Abstract Interpretation of Active Rules and Its Use in Termination Analysis

James Bailey, Lobel Crnogorac, Kotagiri Ramamohanarao and Harald Søndergaard

Dept. of Computer Science, University of Melbourne, Parkville Vic. 3052, Australia
E-mail: {jbailey,lobel,rao,harald}@cs.mu.oz.au

Abstract. The behaviour of rules in an active database system can be difficult to predict, and much work has been devoted to the development of automatic support for reasoning about properties such as confluence and termination. We show how abstract interpretation can provide a generic framework for analysis of active rules. Abstract interpretation is a well-understood, semantics-based method for static analysis. Its advantage, apart from generality, lies in the separation of concerns: Once the underlying semantics has been captured formally, a variety of analyses can be derived, almost for free, as *approximations* to the semantics. Powerful general theorems enable simple proofs of global correctness and uniform termination of specific analyses. We outline these ideas and show, as an example application, a new method for termination analysis. In terms of precision, the method compares favourably with previous solutions to the problem. This is because the method investigates the flow of data rather than just the syntax of conditions and actions.

1 Introduction

An active database is a system which provides the functionality of a traditional database and additionally is capable of automatically reacting to state changes, both internal and external, without user intervention. This functionality is achieved by means of active rules or triggers. Active rules are useful in diverse applications such as workflow systems, audit trails, and maintenance of statistics. When multiple triggers are defined in a system, however, their interactions can be difficult to analyse, and interaction analysis is currently an important area of research. A key property which needs to be examined in this context is that of termination [2, 5, 11, 3, 4, 9, 8]. In this paper, we present an approach to rule analysis, based on abstract interpretation, which goes beyond the checking of syntax. The analysis propagates sophisticated dataflow information about calling patterns and how different rules interact. An important advantage of the technique is that it can be used for examining a wide variety of rule properties. We apply our method to the particular problem of rule termination and are able to obtain more precise answers for many types of rule sets.

The remainder of this paper is structured as follows. We provide a brief survey of termination analysis and an introduction to the principles of abstract interpretation. We then present the semantics of our active database language in a denotational form and also define an extension of it called the collecting semantics. We show how to approximate the collecting semantics using a simple rule language as an example. Next, we put this approximate collecting semantics to use in termination analysis. A discussion of how this could be extended to a more realistic rule language is also given. Finally, we discuss the results obtained and outline some directions for future research.

1.1 Termination

A rule set is said to terminate if, for any initial database state and any initial triggering event, rules cannot activate each other indefinitely. Obviously, this is a desirable property, since infinite triggering behaviour could make a system unusable. Yet it is undecidable in the general case whether an arbitrary rule set terminates. Previous work has suggested the use of either syntactic analysis, local semantic analysis, or a combination of both for detecting termination.

One of the first studies of active rule termination analysis was due to Aiken, Widom and Hellerstein [2] who introduced an abstraction called the *triggering graph*. In a triggering graph, the rules are represented by vertices and the possibility that execution of one rule can cause the execution of another rule is captured by a directed arc between two vertices. An acyclic triggering graph implies definite termination of a set of rules. This idea was refined by Karadimce and Urban [9]. Baralis and Widom [5] suggested another type of graph, the *activation graph*. Again rules are represented by vertices, but this time an arc indicates that the action of one rule may satisfy the condition of another rule. Here too, acyclicity means definite termination, but generating the graph is non-effective. Baralis *et al.* [3] presented the *rule reduction method*, combining the use of triggering and activation graphs. Any vertex which does not have both an incoming triggering and activation arc, can be removed from the graph (along with its outgoing arcs). If this procedure results in all vertices being removed, then the rule set must terminate.

Van der Voort and Siebes [11] consider the termination problem in an object-oriented rule system. The focus is on deciding termination in a fixed number of steps. The proofs make use of typical database states and the reasoning has some resemblance to a dataflow style of analysis. The general termination problem is not discussed, however, and details of the algorithms are unclear.

In an alternative approach by Baralis *et al.* [4], termination analysis happens at run time. The method presented in this paper, although static, does have features in common with this dynamic method, since we are effectively "running" the rule set, albeit in an approximate manner. The dynamic method is fundamentally limited, however, in that it can't analyse rules which can create new constants.

