An Abstract Interpretation Framework for Termination Analysis of Active Rules

James Bailey and Alexandra Poulovassilis

Dept. of Computer Science, Birkbeck College, University of London,
Malet Street, London WC1E 7HX. {james,ap}@dcs.bbk.ac.uk

Abstract. A crucial requirement for active databases is the ability to analyse the behaviour of the active rules. A particularly important type of analysis is termination analysis. We define a framework for modelling the execution of active rules, based on abstract interpretation. Specific methods for termination analysis are modelled as specific approximations within the framework. The correctness of a method can be established by proving two generic requirements provided by the framework. This affords the opportunity to compare and verify existing methods for termination analysis of active rules, and also to develop new ones.

1 Introduction

Active databases are capable of reacting automatically to state changes without user intervention by supporting *active rules* of the form "on *event* if *condition* do *action*". An important behavioural property of active rules is that of *termination* and many methods for analysing termination have been proposed. However, in many cases it is not clear whether a method developed for one active database system would be correct if applied to another. 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 a useful tool in the analysis of imperative, functional and logic programs [1, 10, 16]. In this paper we apply it to developing, and proving the correctness of, techniques for termination analysis of active rules. We develop a general framework for relating real and abstract rule execution. Specific termination analysis techniques are modelled in this framework by defining specific approximations. The correctness of a technique is established by proving two generic requirements provided by the framework. These requirements relate only to individual rules, *not* to recursive firings of rules. The class of active database systems which the framework can model is broad and does not assume a particular rule definition language. The formal nature of the framework allows a smooth adoption of previous work in query satisfiability, incremental evaluation techniques, and approximation techniques. We illustrate the use of the framework by developing two abstractions for static termination analysis of active rules and a third for dynamic termination analysis.

Section 2 presents the framework. This involves defining the concrete execution semantics and their abstract counterpart. Section 3 describes how the framework can be applied to a particular rule language, rule scheduling semantics, and approximation. Sections 4, 5 and 6 give three example applications of the framework. Section 7 discusses and compares these three techniques. Section 8 compares our approach to rule termination analysis with related work. We give our conclusions and directions for future research in Section 9.

2 The Framework

We use a typed functional metalanguage to specify both the actual execution semantics and their abstract counterpart. In this language, the type $List(t)$ consists of lists of values of type t, $P(t)$ of sets of values of type t, $(t_1, \ldots, t_n)$ is the n-product of types $t_1, \ldots, t_n$ and $t_1 \to t_2$ is the function space from t_1 to t_2. The operator $\to$ is right-associative. Function application is left-associative and has higher precedence than any operator. $[\,]$ denotes the empty list and $(x : y)$ a list with head x and tail y. The function $map : (a \to b) \to P(a) \to P(b)$ applies its first argument to each element of its second argument. We also require the following function which "folds" a binary function f into a list:

```
fold : (a->b->a) -> a -> List(b) -> a
fold f x []     = x
fold f x (y:ys) = fold f (f x y) ys
```

Our specifications are reminiscent of a denotational approach [23] in that we represent the database state as a function from a set of identifiers to a semantic domain, and define how this state is transformed during rule execution. However, our specifications are executable. As we will see, this means that the abstract semantics can form the basis for developing practical tests for rule termination.

2.1 The execution semantics

Each active rule is modelled as a four-tuple consisting of an *event query*, a *condition query*, a list of *actions*, and an *execution mode*. The type *Rule* is thus:

```
Rule = (Query,Query,List(Action),Mode)
```

A rule's execution mode encodes information about where on the current schedule the rule's actions should be placed if the rule fires (e.g. Immediate or Deferred scheduling) and to what database state these actions should be bound (e.g. the state in which the condition was evaluated, or the state in which the action will be executed) — see [17] for a description of the scheduling and binding possibilities for active rules. We assume that the currently defined rules are held in a global list $RULES : List(Rule)$ in order of their priority [1]. We also assume that it is possible to derive a *delta query* from each rule action which encodes the change that the action would make to the database state.

Schedules are lists of actions, and database states are modelled as functions from a set of database object identifiers, Id, to a semantic domain, D:

[1] Thus rules are totally ordered and rule firing is deterministic. Handling non-deterministic firings of rules of the same priority is an area of future work.

```
Schedule = List(Action)
DBState  = Id -> D
```

We assume that there is a distinguished value $\emptyset \in D$. A rule *fires* if its event query and condition query both evaluate to a value other than $\emptyset$.

The Id, D, $Query$, $Action$ and $Mode$ types referred to above may clearly be different for different active database systems. In Section 3 we define them for a relational database system, but they can be defined for other types of system, such as object-oriented ones.

There are three kinds of database object identifiers: the set of *data identifiers*, $DataId$, the set of *view identifiers*, $ViewId$, and the set of *binding identifiers*, $BindId$ (so that $Id = DataId \cup ViewId \cup BindId$).

Data identifiers: These are objects over which users can specify queries and actions e.g. the names of base relations and the names of delta relations in a relational database system.

View identifiers: Suppose that $RULES$ contains n rules, so that there are n event queries $eq_1, \ldots eq_n$, n condition queries $cq_1, \ldots, cq_n$ and $m \geq n$ delta queries $dq_1, \ldots dq_m$. Then, the domain of the database state will contain a set of corresponding view identifiers, $e_1, \ldots e_n, c_1, \ldots, c_n, d_1, \ldots d_m$, which are mapped to the current values of their corresponding queries.

Binding identifiers: These record a history of the values that the view identifiers take throughout the rule execution. As we will see below, this history is needed in order to support a variety of rule binding modes.

We specify the rule execution semantics by a function *execSched*, listed below. This takes a database state and a schedule, and repeatedly executes the first action on the schedule, updating the schedule with the actions of rules that fire along the way. If *execSched* terminates, it outputs the final database state and the final, empty, schedule. If it fails to terminate, it produces no output.

The function *exec* (see below) executes the first action, a, on the schedule and returns a new database state. This new state, db' say, is then passed to a function `createSnapshot:DBState->DBState` whose definition is straight-forward and not listed here. *createSnapshot* extends the domain of db' with a new binding identifier for each view identifier $i \in dom(db')$, setting the value of this new binding identifier to be $db'(i)$. These binding identifiers thus create a "snapshot" of the current values of the view objects. The event, condition, and delta queries of rules that fire as a result of the execution of the action a can be bound to this snapshot by *updateSched* (see below). This snapshot is never updated[2].

