On the Decidability of the Termination Problem of Active Database Systems*

James Bailey[†] University of Melbourne

Guozhu Dong[‡] Wright State University

Kotagiri Ramamohanarao[§] University of Melbourne

Abstract

Active database systems enhance the functionality of traditional databases through the use of active rules or ‘triggers’. One of the principal analysis questions for such systems is that of *termination* – is it possible for the rules to recursively activate one another indefinitely, given an initial triggering event. In this paper, we study the decidability of the termination problem, our aim being to delimit the boundary between the decidable and the undecidable. We present results for two broad types of variations, variations in rule syntax and variations in meta level features. Within each of these, we identify members close to the boundary of (un)decidability and also look at the effect of combining members of each type. The maximal decidable class we present is capable of expressing some useful kinds of application requirements, such as checking and repairing inclusion constraints. The work is also interesting from a theoretical point of view, since the context is similar to the *while* query language and the dynamics gives an interesting contrast to Datalog with negation.

1 Introduction

Traditional database systems provide a mechanism for storing large amounts of data and an interface for manipulating and querying this data. They are, however, passive in the sense that their state can only change as a result of outside influences. In contrast, an active database is a system providing the functionality of a traditional database and additionally is capable of reacting automatically to state changes, both internal and external, without user intervention. The rules which define this behaviour are known as *triggers* or *active rules*. Active database systems have been intensively studied for over a decade and many prototypes have been built [38].

Rule definition most commonly follows the Event-Condition-Action (ECA) paradigm. In this, a rule is triggered by an event, in response to which it evaluates a condition and if the condition is true, then performs an action.

*Parts of results of this paper appeared in [8] and [9].

[†]Corresponding author: Department of Computer Science, University of Melbourne, Victoria 3010, Australia, e-mail jbailey@cs.mu.oz.au

[‡]Department of Computer Science and Engineering, Wright State University, Dayton, OH 45435, USA. e-mail gdong@cs.wright.edu

[§]Department of Computer Science, University of Melbourne, Victoria, 3010, Australia. e-mail rao@cs.mu.oz.au

The integration of rules within overall database functionality is defined in the *rule execution model*. Amongst other things, this provides policies for handling simultaneously triggered rules (*pending rule structure*) and for integrating rule processing with database transactions.

Major areas of research in active database systems include rule specification, rule execution models, system architectures, optimisation of rule execution, rule analysis, formal foundations and applications. Since much of this research has been motivated by the promise of greater functionality, rule language proposals have gradually become more and more complex. This has had a negative side though, since when a system contains many rules, overall behaviour may be obscure and reasoning about rule dynamics may become very complicated. One of the most important behavioural properties of rule sets, is that of *termination*.

Statement of the Problem: When several rules are defined in an active database system, there is the possibility that they may mutually activate one another: The action executed by one rule may trigger another rule, this newly activated rule may itself then trigger another rule and so on. Such triggerings could continue infinitely, causing *non termination*. Clearly, such situations should be prevented, since such behaviour could make a system unusable.

There are three principal ways to address this. Firstly, using static analysis, we can try to guarantee *a priori*, that non termination is impossible for a particular rule set. This task is made difficult, due to the complex interactions which can occur among rules. The second approach, is to impose some fixed (hardwired) limit upon the number of rules which can be executed in a triggering sequence - such a method is adopted by commercial database systems such as Oracle and Sybase. While easy to implement, it has the defect that valid rule execution sequences may exceed this limit and be prematurely halted and aborted, an approach unsuitable for applications where correctness and performance is paramount, such as mission critical systems and even banking systems. A third approach involves the imposition of syntactic restrictions on the rule set to ensure that rule execution always terminates. The difficulties of defining such criteria are recognised by the current SQL3 standard for triggers [28], which does not attempt to prescribe methods for ensuring termination.

In this paper, we examine the problem of deciding termination for various classes of active database systems. Since it is obviously undecidable in general, other work which has considered statically analysing termination, has predominantly dealt with either developing sufficient conditions on rule sets to ensure they are terminating (*approximate termination analysis* e.g. [6, 11, 10]), or on designing languages which cannot express non terminating programs [17, 30, 39, 40]. In contrast, one of the purposes of our work, is to identify which features are influential in (un)decidability and investigate the structure of the resulting system. We hope that this information can then be used to help make informed choices in rule system design.

Contributions: Our principal contribution is the identification of a number of types of active rule systems for which termination analysis lies close to the boundary of (un)decidability. Different systems can be distinguished using two general parameters: *rule language* (dealing with the rule syntax) and *rule meta language* (dealing with the rule execution model). Within the first, we identify a powerful decidable class called the safe-cones language, which can satisfy the expressiveness requirements of some practical situations. We then show that minimal extensions to this language result in undecidability. Within the second, we show decidability for systems employing a stack schedule and show undecidability for a queue schedule. This undecidability result also extends to other meta features such as complex events and coupling modes. To our knowledge, this is the first paper to systematically study decidability of termination for active databases. The closest work being that of [35], where decidability of termination in N steps for a simple object oriented language is examined.

Although we focus on active databases, our work has broader applicability to database dynamics generally. The execution of a sequence of active rules can be modelled as a $while$ or $while_N$ [4] program (variants of partial fixpoint logic) and our analysis techniques can then be used to study properties such as termination and satisfiability in these formalisms also. Our results also form a natural adjunct to previous research on optimisation and analysis of logic programs (such as the decidability of boundedness [24, 20]).

Paper Outline: In section 2, we present the preliminaries needed in the paper; we also highlight some subtleties in the definition of termination. In sections 3 and 4, we study the (un)decidability of a class of languages whose definition is based on safety (and number of literals). We identify a particular decidable member called the *safe-cones* language and show how minimal extensions result in the crossing of the decidability boundary. Next, in section 5, we discuss meta features and give decidability results for variations on the pending rule structure, complex events and coupling modes. Section 6 looks at applications of the decidable cases, section 7 discusses related work and section 8 gives a summary.

2 Preliminaries

We start with some basic terms and notations. We assume familiarity with relational databases and some knowledge of active databases. For further background see [3] and [38].

We assume the existence of three disjoint infinite sets: a set **rel** of *predicate* or *relation names*, each with an associated *arity* ≥ 0, a (universal) domain **dom** of *constants* and a set of variables **var**.

For each natural number n, an *n-ary tuple* is a mapping from $\{i \mid 1 \leq i \leq n\}$ to **dom**; an *n-ary relation* is a finite set of n-ary tuples, and its *cardinality* is the number of tuples in it. A *database schema* is a finite subset of **rel**, and its *arity* is the maximal arity of relation names contained in it. A *database instance* (or a *database state*) of a database schema $\mathcal{S}$ is a mapping I such that, for each relation name R in $\mathcal{S}$, then $I(R)$ is an n-ary relation where n is R's arity.

We define the *active domain* of a relation R, denoted by $adom(R)$, to be the set of constants occurring in R. For each database instance I, we define the *active domain* of I as the union of the active domains of its relations.

For each natural number n, a *free tuple* (or a *variable pattern*) of arity n is a mapping from $\{i \mid 1 \leq i \leq n\}$ to **var**. An *atom* is either a *comparison atom* of the form $X = Y$ where X and Y are variables, or a *relational atom* of the form $R(\overline{X})$ where R is a relation name and $\overline{X}$ is a free tuple whose arity matches that of R. Notice that we disallow constants within atoms. We will allow, however, the propositional constant $true$, which is considered to be a special kind of atom. Atoms are also called positive literals. A *comparison literal* is either a comparison atom or its negation; similarly for a *relational literal*.

Update Languages

An update over a relation R is either an insertion or deletion over R. It is represented by an expression of the form

$\pm R(\overline{X}) \leftarrow L_1, \cdots, L_m$

where $R(\overline{X})$ is a relational atom (the head), and $L_1, \cdots, L_m$ is a conjunction of zero or more literals (the body) such that each variable occurring in $L_1, \cdots, L_m$ occurs in at least one of its positive literals (range restriction). An empty body is assumed to equal *true*. The semantics of the update is as follows: First the set Δ containing the answer of $R(\overline{X}) \leftarrow L_1, \cdots, L_m$ as a query of nonrecursive semi-positive datalog with negation is derived; then Δ will be inserted to R if $+$ is present (an insertion over R) and deleted from R if $-$ is present (a deletion over R). At this moment we impose no limitations about variables in the head and those in the body, but we will do so later on. If one or more variables $\overline{Y} = Y_1, \ldots, Y_k$ occur in the head but not in the body $L_1, \ldots, L_m$, then the query semantics is equivalent to that of the query $\pm R(\overline{X}, \overline{Y}) \leftarrow L_1, \cdots, L_m, active\_domain(Y_1), \ldots, active\_domain(Y_k)$ where all variables in $\overline{X}$ occur within $L_1, \ldots, L_m$. An update is said to be *safe* if all of its variables occur in some positive relational literals in the body.

Rule Syntax and Execution Model

We consider rules that have the "ECA" form "*on* event *if* condition *then* action" which satisfy the following requirements (a–c).

(a) An Event is represented by its event_expression, which is either of the form Insert($R_i(\overline{X}), \theta$) or Delete($R_i(\overline{X}), \theta$), where $R_i(\overline{X})$ is a relational atom and θ is a conjunction of inequalities of the form $X_1 \neq X_2$; we will simply write Insert(R_i) or Delete(R_i) if θ is absent (interpreted as $true$) and no variable in $\overline{X}$ occurs more than once. We say that an insert (delete) event_expression is true with respect to an update, if the update inserts (deletes) a non-empty set of tuples, satisfying the comparisons in θ and the implicit equality conditions in $\overline{X}$, into (from) a relation R_i that appears in the event expression. Otherwise the event_expression is false with respect to the update. Note that event expressions are only true if the contents of the relation is actually changed due to the update.

We do not allow bindings to be passed from the rule's event to the rule's condition.

(b) The syntax of a condition is the same as that of the body of an update. i.e. $answer(\overline{X}) \leftarrow L_1, \ldots, L_m$ where $L_1, \ldots, L_m$ is a conjunction of zero or more literals. The condition is $true$ if $answer$ is non empty and *false* otherwise.

(c) An action is a finite sequence of updates.

Example 2.1 The following is a trigger.

On insert(R_1)
If $R_1(X, X, Z), X \neq Z$ then
$-R_2(X, Y) \leftarrow R_3(X, Y, Y, A), A \neq X;$
$+R_8(X, X, Y) \leftarrow R_9(X, Y, Z)$

■

If a rule's action is $\delta_0; \delta_1; \ldots; \delta_n$ and it is chosen for execution in a database state I_0 where its condition is *true*, then the database state after the execution of the rule is $\delta_n(\delta_{n-1}(\ldots \delta_0(I_0) \ldots))$. An event is said to be *raised* at the completion of an action if its event_expression was true with respect to any of the updates δ_0 or δ_1 or $\ldots \delta_n$. A rule is said to be *triggered* when its event is raised [1].

It should be pointed out that there is no passing of values among conditions and updates. This does not lead to a loss of expressive power in the general case, although a loss of power may occur because of this for some of the languages studied in this paper.

We assume rules are totally ordered by some priority scheme. This restriction is not essential, however, and this issue is further discussed in section 6.

When a rule is triggered by an event in a transaction, the rule will be put on a *pending rule structure* which is used to store rules which are awaiting execution later on. This also initiates rule processing from the *initial* database state, using the following steps.

1. If there are no triggered rules pending execution, then exit rule processing and resume the transaction.
2. Select and remove a rule to execute from the pending rule structure.
3. Evaluate the condition of the selected rule.
4. If the condition is *true* then execute the action of the selected rule and goto step 1.