1.2 Abstract Interpretation

Abstract interpretation has proven a useful tool for analysis of imperative, functional and logic programs [1, 7, 10]. It has numerous applications in compiler

$*$	*none*	*odd*	*even*	*any*
none	*none*	*none*	*none*	*none*
odd	*none*	*odd*	*even*	*any*
even	*none*	*even*	*even*	*even*
any	*none*	*any*	*even*	*any*

Input	No branch	Yes branch
none	*none*	*none*
odd	*odd*	*none*
even	*none*	*even*
any	*odd*	*even*

Table 1. Approximating multiplication (left) and the *even* test (right)

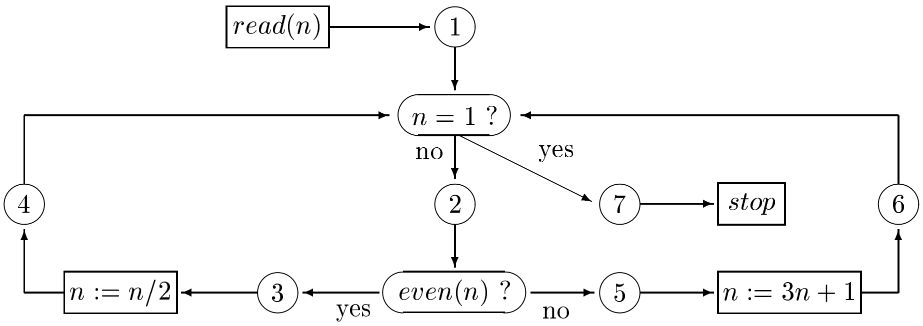

Fig. 1. A flow diagram for the $3n + 1$ example

optimisation and program transformation. The idea is to infer and collect information about a program's run time properties and its operational aspects. This information is known as the collecting semantics of a program.

The semantic domain of a program is approximated by an "abstract" domain, essentially a coarse grained version of the semantic domain. The operations of a program are approximated by abstract operations. Consider a program to be analysed whose semantic domain is $\mathcal{Z}$, the set of integers, and whose operations are the arithmetic operations $\{+, -, *, /\}$, the comparison functions, $\{<, >, =\}$, as well as tests $\{odd(\cdot), even(\cdot)\}$. The semantic domain could be approximated by descriptions $\{$*none*, *even*, *odd*, *any*$\}$ and the arithmetic operations simply according to the well-known "rules of parities". For example, Table 1 shows how multiplication is approximated (left table), as well as the approximation of the test *even* (right table).

Now consider the approximate computation of the flow diagram in Figure 1. Each arc of the diagram corresponds to a program point, and is labelled with a number. The computation involves iterating towards a fixpoint. The iteration proceeds by following arcs from the goal and updating program points until the values at each point have stabilised. Table 2 shows how the states iterate towards a fixpoint assuming n is *any* at point 1. The last step yields a fixpoint. The result of the analysis justifies transformation of the program so as to avoid a number of tests; for example, the statement $n := 3n+1$ can be replaced by $n := (3n+1)/2$.

Step	(1,	2,	3,	4,	5,	6,	7)
0	(*any*,	*none*,	*none*,	*none*,	*none*,	*none*,	*none*)
1	(*any*,	*any*,	*none*,	*none*,	*none*,	*none*,	*odd*)
2	(*any*,	*any*,	*even*,	*none*,	*odd*,	*none*,	*odd*)
3	(*any*,	*any*,	*even*,	*any*,	*odd*,	*even*,	*odd*)
4	(*any*,	*any*,	*even*,	*any*,	*odd*,	*even*,	*odd*)

Table 2. Approximate computation to a fixpoint from an initial state

Even though the actual computation may never terminate, the approximate computation always terminates.This is because the analysis is monotone with respect to the natural ordering of the descriptions, namely $d \leq d'$ iff $(d = \mathit{none}) \vee (d = d') \vee (d' = \mathit{any})$. Annotations are only allowed to "grow" with respect to that ordering. In particular, where two or more arcs run together, the result is taken to be their least upper bound. We return to this point in Section 3.

2 A Collecting Semantics for Active Rules

We use denotational semantics to formally specify the execution of our active database model as a fixpoint computation. Analysis then consists of finding an approximation to the least fixpoint. Our basic execution model is similar to that assumed by Baralis *et al.* [3]:

1. If there is no triggered rule then exit.
2. Select the highest priority triggered rule.
3. Evaluate the condition of the selected rule.
4. If the condition is true then execute the action of the selected rule.

The action that executes in step 4 can trigger further rules by causing events and thus 1-2-3-4 can repeat forever. We assume that an initial triggering event occurs within a transaction and this immediately initiates rule processing which will either continue indefinitely or terminate and allow the next statement in the transaction to execute (essentially immediate coupling mode). In the denotational definition, we have made the following important choices.

– Whenever there are several triggered rules to choose from, we deterministically select one according to a priority scheme. This is quite a sensible choice, since it automatically imposes confluence on the rule set.
– Pending rules are represented as a set. This means that a rule cannot have multiple instances triggered simultaneously.

We now give a denotational definition of the execution model. The definition captures the essence of our model and is similar in spirit to those of Widom [12] and Coupaye and Collet [6].

A database is modelled by a collection of variables. Dbs is the domain of database states. A database state associates each variable with its value. $Rule$ is the domain of rules. Assume that a fixed set R_0 of rules is given. In the following we define the meaning $\mathcal{M}$ of R_0. To keep the definition readable, we do this without direct reference to R_0. We assume that a function, *triggered*, when given a rule r, will yield the set $R \subseteq R_0$ of rules that could be triggered by the execution of r.