The function *schedRules* applies the function *schedRule* to each rule, in order of the rules' priority. *schedRule* determines whether a given rule (eq, cq, as, m) should fire. If the rule's event query eq or its condition query cq evaluate to $\emptyset$ in the current database state, then the rule does not fire and the database state and schedule remain unchanged. Otherwise, *updateSched* (see below) is called to update the database state and schedule.

[2] Clearly in a practical implementation the entire view state does not need to be copied each time, only the changes to it. Moreover, snapshots that are no longer referenced on the schedule can be discarded from the database state.

```
execSched : (DBState,Schedule) -> (DBState,Schedule)
execSched (db,s) =
    if s = []
    then (db,s)
    else execSched (schedRules (execUpdate (db,s)))

execUpdate : (DBState,Schedule) -> (DBState,Schedule)
execUpdate (db,a:s) = (createSnapshot (exec (a,db)),s)

schedRules : (DBState,Schedule) -> (DBState,Schedule)
schedRules (db,s) = fold schedRule (db,s) RULES

schedRule : Rule -> (DBState,Schedule) -> (DBState,Schedule)
schedRule (eq,cq,as,m) (db,s) =
    if (empty eq db) or (empty cq db)
    then (db,s)
    else updateSched (as,m,db,s)
```

Three functions therefore remain to be defined for any specific rule language and execution semantics, *empty*, *exec* and *updateSched*.

`empty:Query -> DBState -> Bool` determines whether a query evaluates to $\emptyset$ with respect to the current database state.

`exec:(Action,DBState) -> DBState` takes an action, a, and a database state, db, and updates the values of the data objects and view objects in db to reflect the effect of a. We assume that if a rule's event query or condition query evaluates to $\emptyset$ then, were they to be scheduled, the rule's actions would have no effect i.e. that $exec\ (a, db) = db$ for any such action a and any database state db. We call such actions *null actions*. Null actions aren't actually scheduled by *execSched* but do need to be considered by the abstract execution semantics in order to reflect all possible concrete executions (see outline proof of Theorem 1 in Section 2.3). An easy way to guarantee that all rule actions satisfy this property is to encode the rule's event and condition queries within each of the rule's actions — note this has no effect on the semantics of a rule. Thus, each rule action has the notional form $if\ \ eq \wedge cq\ \ then\ \ update(dq)$ where eq is the event query, cq the condition query, dq the delta query, and $update$ is some expression. The precise syntax of rule actions will vary from system to system.

`updateSched:(List(Action),Mode,DBState,Schedule)->(DBState,Schedule)` takes a rule's list of actions as, its execution mode m, the current database state db, and the current schedule s, and does the following:

(i) Replaces the event and condition query encoded within each action $a \in as$ by its corresponding snapshot view identifier. This binds the encoded event and condition queries to the current database state.
(ii) If the execution mode m states that the rule's action must also be bound to the current state, then the delta query appearing within each $a \in as$ is also replaced by its corresponding snapshot view identifier.

(iii) The resulting reaction transaction is inserted into the appropriate part of the schedule, as indicated by m.

We assume that this processing is independent of the values that database identifiers are mapped to, so that *updateSched* in fact has a more general signature `(List(Action),Mode,Id->a,Schedule)->(Id->a,Schedule)` where `a` is a type variable. In other words, *updateSched* is polymorphic over the semantic domain D. This means that *updateSched* can also be used in the abstract execution semantics. We assume that *createSnapshot* is similarly polymorphic over D, and it too is also used in the abstract execution semantics.

2.2 The abstract execution semantics

We are now ready to define the abstract counterpart, $execSched^*$, to $execSched$. The definition of $execSched^*$ is listed below. We distinguish abstract types and functions by suffixing their names with a '*'. The abstract database state type is defined by $DBState^* = Id \to D^*$, where D^* is the abstract counterpart to the semantic domain D. In general, D^* will be different for each particular abstraction. There needs to be a distinguished constant $\emptyset^* \in D^*$ which is the abstract counterpart to $\emptyset \in D$. Rules and schedules are syntactic objects which are common to both the concrete and the abstract semantics.

We see that $execSched^*$ is identical to $execSched$ except that it operates on an abstract database state type and that at the "leaves" of the computation the functions *empty* and *exec* are replaced by abstract counterparts $empty^*$ and $exec^*$. `empty*:Query->DBState*->Bool` determines whether a query evaluates to $\emptyset^*$ with respect to an abstract database state. `exec*:(Action,DBState*) ->DBState*` executes an action on the data objects and view objects in an abstract database state. As we discuss further in Section 3, these two functions need to be defined for each specific abstraction.

```
execSched* : (DBState*,Schedule) -> (DBState*,Schedule)
execSched* (db*,s) =
    if s = []
    then (db*,s)
    else  execSched* (schedRules* (execUpdate* (db*,s)))

execUpdate* : (DBState*,Schedule) -> (DBState*,Schedule)
execUpdate* (db*,a:s) = (createSnapshot (exec* (a,db*)),s)

schedRules* : (DBState*,Schedule) -> (DBState*,Schedule)
schedRules* (db*,s) = fold schedRule* (db*,s) RULES

schedRule* : Rule -> (DBState*,Schedule) -> (DBState*,Schedule)
schedRule* (eq,cq,as,m) (db*,s) =
    if (empty* eq db*) or (empty* cq db*)
    then (db*,s)
    else updateSched (as,m,db*,s)
```

2.3 Correctness of the abstract semantics

An abstract database approximates a number of real databases. These possible concretisations are obtained by applying a *concretisation function* (see Section 2.4) to the abstract database.