The action executed in step 4 can cause events and thus trigger further rules. These are added to the pending structure. Thus the steps 1-2-3-4 can loop forever.

Observe that once rule processing begins, the transaction which initiated it becomes suspended - in active database terminology this corresponds to *immediate coupling*. We will examine other kinds of coupling mode in section 5.3.2

Ultimately, we will consider pending rule structures such as sets, stacks and queues, to hold all the activated rules for different treatment strategies. For simplicity, initially in sections 3 and 4 of this paper, we will be using a *singleton* pending structure, which requires that there can only ever be one rule awaiting execution at any given time. If two or more rules are triggered simultaneously, then the one with highest priority is added to the structure and the other(s) discarded. We have chosen this simple semantics initially because it helps us to isolate the effect that variations in rule syntax have on termination decidability.

[1] Observe that an event "on insert(R)" could be raised at the completion of an action which has produced no net change in R. This is because one or more individual updates within the action may still have changed R.

Termination

We now formally define the property of termination for active rules.

Definition 2.2 (a) A set of rules is *globally terminating*, if for any initial database state and triggering event, rule processing terminates; it is *globally non terminating* otherwise.

(b) A set of rules is *locally terminating on event* e, if for any initial database state and a triggering event e, rule processing terminates; it is *locally non terminating on event* e otherwise. ■

Local termination analysis is performed on a rule set with a well defined starting point; so the execution of rules can be essentially regarded as a deterministic $while$ or $while_N$ program [3]. For global termination, we need to analyse programs with a very limited kind of non determinism, since the starting point is not fixed and in fact corresponds to the first rule triggered. We now compare these two notions. Firstly, it is easy to show that deciding local termination is at least as difficult as deciding global termination.

Proposition 2.3 *If local termination is decidable then global termination is decidable.*

Proof: Analysis is conducted for all possible initial triggering events to see whether the rules are locally terminating. The system is globally terminating iff the rules are locally terminating for all possible initial triggering events. ■

Surprisingly, the converse of this proposition is false in general. Intuitively, this is because in order to decide local termination, we may need to conduct some reachability analysis to see whether a cycle can be reached. On the other hand, since we have a weak kind of non determinism in the global case, such a cycle can always be activated by the first triggering event and so reachability analysis is unnecessary. The following result is valid for ECA rules using the syntax and semantics already defined, with the additional proviso that the condition part is now allowed to be a first-order query.

Proposition 2.4 *There exists a class of rule sets for which global termination is decidable and local termination undecidable.*

Proof: We define a class of rule sets where rules within a set are in one of two categories, *regular* rules or *special* rules. Regular rules have the format "on $Delete(R)$ if c then $+R(\overline{X}) \leftarrow R'(\overline{Y})$" where c is a first order query and R and R' are any relation names. Note that no deletions are permitted. Special rules have a fixed format "on $insert(S)$ if *true* then $-S \leftarrow S; +S \leftarrow true$" ($S$ is a zero-arity relation) and have priority higher than any regular rule. Clearly, once a special rule is triggered, rule processing will not terminate. Consequently, a rule set in this class is globally non terminating iff it contains one or more special rules. On the other hand, suppose a regular rule of the form "on $Delete(R)$ if c then $+S \leftarrow R'(X)$" and a special rule whose event is "on $Insert(S)$" are defined. Then this rule set is locally non terminating from (external) event $Delete(R)$ iff c is satisfiable. Hence local termination is undecidable. ■

Previous work (such as [6]) has only considered global termination. In light of the above two results, we believe that local termination is a more suitable (and general) notion for studying decidability. Henceforth, when we refer to termination without any qualification, we will mean local termination.

For analysing termination behaviour, we also need to define what we understand by the term system state. By *system state*, we mean an ordered pair $(I, \mathcal{R})$, where I is a database instance and $\mathcal{R}$ is an instance of the pending rule structure.

Proposition 2.5 *Rule execution will not terminate if some system state* $(I, \mathcal{R})$ *(where* $\mathcal{R} \neq \emptyset$*) occurs twice.*

Proof: If $\mathcal{R} \neq \emptyset$, then execution cannot have halted. Since the semantics is deterministic, the state $(I, \mathcal{R})$ must be repeated infinitely often and so we get non-termination. ■

For certain kinds of structures (and in particular the singleton pending structure), the 'if" can be replaced by an 'iff". This will form the basis of the decision procedures developed later. However, there are cases (such as stack or queue pending structures) where the "if" cannot be replaced by "iff".

Iteration Simulation

We will often use rules to simulate various kinds of state machines. To facilitate this, we sometimes use a procedural description using *while* loops instead of defining individual rules. That this can be done is not surprising, considering the relationships between *while* languages and active rules established in [33].

For instance, consider the statement

$< b_1 >$
While (c) do
$< b_2 >$
End While
$< b_3 >$

where each $< b_i >$ is a sequence of updates and c is a condition. It is equivalent to the following rules (assuming the singleton pending structure) where $e_1, \ldots e_4$ are relations of arity zero

Rule $\mathbf{r_1}$	**Rule $\mathbf{r_2}$**	**Rule $\mathbf{r_3}$**	**Rule $\mathbf{r_4}$**
On $ins(e_1)$	On $ins(e_2)$	On $ins(e_3)$	On $ins(e_4)$
If *true*	If *true*	If *true*	If *true*
$< b_1 >$;	$trigger(e_3) \leftarrow c$;	$< b_2 >$;	$< b_3 >$;
$trigger(e_2)$;	$trigger(e_4)$;	$trigger(e_2)$;	

where r_1 is the first rule triggered, e_1 is a distinguished event that initiates rule execution, and $priority(r_3) > priority(r_4)$. The notation $trigger(e)$ represents an update with respect to which the event expression on relation e is true. *e.g.* a deletion/insertion pair such as $-e \leftarrow e$;$+e \leftarrow true$. The statement $trigger(e_3) \leftarrow c$ can be translated as $-e_3 \leftarrow e_3, c$; $+e_3 \leftarrow c$.

3 Decidability of the Safe-Cones Trigger Language

In this section we introduce the *safe-cones trigger language*. The language is powerful enough to be useful for some practical applications and we prove that termination is decidable for it. In section 4 we will show that termination is undecidable for several languages violating the safe-cones condition.

We establish the decidability result by reducing safe-cones triggers to those in its simplest sublanguage, namely the safe one-literal triggers; we then prove the decidability of the latter by establishing a bounded model property of that sublanguage.

3.1 The safe-cones trigger language

Intuitively, the body of each safe-cones update contains a tree (but not a lattice) of relational literals, where the parent-child relationship corresponds to the superset-subset relationships between their sets of variables, and where the relational literals at the top are positive.

To formalize the notion, we need several auxiliary definitions. The *variable set* of a relational literal L, denoted by $\text{VAR}(L)$, is the set of variables that occur in L. Given a set $\mathcal{L}$ of relational literals, let $\mathcal{V}_{\mathcal{L}}$ be the minimal collection (of variable sets) which contains $\{\text{VAR}(L) \mid L \in \mathcal{L}\}$ and is closed under intersection; its *variable-set collection*, denoted by $\mathcal{V}^+_{\mathcal{L}}$, is defined as $\mathcal{V}_{\mathcal{L}} - \{\emptyset\}$.

For example, $\text{VAR}(R(X,Y)) = \{X,Y\}$. Moreover, for

$\mathcal{L} = \{R_4(X,Y,Z), \neg R_2(Y,X), R_5(A,B), \neg R_1(B,A), R_3(A,C)\}$,

we have $\mathcal{V}^+_{\mathcal{L}} = \{\{X,Y,Z\},\{X,Y\},\{A,B\},\{A,C\},\{A\}\}$; observe that $\{A\}$ is included because of the intersection closure requirement, and that $\{X,Y\}$ is included due to a negative literal.

Given a collection $\mathcal{S}$ of sets and a set S in $\mathcal{S}$, we say S is maximal in $\mathcal{S}$ if there is, in $\mathcal{S}$, no proper superset of S; similarly we define minimal sets of $\mathcal{S}$; moreover, we say S' is a maximal subset of S in $\mathcal{S}$ if $S' \in \mathcal{S}$, $S' \subset S$, and there is no S'' in $\mathcal{S}$ such that $S' \subset S'' \subset S$. ($S' \subset S$ means $S' \subseteq S$ and $S' \neq S$.)

Definition 3.1 A collection $\mathcal{L}$ of relational literals is said to *form cones* if (i) each set in $\mathcal{V}^+_{\mathcal{L}}$ has at most one maximal subset in $\mathcal{V}^+_{\mathcal{L}}$ and (ii) for each maximal set V in $\mathcal{V}^+_{\mathcal{L}}$, there is some positive relational literal L in $\mathcal{L}$ such that $V = \text{VAR}(L)$. The collection of all supersets of a minimal variable set V in $\mathcal{V}^+_{\mathcal{L}}$ is called a *cone*. ■

For example, the following set of relational literals form cones:

$R_4(X,Y,Z), R_1(X,Y), \neg R_2(Y,X), X \neq Z,$
$R_5(A,B), \neg R_1(B,A), R_3(A,C), A \neq C, A \neq B$

It contains these two cones: $\{\{X, Y\}, \{X, Y, Z\}\}$ and $\{\{A\}, \{A, B\}, \{A, C\}\}$; condition (i) of Definition 3.1 is satisfied because each set in the collection has at most one subset in the collection, and condition (ii) of Definition 3.1 is satisfied because (a) $\{X, Y, Z\}$ is the variable set of the positive literal $R_4(X, Y, Z)$, (b) $\{A, B\}$ is that of the positive literal $R_5(A, B)$, and $\{A, C\}$ is that of the positive literal $R_3(A, C)$.

Definition 3.2 The *safe-cones trigger language* consists of triggers of the form "on e if c then a", where e is an event, c is a safe-cones condition, and a is a sequence of safe-cones updates. A condition c is a *safe-cones* condition if it is a conjunction of literals such that (i) its subset of relational literals forms cones and (ii), for each $X \neq Y$ or $X = Y$ in c, X and Y are contained in some common relational literal in c. An update $\pm head \leftarrow body$ is a *safe-cones update* if (a) *body* is a safe-cones condition, and (b) the variable set $\text{VAR}(head)$ is contained in the minimal variable set of a cone. ■

Example 3.3 An example safe-cones update is

$$\begin{aligned} +R_3(X, Y) \leftarrow\ & R_4(X, Y, Z), R_1(X, Y), \neg R_2(Y, X), X \neq Z, \\ & R_5(A, B), \neg R_1(B, A), R_3(A, C), A \neq C, A \neq B \end{aligned}$$

The body, considered above, contains two cones, namely $\{\{X, Y\}, \{X, Y, Z\}\}$ and $\{\{A\}, \{A, B\}, \{A, C\}\}$. The body is also an example of a safe-cones condition. Another example safe-cones update is

$$\begin{aligned} +R(X) \leftarrow\ & R_4(X, Y, Z), R_1(X, Y), \neg R_2(Y, X), X \neq Z, \\ & R_5(A, B), \neg R_1(B, A), R_3(A, C), A \neq C, A \neq B \end{aligned}$$

The difference between this update and the previous one is that $\{X\} = \text{VAR}(R(X))$ is properly contained in a minimal set of the cone $\{\{X, Y\}, \{X, Y, Z\}\}$, not equal to it. A third example safe-cones update is

$$+R(X) \leftarrow T(X, C), T(Y, A), T(A, B).$$

One should compare this with the second non-safe-cones example below to see the subtle differences.