The meaning $\mathcal{M}$ of R_0 is a function which, given a set R of initially triggered rules and an initial database state d, yields a new database state. Since non-termination is a possibility, $\mathcal{M}$ is in fact partial. We model this by allowing for a result, $\perp$, signifying undefinedness. That is,

$$\mathcal{M} : \mathcal{P}(Rule) \to Dbs \to Dbs_\perp$$

where $Dbs_\perp \;=\; Dbs \cup \{\perp\}$ with ordering $d \;\le\; d'$ iff $d \;\in\; \{\perp\;,\; d'\}$

Since rules can activate each other recursively, the meaning $\mathcal{M}$ is naturally given as a fixpoint characterisation. $\mathcal{M}$ is defined to be the least $P : \mathcal{P}(Rule) \to Dbs \to Dbs_\perp$ (wrt the pointwise ordering) satisfying:

$$\begin{aligned} P(R)(d) = {} & \text{if } R = \emptyset \text{ then } d \\ & \text{else let } r = \mathit{select}(R) \\ & \qquad\qquad R' = \mathit{triggered}(r, d) \cup (R \setminus \{r\}) \\ & \qquad\qquad d' = \mathcal{R}(r)(d) \\ & \text{in } P(R')(d') \end{aligned}$$

It is straightforward to show that $\mathcal{M}$ is well-defined. The right-hand side reads as follows: If no rules are pending then the current database state is the final result. Otherwise the rule r with highest priority is selected, and the set of pending rules is updated by removing r and adding whichever rules r triggers. The effect of r on the database state is given by a function $\mathcal{R}$ which gives the meaning of r as a state transformer $Dbs \to Dbs$. A rule consists of an event, condition and a set of actions. It produces a new state and a set of events. The function $\mathcal{R}$ gives the meaning of a rule as the result of applying its actions to the current state. If the condition of the rule evaluates to *false* in the current database state then the resulting state is just the current state. The exact definition of $\mathcal{R}$ depends on the rule language and its definition is of no interest here, but the details would be straightforward. In the next section we consider a specific language.

For analysis purposes we are interested not only in the result of the computation, but also in all the database states that are obtained at intermediate points in the execution. For this reason the denotational definition is changed into a so-called *collecting semantics* [7, 10]. The idea is to use a history to record all the possible combinations (set of pending rules, database state) that occur. The domain $Hist = \mathcal{P}(Rule) \to \mathcal{P}(Dbs)$ is used for this. For example, if at some stage we encounter a set R of pending rules in a current database state d, then the entry for R will be updated to include d. Notice that the rule r which would have been activated next is the rule in R which has the highest priority. This leads to the following definition of a collecting semantics: $\mathcal{C} : \mathcal{P}(\mathit{Rule}) \to \mathit{Dbs} \to \mathit{Hist}$ is defined by

$$\mathcal{C}(R)(d) = \mathcal{C}'(R)(d)(\lambda R\ .\ \emptyset)$$

where $\mathcal{C}'$ is the least $P : \mathcal{P}(\mathit{Rule}) \to \mathit{Dbs} \to \mathit{Hist} \to \mathit{Hist}$ satisfying

$$\begin{aligned} P(R)(d)(h) = \text{let } & h' = \mathit{update}(R)(d)(h) \text{ in} \\ & \text{if } R = \emptyset \text{ then } h' \\ & \text{else let } r = \mathit{select}(R) \\ & \qquad\quad R' = \mathit{triggered}(r,d) \cup (R \setminus \{r\}) \\ & \qquad\quad d' = \mathcal{R}(r)(d) \\ & \text{in } P(R')(d')(h') \end{aligned}$$

Again, it is possible to show that $\mathcal{C}$ is well-defined. The function $\mathit{update} : \mathcal{P}(\mathit{Rule}) \to \mathit{Dbs} \to \mathit{Hist} \to \mathit{Hist}$ handles the updating of history entries. Namely,

$$\mathit{update}(R)(d)(h)(R') = \begin{cases} h(R) \cup \{d\}, & \text{if } R = R' \\ h(R), & \text{otherwise} \end{cases}$$

In other words, $\mathit{update}(R)(d)(h)$ is a new history which is identical to h, except its value for R has had $\{d\}$ added.

The collecting semantics is not finitely computable in general, as infinite sets of database states may be generated. It represents the *ideal* information needed for many analysis tasks. In the next section we consider how to finitely approximate the collecting semantics, so as to obtain correct, albeit less precise, dataflow information in finite time.