Termination is an undecidable property even for simple active rule languages [5]. $execSched^*$ is a *safe* approximation if the equivalence below holds for all $db^* \in DBState^*$ and $s \in Schedule$, where *conc* is the chosen concretisation function and $\sqsubseteq$ the information ordering on $P(DBState, Schedule)$ (see Section 2.5):

$$conc\ (execSched^*\ (db^*, s)) \ \sqsubseteq\ map\ execSched\ (conc\ (db^*, s))$$

The LHS of this equivalence corresponds to the set of possible concrete databases and schedules obtained by first running the abstract execution on (db^*, s) and then deriving all possible concretisations of the resulting abstract database and schedule (note that if $execSched^*$ terminates then all the concretised final schedules will be empty). The RHS corresponds to first deriving all possible concretisations of (db^*, s) and then applying the real execution to each concrete database and schedule pair.

Thus a safe approximation yields no more information than the real execution would. The following theorem states sufficient conditions for this property to hold:

Theorem 1. $execSched^*$ is a safe approximation if
(i) for all actions a and all $db^* \in DBState^*$,
$conc\ (exec^*\ (a, db^*)) \ \sqsubseteq\ map\ exec\ (conc\ (a, db^*))$, and
(ii) for all event or condition queries q and all $db^* \in DBState^*$,
$empty^*\ q\ db^* = True \Rightarrow (\forall db \in conc\ db^*\ .\ empty\ q\ db = True)$

Proof (outline). Since we a using a functional metalanguage, fixpoint induction [23] can be employed. The values of $execSched^*$ and $execSched$ are given by $\sqcup\{F^i(\bot) \mid i \geq 0\}$ and $\sqcup\{G^i(\bot) \mid i \geq 0\}$ respectively, where

$$F = \lambda f.\lambda(db^*, s).if\ (s = [\,])\ then\ (db^*, s)\ else\ f\,(schedRules^*\,(execUpdate^*\,(db^*, s)))$$
$$G = \lambda g.\lambda(db, s).if\ (s = [\,])\ then\ (db, s)\ else\ g\,(schedRules\,(execUpdate\,(db, s)))$$

It is thus sufficient to show that $conc \circ F^i(\bot) \ \sqsubseteq\ (map\ G^i(\bot)) \circ conc$ for all $i \geq 0$. The base case of $i = 0$ is straightforward. For the inductive case, we can show by provisos (i) and (ii) of the theorem that

$$conc \circ schedRules^* \circ execUpdate^* \sqsubseteq (map\ (schedRules \circ execUpdate)) \circ conc$$

It is here that null actions may arise since the concretisation of the abstract database and schedule on the LHS may generate database, schedule pairs in which the schedule contains null actions whereas the corresponding database, schedule pair in the RHS will not include such actions. Using the above equivalence and the induction hypothesis, the inductive case of the theorem follows. ■

2.4 Defining the *conc* functions

A concretisation function is needed for the argument type of each of the functions called by $execSched^*$. We observe from the definition of $execSched^*$ that there are four different argument types: $DBState^*$, $(DBState^*, Schedule)$, $(Action, DBState^*)$, and $(List(Action), Mode, DBState^*, Schedule)$. We use the concretisation function over $DBState^*$ to define the concretisation functions over the other three argument types. *These definitions are independent of the particular abstraction adopted.* We give them here and they can be reused for each specific abstraction considered later in the paper. In particular, given a definition for $conc : DBState^* \rightarrow P(DBState)$, the concretisation functions over $(DBState^*, Schedule)$, $(Action, DBState^*)$ and $(List(Action), Mode, DBState^*, Schedule)$ are respectively defined as follows (we overload the identifier *conc* since its type can always be inferred from context):

$$\begin{aligned}
conc\ (db^*, s) &= \{(db, s) \mid db \leftarrow conc\ db^*\}\\
conc\ (a, db^*) &= \{(a, db) \mid db \leftarrow conc\ db^*\}\\
conc\ (as, m, db^*, s) &= \{(as, m, db, s) \mid db \leftarrow conc\ db^*\}
\end{aligned}$$

2.5 Defining $\sqsubseteq$

The concrete domain D, together with the information ordering on it $\sqsubseteq_D$, need to be defined for a given active database system. For example, in Section 3 we define D for a relational database system. The domain $DBState$ is derived from D. The orderings on $DBState$, $Schedule$, $P(DBState)$ and $P(DBState, Schedule)$ are derived in a standard fashion, *independent of the particular choice of* D. We define these orderings here, and they can be reused for all the abstractions considered later in the paper.

The ordering on $DBState$, $\sqsubseteq_{DBState}$, is the usual ordering on a function space and is defined in terms of $\sqsubseteq_D$ as follows, where $db_1, db_2 \in DBState$:

$$db_1 \sqsubseteq_{DBState} db_2 \quad if \quad \forall i \in Id \,.\, db_1\ i\ \sqsubseteq_D\ db_2\ i$$

The ordering on $P(DBState)$ is as follows for $DB_1, DB_2 \in P(DBState)$:

$$DB_1 \sqsubseteq_{P(DBState)} DB_2 \quad if \quad \forall db_2 \in DB_2 \,.\, \exists db_1 \in DB_1 \,.\, db_1 \sqsubseteq_{DBState} db_2$$

Note that this is the Smyth powerdomain ordering [23]. Since the sets of database states that we consider in our abstract interpretation framework are generated by concretising a particular abstract database state, this reflects the intuition that more precise abstractions give rise to smaller sets of database states.

To define the ordering on $Schedule$ $(= List(Action))$ we require an operator $reduce : Schedule \rightarrow Schedule$ which removes all null actions from a schedule. Then $\sqsubseteq_{Schedule}$ is as follows for $s_1, s_2 \in Schedule$:

$$s_1 \sqsubseteq_{Schedule} s_2 \quad if \quad (reduce(s_1) = reduce(s_2)) \ \wedge\ (length(s_1) \geq length(s_2))$$

Thus s_1, s_2 are identical apart from any null actions they may contain, except that s_2 is no longer, and hence has no worse termination properties, than s_1.

Finally, the ordering on $P(DBstate, Schedule)$ is as follows for $DBS_1, DBS_2 \in P(DBState, Schedule)$: $DBS_1 \sqsubseteq_{P(DBState,Schedule)} DBS_2$ *if*

$$\forall (db_2, s_2) \in DBS_2 \,.\, \exists (db_1, s_1) \in DBS_1 \,.\, db_1 \sqsubseteq_{DBState} db_2 \wedge s_1 \sqsubseteq_{Schedule} s_2$$

Generally, we do not subscript the $\sqsubseteq$ symbol since which of the above orderings is meant can be inferred from context.