Three example non-safe-cones updates are

$$\begin{aligned} &+R(X) \leftarrow tc(X, A, B), tc(X, A, C), tc(X, B, C) \\ &+R(X) \leftarrow T(X, A), T(A, B) \\ &+R_3(X, Y) \leftarrow R_4(X, Y, Z), R_5(A, B), X \neq A \end{aligned}$$

The body of the first update corresponds to the following variable-set collection $\mathcal{V}_1$:

{{X,A,B},{X,A,C},{X,B,C},{X,A},{X,B},{X,C},{X}}.

This update violates the safe-cones condition because its body does not form cones: the set $\{X, A, B\}$ contains two maximal subsets, namely $\{X, A\}$ and $\{X, B\}$, which are in $\mathcal{V}_1$. The second update is not safe-cones because the variable set $\{X\}$ of the head is not contained in the minimal variable set of any cone. The third update is not safe-cones because its body contains the comparison $X \neq A$, but X and A are not contained in a common relational literal in the body. ■

Observe that one can express arbitrary propositional conjunctions, since they do not use variables.

The simplest type of safe-cones triggers are safe one-literal triggers.

Definition 3.4 A *safe one-literal trigger* is a safe-cones trigger having exactly one relational literal in its condition and exactly one relational literal in the body of each of its updates; and we will call such conditions (respectively updates) safe one-literal conditions (respectively updates). ■

The trigger given in Example 2.1 is also a safe one-literal trigger.

3.2 The Main Result

The proof of our main result is by reducing safe-cones triggers to safe one-literal triggers.

Lemma 3.5 *Termination is decidable for the safe one-literal language.*

The proof of this lemma is long and involved and is given in the appendix.

The following lemma will be useful in simplifying proofs.

Lemma 3.6 *Using safe one-literal updates, we can simulate the following relational algebra operations: union ($\cup$), intersection ($\cap$), projection (π), selection (σ_c) where c consists of comparisons (equality and inequality), and set difference ($-$).*

Proof: Clearly a projection and a selection can each be expressed as one safe one-literal update. The union operation $T = R \cup S$ can be done by initialising T to empty and using two insertion updates. The difference operation $T = R - S$ can be done by initialising T to empty and first copying R into T and then deleting all tuples occurring in S from T. The intersection operation $T \leftarrow R \cap S$ is equivalent to $T = R - (R - S)$. ■

Example: We simulate the query $Q = (R \cup S) - (S \cup \Pi_{\overline{M}} T)$ using safe one-literal updates. We first define three temporary relations tmp_1, tmp_2 and tmp_3, where arity($tmp1$)=arity(R), arity(tmp_2)=arity(S) and arity(tmp_3)=arity($\Pi_{\overline{M}} T$). We then perform the following one literal update sequence:

$-Q(\overline{X}) \leftarrow Q(\overline{X})$; $-tmp_1(\overline{X}) \leftarrow tmp_1(\overline{X})$; $-tmp_2(\overline{X}) \leftarrow tmp_2(\overline{X})$; $-tmp_3(\overline{X}) \leftarrow tmp_3(\overline{X})$;
$+tmp_3(\overline{M}) \leftarrow T(\overline{Y})$ (n.b. $M \subseteq Y$)
$+tmp_2(\overline{X}) \leftarrow S(\overline{X})$
$+tmp_2(\overline{X}) \leftarrow tmp3(\overline{X})$
$+tmp_1(\overline{X}) \leftarrow R(\overline{X})$
$+tmp_1(\overline{X}) \leftarrow S(\overline{X})$
$+Q(\overline{X}) \leftarrow tmp_1(\overline{X})$
$-Q(\overline{X}) \leftarrow tmp_2(\overline{X})$
■

We now state and prove the main result.

Theorem 3.7 *Termination is decidable for the safe-cones trigger language.*

Proof: We will prove this result by simulating each safe-cones update by a sequence of safe one-literal triggers. Roughly, we will use one safe one-literal trigger to simulate the query given by a cone in a safe-cones update, and will link these triggers together through some appropriate events.

In this simulation, we use a set of new, scratch-paper relations, denoted by scripted TMP relations. We illustrate this simulation using an example.

$$-R(X,Y) \leftarrow S(X,A,Y,Z), T(X,Y), \neg T_1(X,Y), Q(Y,Z,X,B), X \neq Y, T(W_1,W_2), T(W_1,W_1)$$

There are two cones for the body of this update. A key point to note is that the cone containing $\{X,Y\}$ determines what tuples might be removed from R, whereas the cone containing $\{W_1\}$ determines whether these potential removals should actually occur.

For each variable set, we will have a bounded number of scratch-paper relations, depending on the number of supersets this variable set has. There is only one relational literal for the variable set $\{X,A,Y,Z\}$ and this literal happens to correspond to a maximal variable set; we assign one scratch paper relation, say TMP_{XAYZ}, for it and initialize it to contain the value of (the answer to the query) $S(X,A,Y,Z)$. Similarly, let TMP_{YZXB} be the scratch paper relation for the variable set $\{Y,Z,X,B\}$ and let it be initialized to $Q(Y,Z,X,B)$. We then use some updates and scratch paper relations to find the projection $\text{TMP}^1_{XYZ} = \Pi_{XYZ}(\text{TMP}_{XAYZ})$, the projection $\text{TMP}^2_{XYZ} = \Pi_{XYZ}(\text{TMP}_{YZXB})$, and the intersection $\text{TMP}_{XYZ} = \text{TMP}^1_{XYZ} \cap \text{TMP}^2_{XYZ}$. Then we find the projection $\text{TMP}^1_{XY} = \Pi_{XY}(\text{TMP}_{XYZ})$, the intersection $\text{TMP}^2_{XY} = \text{TMP}^1_{XY} \cap T$, and the difference $\text{TMP}^3_{XY} = \text{TMP}^2_{XY} - T_1$, and finally, $\text{TMP}_{XY} = \sigma_{X \neq Y}(\text{TMP}^3_{XY})$; this is the set of tuples that might be removed. All these operations can be done using safe one-literal updates, by Lemma 3.6; let a_1 represent the sequence of these updates.

Similarly, we can find the value of $\text{TMP}_{W_1} = \Pi_{W_1}(T(W_1,W_2)) \cap \Pi_{W_1}(\sigma_{W_1=W_2}(T(W_1,W_2)))$. Let a_2 represent the sequence of these updates followed by an extra two updates that will raise an event if TMP_{W_1} is non-empty.

We will link these two sequences by having a trigger for each sequence. The trigger for performing a_1 is "on $Insert(\text{TMP}_{W_1})$ if true then a_1". The trigger for performing a_2 is "on ev if true then a_2", where ev is some appropriate event (for linkage or for initiation, depending on whether the safe-cones update we are simulating is the first update in the safe-cones trigger). It is this second trigger which executes first.

One can devise a general procedure to simulate arbitrary safe-cones updates. Essentially, we traverse the cones from maximal variable sets to minimal nonempty ones. For each variable set V, we find the content of its corresponding relation using intersections and selections of relations formed from projections of relations which correspond to V's parent variable sets. The potential tuples for insertion/deletion are given by the relation for the variable set of the head, and these insertions/deletions are executed if relations for all of the minimal nonempty variable sets (of the body) are nonempty. The linkage of the triggers is as illustrated in the previous paragraph. Observe that this procedure uses all conditions in the definition of safe-cones updates. ■

Intuitively, each cone represents a series of containment relationships which can be constructed in a downwards manner using one literal updates. The restrictions placed on cones mean that only one cone may contribute tuples to the result of the update, while the other cones may only control if the update may take place. The interaction between cones can then be captured using the relationships between actions and events in a set of triggers (with one literal updates).

Example: We explicitly show the one literal triggers corresponding to the safe cones trigger example used in the above theorem

on ev
if true
then
$-R(X,Y) \leftarrow S(X,A,Y,Z), T(X,Y), \neg T_1(X,Y), Q(Y,Z,X,B), X \neq Y, T(W_1,W_2), T(W_1,W_1)$

Let the notation $erase(R)$ represent the update $-R(\overline{X}) \leftarrow R(\overline{X})$ (which removes all tuples from R).

Define the sequence of updates a_1 as follows:

$erase(\text{TMP}_{XAYZ}); erase(\text{TMP}_{YZXB}); erase(\text{TMP}^1_{XYZ}); erase(\text{TMP}^2_{XYZ});$
$erase(\text{TMP}_{XYZ}); erase(\text{TMP}^1_{XY}); erase(\text{TMP}^2_{XY}); erase(\text{TMP}^3_{XY});$
$erase(\text{TMP}_{XY}); erase(\text{TMP}_{W_1W_1}); erase(\text{TMP}^1_{W_1}); erase(\text{TMP}^2_{W_1})$
$+\text{TMP}_{YZXB}(Y,Z,X,B) \leftarrow Q(Y,Z,X,B)$
$+\text{TMP}_{X,A,Y,Z}(X,A,Y,Z) \leftarrow S(X,A,Y,Z)$
$+\text{TMP}^1_{X,Y,Z}(X,Y,Z) \leftarrow \text{TMP}_{XAYZ}(X,A,Y,Z)$
$+\text{TMP}^2_{X,Y,Z}(X,Y,Z) \leftarrow \text{TMP}_{YZXB}(Y,Z,X,B)$
$+\text{TMP}_{XYZ}(X,Y,Z) \leftarrow \text{TMP}^1_{XYZ}(X,Y,Z)$
$-\text{TMP}^1_{XYZ}(X,Y,Z) \leftarrow \text{TMP}^2_{XYZ}(X,Y,Z)$
$-\text{TMP}_{XYZ}(X,Y,Z) \leftarrow \text{TMP}^1_{XYZ}(X,Y,Z)$
$+\text{TMP}^1_{XY}(X,Y) \leftarrow \text{TMP}_{XYZ}(X,Y,Z)$
$+\text{TMP}^2_{XY}(X,Y) \leftarrow \text{TMP}^1_{XY}(X,Y)$
$-\text{TMP}^1_{XY}(X,Y) \leftarrow T(X,Y)$
$-\text{TMP}^2_{XY}(X,Y) \leftarrow \text{TMP}^1_{XY}(X,Y)$
$+\text{TMP}^3_{XY}(X,Y) \leftarrow \text{TMP}^2_{XY}(X,Y)$
$-\text{TMP}^3_{XY}(X,Y) \leftarrow T_1(X,Y)$
$+\text{TMP}_{XY}(X,Y) \leftarrow \text{TMP}^3_{XY}(X,Y), X \neq Y$

and the sequence of updates a_2 by

$+\text{TMP}_{W_1W_1}(W_1,W_1) \leftarrow T(W_1,W_1)$
$+\text{TMP}^1_{W_1}(W_1) \leftarrow T(W_1,W_2)$
$+\text{TMP}^2_{W_1}(W_1) \leftarrow \text{TMP}_{W_1W_1}(W_1,W_2)$
$+\text{TMP}_{W_1}(W_1) \leftarrow \text{TMP}^1_{W_1}(W_1)$
$-\text{TMP}^1_{W_1}(W_1) \leftarrow \text{TMP}^2_{W_1}(W_1)$
$-\text{TMP}_{W_1}(W_1) \leftarrow \text{TMP}^1_{W_1}(W_1)$
$-test \leftarrow test$
$+test \leftarrow \text{TMP}_{W_1}(W_1)$

The translated one literal triggers are then

on ev	on $insert(test)$	
if true	if true	■
then a_2	then $a_1; -R(X,Y) \leftarrow \textsc{Tmp}_{XY}(X,Y)$	

As a side remark, we note that we can obtain an analogous result for *while* programs [5] using safe-cones updates, by simulating them using triggers.