3 Approximating the Collecting Semantics

Recall the following basic fixpoint-theoretical notions. A *preordering* on a set D is a binary relation that is reflexive and transitive. A *partial ordering* is a preordering that is antisymmetric. Given a partially ordered set (*poset*) D, for any subset S of D, $a \in D$ is an *upper bound* for S if for all $x \in S$, $x \leq a$. Moreover, a' is the *least upper bound* (lub) of S, if a' is an upper bound for S, and for each upper bound a, $a' \leq a$. When it exists, we denote the (unique) least upper bound of S by $\sqcup S$. Dually, the *greatest lower bound* $\sqcap S$ can be defined. A partially ordered set for which every subset S possesses both a least upper bound and a greatest lower bound is a *complete lattice*. A (possibly empty) subset Y of X is a *chain* if for all $y, y' \in Y, y \leq y' \ \vee \ y' \leq y$.

Our interest in complete lattices should be clear. Although we have not made it explicit in our semantic definitions, the definitions are fixpoint characterisations. Complete lattices thus help ensure well-definedness. The idea in the following is to replace the large lattice of database states by a more crude (smaller) lattice of approximate database states. Once we have made the notion of "approximates" precise, we only need to devise ways of manipulating approximations in a manner faithful to the way the basic operations of the given database language manipulate proper database states.

In the $3n + 1$ example in Section 1 we dealt informally with the notion of approximation. More formally we can specify the meaning of *none*, *even*, *odd*, and *any* by mapping each to the set of integers it stands for:

$$\begin{aligned}
\gamma(\mathit{none}) &= \emptyset \\
\gamma(\mathit{even}) &= \{z \in \mathcal{Z} \mid z \text{ is even}\} \\
\gamma(\mathit{odd}) &= \{z \in \mathcal{Z} \mid z \text{ is odd}\} \\
\gamma(\mathit{any}) &= \mathcal{Z}
\end{aligned}$$

We can also give, for each set of integers, the best approximation available:

$$\alpha(S) = \begin{cases} \mathit{none}, \text{ if } S = \emptyset \\ \mathit{even}, \text{ if } S \neq \emptyset \text{ contains only even numbers} \\ \mathit{odd}, \quad \text{if } S \neq \emptyset \text{ contains only odd numbers} \\ \mathit{any}, \quad \text{if } S \neq \emptyset \text{ contains both even and odd numbers} \end{cases}$$

These two functions form a *Galois connection* which is the basic tool for formalising "correct approximation" in abstract interpretation. We refer to them as the *concretization function* and *abstraction function*, respectively. Similarly the domain and codomain for α are called the concrete and abstract domains, respectively. In the example, α and γ are monotonic and

$$\begin{aligned}
&S \subseteq \gamma(\alpha(S)), \text{ for all } S \in \mathcal{P}(\mathcal{Z}) \\
&d = \alpha(\gamma(d)), \text{ for all } d \in \{\mathit{none}, \mathit{even}, \mathit{odd}, \mathit{any}\}
\end{aligned}$$

More generally, a Galois connection is a quadruple (γ, α, U, V) where U and V are posets and $\alpha(v) \leq u \iff v \leq \gamma(u)$, for all $u \in U, v \in V$. Approximation is defined as follows:

$$u \ \mathit{appr} \ v \iff v \leq \gamma(u).$$

There are standard ways of extending this notion to composite domains, such as Cartesian products and function spaces [10].

The powerful correctness theorem that we have alluded to can now be explained. Assume we have two chains, $\{u_i\}_{i \in I}$ and $\{v_i\}_{i \in I}$, such that u_i *appr* v_i for each $i \in I$. Then $\sqcup_{i \in I} u_i$ *appr* $\sqcup_{i \in I} v_i$. In this sense, local correctness automatically implies global correctness. If we guarantee correct manipulation of approximations by all basic operations, then we automatically approximate least fixpoints correctly. Moreover, the design of approximate operations does not need to be *ad hoc*. For each operation $f : V \to V$ on the concrete domain, the Galois connection determines a unique best approximation $f' : U \to U$, namely $f' = \alpha \circ f \circ \gamma$.

We now make these points more specific, by considering example concrete and abstract domains. The rule language we consider is of the usual Event-Condition-Action form. Variables range over $\mathcal{Z}$.

- An event is of the form $U(X)$ which means "update variable X." It is triggered by any assignment to X.

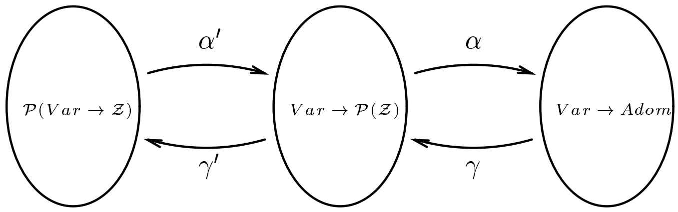

Fig. 3. The two steps needed to approximate sets of database states

- A condition is a conjunction of simple conditions. A simple condition is a test of the form $X = n$ or $X = Y$, where X and Y are variables and n is an integer.
- An action is a sequence of simple actions. A simple action is an assignment of the form $X{:=}Y$ or $X{:=}n$.