3 Applying the Framework

To summarise, in order to use the framework just described, one needs to define:

(i) the concrete domain D,
(ii) the *empty* and *exec* functions,
(iii) the *updateSched* function,
(iv) the abstract domain D^*,
(v) the concretisation function $conc : DBState^* \rightarrow DBState$, and
(vi) the $empty^*$ and $exec^*$ functions.

Parts (i) and (ii) are specific to the particular query/update language, part (iii) to the particular rule scheduling semantics and parts (iv)-(vi) to the particular abstraction. Once an abstraction has been defined, one needs to show that the two requirements for Theorem 1 hold.

For the remainder of the paper we assume a relational database system, with the relational algebra as the query/update language. Referring to point (i) of the framework, in this case the concrete domain is

$$D = P(Const) + P(Const, Const) + P(Const, Const, Const) + \ldots$$

where $Const$ is a possibly infinite set of constants and $+$ is the disjoint union domain constructor [23]. Thus, each member of D is a set of n-tuples, for some $n > 0$. The distinguished constant $\emptyset$ is the empty set, $\{\}$.

Referring to point (ii) of the framework, in order to define *exec* and *empty* we first need some auxiliary sets. Let $Expr$ be the set of relational algebra expressions over D. Let $BaseRel$ be the set of base relation names and $DeltaRel$ the set of delta relation names, so that $DataId = BaseRel \cup DeltaRel$. The syntax of $e \in Expr$, $q \in Query$, $a \in Action$ and $dr \in DeltaId$ is then as follows, where $d \in D$, $i \in Id$ and $r \in BaseRel$:

$$\begin{aligned} e &::= d \mid e_1 \cup e_2 \mid e_1 \times e_2 \mid e_1 - e_2 \mid e_1 \cap e_2 \mid \sigma_\theta e \mid \pi_A e \\ q &::= i \mid q_1 \cup q_2 \mid q_1 \times q_2 \mid q_1 - q_2 \mid q_1 \cap q_2 \mid \sigma_\theta q \mid \pi_A q \\ a &::= insert\ r\ q \mid delete\ r\ q \\ dr &::= +\Delta r \mid -\Delta r \end{aligned}$$

For ease of exposition, we assume that event queries have the form $+\Delta R$ or $-\Delta R$, although in principle they could be arbitrarily complex. An example active rule is thus: on $+\Delta R_1$ if $R_2 \cap R_3$ do $delete\ R_2\ +\Delta R_1$.

With respect to point (iii) of the framework, we do not define a specific rule scheduling semantics for our archetypal relational database system as this is in fact immaterial to the termination analysis techniques that we will explore (hence, we have omitted the *mode* part from the above example active rule). For each abstraction that we consider in subsequent sections of the paper, it thus only remains to address points (iv)-(vi) of the framework.

We recall from Section 2.1 our requirement that if a rule's event query or condition query evaluates to $\emptyset$ then, were they to be scheduled, the rule's actions would have no effect. For any user-specified rule action of the form *insert* $r\ q$, the corresponding delta query is $q - r$ and the rule action can be rewritten to *insert* $r\ \pi_{atts(r)}(eq \times cq \times (q - r))$ where eq and cq are the rule's event query and condition query, respectively. This has the effect of encoding the rule's event and condition queries within the action, without changing the semantics of the action. Similarly, for any user-specified rule action of the form *delete* $r\ q$, the corresponding delta query is $q \cap r$ and the rule action can be rewritten to *delete* $r\ \pi_{atts(r)}(eq \times cq \times (q \cap r))$.

In order to define *exec* and *empty* we assume two auxiliary functions, *eval* and *bind*. The function $eval : Expr \to D$ is a relational algebra evaluator i.e. it evaluates an expression to the set of tuples in D that it denotes. The function $bind : Query \to DBState \to Expr$ substitutes occurrences of identifiers $i \in Id$ in a query by their values in the given database state and returns the resulting expression. *empty* and *exec* are then defined as follows [3]:

$$
\begin{array}{ll}
empty\ q\ db & = (eval\ (bind\ q\ db)\ =\ \{\}) \\
exec\ (insert\ r\ q, db) & = let\ db_1 = db[r \mapsto eval\ (bind\ (r \cup q)\ db), \\
& \qquad\qquad\quad +\Delta r \mapsto eval\ (bind\ (q - r)\ db), -\Delta r \mapsto \{\}, \\
& \qquad\qquad\quad \forall s \in BaseRel, s \neq r.(+\Delta s \mapsto \{\}, -\Delta s \mapsto \{\})] \\
& \quad in\ db_1[\forall i \in ViewId.(i \mapsto eval\ (bind\ iq\ db_1))] \\
exec\ (delete\ r\ q, db) & = let\ db_1 = db[r \mapsto eval\ (bind\ (r - q)\ db), \\
& \qquad\qquad\quad +\Delta r \mapsto \{\}, -\Delta r \mapsto eval\ (bind\ (q \cap r)\ db), \\
& \qquad\qquad\quad \forall s \in BaseRel, s \neq r.(+\Delta s \mapsto \{\}, -\Delta s \mapsto \{\})] \\
& \quad in\ db_1[\forall i \in ViewId.(i \mapsto eval\ (bind\ iq\ db_1))]
\end{array}
$$

Thus, *empty* $q\ db$ binds the query q to the database state db and then calls *eval* to evaluate the resulting expression. *exec* $(insert\ r\ q, db)$ replaces the current value of r in db by the result of evaluating $r \cup q$ w.r.t. db, updates the values of all the delta relations, and updates the values of all the view identifiers, where iq is the query corresponding to a view identifier i. *exec* $(delete\ r\ q, db)$ is similar.

[3] We assume that the queries q in the arguments to *exec* have been rewritten to encode the event and condition queries as described earlier. We use maplets, $a \mapsto b$, to denote members of functions. Given a function f, $f[a_1 \mapsto v_1, \ldots, a_n \mapsto v_n]$ is short-hand for the function $(f - \{a_1 \mapsto f\,a_1, \ldots, a_n \mapsto f\,a_n\}) \cup \{a_1 \mapsto v_1, \ldots, a_n \mapsto v_n\}$.

