list of one or more *actions*. A rule also has a *coupling mode*, which may be either *Immediate* or *Deferred*. With Immediate coupling mode, if the rule fires then its actions are prefixed to the current schedule, while with Deferred coupling mode, they are suffixed. If multiple rules with the same coupling mode fire, then the actions of higher-priority rules precede those of lower-priority ones on the schedule. We thus assume that there is a total ordering imposed on the set of ECA rules as, for example, in the SQL3 trigger standard [23]. We also assume that all rules have the same *binding mode*, whereby any occurrences of *has_occurred* and *change* identifiers appearing in the rule's actions are bound to the database state in which the rule's event-condition query was evaluated.

Our rule execution semantics can be enriched to handle a greater variety of coupling modes and binding modes, but here we confine ourselves to this subset and focus on the use of abstract interpretation for rule analysis. A detailed description of the coupling and binding possibilities for active rules can be found in [26].

We specify our rule execution semantics as a recursive function $execSched$ (see below) which takes a database and schedule, and repeatedly executes the action at the head of the schedule, updating the schedule with the actions of rules that fire along the way. If $execSched$ terminates, it outputs the final database and the final, empty, schedule.

In this specification, $\circ$ denotes function composition, $[\,]$ the empty list, $(x : y)$ a list with head x and tail y, and $++$ is the list append operator. We also assume the following standard function for "left folding" a binary function f into a list:

```
foldl f a []     = a
foldl f a (x:xs) = foldl f (f x a) xs
```

A schedule is a list of *updates*:

```
Schedule = List (Update)
```

where each update is a 4-tuple consisting of a rule action and the values of the rule's event and condition queries at the time that the action was placed on the schedule:

```
Update = (Value, Value, Value, Action)
```

In particular, if E is the event that triggers the rule, i.e. the rule's event part is either *has_occurred_E* or *change_E*, then the first component of the 4-tuple is the value of *has_occured_E*, the second component is the value of *change_E*, and the third component is the value of the rule's condition query, all of these at the time that the rule's actions were placed on the schedule.

The parameter u passed to the function *exec* is an update i.e. a 4-tuple $(oc, ch, cond, a)$. *exec* $(oc, ch, cond, a)$ *db* replaces any *has_occurred* or *change* identifiers occurring within action a by the value of these identifiers at the time that the rule was scheduled i.e. by the parameters *oc* or *ch*, respectively. *exec* then executes action a on the current database *db* and returns the new database, including the updated *change* and *has_occurred* objects[1].

We assume that ECA rules are identified by unique identifiers of type *RuleId*. If E is the event that triggers rule r, then *has_occurred* r returns the query *has_occurred_E*, *change* r returns the query *change_E*, *condition* r returns the rule's condition query, *actions* r its list of actions, and *mode* r its coupling mode. The function *ecq* returns the rule's combined event-condition query.

The function *triggers* takes a rule action, and returns a list comprising the identifiers of the rules that may be triggered by that action, in decreasing order of the rules' priority.

The function *schedRules* applies the function *schedRule* to each rule that may be triggered by a, in decreasing order of the rules' priority. *schedRule* determines whether a given rule fires by invoking the *eval* function to evaluate the rule's event-condition query with respect to the current database. If so, the function *updateSched* is called to update either the schedule's prefix or its suffix, depending on the rule's coupling mode:

```
execSched : (DBState,Schedule) -> (DBState,Schedule)
execSched (db,[])  = (db,[])
execSched (db,u:s) =
    execSched o schedRules (exec u db, u:s)

schedRules : (DBState,Schedule) -> (DBState,Schedule)
schedRules (db,(oc,ch,cond,a):s) =
    let (db,pre,suf) = foldl schedRule (db,[],[]) (triggers a)
    in  (db,pre++s++suf)

schedRule : RuleId -> (DBState,Schedule,Schedule) ->
```

[1] Before making this update, *exec* first checks that the rule's event and condition queries — i.e. the parameter *oc* or *ch* and the parameter *cond* — were *True* at the time the rule was scheduled. For the real rule execution semantics this will trivially be the case since only the actions of such rules are placed onto the schedule. But in order to show the correctness of the abstract semantics, we allow the concrete executions that they represent to place on the schedule the actions of rules whose event or condition query may be *False*. The concrete rule execution must ensure that such "null actions" have no affect on the database or on the subsequent rule firings. We take up this point again in the footnote in Section 4 below.

```
                      (DBState,Schedule,Schedule)
schedRule i (db,pre,suf) =
    if (eval (ecq r) db) = False
    then (db,pre,suf)
    else updateSched (eval (has_occurred r) db,
                      eval (change r) db,
                      eval (condition r) db,
                      actions r,mode r,pre,suf)

updateSched (oc,ch,cond,actions,Immediate,pre,suf) =
    (db,pre ++ [(oc,ch,cond,a) | a <- actions], suf)

updateSched (oc,ch,cond,actions,Deferred, pre,suf) =
    (db,pre, suf ++ [(oc,ch,cond,a) | a <- actions])
```

Example 1. Assume a relational active database and two ECA rules both with Immediate coupling mode:

Rule 1: on *change_insertion_R* if $(R \cap S) \times (R \cap T)$ do *delete* R T

Rule 2: on *change_deletion_R* if *true* do *insert* R $(S - T)$

Only an *insert* or *delete* statement on R can trigger one of the above rules.

Suppose first that a transaction executes an *insert* statement *insert* R e, for some expression e. This update is represented by an initial singleton schedule $[(Unknown, Unknown, Unknown, insert\ R\ e)]$ being passed to *execSched*. *exec* will execute the action *insert* R e on the database. If this results in one or more new tuples being inserted into R, then Rule 1 will be triggered. If its condition is true, then *execSched* will be invoked recursively with the new schedule $[(oc, ch, True, delete\ R\ T)]$ where *oc* and *ch* are respectively the current values of *has_occurred_insertion_R* and *change_insertion_R* (2nd invocation of *execSched*). *exec* will execute the action *delete* R T, which will result in $(R \cap T)$ being deleted from R. Thus Rule 2 will be triggered, and its condition is trivially true. The schedule becomes $[(oc, ch, True, insert\ R\ (S - T)]$ where *oc* and *ch* are respectively the current values of *has_occurred_deletion_R* and *change_deletion_R* (3rd invocation of *execSched*). *exec* will execute the action *insert* R $(S - T)$. If this results in new tuples being inserted into R then Rule 1 will be triggered again. At this point though, Rule 1's condition is $False$ because $R \cap T = \emptyset$. The schedule becomes empty and *execSched* terminates (at the 4th invocation of *execSched*).