As an example abstract domain let us use the set

$$Adom = \mathcal{Z} \cup \{\bot, neg, pos, nonpos, nonzero, nonneg, \top\}.$$

The function $\gamma : Adom \to \mathcal{P}(\mathcal{Z})$, defined by

$$\gamma(d) = \begin{cases} \emptyset, & \text{if } d = \bot \\ \{d\}, & \text{if } d \in \mathcal{Z} \\ \{n \in \mathcal{Z} \mid n < 0\}, & \text{if } d = neg \\ \{n \in \mathcal{Z} \mid n > 0\}, & \text{if } d = pos \\ \{n \in \mathcal{Z} \mid n \leq 0\}, & \text{if } d = nonpos \\ \{n \in \mathcal{Z} \mid n \neq 0\}, & \text{if } d = nonzero \\ \{n \in \mathcal{Z} \mid n \geq 0\}, & \text{if } d = nonneg \\ \mathcal{Z}, & \text{if } d = \top \end{cases}$$

gives the meaning of elements of *Adom*. The domain's Hasse diagram is shown in Figure 2. The ordering is induced by the subset ordering on $\mathcal{P}(\mathcal{Z})$.

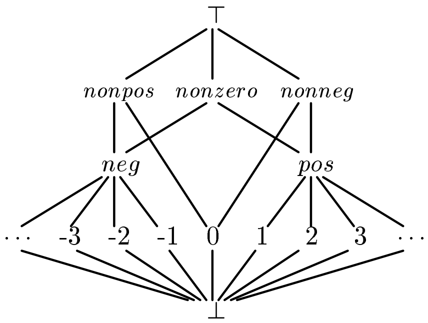

Fig. 2. The abstract domain *Adom*

The collecting semantics is approximated in two steps, as indicated in Figure 3. Assuming a finite set Var of database variables, a database state is a finitely based function in $Dbs = Var \to \mathcal{Z}$. First the domain $\mathcal{P}(Dbs)$ is replaced by $Var \to \mathcal{P}(\mathcal{Z})$. This involves an approximation by which the set of functions F is approximated by the single function f' defined by

$$f'(V) = \{n \mid f(V) = n \ \text{ for some } f \in F\}.$$

(It is straightforward to show that this determines a Galois connection between the two domains.) Next the domain $Var \to \mathcal{P}(\mathcal{Z})$ is replaced by $Var \to Adom$. Here the approximation is induced by the *appr* relation: the function $f' : Var \to Adom$ approximates $f : Var \to \mathcal{P}(\mathcal{Z})$ if $f'(V)$ *appr* $f(V)$ for all variables V.

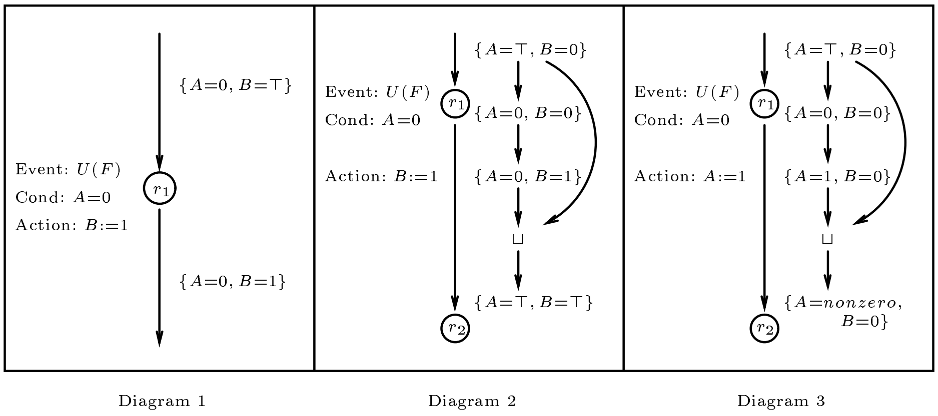

Fig. 4. Approximate execution of rules

Example 1. Let us consider the approximate execution of rules. Recall that a rule is seen as a database state transformer. The meaning of a rule here has type $(Var \rightarrow Adom) \rightarrow (Var \rightarrow Adom)$ and a denotational definition is straightforward. We omit the details, giving instead, in Figure 4, a few examples of "approximate" rule behaviour.

Diagram 1 shows the result of executing a rule. In the initial database state, A is 0, while nothing is known about B. This is sufficient to guarantee that the action B:=1 is executed, so the resulting database state has $A = 0$ and $B = 1$.

Diagram 2 considers the case where a rule's condition involves an indefinite variable A. We proceed by case analysis. In case $A = 0$, action B:=1 results in a database state which has $A = 0$ and $B = 1$. Alternatively, no action is taken, so $A = \mathit{nonzero}$ and $B = 0$. The least upper bound of these two (the junction labelled '$\sqcup$' represents this operation) is the state $A = \top$ and $B = \top$.