4 Abstraction 1: An Abstraction for Static Termination Analysis

In this section we define an abstraction for statically inferring the termination of a set of rules and show that it is a safe approximation. We first address points (iv)-(vi) of the framework. The abstract domain D^* consists of the set of relational algebra queries $q \in Query$ as defined in the previous section, together with the value $\emptyset^* = \{\}$. $exec^*$ replaces the abstract value of each data identifier by a new abstract value, obtained by binding to the current abstract database state the query that would have been evaluated by $exec$ in the concrete semantics:

$$\begin{aligned}
exec^*\ (insert\ r\ q, db^*) = {} & let\ db_1^* = db^*[r \mapsto bind\ (r \cup q)\ db^*, \\
& \quad +\Delta r \mapsto bind\ (q - r)\ db^*, -\Delta r \mapsto \{\}, \\
& \quad \forall s \in BaseRel, s \neq r.(+\Delta s \mapsto \{\}, -\Delta s \mapsto \{\})] \\
& in\ db_1^*[\forall i \in ViewId.(i \mapsto bind\ iq\ db_1^*)] \\
exec^*\ (delete\ r\ q, db^*) = {} & let\ db_1^* = db^*[r \mapsto bind\ (r - q)\ db^*, \\
& \quad +\Delta r \mapsto \{\}, -\Delta r \mapsto bind\ (q \cap r)\ db^*, \\
& \quad \forall s \in BaseRel, s \neq r.(+\Delta s \mapsto \{\}, -\Delta s \mapsto \{\})] \\
& in\ db_1^*[\forall i \in ViewId.(i \mapsto bind\ iq\ db_1^*)]
\end{aligned}$$

The concretisation function $conc : DBState^* \to DBState$ when applied to an abstract database state db^* returns a set of concrete database states db; each db is obtained by first choosing a function $\rho_{db} : Id \to D$ which defines an initial value for each database identifier, and then setting $db\ i = eval\ (bind\ (db^*\ i)\ \rho_{db})$ for all $i \in Id$. The remaining $conc$ functions are derived as in Section 2.4.

Defining $empty^*$ amounts to selecting a method for analysing the satisfiability of queries. Satisfiability is undecidable for the general relational algebra, so we must ensure that the queries being tested are expressed in some decidable fragment. One decidable fragment consists of queries where the argument to projection operations does not contain any occurrence of set-difference [22]. To guarantee that the queries passed to $empty^*$ have this form, we use a function $normalise$ that safely rewrites a query into an approximating interval of two queries. This function applies bottom-up the following rules except that the rule for $e_1 - e_2$ becomes $[\{\}, b]$ for any sub-expression $e_1 - e_2$ occurring within the scope of a projection operation. In these rules, a, b, c, d are relational algebra expressions, $a \subseteq b$, $c \subseteq d$, $e_1 = [a, b]$ and $e_2 = [c, d]$:

$$\begin{array}{ll}
e_1 \cup e_2 \quad = [a \cup c, b \cup d] & e_1 - e_2 = [a - d, b - c] \\
e_1 \cap e_2 \quad = [a \cap c, b \cap d] & \sigma_\theta(e_1) \ = [\sigma_\theta(a), \sigma_\theta(b)] \\
\pi_A(e_1) \quad = [\pi_A(a), \pi_A(b)] & e_1 \times e_2 = [a \times c, b \times d] \\
[[a, b], [c, d]] = [a, d] &
\end{array}$$

e.g. $normalise(\pi_x((R_1 \times R_2) - R_3) \times (R_2 - R_1)) = [\emptyset, \pi_x(R_1 \times R_2) \times (R_2 - R_1)]$. The definition of $empty^*$ is thus:

$$empty^*\ q\ db^* \ = \ satisfiable\ (normalise\ (bind\ q\ db^*))$$

where $satisfiable$ applies the satisfiability test of [22] to the upper query of the interval output by *normalise*.

The following theorem is then straight-forwardly shown:

Theorem 2. *execSched** is a safe approximation of *execSched*, given the above definitions of D^*, *conc*, *exec** and *empty**.

Example 1. Consider the following two rules:
Rule 1: on $+\Delta R$ if $(R \cap S) \times (R \cap T)$ do *delete* R T
Rule 2: on $-\Delta R$ if *true* do *insert* R $(S - T)$

If rule 1 is the first rule triggered, then a trace of the abstract execution on each successive call to *execSched** is as follows, where 'Iter' denotes the current iteration, $action(i)$ the action of the i^{th} rule and I is an arbitrary unused identifier denoting that $+\Delta R$ is satisfiable at the first iteration:

Iter	Database State					Schedule
	R	S	T	$+\Delta R$	$-\Delta R$	
1	R_0	S_0	T_0	I	$\{\}$	$[action(1)]$
2	$R_0 - T_0$	S_0	T_0	$\{\}$	$R_0 \cap T_0$	$[action(2)]$
3	$(R_0 - T_0) \cup (S_0 - T_0)$	S_0	T_0	$(S_0 - T_0) - (R_0 - T_0)$	$\{\}$	$[action(1)]$

Note that in this example the execution mode of the rules is immaterial since at most one rule is ever triggered. At iteration 3, the condition of rule 1 expands to the expression $(((R_0 - T_0) \cup (S_0 - T_0)) \cap S_0) \times (((R_0 - T_0) \cup (S_0 - T_0)) \cap T_0)$, which is unsatisfiable. Hence *execSched** terminates after three iterations and definite rule termination can be concluded if rule 1 is the first rule triggered. ■

Of course in general there is no guarantee that *execSched** itself will terminate and so we need a criterion for halting it (concluding in such a case that the concrete rule execution may fail to terminate). A simple way is to choose *a priori* a bound on the number of iterations of *execSched**. Increasing this bound allows more precision but must be balanced against the time and space resources available. Another strategy for halting *execSched** is to choose a more limited abstract domain D^* that makes $DBState^*$ finite. *execSched** can then be halted if a repeating argument (db^*, s) is detected since in such a case *execSched** would not terminate. We demonstrate such an abstraction in the next section.