Suppose now that a transaction executes a *delete* statement *delete* R e, for some expression e. This update is represented by an initial singleton schedule $(Unknown, Unknown, Unknown, delete\ R\ e)$ being passed to *execSched*. *exec* will execute this action on the database. If this results in one or more tuples being deleted, then Rule 2 will be triggered. Its condition is true, and so *execSched* will be invoked recursively with the new schedule $[(oc, ch, True, insert\ R\ (S -$

$T)]$, where *oc* and *ch* are the current values of *has_occurred_deletion_R* and *change_deletion_R*. Execution then proceeds as above and it is easy to see that *execSched* will terminate after at most 5 invocations. ■

2.1 Discussion

The above rule execution semantics encompass the functionality of statement-level AFTER triggers in the SQL3 standard [23]. SQL3 uses syntactic triggering, so the event parts of its triggers are captured by our *has_occurred* objects. In SQL3, rule conditions are evaluated against the database state that the action will be executed on, rather than the database state in which the rule was triggered. This different binding mode can be captured in our semantics by syntactically modifying each rule so that its condition is encoded which each of its actions and its condition part becomes just *true*. In this way, the actions of any triggered rule will be placed on the schedule by *updateSched* and it is *exec* that will determine whether any give action will actually be applied to the database.

Our semantics also encompass semantic triggering and Deferred rule coupling, which are not supported in SQL3.

The above semantics do not encompass SQL3's BEFORE triggers but could easily be extended to allow these, and the analysis techniques that we describe here would also apply to such rules.

The above semantics assume that all ECA rules are *set-oriented* as opposed to *instance-oriented*. When a set-oriented rule is triggered by some event E and is then scheduled for execution, one copy of its actions is placed on the schedule, each action being tupled up with the current values of the *has_occurred_E* and *change_E* objects. SQL3's statement-level triggers are set-oriented.

When an instance-oriented rule is triggered by some event E, one copy of its actions is placed on the schedule for each $c \in$ *change_E* for which the rule's condition evaluates to $True$. SQL3's row-level triggers are instance-oriented. Our semantics and analysis techniques can be extended to instance-oriented rules and we discuss how in Section 6 below.

Finally, the above semantics assume that there is a total ordering imposed on the set of ECA rules, as in SQL3. It would be straightforward to extend our semantics to allow multiple rules of equal priority: if multiple rules of the same priority were to fire, then their actions would be placed on the schedule in an arbitrary order. It is generally assumed with such rules that they are well-defined, in the sense that the same final database state will result when rule execution terminates irrespective of the order in which such rules are scheduled. A rule set that satisfies this property is termed *confluent* and several techniques have been developed for detecting confluent rule sets [**?**], including using abstract interpretation [6].

3 The Abstract Execution Semantics

The abstract counterpart to $execSched$ is $execSched^*$, given below. $execSched^*$ is identical to $execSched$ except that it operates on abstract databases and abstract schedules, and that at the "leaves" of the computation the functions *eval* and *exec* are replaced by their abstract counterparts *eval** and *exec**. We distinguish abstract types and functions from their concrete counterparts by suffixing their names with a '*'.

An abstract database consists of a set of identifiers and an abstract value associated with each identifier. Generally, these abstract values will be drawn from different domains for different abstractions.

An abstract schedule is a list of abstract updates:

```
Schedule* = List (Update*)
```

Each abstract update is a 4-tuple, consisting of a rule action, and the abstract values of the rule's event and condition queries at the time that the rule fired:

```
Update* = (Value*, Value*, Value*, Action)
```

An abstract database approximates a number of real databases and an abstract schedule a number of real schedules. These possible concretisations are given by a *concretisation function*, *conc*. This *conc* function depends on the specific abstraction and is generally different for different abstractions. We will be seeing some examples in Section 5 below.

Rules, queries and actions are syntactic objects which are common to both the concrete and the abstract semantics. The functions *triggers*, *has_occurred*, *change*, *condition*, *ecq*, *actions*, *mode* and *updateSched* are thus the same in both semantics.

```
execSched* : (DBState*,Schedule*) -> (DBState*,Schedule*)
execSched* (db*,[])  = (db*,[])
execSched* (db*,u*:s*) =
    execSched* o schedRules* (exec* u* db*, u*:s*)

schedRules* : (DBState*,Schedule*) -> (DBState*,Schedule*)
schedRules* (db*,(oc*,ch*,cond*,a):s*) =
    let (db*,pre*,suf*) = foldl schedRule* (db*,[],[]) (triggers a)
    in  (db*,pre* ++ s* ++ suf*)

schedRule* : RuleId -> (DBState*,Schedule*,Schedule*) ->
                       (DBState*,Schedule*,Schedule*)
schedRule* i (db*,pre*,suf*) =
    if (eval* (ecq r) db*) = False
    then (db*,pre*,suf*)
    else updateSched (eval* (has_occurred r) db*,
                      eval* (change r) db*,
                      eval* (condition r) db*,
                      actions r,mode r,pre*,suf*)
```

Note that in general there is no guarantee that *execSched** will terminate and so a criterion is needed for halting it. A simple way is to choose *a priori* a bound on the number of recursive calls of $execSched^*$. Increasing this bound allows more precision and can be balanced against the time available for undertaking the analysis.

An alternative strategy for halting $execSched^*$ is to maintain a history of the $(db^*, head\ s^*)$ arguments passed to $execSched^*$ and to halt if a repeating argument is detected, since in such a case we know that $execSched^*$ would continue to process the same sequence of rules *ad infinitum*.

Before looking at some specific abstractions, we consider first the issue of the correctness of the abstract semantics.

4 Correctness of the Abstract Semantics

Suppose the following two conditions hold for any event-condition query q, abstract update u^*, and abstract database db^*:

$$conc\ (exec^*\ u^*\ db^*) \supseteq \{exec\ u\ db \mid (u, db) \in conc\ (u^*, db^*)\} \quad (1)$$

$$eval^*\ q\ db^* = False \Rightarrow (\forall db \in conc\ db^*\ .\ eval\ q\ db = False) \quad (2)$$

Condition (1) states that the abstract database created by an abstract update $exec^*\ u^*\ db^*$ represents all the databases that could be created by $exec\ u\ db$ for any concretisation of (u, db) of (u^*, db^*) (and possibly other databases as well).

Condition (2) states that if an event-condition query evaluates to $False$ with respect to an abstract database db^*, then it evaluates to $False$ with respect to all concretisations db of db^*.