Diagram 3 shows an example of a *self-disactivating* rule. The rule r_1 makes its own condition *false*. The resulting approximate database state is sufficiently precise that we can detect self-disactivation in this case. It is obtained as the least upper bound of $\{A = 1, B = 0\}$ and $\{A = \mathit{nonzero}, B = 0\}$. The latter state is a *sharpening* of the original $\{A = \top, B = 0\}$, incorporating the assumption that the condition $A = 0$ was false. ■

The example illustrated the approximate execution of rules. The result of executing an entire set of rules is determined simply by approximating the collecting semantics. In particular, the function that updates the "history" remains the same except that its type is $\mathcal{P}(Rule) \rightarrow Dbs' \rightarrow Hist' \rightarrow Hist'$, where $Dbs' = Var \rightarrow Adom$ and $Hist' = \mathcal{P}(Rule) \rightarrow \mathcal{P}(Var \rightarrow Adom)$.

4 Termination Analysis

A rule set is said to terminate if, for any initial database state and any initial triggering event, rules cannot activate each other indefinitely. A termination detection algorithm will typically return either an answer that the rule set *definitely* terminates or an answer that it *may* terminate. The approximate semantics developed so far can be used for this type of analysis. Essentially, if, during approximate execution, we never encounter the same (pending rule set, approximate database state) combination twice, then the rules must terminate. Only finitely many such combinations are possible, in spite of *Adom* being infinite, this is because rules are unable to create new constants. Ways to relax this restriction are examined in section 5.

We now look at two examples of termination analysis. After that we give an algorithm based on what we have developed so far. We stress that, although the first example is taken from a paper which only deals with a 0–1 language, we read the rules as elements of a more expressive language, in which variables can take any integer value. Still we obtain comparable or better precision. That is, we improve expressiveness and precision simultaneously.

Example 2. This example of non-terminating behaviour is due to Baralis *et al.* Figure 5 shows the rule system.

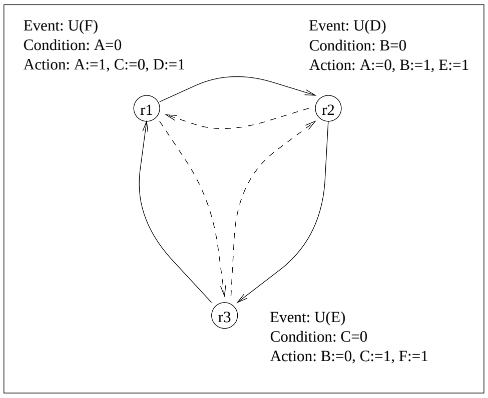

Fig. 5. Triggering/activation graph

A solid arc from one rule to another indicates that the first rule's action can trigger the event of the second rule (the *triggering graph* [2]). For example, there is a solid arc from r_3 to r_1 because r_3 contains the action F:=1 which causes the event of r_1 to trigger. The dashed arcs represent the *activation graph* [5]. Thus, there is a dashed arc from rule r_2 to rule r_1 because r_2 contains the action A:=0 which can make the condition of r_1 true. The triggering and activation graphs are used in the *rule reduction algorithm* [3]. In this example the rule reduction algorithm cannot eliminate any rules and so the rule set is deemed possibly non-terminating. This is a precise statement since this rule set is non-terminating in general (for example, if A and B are both 0 initially and r1 is triggered).

In Table 3, trace 1 gives the trace of execution for a method based on abstract interpretation. After six steps our algorithm detects a repeating state (3 and 6) and possible non-termination is signalled. ■

trace 1

Step	A	B	C	D	E	F	set
0	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\{r_1\}$
1	1	$\top$	0	1	$\top$	$\top$	$\{r_2\}$
2	0	1	0	1	1	$\top$	$\{r_3\}$
3	0	0	1	1	1	1	$\{r_1\}$
4	1	0	0	1	1	1	$\{r_2\}$
5	0	1	0	1	1	1	$\{r_3\}$
6	0	0	1	1	1	1	$\{r_1\}$

trace 2

Step	A	B	C	D	E	F	set
0	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\{r_1\}$
1	1	$\top$	0	1	$\top$	$\top$	$\{r_2\}$
2	0	1	1	1	1	$\top$	$\{r_3\}$

Table 3. Traces of execution

Example 3. Now change r_2 slightly, adding C:=1 to its actions. Trace 2 shows the trace of the resulting computation. Our algorithm signals termination after two steps, outperforming the rule reduction algorithm, as the triggering/activation graph remains the same as in Figure 5. Our algorithm yields a better result because it is able to mimic the dataflow of the real execution. Note that we only show traces assuming that r_1 was the first rule activated. In practice it would be necessary to give additional traces for when r_2 or r_3 is initially activated. ■