4 The Undecidability of the Semijoinable Trigger Languages

In this section we define the *semijoinable* trigger languages and then establish the generic result that termination is undecidable for all such languages. This powerful generic result implies that termination is undecidable for three semijoinable languages, each of which violating the safe-cones condition in a minimal way, namely safe two-literal, unsafe-insert safe-delete one-literal, and safe-insert unsafe-delete one-literal. Thus the generic result identifies the ability of defining semijoin as influential regarding the decision problem of termination.

We now define the semijoinable trigger languages. Recall that the semijoin $R(\overline{X}) \ltimes S(\overline{Y})$ is defined as $\Pi_{\overline{X}}(R(\overline{X}) \bowtie S(\overline{Y}))$. Where $\bowtie$ is the natural join operator (which reduces to cartesian product if $\overline{X} \cap \overline{Y} = \emptyset$).

Definition 4.1 A trigger language is called *semijoinable* if it can simulate the safe one-literal language and it has the ability to calculate semijoins. ■

We first give the main undecidability result, and will give examples of the semijoinable trigger languages later.

Theorem 4.2 *Termination is undecidable for semijoinable trigger languages.*

Before turning to the proof, we first list some corollaries here and in the next subsection.

Since the semijoin can be expressed as the projection of an equality-based selection of the cross product of the two input relations, and since both projection and selection can be defined by safe one-literal triggers (see Lemma 3.6), we get the following:

Corollary 4.3 *Termination is undecidable for any trigger language which is at least as powerful as the safe one-literal language and which can define the cross product of two relations.*

4.1 Corollaries for minimal non-safe-cones triggers

For each natural number k, a condition is said to be *k-literal* if it contains at most k relational literals, and an update is said to be *k-literal* if its body contains at most k relational literals. Recall that an update is called *safe* if each of its variables occurs in some positive relational literal in the body.

For example, "$R_1(X,Y), R_2(Y,Z), X \neq Z$" is a safe, two-literal condition; "$R_1(X,Y), X \neq Z$" is an unsafe, one-literal condition; "$-R_4(X) \leftarrow R_3(X,A), R_4(A)$" is a safe, two-literal (deletion) update; and "$+R_8(U,X) \leftarrow R_9(X,Z), Z \neq X$" is an unsafe, one-literal (insertion) update.

We now introduce three trigger languages, which differ in the number of literals in the updates and the safety of the updates. While the safe one-literal trigger language is the simplest sublanguage of safe-cones triggers, these three languages are minimal "violations" of safe-cones triggers.

Definition 4.4 (a) The safe two-literal language consists of triggers whose conditions and updates are safe and 2-literal.

(b) The safe-insert unsafe-delete one-literal language consists of triggers, where the condition is safe and 1-literal, the update is 1-literal, and the insertion is safe. (There is no safety restriction on the deletion.)

(c) The unsafe-insert safe-delete one-literal language consists of triggers where the condition is safe and 1-literal, the update is 1-literal, and the deletion is safe. (There is no safety restriction on the insertion.) ■

Example 4.5 We now give several example triggers: (i) is a safe-insert unsafe-delete one-literal trigger, (ii) is a unsafe-insert safe-delete one-literal trigger, and (iii) is a safe two-literal trigger.

On Insert(R)
If M(X,Y) Then
$+Q(A,B) \leftarrow G(B,A,X)$;
$-G(X,U,X) \leftarrow T(Y,X)$
(i): A safe-insert unsafe-delete one-literal trigger

On Insert(R)
If M(X,Y) Then
$+Q(A,B) \leftarrow G(B,X)$;
$-G(X,X) \leftarrow T(Y,X)$
(ii): A unsafe-insert safe-delete one-literal trigger

On Insert(R)
If M(X,Y) Then
$+Q(A,B) \leftarrow G(B,X), T(X,A)$;
$-G(X,X) \leftarrow T(Y,X)$
(iii): A safe two-literal trigger

■

By showing their ability in defining semijoins through their updates, we get the following:

Theorem 4.6 *Termination is undecidable for the following trigger languages:*

a. Safe two-literal.

b. Safe-insert unsafe-delete one-literal.

c. Unsafe-insert safe-delete one-literal.

Proof: By Theorem 4.2, it suffices to show that each of these trigger languages can do the semijoin operation $R(\overline{X}) \ltimes S(\overline{Y})$. For (a), this semijoin can be done by doing (a.1) the cross product of R and S, (a.2) an equi-join on those columns of R and S corresponding to variables occurring in both $R(\overline{X})$ and in $S(\overline{Y})$, and (a.3) a projection. For (b), let tmp and $result$ be workspace relations having the same arity as R. Then this semijoin can be done using the following after erasing tmp and $result$:

$+tmp(\overline{X}) \leftarrow R(\overline{X});$
$+result(\overline{X}) \leftarrow R(\overline{X});$
$-tmp(\overline{X}) \leftarrow S(\overline{Y});$
$-result(\overline{X}) \leftarrow tmp(\overline{X})$

Observe that $result$ contains the tuples in $R(\overline{X}) \ltimes S(\overline{Y})$ at the end of the computation. For (c), observe that we can simulate updates of type (b) using updates of type (c). For example,

$-P(X, A) \leftarrow Q(X, Y, Z)$

is equivalent to

$+tmp(X, A) \leftarrow Q(X, Y, Z);$
$-P(X, A) \leftarrow tmp(X, A)$

■

4.2 Proof of Theorem 4.2

The basic idea of the proof is to establish a connection between our termination problem and the undecidable halting problem of two counter machines (2CM's). Given any description of a 2CM and its computation starting from zero counters, we show how to a) encode this description in database relations and b) define rules to check this description. We write our rules in such a way that they are locally non-terminating if and only if the 2CM halts. Note that since the state of the database is arbitrary at the time the first rule is triggered, the rules first have to check whether the relations contain a consistent description of the 2CM. This accounts for most of the detail.

The simulation is similar to one in [29, 19], but with some important differences regarding the interpretation of relations. Recall that a 2CM is a deterministic finite state machine with two non-negative counters. The machine can test whether a particular counter is zero or non-zero.

The transition function has the form

$$\delta : S \times \{=, >\} \times \{=, >\} \rightarrow S \times \{minus, plus\} \times \{minus, plus\}$$

For example, the statement $\delta(4, =, >) = (2, plus, minus)$ means that if the machine is in state 4 with counter 1 equal to 0 and counter 2 greater than 0, then go to state 2 and add one to counter 1 and subtract one from counter 2. It is known that the halting problem for 2CMs is undecidable for the situation where the counters are set to zero in the initial state [25].

The computation of the machine is stored in the relation $config(T, S, C_1, C_2)$, where T is the time, S is the state and C_1 and C_2 are values of the counters. The states of the machine can be described by "simulated" integers $0, 1 \ldots, h$, where 0 is the initial state and h the halting (accepting) state. The first configuration of the machine is $config(0, 0, 0, 0)$ and thereafter, for each move, the time is increased by one and the state and counter values changed according to the transition function.

The relation $succ(X, Y)$ is used to represent the successor relation and $R_0(X)$ contains a representation for the constant 0. The constants in $succ$ are used for representing the 2CM states, times and counter values. Since there is no guarantee that these relations represent what we wish them to, we need to devise a method of checking their correctness. A major limitation of using the semijoinable trigger languages for such checking, is that they cannot express inequality between constants in different tuples. Consequently, we cannot do simple things like testing whether a relation contains more than one constant. Since the simulation in [29, 19] depends on this feature for checking the goodness of the $succ$ relation (amongst other things), we need to devise testable conditions on $succ$ and R_0 that are less stringent.

$R_0(X)$ is interpreted as $X = 0$; there may, however, be several X's for which this is true. $succ(X, Y)$ is interpreted as $Y = X + 1$. For a given X, there may be several Y's for which this is true. So we need to think of $succ$ as representing a kind of *partial* order on constants, instead of the usual *total* order. Suppose we are trying to use the constants in $succ$ to represent the constants $0, 1, \ldots, k$. Let f be the function mapping each of these numbers to the set of all possible representations it may have in $succ$. That is, $f(0)$ is the set of all constants having no predecessor; and, inductively, $f(i + 1)$ is the set of all constants having some predecessor in $f(i)$. We need to ensure that the $succ$ relation is acyclic, or equivalently, $\forall i, j \in [0..k]$ $(i \neq j \Leftrightarrow f(i) \cap f(j) = \emptyset)$.

We need some more relations in our simulation; all of them have arity 1 unless otherwise specified:

- $R_0, R_1, \ldots, R_h$: R_i contains *all* constants representing the state i (*i.e.* all values of $f(i)$).
- $last\_time$: contains *all* constants representing the last time cycle examined.
- $last\_state$: contains *all* constants representing the state which occurred at $last\_time$.
- $last\_C_1$,$last\_C_2$: $last\_C_i$ contains *all* constants representing the *value* of the i^{th} counter at $last\_time$.
- $current\_time$: contains *all* constants representing the successor of $last\_time$.
- $reach$: contains *all* constants representing times which are reachable from the initial ones.
- $nonzero$: contains all constants in $succ$ which are not in R_0.
- bad : has arity 0 and is used to indicate whether the database has an invalid computation. bad will be made true if we find an invalid computation (i.e the database doesn't contain a model we desire), otherwise it will stay false.

We also use some other relations not listed here; these will be explained when they are needed. We now outline an algorithm for checking the correctness of the various relations. It can be expressed using a set of triggers of the semijoinable trigger language (see Lemma 3.6 and the linkage technique of Theorem 3.7).

1 Initialise Relations;
2 Construct $R_0, \ldots, R_h$;
3 Check that there are no cycles in $succ$;
4 Check goodness of $config$ at time zero;
5 if $bad \neq true$
$reach = R_0$;
6 $last\_time(X) = R_0(X)$;
7 $current\_time = \Pi_x(succ(Y, X) \ltimes R_0(Y))$;
8 while ($current\_time \neq \emptyset \wedge bad \neq true$)
8.1 for each tuple t in $config$ such that the time is $current\_time$
for each transition δ
Check that if δ is applicable then the transition to t is correct
end for
end for
8.2 if all tuples correct then
$reach = reach \cup current\_time$
else $\{bad = true; reach = \emptyset\}$;
8.3 $last\_time = current\_time$
8.4 $current\_time = \Pi_X(succ(Y, X) \ltimes current\_time(Y))$
end while
9 if ($bad \neq true$ and there is a time in $reach$ for which $config$ is in the halting state)
loop infinitely;
else End;

We will describe the logic needed for each of the components 1-9 of the algorithm. Each component M is implementable by either a single rule or a set of rules $t_1^M, \ldots, t_f^M$ where t_1^M is the first rule that executes in component M and t_f^M is the last rule that executes in component M. Sequencing between components M_i and M_{i+1} is achieved by defining the event expression of t_1^{M+1} and the action of t_f^M such that the event expression is made true by the action.

1. Initialising Relations: For many of the relations we are using, it is necessary for them to be empty initially. This can be achieved by the appropriate $erase$ statement, for example $-R(X) \leftarrow R(X)$ erases everything in R. Relations to be emptied include $R_0, \ldots, R_h$, $sofar$, $nonzero, reach$ and bad should be made false. We also have a bounded number of relations, which respectively will hold some subsets of $config$ and $succ$, called $succ_1$,$succ_2$, ...,$config_1$,$config_2$, which also need to be made empty initially. bad is also made false initially.

Component 1 thus consists of a single rule which executes a sequence of erase updates and then raises an event which triggers the first rule of component 2.

2. Constructing $R_0, \ldots, R_h$: We wish to put each group of constants corresponding to one of the states $[0, h]$ in its own relation. We can construct the R_0 relation using two projections:

$+R_0(X) \leftarrow succ(X, Y)$ /* insert all candidate constants */
$-R_0(X) \leftarrow succ(Y, X)$ /* remove the ones which have a predecessor */

We now construct the R_1 relation to contain the successors of '0'.