In combination, these two conditions imply that (a) the abstract database at each iteration of $execSched^*$ represents a superset of the set of concrete databases that may arise at the same iteration of $execSched$, and that (b) the set of rules firing at any iteration of $execSched^*$ is a superset of the set of rules that may fire at the same iteration of $execSched$[2].

Thus, if a particular abstraction satisfies (1) and (2) above, then the abstract execution from any initial abstract state (db^*, s^*) represents a *superset* of the set of real executions from the set of real initial states $(db, s) \in conc\ (db^*, s^*)$. Thus the abstraction is *safe* (i.e. conservative) for all properties P such that if P holds for a set of real executions S then it must also hold for each subset of S. Examples of such properties are the following:

[2] There is a subtlety here in that the formal proof of (a) and (b) (given in [7]) requires that all the concrete executions proceed "in step" with the abstract execution, with the same set of rule actions being scheduled at each step. In reality real rule executions will cause different combinations of rules to fire. However, as discussed in the footnote in Section 2, we can "pad out" real schedules with the actions of rules which have not in fact fired and such actions will have no effect on the database state or on subsequent rule firings. Such"padded out" schedules arising from the concretisation of abstract schedules are thus semantically equivalent to non-padded out real schedules.

– **Termination**: If $execSched^*$ (db^*, s^*) terminates, then so must all $execSched$ (db, s) such that $(db, s) \in conc\ (db^*, s^*)$.
– **Rule Not Fired**: If a rule r does not fire during the execution of $execSched^*$ (db^*, s^*), then it cannot fire during the execution of any $execSched\ (db, s)$ such that $(db, s) \in conc\ (db^*, s^*)$.
– **Rule Sequence Not Fired**: If rules $r_1, \ldots, r_n$ do not fire in this sequence during the execution of $execSched^*$ (db^*, s^*), then this sequence of rules cannot fire during the execution of any $execSched\ (db, s)$ such that $(db, s) \in conc\ (db^*, s^*)$.

An abstraction may additionally satisfy the following pair of stronger conditions:

$$conc\ (exec^*\ u^*\ db^*) = \{exec\ u\ db \mid (u, db) \in conc\ (u^*, db^*)\} \tag{3}$$
$$eval^*\ q\ db^* = False \Leftrightarrow (\forall db \in conc\ db^*\ .\ eval\ q\ db = False) \tag{4}$$

In this case the abstract execution represents exactly the set of real executions from each pair $(db, s) \in conc\ (db^*, s^*)$ and the abstraction is both safe and also *precise*.

Additional properties can be analysed with a precise abstraction. In particular, for any property P as above, if $\neg P$ holds for $execSched^*$ (db^*, s^*) then $\neg P$ must hold for some $execSched\ (db, s)$ such that $(db, s) \in conc\ (db^*, s^*)$. Examples of $\neg P$ are:

– **Non-termination**: If some iteration of $execSched^*$ (db^*, s^*) encounters the same abstract database and abstract update at the head of the schedule as an earlier iteration, then $execSched^*$ will fail to terminate. With a precise abstraction this means that some $execSched\ (db, s)$ such that $(db, s) \in conc\ (db^*, s^*)$ will also pass twice through the same concrete database and concrete update at the head of the schedule and hence will also fail to terminate.
– **Rule Fired**: If a rule r fires during the execution of $execSched^*$ (db^*, s^*), then it must fire during the execution of some $execSched\ (db, s)$ such that $(db, s) \in conc\ (db^*, s^*)$.
– **Rule Sequence Fired**: If rules $r_1, \ldots, r_n$ fire in this sequence during the execution of $execSched^*$ (db^*, s^*), then this same sequence of rules must fire during the execution of some $execSched\ (db, s)$ such that $(db, s) \in conc\ (db^*, s^*)$.

4.1 Discussion

The information provided directly by $execSched^*$ is for concretisations of the initial abstract database and initial abstract schedule. A broader question is does the analysis property hold for *any* initial database and initial finite schedule?

To capture all possible initial databases we can use as the initial abstract database the "least informative" element of $DBState^*$: let us denote it by db_0^*.

This abstract database maps each database object Obj to an abstract value Obj_0 which concretises to all possible valid real values for Obj.

To capture all possible initial finite schedules, let us consider first properties P as above i.e. such that if P holds for a set of real executions S then it must also hold for each subset of S. Let us also suppose first that all rules have Immediate coupling mode.

In this case, it is sufficient to run $execSched^*$ *on each possible singleton schedule* $[(Unknown, Unknown, Unknown, a)]$ *for each action* a *that may trigger one or more rules in the rule set.*

Properties P as above hold for all initial finite schedules if they hold for all such singleton schedules. To see why this is, consider any initial finite abstract schedule

$$s^* = [u_1^*, u_2^*, \dots, u_n^*]$$

Running $execSched^*$ (db_0^*, s^*) would result in the following sequence of abstract updates being applied to db_0^*, where each T_i is the (possibly empty) sequence of abstract updates arising from the execution of rules that have fired as a result of executing u_i^*:

$$u_1^*, T_1, u_2^*, T_2, \dots, u_n^*, T_n$$

If P holds for $execSched^*$ $(db_0^*, [u_i^*])$ then it must hold for T_i, since db_0^* is the least informative abstract database state and hence $execSched^*$ $(db_0^*, [u_i^*])$ represents a superset of the set of real executions represented by T_i.

Thus, if P holds for all $execSched^*$ $(db_0^*, [u_1^*])$, ..., $execSched^*$ $(db_0^*, [u_n^*])$ it must hold for all $T_1, \dots, T_n$, and thus must hold for $execSched^*$ (db_0^*, s^*). Thus P must hold for all $execSched$ (db, s) such that $(db, s) \in conc$ (db_0^*, s^*).

Finally, for any initial finite abstract schedule $[u_1^*, u_2^*, \dots, u_n^*]$, it is sufficient to run $execSched^*$ $(db_0^*, [u_i^*])$ only for those updates u_i^* whose action may trigger one or more rules, since for all other updates u_j^* we know that T_j will be empty. Moreover, the behaviour of any such update $u_i^* = (oc^*, ch^*, cond^*, a)$, is encompassed by running $execSched^*$ $(db_0^*, [(Unknown, Unknown, Unknown, a)]$ since these $Unknown$ truth values for the initial event and condition queries incorporate the possible truth values of oc^*, ch^* and $cond^*$.