5 Abstraction 2: A Coarser Abstraction for Static Termination Analysis

We now set D^* to be the two-valued domain $\{Unknown, False\}$, where $\emptyset^* = False$. $conc(db^*)$ returns the set of databases db such that

- the values assigned to the data identifiers and view identifiers are consistent,
- $db(i) = \emptyset$ if $db^*(i) = False$

*empty** q db^* returns $True$ if $db^*(q)$ is $False$. Otherwise it returns $False$.

To define *exec** we need to distinguish between view identifiers corresponding to event queries, condition queries and delta queries, so we assume that

$ViewId = EqId \cup CqId \cup DqId$. $exec^*$ uses a function $infer(condition, update)$ to infer the new value of a condition query. The $infer$ function returns either (a) $Unknown$ if the condition *may* be true after the update executes, or (b) $False$ if the condition can *never* be true after the update executes. This kind of inference corresponds to the determination of arcs in an activation graph [7] and so a method such as the propagation algorithm described in [8] can be used to determine the effect of updates on conditions (with exponential complexity in the worst case). This, of course, is a non-effective method, and so $infer$ will in general sometimes return $Unknown$ when it could have returned $False$ (but if not, we term it a *perfect* inference function).

$exec^*$ $(insert\ r\ q, db^*)$ sets the values of r and $+\Delta r$ to $Unknown$ and the values of all other delta relations to $False$. It then updates the values of the view identifiers by setting to $Unknown$ the values of all event queries of the form $+\Delta r$ and to $False$ the values of all other event queries, using $infer$ to infer the new values of all the condition queries, and setting to $Unknown$ the values of all the delta queries:

$$
\begin{aligned}
&exec^*\ (insert\ r\ q, db^*)\ = \\
&\quad let\ db_1^* = db^*[r \mapsto Unknown, +\Delta r \mapsto Unknown, -\Delta r \mapsto False, \\
&\qquad\qquad\qquad \forall s \in BaseRel, s \neq r.(+\Delta s \mapsto False, -\Delta s \mapsto False]\ \ in \\
&\quad db_1^*[\forall e_i \in EqId, eq_i = +\Delta r.e_i \mapsto Unknown, \\
&\qquad \forall e_i \in EqId, eq_i \neq +\Delta r.e_i \mapsto False, \\
&\qquad \forall c_i \in CqId.c_i \mapsto infer(cq_i, insert\ r\ q), \\
&\qquad \forall d_i \in DqId.d_i \mapsto Unknown]
\end{aligned}
$$

Actions of the form $delete\ r\ q$ are handled similarly.

Theorem 3. $execSched^*$ is a safe approximation of $execSched$, given the above definitions of D^*, $conc$, $exec^*$ and $empty^*$,

The initial abstract database state now maps all identifiers to $Unknown$, apart from event queries that could not have triggered the initial rule and their corresponding delta relations, all of which are mapped to $False$.

Example 2. Consider again the rules defined in Example 1. The following is a trace of the abstract execution when rule 1 is triggered, assuming a perfect inference function ($cond(i)$ denotes the condition query of the i^{th} rule):

Iter	Database state		Schedule
	$cond(1)$	$cond(2)$	
1	$Unknown$	$Unknown$	$[action(1)]$
2	$False$	$Unknown$	$[action(2)]$
3	$Unknown$	$Unknown$	$[action(1)]$

At iteration 2, rule 1's condition becomes $False$, since its action will always falsify $R \cap T$. At iteration 3 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 $infer$ cannot make use of the fact that $R \cap T$ was previously $False$). The third state is thus a repetition of the first, so *execSched** is halted and possible rule non-termination is concluded if rule 1 is the first rule triggered.

Example 3. Suppose now we change the condition of rule 1 to be just $R \cap T$. The following trace is then obtained when rule 1 is the first rule triggered, again assuming a perfect inference function:

Iter	Database state		Schedule
	cond(1)	cond(2)	
1	$Unknown$	$Unknown$	$[action(1)]$
2	$False$	$Unknown$	$[action(2)]$
3	$False$	$Unknown$	$[action(1)]$

Changing rule 1's condition means that rule 2's action can no longer alter the truth value of this condition. Thus, *execSched** terminates at iteration 3, since rule 1's condition is $False$, and definite rule termination is concluded if rule 1 is the first rule triggered. ∎

The above $infer$ function uses only the syntax of conditions and actions to deduce new values for the event and condition queries. This motivates the development of our third abstraction which uses a more refined inferencing method and also more precise information about the initial abstract database state derived from the current concrete database state i.e. our third abstraction is useful for *dynamic* termination analysis.

6 Abstraction 3: An Abstraction for Dynamic Termination Analysis

As a third example application of our framework, we now show how to use the abstract execution semantics for dynamic, as opposed to static, termination analysis. This can give more precision because in a dynamic setting the initial database state can be described more accurately rather than just consisting of unknown information. This is reflected by the inclusion of the value $True$ in the abstract domain.

D^* is now the three-valued domain $\{True, Unknown, False\}$, where $\emptyset^* = False$. $conc(db^*)$ now returns the set of databases db such that

- the values assigned to the data identifiers and view identifiers are consistent,
- $db(i) = \emptyset$ if $db^*(i) = False$,
- $db(i) \neq \emptyset$ if $db^*(i) = True$.

$empty^*\ q\ db^*$ returns $True$ if $db^*(q)$ is $False$. Otherwise it returns $False$.

$exec^*$ calls an inference function $infer\_delt$ to compute the new truth value of the delta queries. This either (a) returns $True$ if a delta query was previously $True$ and the update to r can never cause deletions from it, or (b) returns $False$

if the delta query was previously $False$ and the update can never cause insertions to it, or (c) returns $Unknown$ otherwise. Such logic is implementable by incremental propagation techniques such as [20, 11] which perform query rewriting to determine if insertions or deletions can have an effect on an expression, and has polynomial complexity. Once again, this is in general a non-effective method and the $infer\_delt$ function will sometimes have to return $Unknown$ instead of $True$ or $False$ in order to be safe.