$+R_1(Y) \leftarrow succ(X, Y) \ltimes R_0(X)$

R_1 now contains all the successors of '0' - *i.e.* '1'. We can similarly construct $R_2, R_3 \ldots, R_h$ using the further updates. If any of $R_0, \ldots, R_h$ is empty, then we make $bad = true$. This is testable by executing some further updates that use auxiliary $test$ relations:

$+test_0 \leftarrow true$
$-test_0 \leftarrow R_0(X_0)$
$+bad \leftarrow test_0$
$+test_1 \leftarrow true$
$-test_1 \leftarrow R_1(X_0)$
$+bad \leftarrow test_1$
$\ldots$
$+test_h \leftarrow true$
$-test_h \leftarrow R_h(X_0)$
$+bad \leftarrow test_h$

Component 2 thus consists of a single rule which executes the sequence of updates described and then raises an event which triggers the first rule of component 3.

3. Cycle Check: We check that the $succ$ relation contains no cycles. This can be done using a while loop. The relation $sofar$ is used to record constants already examined.

3.1 $+sofar(X) \leftarrow R_0(X)$
3.2 $+current(Y) \leftarrow succ(X,Y) \ltimes R_0(X)$
3.3 while ($current \neq \emptyset$)
3.4 if $current \cap sofar \neq \emptyset$
$bad = true; erase(current);$
else
$sofar = sofar \cup current;$
$tmp = current; erase(current);$
$+current(Y) \leftarrow succ(X,Y) \ltimes tmp(X);$
end while

3.1 and 3.2 are straightforward updates. 3.4 can be implemented by the following update sequence:

$-test \leftarrow test$
$+test \leftarrow current(X) \ltimes sofar(X)$
$+bad \leftarrow test$
$-current(X) \leftarrow current(X) \ltimes test$
$+test_2 \leftarrow true$
$-test_2 = test$
$sofar = sofar \cup (current \ltimes test_2);$
$-tmp(X) \leftarrow tmp(X) \ltimes test_2$
$+tmp(X) \leftarrow current(X) \ltimes test_2$
$-current(X) \leftarrow current(X) \ltimes test_2;$
$+current(Y) \leftarrow (succ(X,Y) \ltimes tmp(X)) \ltimes test_2;$

The while loop in 3 thus has the form

$< b_1 >$
while(c) do
$< b_2 >$
End While
$< b_3 >$

where b_1, b_2, b_3 are sequences of updates. It has already been shown how to simulate such a structure in section 2, using a set of four triggers. Component 3 thus consists of such a set of four rules. The last rule to execute from this set should raise an event that triggers the first rule of trigger component 4.

4. Check goodness of $config$ at time zero: We now check that the configuration of the machine at time zero is equivalent to $config(0, 0, 0, 0)$. We first populate $config_1$ to contain only the tuples from $config$ with '0' as a first argument.

$+config_1(T, S, C_1, C_2) \leftarrow config(T, S, C_1, C_2) \ltimes R_0(T)$

The relation $nonzero$ contains all the constants from $succ$ which are not in R_0 (easily expressible). If any of the following deletions succeed (in changing the state of $config_1$), we will make *bad=true*.

$-config_1(T, S, C_1, C_2) \leftarrow config_1(T, S, C_1, C_2) \ltimes nonzero(S)$
$-config_1(T, S, C_1, C_2) \leftarrow nonzero(C_1)$
$-config_1(T, S, C_1, C_2) \leftarrow nonzero(C_2)$

If none of these deletions succeeds, then the configuration at time zero is correct.

Component 4 consists of two rules. The first rule in component 4 performs the updates discussed above (call this sequence α) and then raises an event which triggers the first rule of component 5. The second rule in component 4 performs the task of setting bad to $true$ if a deletion on $config_1$ has succeeded:

First Rule	Second Rule
on event	on delete($config_1$)
if true	if true
then	then
α;	$+bad \leftarrow true$;
trigger(component_5)	trigger(component_5)

where the priority of this second rule is greater than the priority of the first rule in component 5 (and thus it will be the one chosen for execution if both get triggered simultaneously).

5-7. These components are just simple updates within a single rule, with the last update raising an an event which triggers the first rule of component 8.

8. Transition Checking: Transitions are checked by examining successive configurations of the machine in $config$. We need to check that the transition which occurred between the $last\_time$ and the $current\_time$ (equal to $last\_time + 1$) is correct. If this the case for every ($last\_time$, $current\_time$) pair, then it follows inductively that all transitions are correct. We will calculate the state which occurred for $last\_time$ and put it in the relation $last\_state$.

$+candidate\_state(S) \leftarrow config(T, S, C_1, C_2) \ltimes last\_time(T)$

$candidate\_state$ contains some of the constants which represent the number identifying the state at $last\_time$. It may not be complete, however, since $config$ may only use some of them. To obtain the others, we try comparing it with $R_0, \ldots, R_h$ until we get a non empty intersection.

for i=1 to h
    if $R_i \cap candidate\_state \neq \emptyset$
        $last\_state = R_i$; $exit$;
end for

This for loop is expressible by the updates

$-flag1 \leftarrow flag1$
$+flag1 \leftarrow R_1(X), candidate\_state(X)$
$-last\_state(X) \leftarrow last\_state(X), flag1$
$+last\_state(X) \leftarrow R_1(X), flag1$
$-flag2 \leftarrow flag2$
$+flag2 \leftarrow R_2(X), candidate\_state(X)$
$-flag2 \leftarrow flag1$
$-last\_state(X) \leftarrow last\_state(X), flag2$
$+last\_state(X) \leftarrow R_2(X), flag2$
$\vdots$

where the $flag$ variables are used for doing the if test and 'exiting' the for loop appropriately.

Similar updates are needed for constructing $last\_C_1$ and $last\_C_2$. We again construct a candidate relation (say $candidate\_C_1$) and then enlarge this by finding the appropriate 'stratum' of constants from $succ$. This can be done using a while loop similar to that used in the cycle checking section. e.g. For the case of $last\_C_1$ we do:

$current(X) = R_0(X)$
$erase(last\_C_1)$
$erase(candidate\_C_1)$
$+candidate\_C_1(C_1) \leftarrow config(T, S, C_1, C_2) \ltimes last\_time(T)$
$+last\_C_1(X) \leftarrow current(X) \ltimes candidate\_C_1(X)$
while ($last\_C_1 = \emptyset$)
    $tmp = current; erase(current);$
    $+current(Y) \leftarrow succ(X, Y) \ltimes tmp(X)$
    $+last\_C_1(X) \leftarrow current(X) \ltimes candidate\_C_1(X)$
end while

The while loop here can be implemented in a similar way to the one in component 3.4.

We now need to find if a transition is applicable to the $(last\_state, last\_C_1, last\_C_2)$ "tuple". Suppose the transition is $\delta(j, >, =) = (j', minus, plus)$. The following correspondences hold

1) $last\_state = j \Leftrightarrow last\_state \cap R_j \neq \emptyset$
2) $C_1 > 0 \Leftrightarrow last\_C_1 \cap R_0 = \emptyset$
3) $C_2 = 0 \Leftrightarrow last\_C_2 \cap R_0 \neq \emptyset$.

Using these equivalences, we can check whether the condition of the above transition is satisfied with the statement

$erase(satisfied); erase(sat1); +sat2 \leftarrow true; erase(sat3);$
$+sat1 \leftarrow last\_state(X) \ltimes R_j(X)$
$-sat2 \leftarrow last\_C_1(X) \ltimes R_0(X)$
$+sat3 \leftarrow last\_C_2(X) \ltimes R_0(X)$
$+satisfied \leftarrow (sat1 \ltimes sat2) \ltimes sat3$

If the transition is not applicable, then we ignore it and check the next one; but if it is, then we need to determine whether $config$ correctly represents the tuple $(j', C_1 - 1, C_2 + 1)$ at the current time value. We compute $current\_state$ which is the state(s) which occur(s) for the current time value.

$+current\_state(S) \leftarrow config_5(T, S, C_1, C_2) \ltimes current\_time(T)$

If $current\_state - R_{j'} \neq \emptyset$, then make $bad = true$, since this would mean there is a wrong state occurring at $current\_time$. Assuming the state is correct, we then have to check the new counter values are correct. This can be done in a similar way. Thus the full logic for checking if the above transition was done correctly would be:

if (transition_condition_satisfied) {
$+current\_state(S) \leftarrow config_5(T, S, C_1, C_2) \ltimes current\_time(T)$
$-current\_state(S) \leftarrow R_{j'}(X)$
$+bad \leftarrow current\_state(S)$
$erase(tmp1); erase(tmp2)$
$tmp1(X) \leftarrow succ(X, Y) \ltimes last\_C_1(Y)$
$tmp2(Y) \leftarrow succ(X, Y) \ltimes last\_C_2(X)$
$+current\_C_1(C_1) \leftarrow config_5(T, S, C_1, C_2) \ltimes current\_time(T)$
$+current\_C_2(C_2) \leftarrow config_5(T, S, C_1, C_2) \ltimes current\_time(T)$
$-current\_C_1(C_1) \leftarrow tmp1(C_1)$
$-current\_C_2(C_2) \leftarrow tmp2(C_2)$
$bad \leftarrow current\_C_1(X)$
$bad \leftarrow current\_C_2(X)$ }

The logic needed for 8.2-8.4 is straightforward. Component 8 then consists of triggers to implement the outer while loop (in the same way as while loops discussed earlier), with the body of the loop consisting of the updates discussed for 8.1-8.4. Upon finishing the while loop, an event is raised which triggers the first rule in component 9.

9. Halt Check: After exiting the main $while$ loop of the 2CM simulation, we need to check whether there is a time in $reach$ for which $config$ is in the halting state. To do this, we first erase $config_7$. Then do the update

$$+config_7(T, S, C_1, C_2) \leftarrow (config_6(T, S, C_1, C_2) \ltimes reach(T)) \ltimes R_h(S)$$

If $config_7$ is not empty, then the halting state is reachable from the initial state. If $config_7$ is non empty and $bad \neq$ true then we trigger a rule which loops infinitely e.g.

```
on e
if true
trigger(e)
```

It is now clear that the 2CM halts iff the triggers we constructed do not terminate when when the first rule in component 1 is initially triggered. ■

5 Meta Features of Rule Execution

5.1 Overview

We have so far concentrated on varying the language features of an active rule system using a simple execution semantics; in essence, under that semantics, an active rule system is executed like that of a *while*-like language [31]. Aspects of active databases which distinguish them from typical query languages, however, are their meta level features (i.e. a sophisticated execution model). These include managing the pending structure of rules, flexible methods for detecting/triggering events and controlling the timing of rule action execution (coupling modes). Henceforth, we will regard an active rule system as specified using two languages, the rule language $\mathcal{L}$ for specifying the syntax of events, conditions and actions and the meta language $\mathcal{M}$ for specifying higher order features.

For the analysis in sections 3 and 4, we fixed $\mathcal{M}$ as the semantics defined in section 2 and then varied $\mathcal{L}$. Conversely, when analysing meta features, we will fix $\mathcal{L}$ and vary $\mathcal{M}$.

For making this fixed choice of $\mathcal{L}$, we begin by defining a class of decidable rule languages called *bounded model languages*. These are languages where system behaviour on arbitrary instances can be simulated by representative instances using a bounded number of constants.