Let us consider now the case where some rules may have Deferred coupling mode. In this case, the execution of each u_i^* may result both in an "immediate" sub-transaction such as T_i above which is scheduled immediately after u_i^* and also in a "deferred" sub-transaction which is appended to the current schedule. The abstract database state that deferred sub-transactions encounter during the execution of $execSched^*$ (db_0^*, s^*) may thus be different from the abstract database state that they encounter during the execution of each individual $execSched^*$ $(db_0^*, [u_i^*])$. Thus, for a safe analysis, the abstract database state must be reinitialised to db_0^* at the start of each deferred sub-transaction.

Let $execSched^{*m}$ be $execSched^*$ modified so as to reinitialise the abstract database state to db_0^* at the start of each deferred sub-transaction. Then, properties P as above hold for all initial finite schedules if they hold for all singleton schedules with the modified abstract semantics $execSched^{*m}$.

For properties $\neg P$ and precise abstractions, it is not sufficient to execute *execSched** on singleton schedules. For example, a rule set may in general be non-terminating for some singleton schedule $[u_i^*]$ but for the specific databases that will arise as a result of first executing the schedule $[u_1^*, \ldots, u_{i-1}^*]$ it may actually always terminate. Similarly, a rule or rule sequence may in general fire during the execution of $[u_i^*]$ but for the specific databases that will arise as a result of first executing $[u_1^*, \ldots, u_{i-1}^*]$ the rule or rule sequence may actually never fire.

Hence, such properties can only be analysed for concretisations of a *specific* abstract schedule. This can none-the-less be useful in a *dynamic* setting, where there is indeed a specific current schedule that we wish to analyse. This is the setting in [9], where we discuss dynamic termination analysis of ECA rules. We refer the reader to this paper for further discussion of dynamic ECA rule analysis and how our abstract interpretation approach compares with other approaches.

As a final remark, we note that *execSched** as defined above assumes that the ECA rules are totally ordered. If this is not the case and at some point multiple rules of the same priority fire, then the abstract execution would need to branch into a separate execution pathway for each possible permutation of these rules. We specified such a branching semantics in [6] and we refer the reader to that paper for details. We consider an example of a branching abstract execution in Section 6 below.

5 Some Abstractions

In [7], we discussed the use of abstract interpretation for rule termination analysis and we identified three specific abstractions. Here we review again these abstractions, and note that they can be used more generally for analysing other rule properties, as discussed above.

The first abstraction we presented in [7] is precise for rule languages for which query satisfiability is decidable. This abstraction maintains an abstract database state consisting of an expression for each database object. These expressions are expanded' by *exec** as rule actions are applied to them. Satisfiability of a rule's event-condition query determines whether the rule may fire in the current abstract database state. Thus, *eval** is a query satisfiability test which, if satisfiability is decidable for the particular rule language, will return either $True$ or $False$.

If satisfiability is not decidable, *eval** may in addition return $Unknown$. In this case we get a safe but non-precise abstraction. This is because an $Unknown$ truth value for an event-condition query is treated in the same way as $True$ by *execSched** i.e. if it is not known whether a rule would fire, the abstract semantics assumes that it *will* fire in some real execution whereas it may in fact not fire in any real execution.

Example 2. Consider again the rules of Example 1. The following is a trace of the abstract execution with an initial singleton schedule consisting of the action *insert* R ?. Here, "?" is an unused identifier denoting the set of tuples being inserted into R. As with the other database objects, its initial abstract value is $?_0$ which concretises to all possible valid real values. We do not show it below as it cannot be changed by any of the rules. "Iteration i" means the i^{th} invocation of $execSched^*$. For simplicity we show only the actions on the schedule, not the abstract values of the event and condition queries:

Iteration	Abstract database state			Schedule
	R	S	T	
1	R_0	S_0	T_0	$[insert\ R\ ?]$
2	$R_0 \cup ?_0$	S_0	T_0	$[delete\ R\ T]$
3	$(R_0 \cup ?_0) - T_0$	S_0	T_0	$[insert\ R\ (S-T)]$
4	$((R_0 \cup ?_0) - T_0) \cup (S_0 - T_0)$	S_0	T_0	$[]$

At iteration 1, Rule 1 is triggered because the expression $? - R$ is satisfiable and tuples may indeed have been inserted into R. The condition of Rule 1 is also satisfiable, and so the action of Rule 1 is placed on the schedule. After *delete* $R\ T$ is applied to R at iteration 2, Rule 2 is triggered because the expression $R \cap T$, which we term Rule 1's *delta query*, is satisfiable at this stage and so tuples may indeed have been deleted from R. The condition of Rule 2 is trivially true, and so its action is placed on the schedule. After the update *insert* $R\ (S-T)$ is applied to R at iteration 3, Rule 1 is triggered because Rule 2's delta query, $(S-T)-R$, is satisfiable at this stage and so tuples may indeed have been inserted into R. However, the condition of Rule 1 now expands to $(((R_0 \cup ?_0) - T_0) \cup (S_0 - T_0) \cap S_0) \times (((R_0 \cup ?_0) - T_0) \cup (S_0 - T_0) \cap T_0)$, which is not satisfiable because the second argument of this cartesian product is not satisfiable. The schedule thus becomes empty and $execSched^*$ terminates at iteration 4.

Similarly, the following is a trace of the abstract execution with an initial singleton schedule consisting of the action *delete* R ?, and we see that this terminates at iteration 5:

Iteration	Abstract database state			Schedule
	R	S	T	
1	R_0	S_0	T_0	$[delete\ R\ ?]$
2	$R_0 - ?_0$	S_0	T_0	$[insert\ R\ (S-T)]$
3	$(R_0 - ?_0) \cup (S_0 - T_0)$	S_0	T_0	$[delete\ R\ T]$
4	$((R_0 - ?_0) \cup (S_0 - T_0)) - T_0$	S_0	T_0	$[insert\ R\ (S-T)]$
5	$(((R_0 - ?_0) \cup (S_0 - T_0)) - T_0) \cup (S_0 - T_0)$	S_0	T_0	$[]$

Thus, we can conclude that this rule set is definitely a terminating one. ■

Testing satisfiability is generally expensive, and the second and third abstractions we presented in [7] undertake a cheaper and less precise analysis by maintaining in the abstract database only the truth values of specific queries, and not the entire history of updates that have have been applied to the database objects. Both these abstractions are safe. They are also precise provided $eval^*$ returns only $True$ or $False$, not if $eval^*$ returns $Unknown$.

With the second abstraction, the abstract database consists of an identifier corresponding to each event query and each condition query in the rule set. These identifiers are assigned values from the 2-valued domain $\{False, Unknown\}$. $conc(db^*)$ returns all databases db satisfying the following conditions:

– the values assigned to database objects in db are consistent with the abstract values assigned to the event and condition queries in db^*;
– if the abstract value of an event-condition query, q, is $False$ then its concrete value must also be $False$ i.e.
$eval^*\ q\ db^* = False \Rightarrow eval\ q\ db = False$.