The new value of the condition queries is similarly given by a function $infer\_con$. If the new value of the delta query for the update is $False$, then the value of the condition is unchanged. Otherwise, $infer\_con$ either (a) returns $True$ if the condition was previously $True$ and the update to r can never cause deletions from it, or (b) returns $False$ if the condition was previously $False$ and the update can never cause insertions to it, or (c) returns $Unknown$ otherwise. The new value of the event queries is given by a function $infer\_ev$ which sets the value of an event query to the value of its corresponding delta relation.

$exec^*$ $(insert\ r\ q, db^*)$ infers new values for all the delta queries, sets the value of $+\Delta r$ to that of the delta query corresponding to the action $insert\ r\ q$ (this query is denoted by dq below), sets the value of $-\Delta r$ to $False$, sets the value of all other delta relations to $Unknown$, and infers new values for the view identifiers corresponding to event queries and condition queries:

$$
\begin{aligned}
& exec^*\ (insert\ r\ q, db^*)\ = \\
& \quad let\ db_1^* = db^*[\forall d_i \in DqId.d_i \mapsto infer\_delt(db^*\ d_i, insert\ r)]\ in \\
& \quad let\ db_2^* = db_1^*[+\Delta r \mapsto db_1^*\ dq, -\Delta r \mapsto False, \\
& \qquad \forall s \in BaseRel, s \neq r.(-\Delta s \mapsto Unknown, +\Delta s \mapsto Unknown)]\ in \\
& \quad db_2^*[\forall e_i \in EqId.e_i \mapsto infer\_ev(db_2^*), \\
& \qquad \forall c_i \in CqId.c_i \mapsto infer\_con(db_2^*\ c_i, db_2^*\ dq, insert\ r)]
\end{aligned}
$$

Actions of the form $delete\ r\ q$ are handled similarly.

Theorem 4. *$execSched^*$ is a safe approximation of $execSched$, given the above definitions of D^*, $conc$, $exec^*$ and $empty^*$.*

The initial abstract database state is the same as for Abstraction 2, except that (a) event queries that could have triggered the initial rule, and their associated delta relations, are now $True$ instead of $Unknown$, and (b) the values of the condition and delta queries will be one of $\{True, False, Unknown\}$ depending on the concrete run-time state when abstract execution is invoked.

Example 4. Consider again the rules 1 and 2 as originally defined in Example 1. For an initial database state in which rule 1's condition is known to be $True$ and rule 2's to be $False$, we obtain the following trace, assuming a perfect inference function:

Iter	Database state		Schedule
	$cond(1)$	$cond(2)$	
1	$True$	$False$	$[action(1)]$
2	$Unknown$	$False$	$[action(2)]$

Thus *execSched** terminates at iteration 2 because of the falsity of rule 2's condition. If, however, this was initially $True$, the trace would be as shown below. The last state is now a repetition of the second and so possible rule non-termination is concluded:

Iter	Database state		Schedule
	$cond(1)$	$cond(2)$	
1	$True$	$True$	$[action(1)]$
2	$Unknown$	$True$	$[action(2)]$
3	$Unknown$	$True$	$[action(1)]$
4	$Unknown$	$True$	$[action(2)]$

7 Discussion and Comparison of Abstractions 1—3

The termination analysis directly provided by *execSched** is for a given set of rules and a given initial schedule. For example, in Examples 1–4 the initial schedule consisted of the action of one of the rules. A broader question is, will a given set of rules terminate for *any* finite initial schedule? For reasons of space we cannot give an in-depth treatment of this issue here and will do so in a forthcoming paper. However, briefly, if all rules have Immediate coupling mode then it is sufficient to run *execSched** on each possible singleton schedule i.e. on two schedules $[insert\ R\ I]$ and $[delete\ R\ I]$ for each base relation R in the database; *execSched** terminates on all finite schedules if and only if it terminates on all singleton schedules. If some rules may have Deferred coupling mode, let $execSched^{*m}$ be *execSched** modified so as to reinitialise the abstract database state at the start of each deferred sub-transaction; then, *execSched** terminates on all finite schedules if $execSched^{*m}$ terminates on all singleton schedules. Unlike with Immediate-only coupling mode, this test is safe but may not be precise. Indeed, the general undecidability of termination of queue schedules [4] means that a precise test cannot be devised.

We now draw some comparisons between Abstractions 1–3. In general, none of them subsumes any of the others in the sense that all three can detect cases of definite termination that the other two cannot. Abst 3 is not directly comparable to Absts 1 and 2 since it will have available information about the initial database state. Abst 1 and 2 are distinct in that they use different satisfiability tests and so the resulting behaviour of *empty** is different for each. With Abst 1, $exec*$ maintains a precise record of the execution history (thus requirement (i) of Theorem 1 is actually an equality) and all the approximation is carried out by the *empty** function. Limiting the abstract domain in order to be able to detect repeating states, as was done for Absts 2 and 3, moves the approximation functionality from *empty** to $exec^*$.

We note that our framework provides a useful basis for determining the relative costs of different analysis methods, as these will only differ in their $exec^*$ and *empty** functions. The table below summarises the worst case costs for Absts 1–3. The costs of *empty** for Abst 1 and $exec^*$ for Abst 2 are both exponential in the size of the expressions input to them. However, the size of expressions input to the latter is constant, whereas for the former the expression size grows at each iteration. Thus, Abst 1 is likely to become more expensive in practice. We thus envisage Abst 2 being applied in scenarios where the rule set is very large (and hence a cheap analysis technique is required) whereas Abst 1 is more suitable for a deeper analysis of a small rule set. Its low computational cost makes Abst 3 suitable for run-time termination analysis.

Cost of Execution for N Rules			
	Time		
Abst	$empty^*$	$exec^*$	Space
1	Exponential	$\mathcal{O}(1)$	Unbounded
2	$\mathcal{O}(1)$	Exponential	$\mathcal{O}(2^N)$
3	$\mathcal{O}(1)$	Polynomial	$\mathcal{O}(3^N)$

8 Related Work

Previous methods of active rule analysis have often been closely linked to specific rule languages. In contrast, the framework that we have presented here does not assume any particular rule definition language. Our rule execution semantics can simulate most standard features of active database systems and prototypes. A detailed comparison is beyond the scope of this paper, but features such as SQL3's statement-level and row-level AFTER triggers [13], and variations of event consumption and rule coupling modes are either directly available or can be simulated within the framework. An alternative to the functional specification approach we have adopted would be to use *while* or $while_N$ programs [18] for the concrete execution semantics, and derive abstract versions of these. The advantage of a functional approach is that parameters such as rule language and scheduling strategy are explicit. The advantage of *while* programs would be easier identification of rule cycles and monotonicity of relations.