Definition 5.1 A rule language $\mathcal{L}$ is called a *bounded model language* with respect to a meta language $\mathcal{M}$ if, for every rule program P written in $\mathcal{L}$ and $\mathcal{M}$, there is an effectively computable $k < \infty$ (depending only on the rules) satisfying: For every instance I, there is another instance I' using $\leq k$ constants such that P terminates on I iff P terminates on I'. ■

In other words, the termination behaviour of a set of rules written in a bounded model rule language is completely determined by a specific (finite) set of database instances. The safe-cones language is an important example of a bounded model language, for all the meta languages we consider in this paper. Other examples are given in section 7.

Another bounded model language that we will need is a simple language we call the 0-1 rule language. This language is a trigger language using only 0-ary relations. To simplify the discussion, we will use binary valued variables to denote 0-ary relations.

Definition 5.2 The *0-1 rule language* consists of triggers such that

- events are of the form $U(A)$ which we understand to mean "the variable A has had its value changed";
- conditions are conjunctions of simple conditions, where a simple condition is a test of the form $A = 0$ or $A = 1$;
- an action is a sequence of simple actions, where a simple action is an update of the form $X = x \leftarrow c$ where $x \in \{0, 1\}$ and c is a condition.

■

Remark: Observe that the 0-1 conditions and actions can be expressed using the rule formalism we have already defined. The events are equivalent to statements of the form 'on insert(A) or on delete(A)", which strictly speaking is a generalisation of the previous event formalism. However, this notation has only been used for readability and it is possible to rewrite any set of 0-1 rules into another set which instead just uses events of the form "on insert(A)" or "on delete(A)".

An example of a 0-1 rule is:

On U(A)
If $C = 0 \ \wedge D = 1 \ \wedge \ T = 1$ then
$T = 0 \leftarrow E = 1 \wedge F = 0 \wedge G = 0$;
$E = 1 \leftarrow F = 1 \wedge E = 0$

Note that events of the form $U(A)$ will always be triggered by an action of the form $A = 0; A = 1$. Despite its simplicity, the 0-1 language is essentially equivalent to every bounded model language, for the purposes of analysing termination:

Theorem 5.3 *Let $\mathcal{S}_1$ be a rule system using a bounded model rule language and a meta language $\mathcal{M}$, and let $\mathcal{S}_2$ be a rule system using the 0-1 rule language and a meta language $\mathcal{M}$. Termination is decidable for $\mathcal{S}_1$ iff termination is decidable for $\mathcal{S}_2$.*

Proof: $\Leftarrow$: Suppose termination is decidable for $\mathcal{S}_2$. We show how to translate the rules of $\mathcal{S}_1$ into 0-1 rules and the database instance for $\mathcal{S}_1$ into a 0-1 database instance. Let k be the maximum number of constants needed to characterise the bounded model language (as per definition 5.1). Since k is finite, there are a bounded number of possible database states for $\mathcal{S}_1$ and a bounded number of tuples that can ever be constructed. For each of these tuples, we use a 0-1 variable to record its presence or absence in a particular relation. The actions of rules in $\mathcal{S}_1$ cause transitions between states via insertions/deletions, and this can be captured by a sequence of 0-1 updates which check the current state and then change the values of the variables accordingly, to reflect the semantics of the update (like the transitions of a state machine). The event and condition parts are handled similarly to the actions.

$\Rightarrow$: Since the 0-1 language is a bounded model language, the result follows trivially. ■

Given the above result, when considering meta features, we will henceforth use the 0-1 language and the class of bounded model rule languages interchangeably. This is because any (un)decidability result which holds for one, will also hold for the other. Also, the basic nature of the 0-1 language means that any undecidability results for it also carry across to rule languages (not just bounded model ones) of greater expressiveness.

5.2 Decidability Results for Meta Features

We begin by considering the pending structure of the rule system. This is a repository for rules awaiting execution. A triggered rule is *put* onto the structure. Rules are removed by performing a *select* operation on the structure. Choices we will consider are

- *Singleton* - This has hitherto been our default choice for execution. The structure can contain at most one rule. When several rules are simultaneously triggered, only the one of the highest priority is put onto the structure.
- *Set* - We retain at most one instance of any rule. When a rule needs to be selected, the one with the highest priority is chosen. The active database prototype Starburst [37] uses this approach.
- *Stack* - This may contain multiple instances of rules. Newly triggered rules are placed on top of the stack in order of high-to-low priority. Rule selection is done by removing the rule on top of the stack. The active database prototype NAOS [16] uses this approach and the current SQL3 rule semantics [28] can be simulated using a stack structure.
- *Queue* - This may contain multiple instances of rules. Newly triggered rules are placed at the tail of the queue in order of priority. Rule selection is done by removing the rule at the head of the queue. The active database prototype HiPAC [18] uses this approach.

The singleton and set structure are similar, since there is an upper limit to the number of rules that may be contained in the structure. We call these structures *bounded rule structures* (containing $\leq f(n)$ rule instances, where n is the number of rules and f is some function). This property of boundedness yields decidability when used in conjunction with bounded model languages.

Theorem 5.4 *Termination is decidable for every trigger system with a bounded model rule language and a bounded pending rule structure.*

Proof: Since the language has a bounded model property, we can simulate it using a bounded number of constants and look for repeating system states. Since the pending structure is bounded, the number of possible states for it is finite also. The 'if' in proposition 2.5 thus becomes an "iff". ■

The other two structures listed above, the stack and queue, are not bounded however. It is therefore not possible in general to prove decidability of termination using finiteness arguments. Interestingly, for the case of the stack, since rules are added to the structure in a restricted manner, it is possible to prove decidability.

Theorem 5.5 *Termination is decidable for a trigger system using a bounded model rule language and a stack pending structure.*

Proof: We begin with a few definitions to aid in describing execution of rules using stacks and the associated termination analysis. Each stack is a list of rules, with the head of the list corresponding to the top of the stack; we will treat list and stack as synonyms in this proof. Each rule occurrence r in the list is treated as having two attributes: i) the name of the rule and ii) a timestamp of the rule, which records the iteration number of when this rule was placed on the stack (we assume that rule execution begins at iteration zero and the iteration number thereafter is incremented after the completion of a rule's action executing). Two rules are *name equivalent* if they have the same name and *strongly equivalent* if they have the same name and timestamp. Two lists of the same length are name (strongly) equivalent if the corresponding elements of the two lists are name (strongly) equivalent. The operator $=$ denotes strong equivalence and $\cong$ name equivalence. We use $w_1, w_2, \ldots, q$ to denote lists of rules and $w_i.w_j$ denotes the composition of the lists w_i and w_j.

For two stacks s_1 and s_2, we say that $s_1 \subseteq s_2$ if $s_1 = w_1.q$ and $s_2 = w_2.w_3.q$, where (i) $w_2 \cong w_1$, (ii) q is a list representing the longest strongly equivalent suffix shared by s_1 and s_2, and (iii) w_1 is a list representing the longest name equivalent prefix shared by s_1 and s_2.

We now show that the rule execution does not terminate if (*) during execution there occur two distinct system states $S_1 = (db_1, s_1)$ and $S_2 = (db_2, s_2)$, where $db_1 = db_2$ and $s_1 \subseteq s_2$. Indeed, assume that (*) is true and S_2 occurs after S_1. Executing the same rule in db_1 or db_2 has equivalent effect since $db_1 = db_2$. Rewriting s_1 and s_2 according to the definition of containment, w_2 executes and eventually yields the stack $w_5.w_4.w_3.q$ in db_1, where $w_5 \cong w_1$ ($w_3.q$ remains unaffected in the same way as q was unaffected moving from S_1 to S_2). w_5 then executes and we eventually get the stack $w_7.w_6.w_4.w_3.q$ in db_1, where $w_7 \cong w_1$. This process will repeat infinitely, yielding non termination.

We next show that, after the rules are executed some bounded number (determined below) of iterations, there are guaranteed to be two distinct system states $S_1 = (db_1, s_1)$ and $S_2 = (db_2, s_2)$, where $db_1 = db_2$ and $s_1 \subseteq s_2$.

Firstly, let N be the number of rules and let n be the total number of database states (since the rule language is bounded model, n can be determined from the bounded number of constants in definition 5.1). Let T_l denote the total number of distinct system states having stacks of length between 1 and l. Then $T_l = \Sigma_{i=1}^{l} n \times N^i$, since there are at most $n \times N^i$ distinct system states having stacks of length i.

Suppose we begin execution using one of the n states as the initial state and one of the N rules as the initial triggering rule. If execution proceeds for $T_l + 1$ iterations, then either a) the final size of the stack at iteration $T_l + 1$ is $> l$, or b) there is an intermediate stack of size $> l$, or c) there is a repetition of system states. To see this, an equivalent (and more obvious) statement is: If the final size of the stack is $\leq l$ and all intermediate stacks are $\leq l$ and there is no repetition of system states, then execution lasted for $\leq T_l$ iterations.

Now if c) is true, then we are done, since a repeating state has occurred. Otherwise, there must have existed a stack of length $> l$ sometime during the execution. Furthermore, there must exist a subsequence of system states that lead to the system state with stack length $> l$. Call these states $S_1, S_2, \ldots, S_p$ where $\lceil l/N \rceil \leq p$, the stack has size $> l$ in S_p, and for all $1 \leq i < j \leq p$, i) S_i occurred sometime earlier than S_j and ii) the size of the stack in S_i is less than the size in S_j and the stack does not shrink below the size it had in S_i, between the occurrence of S_i and S_j. This follows from the fact that the initial size of the stack is 1 and at most N rules can be placed on the stack per iteration. Therefore the least number of iterations it could take to grow to a size $> l$ (i.e. grow by at least an extra l rules) is $\lceil l/N \rceil$. Observe that the subsequence $S_1, S_2, \ldots, S_p$ can be found by starting with the complete sequence of system states leading from S_1 to S_p and then deleting system states where there exists a later element in the sequence having a smaller stack.

Now, if we choose $l = n \times N \times (N+1) + 1$, then $p > n \times (N+1)$. In such a case, it then follows that there must exist a database state db in the sequence $S_1, \ldots, S_p$ which is repeated $N+1$ times. Since there are only N possible heads the stacks can have, there exist two system states S_a and S_b in this sequence where db occurs and the stacks have the same rule e as the head: $S_a = (db, e.f)$ (where $e.f$ represents a stack with singleton rule e at the head and f is the list of rules comprising the tail) and $S_b = (db, e.g)$ for some f and g. Now, as a consequence of condition ii) above, f must be a suffix of g. It must therefore be the case that $e.f \subseteq e.g$. Relating this back to our decision procedure, we know that the rule set is non terminating on the initial database state iff rule execution can proceed as far as $T_l + 1$ iterations. We therefore just need to execute the rules on all n initial database states and see if any of these executions lasts for $T_l + 1$ iterations. If so, then the rule set is locally non terminating, otherwise it is terminating.

The complexity of this termination analysis is indicated by the number of iterations needed, which is

$1 + T_l \leq 1 + \Sigma_{i=1}^{l} n \times N^i$ (as discussed several paragraphs ago)
$= 1 + \Sigma_{i=1}^{1+n \times N \times (N+1)} n \times N^i$ (because we chose $l = 1 + n \times N \times (N+1)$)
$= 1 + n \times \Sigma_{i=1}^{1+n \times N \times (N+1)} N^i$
$\leq 1 + n \times (1 + n \times N \times (N+1)) \times N^{1+n \times N \times (N+1)}$
$= O(n^N \times N^{n \times N^2})$.

Observe that n varies for different rule languages; it is 2^m for the 0-1-language, where m is the number of binary variables used by the program under consideration. ■

Theorems 5.4 and 5.5 are about the most powerful decidable configurations considered in this paper. Our focus now turns to meta features which cause termination to become undecidable.