The algorithm follows the outline provided by the collecting semantics, but additional features have been added to obtain a precise termination analysis.

```
For each rule r0 do
   For each set R do h[R] := ∅
   For each variable V do d[V] := ⊤
   R = {r0}
   While R ≠ ∅
      if d ∈ h[R] then (* we have met this state before *)
         report "rules may not terminate"; exit
      else h[R] := h[R] ∪ {d}
      r := select(R)
      R' := triggered(r) ∪ (R \ {r})
      if must_satisfy(d, cond(r)) then d := eval(action(r), d)
      else if may_satisfy(d, cond(r)) then
         if R = {r} then d := eval(action(r), d) (*assume cond(r) true *)
         else d := sharpen(d, cond(r)) ⊔ eval(action(r), d);
              R' := mark(triggered(r)) ∪ (R \ {r})
      else R' := R \ {r} (* must not satisfy *)
      R := R'
   end While
report "rules must terminate"
```

Since database states are approximate, the evaluation of a rule condition can have three outcomes:

1. The function *must_satisfy* takes an approximate database state d and a condition c and returns true iff c holds for *every* database state approximated by d and the selected rule is not marked. If it returns true then the function *eval* is called to yield the state that results after the rule's action has been executed.
2. The function *may_satisfy* takes the same arguments and returns true iff c holds for *some* database state approximated by d. In general we should then take the least upper bound of two possible results: d itself (if r is not activated) and $eval(action(r), d)$ (in case r is activated). However, we can improve on this: If r was the only rule pending then we can disregard the result of not activating r, since it will not be used. Furthermore, knowing that d does not satisfy the condition of r can sometimes allow a *sharpening* of d before taking its least upper bound with $eval(action(r), d)$. The function *sharpen* is for this purpose. For example, with $d(A) = \top$ and the condition $A = 0$, we know that $d(A) = nonzero$ is correct when the test fails. For safety reasons, sharpening is constrained to only have an effect when there are no more than two rules in the pending set and a rule is marked if triggered as the result of a *may_satisfy* rule.
3. In the remaining case, the rule will definitely not be activated, so it is simply removed from the set of pending rules. No new rules can be triggered.

Notice that the algorithm assumes that exactly one rule is initially triggered. However, it is a simple modification to allow arbitrary initial configurations, by making R and d parameters.

Theorem 1. *If our algorithm signals that the rule set is terminating, then the rule set terminates for any initial database state and triggering event.* ■

Theorem 2. *The abstract interpretation algorithm always terminates.* ■

Let us compare our algorithm with the rule reduction method for a language where variables can only take the values 0 or 1. For the rule reduction method to remove a rule, either it has no incoming triggering arc or no incoming activation arc. In case of no incoming triggering arc, no other rule can trigger it and thus our algorithm will never be able to consider it for execution. In case of no incoming activation arc, the rule must be self-disactivating and no other rule's action can make it true. Hence, our algorithm executes it a finite number of times and without state repetition. Let us state this result as a lemma:

Lemma 3. *Every rule that is eliminated by the rule reduction algorithm is executed a finite number of times and without state repetition by the abstract interpretation algorithm.* ■

We now claim that the abstract-interpretation-based algorithm is at least as precise as the rule reduction method for simple conditions in the 0-1 rule language. To prove this, we only need to show that it signals termination whenever the rule reduction algorithm reports termination. Suppose the rule reduction algorithm reports termination. This implies it is able to reduce the input set of rules to the empty set. This means that every rule in the rule set is eventually eliminated. By Lemma 3, the abstract interpretation algorithm will eventually be unable to execute any rule and thus it signals termination. Thus we have:

Proposition 4. *The abstract interpretation algorithm, applied to the 0-1 rule language is uniformly at least as precise as the rule reduction method provided the rule set only contains simple conditions.* ∎

Note that for simple triggering graphs (that is, where each rule can trigger at most one other rule), Proposition 4 also holds for compound conditions. In the case of non-simple graphs, we can modify our algorithm such that we keep a disjunction of variable dependencies and thus still achieve greater precision.

5 Extending the Rule Language

The previous section defined an algorithm to handle a language which was unable to construct new constant values. It is straightforward to extend this to a language which uses arithmetic and inequalities, however, provided we are willing to sacrifice some precision. The extension necessitates we alter the way we update our history of database states. Instead of adding a new state to the set of previous states, we only record one entry for each pending set and incrementally refine its value by taking the least upper bound with new states as they arise. This implies some loss of precision in our analysis, but guarantees termination. The advantage is that termination and correctness of this new algorithm come almost for free, owing to the framework's flexibility. Also, by more complex choices of abstract domain, we can regain precision at the expense of efficiency.

We now consider how we would approximate the collecting semantics of a more complex rule language. We wish to examine rules which can perform relational algebra operations on a relational database. For this, it is necessary that we be able to represent the state of the database as we conduct our analysis. The following theorem states that we could do this using just variables which have the value 0 or 1, subject to certain restrictions.