In [14], some general ideas regarding approximation in databases are discussed, motivated by the classification of methods for incompleteness of information. Other related work [9] looks at approximate query evaluation in the context of what precision is theoretically possible when approximating Datalog by conjunctive or first-order queries. Our work is complementary to [14, 9] in that we have developed a framework designed specifically for active rule analysis and we have developed three example abstractions for a first-order language. These abstractions serve to illustrate the application and usefulness of our framework, as well as being useful heuristics for rule termination analysis in their own right.

Regarding other work in rule termination analysis, [12] proposed using satisfiability tests between pairs of rules to refine triggering graphs. This can be interpreted as a less sophisticated version of the $empty^*$ function of our Abst 1 which does satisfiability testing for sequences of rule conditions and actions. Thus, in Example 1, if we only considered the relationship between pairs of rules, we would derive the information that rules 1 and 2 can mutually trigger each other. By itself, this information does not allow us to make any conclusions about termination behaviour. However, Example 1 demonstrated that by doing satisfiability testing on an accumulated "history" expression, it can be concluded that rule execution must terminate if rule 1 is the initially triggered rule.

Abst 2 does focus on pairwise relationships between rules and so is similar to graph-based analysis techniques based on the notions of triggering and activating rules [2, 8, 7, 12]. Abst 2 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 do not arise naturally from the specifications of the techniques themselves.

Using abstract interpretation for termination analysis of active rules was first considered in [3], for a theoretical rule language manipulating variables rather than relations and for a less expressive execution semantics. In [6] we refined that work to a full relational language and explored Abst 1 for the PFL active database system [21, 19] i.e. assuming a specific `updateSched` function.

Of course an alternative to developing techniques for rule termination analysis is to design rule languages which *a priori* cannot give rise to non-terminating computations e.g [25, 15, 24]. However,the current SQL3 proposal [13] allows non-terminating rule sets to be defined. Thus we believe that developing techniques for rule termination analysis is both necessary and relevant to current practice.

9 Summary and Further Work

We have defined a framework for the abstract execution of active rules, and have illustrated its application by developing and showing the correctness of three techniques for rule termination analysis. Applying the framework is a matter of addressing points (i)-(vi) in Section 3 and proving that the two requirements of Theorem 1 hold. We envisage the framework as a useful general tool for the development, comparison and verification of abstraction-based analysis and optimisation methods for active databases. Of the three abstractions that we have presented here, the first provides a detailed static analysis, the second a cheaper static analysis and the third a cheap dynamic analysis.

We are currently implementing a rule analysis tool that supports our framework. Avenues for further work include: classifying other active rule analysis methods using our framework, and thereby verifying their correctness and domain of applicability; extending the framework to allow abstraction at different granularities of database objects, abstraction over schedules, and non-deterministic firings of rules with the same priority.

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, 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, 1997.
4. J. Bailey, G. Dong, and K. Ramamohanarao. Structural issues in active rule systems. In *Proc. ICDT'97, LNCS 1186*, pages 203–214, 1997.
5. J. Bailey, G. Dong, and K. Ramamohanarao. Decidability and undecidability results for the termination problem of active database rules. In *Proc. PODS'98*, pages 264–273, 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. E. Baralis, Ceri. S., and S. Paraboschi. Compile-time and runtime analysis of active behaviors. *IEEE Trans. on Knowledge and Data Engineering*, 10(3):353–370, 1998.
8. 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.
9. S. Chaudhuri and P. Kolaitis. Can datalog be approximated ? *Journal of Computer and System Sciences*, 55, 1997.
10. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic Programming*, 13(2&3):103–179, 1992.
11. T. Griffin, L. Libkin, and H. Trickey. A correction to "Incremental recomputation of active relational expressions" by Qian and Wiederhold. *IEEE Trans. on Knowledge and Data Engineering*, 9(3):508–511, 1997.
12. A. Karadimce and S. Urban. Refined triggering graphs: A logic based approach to termination analysis in an active object-oriented database. In *Proc. 12th ICDE*, pages 384–391, 1996.
13. K. Kulkarni, N. Mattos, and R. Cochrane. Active database features in SQL3. In [17].
14. L. Libkin. Approximation in databases. In *Proc. ICDT'95*, pages 411–424, 1995.
15. B. Ludäscher, U. Hamann, and G. Lausen. A logical framework for active rules. In *Proc. 7th International Conference on Management of Data*, Pune, India, 1995.
16. K. Marriott, H. Søndergaard, and N. D. Jones. Denotational abstract interpretation of logic programs. *ACM TOPLAS*, 16(3):607–648, 1994.
17. N. Paton, editor. *Active Rules in Database Systems.* Springer-Verlag, 1999.
18. P. Picouet and V. Vianu. Semantics and expressiveness issues in active databases. In *Proc. PODS'95*, pages 126–138, 1995.
19. A. Poulovassilis, S. Reddi, and C. Small. A formal semantics for an active functional DBPL. *Journal of Intelligent Information Systems (special issue on Active Database Systems)*, 7(2):151–172, 1996.
20. X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Trans. on Knowledge and Data Engineering*, 3(3):337–341, 1991.
21. S. Reddi, A. Poulovassilis, and C. Small. Extending a functional DBPL with ECA-rules. In *Proc. RIDS'95*, pages 101–115, 1995.
22. Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM*, 27(4):633–655, 1980.
23. D.A. Schmidt. *Denotational Semantics.* Allyn and Bacon, 1986.
24. C. Zaniolo. Active database rules with transaction-conscious stable-model semantics. In *Proc. DOOD'95*, pages 55–72, 1995.
25. Y. Zhou and M. Hsu. A theory for rule triggering systems. In *Proc. EDBT'90, LNCS 416* pages 407–422, 1990.