5.3 Undecidability Results for Meta Features

We begin by examining the case of the queue pending structure. The difference from the stack is that, because rules are added to the tail rather than the head, there is no criterion for detecting "similar" queues. In fact, as we now show, the property of termination is undecidable.

Theorem 5.6 *Termination is undecidable for the 0-1 rule language using a queue pending structure.*

Proof: We will show how to build a set of 0-1 triggers with queue to simulate any Post machine [34, 15], a device which is as computationally powerful as the Turing machine. The Post machine is like a pushdown automaton which uses a queue instead of a stack. It consists of an alphabet of input symbols and a number of states including a START and one or more *accepting* states. In each state one then moves to another state after reading the front of the queue and removing a symbol (if one exists) and then optionally adding an element to the tail of the queue. The machine does not have a separate input tape unit, but rather the input string is initially loaded into the queue before execution. The machine halts when it enters an accepting state or encounters a state, symbol pair for which no transition is defined. A string is accepted if the machine halts in an accepting state. Termination is undecidable for Post machines on the empty string. We can therefore use an empty input in our simulation. We will use the pending structure of the active database to simulate the Post machine queue and we will show how to define various rules which replicate the machine's transitions.

A Post machine's transitions have the form (p, a, q, b), which says that, if p is the machine's current state and a is the symbol at the head of the queue, then the machine will go to the new state q and it will append to the tail of the queue the symbol b. The symbol b may equal ϵ which indicates nothing is to be added to the tail of the queue. The symbol a may equal ϵ which indicates the queue is currently empty.

To translate this machine into 0-1 rules, we define the following variables.

- A special variable V_{accept} to indicate an accepting state.
- A special variable V_s to indicate the START state.
- A special variable V_ϵ, to allow us to recognise the empty word.
- A special variable V_{flag} to help with mutual exclusion.
- For each machine symbol a, the variable V_a.
- For each machine state p, the variable V_p.

We group transitions together according to symbol. Suppose the group for symbol a is the following:

(p, a, p_1, w_p)
(q, a, q_1, w_q)

These can be translated into the following package of rules.

Rule r_a	Rule r_{a_p}	Rule r_{a_q}
On U(V_a)	On U(V_a)	On U(V_a)
If true then	If V_p=1 and $V_{flag} = 1$ then	If V_q=1 and V_{flag}=1 then
V_{flag}=1	V_p=0 ; V_{p_1}=1 ;	V_q=0 ; V_{q_1}=1 ;
	$V_{w_p} = 1; V_{w_p} = 0$; V_{flag}=0	$V_{w_q} = 1; V_{w_q} = 0$; V_{flag}=0

The variable V_{flag} ensures that only one of r_{a_p} and r_{a_q} is executed. Rule r_a sets V_{flag} so that other rules may use it. These rules are ordered so that priority(r_a) $>$ priority(r_{a_p}) $>$ priority(r_{a_q}). If p is an accepting state, then we also include the action $V_{accept} = 1$ in rule r_{a_p}, similarly for state q and rule r_{a_q}. Statements such as $V_{w_p} = 1; V_{w_p} = 0$; are there to trigger the rule r_{w_p} (since this is guaranteed to produce a change in the variable V_{w_p}); Note that we can always add some extra transitions to the Post machine to ensure that w_p is a single letter and not a sequence of letters (these extra transitions would add one letter at a time).

We also need a rule to empty the queue if an accepting state is entered. Continuing with the above example, suppose p is an accepting state. Then we have the following rule, whose priority is less than that of r_a and larger than that of r_{a_p}, to ensure that none of the rules on the queue can trigger another rule:

Rule $r_{a_{accept}}$
On U(V_a)
If V_{accept}=1 then
V_{flag}=0

We have thus shown how the state transitions of the Post machine can be replicated by 0-1 rules. To complete the picture, we need to explain how the machine is initialised. The first action to happen needs to have the form $V_{accept} = 0; V_{p_1} = 0; ...; V_{p_m} = 0; V_s = 1; V_\epsilon = 1; V_\epsilon = 0$;, where $p_1, ..., p_m$ is an enumeration of all the states of the Post machine. This ensures that we begin in the starting state with the empty word ϵ on the queue, and all variables are appropriately initialised. Observe that the execution of a 0-1 rule system halts once the queue is empty. ■

Observe that although termination is undecidable for this configuration, termination in N steps is in fact decidable. We next show that the above theorem can be used to derive undecidability results for other types of meta features - complex events and coupling modes.

5.3.1 Complex Events

Many active rule languages have a facility for specifying complex events (e.g. [22, 21, 12]). These are combinations of various primitive events. One needs to be careful, however, about specifying their semantics, since even seemingly simple operators may have a variety of interpretations [14].

The operator we will consider is the *cumulative event sequence operator*. An event E=e1;e2 is raised if e2 is raised and the event $e1$ was raised sometime earlier. The consumption semantics further specifies what occurrences of $e1$ need be considered when determining if E should be raised. Cumulative semantics means intuitively that we 'match an e2 with each unmatched e1 before it' (in applications this could correspond to pairing all preceding deposits to a big withdrawal). Figure 1 A) illustrates this with an example event history (where time flows left to right) having six different occurrences of the event $e1; e2$. The numeric labels on the arcs indicate the complex event ordering. This ordering is derived by considering the time of occurrence of the complex event's first (i.e. $e1$) component. So in the figure, 1 occurs before 2, 2 occurs before 3 etc. Part B) shows a situation where three complex events have been defined: $e4; e7$ and $e5; e7$ and $e6; e7$. Once again, the labels on the arcs indicate the order in which the events occur. In both cases, when events are triggered simultaneously, they are pushed onto the stack in order of most recent firing (i.e push 3, then push 2, then 1).

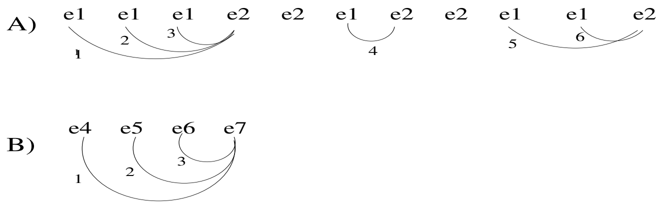

Figure 1: Cumulative Consumption Semantics

Suppose we assume that the rule system has the power to recognise a complex event of this type. The following theorem tells us that it makes termination undecidable. This is because the system has become as powerful as when we had a queue earlier.

Theorem 5.7 *Termination is undecidable for a 0-1 trigger system using a stack and the cumulative event sequence operator.*

Proof (sketch): We show how it is possible to use the complex event capability to make the stack behave like a queue. It then follows from theorem 5.6 that termination is undecidable.

Given a Post machine, we first define a set of active rules as was done in theorem 5.6. Call these rules $r_1, r_2, \ldots, r_n$. For each such r_i, if its event part was "On e_i", we now modify it to "On $e_i; e_{bottom}$". "On $e_i; e_{bottom}$' is a complex event using cumulative consumption semantics. It therefore will be raised when e_{bottom} is raised, provided the e_{bottom} can be paired with an 'unmatched' e_i. The condition and action of the rule are left unchanged.

Since we wish to simulate a queue, it is necessary to be able to place a newly triggered rule at the bottom, rather than the top of the stack. Before a rule r_i can be placed on the bottom, the stack must first be emptied of all the rules currently on it. This can be achieved by a) using the complex event capability to act as a memory for what these rules were and b) adding some extra logic to the definitions of the rule packages.

From the way rule packages were defined in theorem 5.6, it is possible for at most one event to be triggered by the package (due to the flag variable enforcing a kind of mutual exclusion). This property is used below. We now describe the execution behaviour, demonstrating the extra logic that needs to be contained in the rule packages.

Let the state of the pending structure and database at some point in time be

$[P_1, P_2, \ldots, P_n, M], mstate = s_1$

where each P_i is a rule package for the machine symbol a_j and M corresponds to a special rule package which is always at the bottom of the stack. $mstate$ is a variable used to indicate which state the Post machine is in. Suppose the Post machine transition (s_1, a_i, s_2, a_x) is applicable at this point. Under queue semantics, the effect of P_1 would be to trigger P_x, placing it at the end of the queue and changing the machine state from s_1 to s_2, reaching the configuration of

$$[P_2, P_3, \ldots, P_n, P_x, M], mstate = s_2.$$

We now sketch the sequence of steps needed to reach this machine state. For achieving this, two mutually exclusive modes of operation will be used, *normal mode* and *memory* mode. The behaviour of these modes will be demonstrated in the following trace - assuming without loss of generality that $normal$ mode is initially true and there are no unconsumed events in the event history.

$$[P_1, \ldots, P_n, M], mode = normal, mstate = s_1$$

In $normal$ mode, rather than generating an event e_x to trigger package P_x, the actions of P_1 cause the value of e_x to be saved in a variable named $recent$. The actions also cause $mstate$ to change from s_1 to s_2. $normal$ mode is then made false and $memory$ mode made true.

$$[P_2, \ldots, P_n, M], mode = memory, mstate = s_2, recent = e_x$$

Recall that all rules in a package P_i have events of the form "On $e_i; e_{bottom}$". In $memory$ mode, the only effect of the rules in a package P_i is to trigger the single event e_i. In the trace below, we also include a relevant portion of the event history.

$$[P_3, \ldots, P_n, M], mode = memory, mstate = s_2, recent = e_x, evthistory = [e_2]$$
$$[P_4, \ldots, P_n, M], mode = memory, mstate = s_2, recent = e_x, evthistory = [e_2, e_3]$$
$$[P_5, \ldots, P_n, M], mode = memory, mstate = s_2, recent = e_x, evthistory = [e_2, e_3, e_4]$$
$$\vdots$$
$$[M], mode = memory, mstate = s_2, recent = e_x, evthistory = [e_2, \ldots, e_n]$$

The marker rule package M triggers the event e_x, followed by e_m (a special event), followed by e_{bottom} (the value of $recent$ can be used to indicate the identity of e_x). It also changes the mode back to $normal$.

$$[], mode = normal, recent = e_x, mstate = s_2, evthistory = [e_2, \ldots, e_n, e_x, e_m, e_{bottom}]$$

The current state of the event history is now such that a number of complex events (of the form $e_i; e_{bottom}$) become triggered, and the corresponding rule packages are then placed on the stack. Note that all rules in package M are defined such that their event expression is "on $e_m; e_{bottom}$".

$$[P_2, P_3, \ldots, P_n, P_x, M], mode = normal, mstate = s_2$$

We have now achieved our objective of modifying the stack to place P_x after P_n and we are back in normal mode, with no partially consumed complex events. We have not given precise definitions of the internals of the rule packages, but it should be clear that the logic needed is easily implementable by the 0-1 trigger language. It is thus possible to simulate queue semantics by using the complex event capability and hence termination is undecidable. ■

5.3.2 Coupling Modes

We now turn our attention to the timing of activation of the components in an E-C-A rule. Current active database systems address this by incorporating the notion of coupling modes [18]. These can be thought of as another type of meta feature. Each rule can be triggered using a variety of couplings. In this paper, we consider two important types: *immediate coupling* and *deferred coupling*. We can model this as the existence of two pending structures, the *current* pending structure and the *postponed* pending structure. The former stores all the rules awaiting execution currently. The latter stores rules that are to be executed once the current pending structure becomes empty. If a rule has *immediate* coupling, then it is placed into the current pending structure when triggered. If a rule has *deferred* coupling, then it is placed into the postponed pending structure when triggered. Both pending structures have the same semantics - e.g. both are queues or both are stacks. Once the current structure becomes empty, the postponed structure becomes the current structure and a new, empty, postponed structure is created. In actual systems, immediate mode is used to ensure that rule execution will take place within the body of the transaction, before execution of the next top level transaction statement. Deferred mode is used to postpone rule execution until the end of a transaction, just before the commit phase. The deferred semantics we consider is very similar to that used in HiPac [18] (but not the same, since HiPac makes rules that have been deferred execute in parallel, rather than sequentially).