Theorem 5. *Suppose we have a relational database with a schema such that every attribute of a relation has a finite, ordered domain. It is possible to represent the contents of any possible database state by a finite number of 0-1 variables.* ∎

This is true because it is possible to finitely enumerate the tuples that could occur. The presence of a tuple can be signified by '1' and its absence by '0'. The order on attributes allows us to construct an index to access any combination of attribute values. The truth value of a relational algebra condition can be determined by testing the values of the appropriate 0-1 variables, and the effect of a relational algebra action can be captured by appropriate assignments.

Therefore, in principle, if we have finite, ordered domains, we can use the method previously described to analyse for properties in a rule system which uses relational algebra. Obviously, the complexity of this procedure would be prohibitive for large domains, so we now sketch ways in which it could be reduced. The first thing to note is that we only need to simulate relations which are used by rules, the rest can be ignored. Second, if we are analysing for termination, then as a first pass one of the graph based methods such as the rule reduction method can be used to highlight which rules *may* be part of cycles, all other rules can be disregarded. The abstract interpretation method can then be used to conduct a more refined termination analysis of the relevant rules. Third, we need not store every previous database state in a history, but can in fact store delta relations as in [4] which describe the differences between successive states. Fourth, if we have information on some semantic properties of the data, such as functional dependencies and constraints, this can be used to eliminate further potential database states. Fifth, if the complexity is still too high, we can sacrifice some precision by abstracting the attribute domains (e.g by range compression).

6 Conclusion

We have suggested abstract interpretation as a tool to help solve static analysis problems for active rules. The approach has a number of advantages:

- The approach is generic, so that the same formal machinery can help solve a variety of problems.
- It is well-understood and based on a solid mathematical foundation.
- Termination (of the analysis) does not require complicated *ad hoc* proofs. Simple properties of the abstract domain (finiteness, ascending chain property, and others) suffice to guarantee termination.
- Correctness proofs are simplified, since global correctness follows automatically whenever basic operations are approximated correctly.
- The technique allows/forces one to think in semantic terms and conveniently separate two concerns: approximation and implementation [10].
- The technique can be used for analysis of other properties in a rule system such as the detection of redundant rules or determination of the reachability of particular database states as a result of trigger activation (which has applications in the static analysis of transactions).

This paper contains an example of an abstract domain and its use for termination analysis for active rules. The intention has been to demonstrate applicability and show how, by simple means, we can improve on previous "syntactic" methods. However, the termination analysis is far from definitive. Much more work is needed to design abstract domains that are useful for practical active databases. Here 'useful' means both 'sufficiently expressive' and 'efficiently realisable'. One possibility that should not be overlooked is the mixing of technologies to form hybrid methods, so that for example relatively expensive abstract interpretation is replaced by cheaper methods once analysis time exceeds a given limit.

Although we sketched in Section 5 how our method can be extended for more complex cases, it is not clear how feasible this will be when extensions for relational algebra are incorporated. There are several other avenues for future work:

- Exploring other abstract domains, in particular for interval analysis.
- Experimentally comparing the precision of termination analysis using abstract interpretation with that of current syntactic methods.
- Exploring the effect of non-determinism.
- Applying abstract interpretation to other problems in active databases, such as confluence and optimisation of conditions.

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. Behavior of database production rules: Termination, confluence and observable determinism. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 59–68. ACM Press, 1992.
3. E. Baralis, S. Ceri, and S. Paraboschi. Improved rule analysis by means of triggering and activation graphs. In T. Sellis, editor, *Rules in Database Systems*, Lecture Notes in Computer Science 985, pages 165–181. Springer-Verlag, 1995.
4. E. Baralis, S. Ceri, and S. Paraboschi. Run-time detection of non terminating active rule systems. In *Proceedings of the Fourth International Conference on Deductive and Object Oriented Databases*, pages 38–54, Singapore, 1995.
5. E. Baralis and J. Widom. An algebraic approach to rule analysis in expert database systems. In *Proceedings of the 20th International Conference on Very Large Databases*, pages 475–486, Santiago, Chile, 1994.
6. T. Coupaye and C. Collet. Denotational semantics for an active rule execution model. In *Proceedings of the Second International Workshop on Rules in Database Systems*, pages 36–50, Athens, Greece, 1995.
7. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2&3):103–179, 1992.
8. A. Karadimce and S. Urban. Conditional term rewriting as a formal basis for analysis of active database rules. In *Proceedings of the Fourth International Workshop on Research Issues in Data Engineering*, pages 156–162, Houston, Texas, 1994.
9. A. Karadimce and S. Urban. Refined triggering graphs: A logic based approach to termination analysis in an active object-oriented database. In *Proceedings of the 12th International Conference on Data Engineering*, New Orleans, Louisiana, 1996.
10. 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.
11. L. Van der Voort and A. Siebes. Termination and confluence of rule execution. In *Proceedings of the Second International Conference on Information and Knowledge Management*, Washington DC, 1993.
12. J. Widom. A denotational semantics for Starburst production rule language. *ACM SIGMOD Record*, 21:4–9, 1992.