Given an event-condition query q and abstract database db^*, $eval^*\ q\ db^*$ looks up in db^* the current truth value of the event query and the condition query and returns their logical conjunction.

$exec^*\ (oc^*, ch^*, c^*, a)\ db^*$ updates the abstract database as follows. It first sets the values of all event queries that may be affected by a to $Unknown$ and the values of all other event queries to $False$. It then uses a function $infer\_cond\ q\ a$ to infer a new abstract value for each condition query q. This returns:

(a) $Unknown$ if the condition may be true after the action executes;
(b) $False$ if the condition can never be true after the action executes.

A method such as the propagation algorithm described in [13] can be used to determine the effect of actions on conditions.

Example 3. Consider again the rules of Example 1. The following is a trace of the abstract execution with an initial singleton schedule consisting of the action $insert\ R\ ?$, where e_i denotes the event query of rule i and c_i the condition query of rule i:

Iteration	Abstract database state				Schedule
	e_1	c_1	e_2	c_2	
1	U	U	U	U	$[insert\ R\ ?]$
2	U	U	F	U	$[delete\ R\ T]$
3	F	F	U	U	$[insert\ R\ (S-T)]$
4	U	U	F	U	$[delete\ R\ T]$

At iteration 3, Rule 1's condition becomes $False$, since its action will always falsify $R \cap T$. At iteration 4 the execution of Rule 2's action makes Rule 1's condition $Unknown$ again because only the overall truth values of conditions are recorded, so $infer\_cond$ cannot make use of the fact that $R \cap T$ was previously $False$. The fourth state is thus a repetition of the second, so $execSched^*$ is halted and possible rule non-termination is concluded. ■

Example 4. Suppose now we change the condition of rule 1 to be just $R \cap T$. Then the following is a trace of the abstract execution with an initial singleton schedule consisting of the action $insert\ R\ ?$:

Iteration	Abstract database state				Schedule
	e_1	c_1	e_2	c_2	
1	U	U	U	U	$[insert\ R\ ?]$
2	U	U	F	U	$[delete\ R\ T]$
3	F	F	U	U	$[insert\ R\ (S-T)]$
4	U	F	F	U	$[]$

Now rule 2's action can no longer alter the truth value of rule 1's condition and *execSched** terminates at iteration 4, since rule 1's condition is $False$.

Similarly, the following is a trace of the abstract execution with an initial singleton schedule consisting of the action *delete R ?*:

Iteration	Abstract database state				Schedule
	e_1	c_1	e_2	c_2	
1	U	U	U	U	$[delete\ R\ ?]$
2	F	U	U	U	$[insert\ R\ (S-T)]$
3	U	U	F	U	$[delete\ R\ T]$
4	F	F	U	U	$[insert\ R\ (S-T)]$
5	U	F	F	U	$[]$

Again *execSched** terminates. Thus, this cheaper analysis can be used to conclude that the modified rule set is a terminating one. ■

The above function $infer\_cond$ uses only the syntax of conditions and actions to deduce new values for the event and condition queries. The third abstraction we describe in [7] incorporates a more refined inferencing method which makes use of information about the previous values of queries. Its increased precision is reflected by the inclusion of the value $True$ in the abstract domain, as well as $Unknown$ and $False$.

This abstraction assumes that for each rule action a_i there is a set of events $E_1, \ldots, E_n$ which the execution of a_i may cause to occur, and that for each such E_j we can write a query, $q\_a_i\_E_j$, which defines the value of $change\_E_j$ resulting from action a. We term these queries the actions' *delta queries*. For example, in a relational setting, an action *insert R q* can cause only event *insertion_R* to occur and the delta query is $q - R$; similarly, an action *delete R q* can cause only event *deletion_R* to occur and the delta query is $R \cap q$; updates to relations are a bit more complex, but the same principle applies.

With this abstraction, the abstract database again consists of an identifier corresponding to each event and condition query in the rule set. However, there is now also an identifier corresponding to each delta query. These identifiers are all assigned values from the 3-valued domain $\{True, False, Unknown\}$ in the abstract database.

For this abstraction, $conc(db^*)$ consists of all the databases db satisfying the following conditions:

- the values assigned to database objects in db are consistent with the abstract values assigned to the event, condition and delta queries in db^*;
- if the abstract value of an event-condition query, q, is $False$ then its concrete value must also be $False$ i.e.
$eval^*\ q\ db^* = False \Rightarrow eval\ q\ db = False$.
- if the abstract value of an event-condition query, q, is $True$ then its concrete value must also also $True$ i.e.
$eval^*\ q\ db^* = True \Rightarrow eval\ q\ db = True$.

Given an event-condition query q and abstract database db^*, $eval^*\ q\ db^*$ again looks up in db^* the current truth values of the event query and the condition query and returns their logical conjunction.

$exec^*\ a^*\ db^*$ updates the query values in the abstract database by using incremental propagation techniques which perform query rewriting to determine if updates can have an effect on an expression [28, 18]. We refer the reader to [7, 9] for full details of our inferencing method.

This third abstraction is again not powerful enough to detect termination of the original rule set from Example 1. But it can detect more cases of termination than our second abstraction because it can make use of more precise information about the abstract database state by distinguishing between $True$ and $Unknown$ query values. We give an example that uses this third abstraction in Section 6 below.

Distinguishing between $True$ and $Unknown$ query values also gives more precision in a dynamic setting by allowing the current real database state to be described more accurately. In [9], we describe the practical use of this third abstraction for dynamic analysis of rule termination. In [8], we describe how it can also be used for optimising ECA rule execution, both in a static and in a dynamic setting.

6 Handling instance-oriented rules

So far we have assumed that all rules are set-oriented ones. Our rule execution semantics and analysis techniques can be extended to apply also to instance-oriented rules and we now discuss how.

The concrete semantics, *execSched*, is easily extended to also allow for instance-oriented rules. If some rule r has fired and it is an instance-oriented rule then for each action a of r, *updateSched* places onto the schedule a list of updates $[(oc, \{c_1\}, cond, a)]$, $[(oc, \{c_2\}, cond, a)]$, $\ldots$ where oc is the current value of *has_occurred* r, $\{c_1, c_2, \ldots\}$ the current value of *change* r and *cond* the current value of *condition* r.

The c_i are placed onto the schedule in an arbitrary order since it is usually assumed that, if used, instance-oriented rules are well-defined in the sense that the same final database state will result when rule execution terminates irrespective of the order in which individual changes arising from instance-level rules are scheduled.

A condition guaranteeing this is that for all pairs c_i, c_j, the queries that are evaluated during $execSched(db, [(oc, \{c_i\}, cond, a)])$ are independent of the database updates that take place during $execSched(db, [(oc, \{c_j\}, cond, a)])$. Techniques for analysing independence of queries from updates can be used to determine whether this is the case for a given set of rules [**?**].

The question is how should $execSched^*$ be modified to analyse instance-oriented rules, and in particular how many copies of each rule action should be placed onto the abstract schedule? Assuming the above condition holds, then it is sufficient to place just one copy of the rule action onto the abstract schedule i.e. $execSched^*$ remains exactly the same as for set-oriented rules. $execSched^*$ will represent by one abstract execution pathway the multiple independent execution paths that can occur when instance-oriented rules are present.

Example 5. This example is taken from [16]. An object-oriented active database contains a class *Account* which has attributes $overdraft$, *rate*, *capacity*, and *type*. There are four instance-oriented ECA rules, all with Immediate coupling mode. Rules 2 and 3 have the same priority, as have rules 1 and 4. Rule 1 can trigger rules 2 and 3, rule 2 can trigger rules 1 and 4, and rules 3 and 4 do not trigger any other rule.

In rule 1, the event $update(Account.overdraft)$ is synonymous with our earlier terminology $has\_occurred\_update\_of\_Account.overdraft$, and this is syntactic triggering. The same applies to the other rules. In rule 1, the *changed* object is synonymous with our terminology $change\_update\_of\_Account.overdraft$. Since the rule is an instance-level one, we assume here that *changed* returns a single *Account* object to which the update has been applied. A similar syntax is used in the other rules.

Rule 1: on $update(Account.overdraft)$
if $changed.rate = 8$
do $update(changed.capacity, 6000)$

Rule 2: on $update(Account.capacity)$
if $changed.rate = 4$
do $update(changed.overdraft, 20)$

Rule 3: on $update(Account.capacity)$
if $changed.type = stocks\_account$
do $update(changed.rate, 4)$

Rule 4: on $update(Account.overdraft)$
if $changed.type = standard\_account$
do $update(changed.rate, 8)$

Two kinds of actions can trigger this rule set: $update(Account.overdraft, ?)$ and $update(Account.capacity, ?)$.

Let us consider first an initial singleton abstract schedule consisting of the action $a_0 = update(Account.overdraft, ?)$.

We give below the abstract execution trace, assuming the third abstraction from the previous section. We do not show the delta query values as these are all $True$ throughout the execution (because in this rule language *update* actions are always deemed to happen, irrespective of whether they change the actual value of the attribute).

At iteration 2, the abstract execution branches because both rules 1 and 4 are triggered and we need to consider both orders in which their actions may be scheduled for execution — traces 2^1 - 5^1 and 2^2 - 5^2 below.

Iteration	Abstract database state								Schedule
	e_1	c_1	e_2	c_2	e_3	c_3	e_4	c_4	
1	F	U	F	U	F	U	F	U	$[a_0]$
2^1	T	T	F	F	F	U	T	U	$[a_1, a_4]$
3^1	F	T	T	F	T	U	F	U	$[a_3, a_4]$
4^1	F	U	F	U	F	U	F	U	$[a_4]$
5^1	F	U	F	U	F	U	F	U	$[]$
2^2	T	T	F	F	F	U	T	U	$[a_4, a_1]$
3^2	F	T	F	F	F	U	F	U	$[a_1]$
4^2	F	T	T	F	T	U	F	U	$[a_3]$
5^2	F	U	F	U	F	U	F	U	$[]$

At iteration 2^1, c_1 must be true since a_1 has just been placed on the schedule. Thus c_2 must be false. So although a_1 triggers both rules 2 and 3, only a_3 is placed on the schedule. c_3 is unknown at this point, so in a real execution a_3 may or may not occur, depending on whether the object has $type = stocks\_account$. So at iteration 4^1, the values of c_1 and c_2 have become unknown again. a_3 triggers no rules, so a_4 has come to the head of the schedule at iteration 4^1. a_4 triggers no rules, and this execution branch terminates at iteration 5^1.

At iteration 2^2, c_1 is again true and c_2 false. Execution of a_4 does not change these values and triggers no rules. Thus at iteration 3^2, a_1 has come to the head of the schedule. Execution of a_1 triggers both rules 2 and 3, but because c_2 is false only a_3 is placed on the schedule. This execution branch thus terminates at iteration 5^2.

Similarly, this is the abstract execution trace with an initial abstract schedule consisting of the action $a_0 = update(Account.capacity, ?)$. At iteration 3^2 the abstract execution branches a second time, into $4^{2,1}$ and $4^{2,2}$, depending on the order that a_1 and a_4 are scheduled after a_2 executes. We notice that these states are exactly the same as 2^1 and 2^2 above, and so these two branches would proceed exactly as 2^1 - 5^1 and 2^2 - 5^2 above.

Iteration	Abstract database state								Schedule
	e_1	c_1	e_2	c_2	e_3	c_3	e_4	c_4	
1	F	U	F	U	F	U	F	U	$[a_0]$
2^1	F	F	T	T	T	U	F	U	$[a_2, a_3]$
3^1	T	F	F	T	F	U	T	U	$[a_4, a_3]$
4^1	F	U	F	U	F	U	F	U	$[a_3]$
5^1	F	U	F	U	F	U	F	U	$[]$
2^2	F	F	T	T	T	U	F	U	$[a_3, a_2]$
3^2	F	U	F	U	F	U	F	U	$[a_2]$
$4^{2,1}$	T	T	F	F	F	U	T	U	$[a_1, a_4]$
$4^{2,2}$	T	T	F	F	F	U	T	U	$[a_4, a_1]$

Thus, we can conclude that this rule set is a terminating one. ■

7 Other approaches to ECA rule analysis

Many previous methods for ECA rule analysis have been based on analysing the triggering and activation relationships between pairs of rules. In these approaches, a rule r_i *may trigger* a rule r_j if execution of the action of r_i may generate an event which causes the event part of r_j to become $True$. A rule r_i *may activate* another rule r_j if r_j's condition may be changed from $False$ to $True$ after the execution of r_i's action. A rule r_i *may activate* itself if its condition may be $True$ after the execution of its action.

A *triggering graph* [2] represents each rule as a vertex, and there is a directed arc from a vertex r_i to a vertex r_j if r_i *may trigger* r_j. Acyclicity of the triggering graph implies definite termination of rule execution. Triggering graphs can also be used for deriving rule reachability information. An *activation graph* [13] also represents rules as vertices. In this case there is a directed arc from a vertex r_i to a vertex r_j if r_i *may activate* r_j. Acyclicity of this graph also implies definite termination of rule execution.

Previous methods of ECA rule analysis have often been closely linked to specific rule languages. In contrast, the abstract interpretation framework that we have described here does not assume any particular rule language. Moreover, our rule execution semantics can simulate most standard features of active database systems and research prototypes.

[21] proposed using satisfiability tests between pairs of rules to refine triggering graphs. This can be interpreted as a less sophisticated version of the first abstraction we described in Section 5. For example, in Example 1, if we only considered the relationships between pairs of rules, we would derive the information that rules 1 and 2 can cause each other to fire. By itself, this information does not allow us to make any conclusions about the rules' termination behaviour. However, we showed in Example 1 that by doing satisfiability testing on accumulated "history" expressions, we can conclude that this rule set is indeed a terminating one.

The second abstraction we described in Section 5 does focus on pairwise relationships between rules and so is similar to graph-based analysis techniques based on triggering and activation relationships between rules [2, 13, 12, 21, 15]. Our abstraction is not identical to these, however, since in their pure form graph-based techniques do not "execute" the rule set and so have no notion of control flow. This can cause loss of precision for rule sets where the order of execution prohibits certain activation states of conditions. Extensions to these techniques are certainly possible to recognise such situations, but they do not arise naturally from the specifications of the techniques themselves.

Couchot gives in [16] a detailed review of the main methods for rule termination analysis and discusses their limitations in capturing control flow. He proposes an extension of the Refined Triggering Graph method for object-oriented databases [21], and this extension can detect definite termination of the rule set in Example 5. In this chapter, we have shown how our abstract interpretation can also be used to effectively analyse such rule sets, due to its inherent incorporation of control flow information, and moreover our abstract interpretation framework achieves this in a language-independent setting.

8 Analysing XML ECA rules

We conclude this chapter by describing how our framework can also be applied to analysing ECA rules on XML databases. In [10, 11] we describe an ECA rule language for XML which uses the XPath and XQuery languages to specify events, conditions and actions.

The event part of an ECA rule in this language is an expression of the form

$$\texttt{INSERT}\ e$$

or

$$\texttt{DELETE}\ e$$

where e is an XPath expression which evaluates to a set of nodes. The rule is triggered if this set of nodes includes any node in a new sub-document in the case of an insertion, or in a deleted sub-document in the case of a deletion. We note that this is semantic triggering i.e. rules are triggered only if changes occur to the database

Thus, using our earlier terminology, there is an identifier $change\_INSERT\_e$ for each event of the form `INSERT` e appearing in the rule set, and an identifier $change\_DELETE\_e$, for each event of the form `DELETE` e appearing in the rule set. Since only semantic triggering is supported in this rule language, we can ignore the *has_occurred* identifiers.

A system-defined variable, `$delta`, is available for use within the condition and action parts of the rule. Its set of instantiations is the set of new or deleted nodes returned by e. Thus, `$delta` is synonymous with the *change* identifier corresponding to the rule's event part.

The condition part of an ECA rule is either the constant `TRUE`, or one or more XPath expressions connected by the boolean connectives `and`, `or`, `not`.

This condition is evaluated on each XML document in the database which has been changed by an event of the form specified in the rule's event part. If the condition references the `$delta` variable, it is evaluated once for each instantiation of `$delta` — this is instance-oriented triggering. Otherwise, the condition is evaluated just once for each document — set-oriented triggering.

The action part of an ECA rule is a sequence of one or more actions:

$$action_1;\ \ldots;\ action_n$$

These actions are executed on each XML document which has been changed by an event of the form specified in the rule's event part and for which the rule's condition query evaluates to *True*. An ECA rule fires if this set of documents is non-empty.

Each $action_i$ above is an expression of the form

$$\texttt{INSERT}\ r\ \texttt{BELOW}\ e$$

or

$$\texttt{DELETE}\ e$$

where r is an XQuery expression and e an XPath expression[3].

In an `INSERT` action, the expression e specifies the set of nodes immediately below which new sub-document(s) will be inserted. These sub-documents are specified by the expression r.

If there is no occurrence of `$delta` in e or r, the action is executed at most once on each document each time the rule fires (set-oriented triggering). If there is an occurrence of `$delta` in e or r, the action is executed once for each possible instantiation of `$delta` for which the rule's condition query evaluates to *True* (instance-oriented triggering).

In a `DELETE` action, the expression e specifies the set of nodes which will be deleted (together with their sub-documents). Again, e may reference `$delta` and both set-oriented and instance-oriented triggering are possible.

8.1 Analysing Rule Behaviour

In [10, 11] we describe techniques for inferring triggering and activation relationships between pairs of our XML ECA rules (with the techniques described in [11] superceding those in the earlier paper [10]). As with the inferencing techniques that we used in the second and third abstractions of Section 5, these techniques for inferring triggering and activation relationships between XML ECA rules can be "plugged into" our abstract interpretation framework in order to analyse such rules. To illustrate, consider the following pair of rules:

```
Rule 1: on INSERT /a/*
        if /a/b and /a/c
```

[3] This is a slight simplification of the syntax of actions and we refer the reader to [11] for the full syntax.

```
        do DELETE /a/b

Rule 2: on DELETE /a/*
        if TRUE
        do INSERT <c/> BELOW /a
```

Let the abstract database state consist of a truth value *False* or *Unknown* for each rule's event query and also for each XPath expression appearing within the rules' condition parts. Consider first a singleton schedule consisting of the action a_0 = `INSERT ? BELOW /a`.

A trace of the abstract execution is as follows, where for $i = 1, 2$, e_i and a_i respectively denote the event query and action of rule i, $c_{1,1}$ is the first conjunct of rule 1's condition and $c_{1,2}$ the second conjunct of rule 1's condition. Rule 2's condition is trivially *True* throughout so we don't show it:

Iteration	e_1	$c_{1,1}$	$c_{1,2}$	e_2	Schedule
1	U	U	U	U	$[a_0]$
2	U	U	U	F	$[a_1]$
3	F	F	U	U	$[a_2]$
4	U	F	U	F	$[]$

Abstract execution of a_1 at iteration 2 causes $c_{1,1}$ to become *False*, using the analysis techniques from [11]. After the execution of a_2 at iteration 3, $c_{1,1}$ remains *False* because a_2 cannot alter it. The falsity of c_1 means that rule 1 cannot fire at this point and the abstract execution terminates.

Similarly, given an initial singleton schedule consisting of the action a_0 = `DELETE /a/?`, a trace of the abstract execution of the rules is as follows:

Iteration	e_1	$c_{1,1}$	$c_{1,2}$	e_2	Schedule
1	U	U	U	U	$[a_0]$
2	F	U	U	U	$[a_2]$
3	U	U	U	F	$[a_1]$
4	F	F	U	U	$[a_2]$
5	U	F	U	F	$[]$

This time, execution of a_1 at iteration 3 causes $c_{1,1}$ to become *False*. Again it remains *False* at iteration 4, and so rule 1 cannot fire at this point. Overall, we therefore conclude that this is a terminating rule set. ■

9 Concluding Remarks

ECA rules have the potential to improve the robustness and maintainability of database applications by abstracting common reactive functionality and placing it under the management of the DBMS. However, it is recognised that the lack of sufficiently powerful analysis techniques for ECA rules is still an impediment to their wider use. Our recent work has shown that abstract interpretation is a promising analysis technique for ECA rules and can readily be deployed within practical rule analysis tools.

A major feature of our abstract interpretation approach is its generality: it can be applied to any active DBMS whose rule execution semantics have been specified as a function that takes an initial database and an initial list of rule actions to be executed, and returns the resulting database when there are no more actions left to execute. Different abstraction methods can be employed within our framework, and the correctness of a particular method is established by proving a small set of generic requirements. The framework can be used both statically and dynamically.

In other recent work [8] we have also been investigating applying both abstract interpretation and *partial evaluation* [20, 19] to functional rule execution semantics in order to derive a number of rule optimisations. In particular, we use abstract rule execution from the current abstract database state in order to infer information about possible sequences of rule actions that will be executed in the future. Partial evaluation is then applied to the rule execution semantics in order to produced a specialised version for each of these possible sequences of actions. This gives the opportunity to optimise rule execution for each particular sequence of actions, for example by abstracting common sub-queries from the set of event-condition queries that will be evaluated.

References

1. S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.
2. A. Aiken, J. Widom, and J. M. Hellerstein. Static analysis techniques for predicting the behavior of active database rules. *ACM TODS*, 20(1):3–41, 1995.
3. J. Bailey. *On the Foundations of Termination Analysis of Active Database Rules.* PhD thesis, Department of Computer Science, University of Melbourne, September 1997.
4. J. Bailey, L. Crnogorac, K. Ramamohanarao, and H. Søndergaard. Abstract interpretation of active rules and its use in termination analysis. In *Proc. ICDT'97, LNCS 1186*, pages 188–202, Delphi, Greece, 1997.
5. J. Bailey, G. Dong, and K. Ramamohanarao. Decidability and undecidability results for the termination problem of active database rules. In *Proc. ACM PODS'98*, pages 264–273, Seattle, Washington, 1998.
6. J. Bailey and A. Poulovassilis. Abstract interpretation for termination analysis in functional active databases. *Journal of Intelligent Information Systems*, 12(2/3):243–273, 1999.
7. J. Bailey and A. Poulovassilis. An abstract interpretation framework for termination analysis of active rules. In *Proc. 7th Int. Workshop on Database Programming Languages (DBPL-7), LNCS 1949*, pages 249–266, Scotland, 1999.
8. J. Bailey, A. Poulovassilis, and C. Courtenage. Optimising active database rules by partial evaluation and abstract interpretation. In *Proc. 8th Int. Workshop on Database Programming Languages (DBPL-8), LNCS 2397*, pages 300–317, Rome, 2001.

9. J. Bailey, A. Poulovassilis, and P. Newson. A dynamic approach to termination analysis for active database rules. In *Proc. 1st Int. Conf. on Computational Logic (DOOD'2000 stream), LNCS 1861*, pages 1106–1120, London, 2000.
10. J. Bailey, A. Poulovassilis, and P.T. Wood. Analysis and optimisation of event-condition-action rules on XML. *Computer Networks*, 39(3):239–259, 2002.
11. J. Bailey, A. Poulovassilis, and P.T. Wood. An event-condition-action language for XML. In *Proc. WWW 2002*, pages 486–495, Hawaii, 2002.
12. E. Baralis, S. Ceri, and S. Paraboschi. Compile-time and runtime analysis of active behaviors. *IEEE Transactions on Knowledge and Data Engineering*, 10(3):353–370, 1998.
13. E. Baralis and J. Widom. An algebraic approach to rule analysis in expert database systems. In *Proc. VLDB'94*, pages 475–486, Santiago, Chile, 1994.
14. S. Ceri, R. Cochrane, and J. Widom. Practical applications of triggers and constraints: Success and lingering issues. In *Proc. VLDB'2000*, pages 254–262, Cairo, 2000.
15. S. Ceri and P. Fraternalli. *Designing Database Applications with Objects and Rules: The IDEA Methodology*. Addison-Wesley, 1997.
16. A. Couchot. Improving the refined triggering graph method for active rules termination analysis. In *Proc. BNCOD 2002, LNCS 2405*, pages 114–133, Sheffield, 2002.
17. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic Programming*, 13(2&3):103–179, 1992.
18. T. Griffin, L. Libkin, and H. Trickey. A correction to "Incremental recomputation of active relational expressions" by Qian and Wiederhold. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):508–511, 1997.
19. N Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, 1996.
20. N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
21. A. Karadimce and S. Urban. Refined triggering graphs: A logic based approach to termination analysis in an active object-oriented database. In *Proc. ICDE'96*, pages 384–391, New Orleans, 1996.
22. A. Kotz-Dittrich and E. Simon. Active database systems: Expectations, Commercial Experience and Beyond. In Paton [26], pages 367–404.
23. K. Kulkarni, N. Mattos, and R. Cochrane. Active database features in SQL3. In Paton [26], pages 197–219.
24. K. Marriott, H. Søndergaard, and N. D. Jones. Denotational abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 16(3):607–648, 1994.
25. F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
26. N. Paton, editor. *Active Rules in Database Systems*. Springer-Verlag, 1999.
27. A. Poulovassilis, S. Reddi, and C. Small. A formal semantics for an active functional DBPL. *Journal of Intelligent Information Systems*, 7(2):151–172, 1996.
28. X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, 1991.
29. S. Reddi, A. Poulovassilis, and C. Small. Extending a functional DBPL with ECA-rules. In *Proc. 2nd Int. Workshop on Rules in Database Systems (RIDS'95), LNCS 985*, pages 101–115, Athens, 1995.

30. J. Widom and S. Ceri. *Active Database Systems.* Morgan-Kaufmann, San Mateo, California, 1995.