In our semantics described in section 2, we effectively assumed immediate coupling for all rules and thus only the current pending structure was needed. If we allow deferred coupling, then we can get increased power which results in undecidability.

Theorem 5.8 *Termination is undecidable for 0-1 trigger systems using a stack and rules with deferred coupling.*

Proof (sketch): The proof is similar to that of theorem 5.7. We show how to use the deferred coupling capability to simulate the semantics of the queue, and so carry out the Post machine simulation given in theorem 5.6.

Given a Post machine, we define a set of active rules according to the scheme of theorem 5.6. Each of these rules has deferred coupling, and so, when triggered will always be placed in the postponed stack.

The state of the pending structures is described by the notation $[x_1, x_2, \ldots | y_1, y_2, \ldots]$ where the sequence to the left of the $|$ is the state of the 'current' stack (known as $stack_{curr}$) and the sequence to the right of $|$ represents the state of the 'postponed' stack (known as $stack_{post}$). x_1 and y_1 are the "heads" of the respective stacks.

Without loss of generality, let the state of the pending structures and database at some point in time be

$$[P_1, \ldots, P_n, M||], mstate = s_1$$

where each P_i is a rule package for the machine symbol a_i and M is a distinguished rule package that marks the bottom of $stack_{curr}$. $mstate$ is a variable used to indicate which state the Post machine is in. Now suppose the Post machine transition (s_1, a_i, s_2, a_x) is applicable at this point. Under queue semantics, the effect of P_1 would be to trigger P_x, placing it at the end of the queue and changing the machine state from s_1 to s_2, reaching the configuration of

$$[P_2, P_3, \ldots, P_n, P_x, M||], mstate = s_2.$$

We now sketch the sequence of steps needed to achieve this machine state. Similar to the proof of theorem 5.7, we use two mutually exclusive modes of operation, $normal$ mode and $duplicating$ mode. The meaning of these modes will be described in the subsequent trace - assuming without loss of generality that $normal$ mode is initially true. Let the state of the system be as follows:

$$[P_1, \ldots, P_n, M|], mstate = s_1, mode = normal$$

Recall that in the simulation of theorem 5.6, rule packages cause at most one event (due to the $flag$ variable enforcing mutual exclusion). For the present situation, when in $normal$ mode, rather than generating an event e_x to trigger package P_x, the actions of P_1 cause the value P_x to be saved in a variable named $recent$. The actions also cause $mstate$ to become s_2, the mode to become $duplicating$, and the rule package M'' to be triggered.

$$[P_2, \ldots, P_n, M|M''], mstate = s_2, mode = duplicating, recent = P_x$$

In duplicating mode, each rule package P_i just retriggers a (deferred) version of itself.

$$[P_3, \ldots, P_n, M|P_2, M''], mstate = s_2, mode = duplicating, recent = P_x$$
$$[P_4, \ldots, P_n, M|P_3, P_2, M''], mstate = s_2, mode = duplicating, recent = P_x$$
$$\vdots$$
$$[M|P_n, \ldots, P_2, M''], mstate = s_2, mode = duplicating, recent = P_x$$

The execution of rule package M has two effects: i) It triggers the package saved in the value $recent$ (in this case P_x) and ii) also triggers another rule package M' (whose effect is described below).

$$[|M', P_x, P_n, \ldots, P_2, M''], mstate = s_2, mode = duplicating, recent = P_x$$

Since all rules in $stack_{curr}$ have executed, $stack_{curr} = stack_{post}$ and $stack_{post} = []$.

$$[M', P_x, P_n, \ldots, P_2, M''|], mstate = s_2, mode = duplicating$$

The effect of executing rule package M' is to trigger package M.

$$[P_x, P_n, \ldots, P_2, M''|M], mstate = s_2, mode = duplicating$$

Each rule package P_i just retriggers (a deferred version) of itself as before.

$$[P_n, \ldots, P_2, M''|P_x, M], mstate = s_2, mode = duplicating$$
$$[P_{n-1}, \ldots, P_2, M''|P_n, P_x, M], mstate = s_2, mode = duplicating$$
$$[P_{n-2}, \ldots, P_2, M''|P_{n-1}, P_n, P_x, M], mstate = s_2, mode = duplicating$$
$$\vdots$$
$$[M''|P_2, \ldots, P_n, P_x, M], mstate = s_2, mode = duplicating$$

The effect of executing rule package M'' is to change from $duplicating$ mode back to $normal$ mode.

$$[|P_2, \ldots, P_n, P_x, M], mstate = s_2, mode = normal$$

Since all rules in $stack_{curr}$ have executed, $stack_{curr} = stack_{post}$ and $stack_{post} = []$.

$$[P_2, \ldots, P_n, P_x, M|], mstate = s_2, mode = normal$$

We have thus succeeded in placing P_x at the 'bottom' of $stack_{curr}$, while preserving the rules that were on $stack_{curr}$ originally. We are back in $normal$ mode and $stack_{post}$ is empty again. It is thus possible to simulate queue semantics using the deferred coupling capability and hence termination is undecidable. ■

6 Applications

We now examine some applications of the decidable configurations.

Safe Cones as a Query Language: In this paper we have presented the safe cones language as an update language, but it can just as easily be used as a query language. Since in the proofs we have given an equivalence class characterisation, it follows that problems such as containment and equivalence are decidable for safe cones queries.

In the context of information integration, an important problem is the ability to determine whether a query Q can be answered using a set of materialised views $V_1, \ldots, V_n$. Past work (e.g. [1]) has primarily focussed on considering view and query languages which are negation free. The safe cones language, however, can express negation and can indeed be used for this problem. Indeed, if the query Q and the views $V_1, \ldots, V_n$ can be expressed using the safe cones language, then it is possible to determine whether Q can be rewritten using $V_1, \ldots, V_n$. To see why this is so, there is a result from [1] which states that Q can be rewritten using a view V iff $\pi_\emptyset(Q) \subseteq \pi_\emptyset(V)$. So, since we are able to decide containment, we are thus able to decide if a rewriting is possible.

Inclusion Relationships: The simplest type of safe-cones triggers, the safe one-literal triggers, are well suited for enforcing and checking inclusion relationships. Standard inclusion dependencies of the form $R[A_1, \ldots, A_m] \subseteq S[B_1, \ldots, B_m]$ are easily expressed and decidability questions such as the implication problem for a set of dependencies can be straightforwardly translated. The full expressiveness of the safe-cones language can then be seen as a way of specifying more generalised inclusion relationships. Active database rules have been used as a mechanism for both checking integrity constraints and repairing violations of them [13]. Safe-cones triggers are therefore an obvious choice for checking inclusion dependencies and also for repairing (updating) the database if inconsistency does occur.

SQL Execution Model: As already noted, the stack execution semantics for sets of rules is equivalent to that used by SQL3 row-level triggers. Our results therefore imply that termination is decidable for SQL3 row-level triggers using safe cone queries within the condition and within the body of action [28].

Other Kinds of Analysis: The results we have presented can also be related to other properties of interest for active rules. The techniques used to prove our maximal decidability result for bounded model languages and bounded pending/stack structures (theorems 5.4 and 5.5), can be used to prove that *confluence* is also decidable for these systems, assuming the total order on rules is relaxed (recall that rule execution is confluent if the final state is unique, irrespective of what non deterministic choices are made when selecting the next rule from the pending structure to execute). This follows from the characterisation of the language via equivalence classes. If we define a property of *reachability* for active rules (*i.e.* can a rule be triggered as a consequence of some other rule being triggered), it is also possible to show that our (un)decidability results remain true if we replace the word "termination" by "reachability" in the relevant theorems. Furthermore, all our decidability and undecidability results apply to both termination and *satisfiability* for (the appropriate fragments of) the $while$ and $while_N$ languages.

7 Related Work

The safe-cones language is a class lying close to the boundary of decidability. This raises the question of whether there is some alternative "natural" way of varying updates, which does not rely on safety or the number of literals, yet does not sacrifice decidability. The answer is yes, provided we are willing to accept a reduction in the arity of our relations. Work in [9] (further extended in [7]) discusses languages which use unary views as building blocks. Updates may read only from unary views and both read/write from/to unary (base) relations. The view mechanism is used to give restricted access to higher arity relations (e.g. $V(X) \leftarrow R(X,Y), S(Y,Z), T(Z,X)$). For trigger languages which can use unary views of conjunctive queries, termination is decidable (and furthermore the language is a bounded model one). Extending the body of the view to use negation or inequality causes termination to become undecidable.

The work in section 5 of this paper is based upon that in [8]. The emphasis, however, is somewhat different. In [8], results were obtained on the expressiveness of rule systems measured by their ability to recognise various event histories. Termination theorems were then given as corollaries. Here, in contrast, our focus has been on termination and thus we have not related the expressiveness details for configurations we have examined.

In [31, 32], Picouet and Vianu presented the concept of the relational machine as useful for simulating an active database. It is essentially a Turing machine which has restricted access to a relational store via first order queries and is designed to capture the spirit of a database query language embedded in a host programming language such as C. An active database system is modelled by two relational machines, one replicating the external query system and the other duplicating the set of active rules. Using this model, statements can be made about the expressiveness of various simplified prototype systems. Some of the elements we have examined (*e.g.* coupling modes, pending sets) overlap with ones they have looked at, but their results do not directly address the question of rule termination. Thus, our work can be seen as complementary to theirs, since both are concerned with exploring and clarifying the fundamental behaviour of rule systems. The same is also true of [26], where a programming language which employs the delayed update or *delta* is defined. This can be used to express the semantics of certain active database systems.

In [27], methods for specifying meta features to manage execution of the rule set as a whole are presented. Although we also consider meta features, our interest is primarily in how they impact upon termination and not on how to analyse them as an entity in themselves. Supplementary to this is a recent work by Wang et al [36], where the property of confluence in the presence of meta rules is examined.

The techniques used to prove decidability of termination in section 3 depended upon an ability to analyse equivalence classes of the language. This idea of characterising the behaviour of a language by its equivalence classes, has also been used in other contexts. In [5], it is shown how one can define a fixpoint query to extract equivalence classes for a *while* program and order them. The number of equivalence classes for a given instance I is denoted $\#_k(I)$, where k is the number of free variables a query may have. It is observed that for the case of all unary input, $\#_k(I)$ is a constant independent of I. The decidability result of theorem 3.5 can be seen as

Safe cones	Decidable
Safe Two Literal	Undecidable
Safe-insert unsafe-delete one-literal	Undecidable
UnSafe-insert safe-delete one-literal	Undecidable

Table 1: Summary of Decidability Results for Language Syntax

Bounded Structure	Decidable
Stack	Decidable
Queue	Undecidable
Stack + Complex Events	Undecidable
Stack + Coupling Modes	Undecidable

Table 2: Summary of Decidability Results for Meta Features

a generalisation of this, since for the safe-cones language, $\#_k(I)$ is also a constant independent of I. To put it another way, we have identified a new fragment of the *while* language whose equivalence class $\#_k$ is instance independent. Furthermore, our proofs showed that we could construct a representative for each satisfiable combination of equivalence classes in an instance, using a bounded number of constants.

8 Summary and Further Work

Tables 1 and 2 summarise our results on rule updates and meta features. Looking at these, we can see that the most powerful decidable configuration is a system using the safe-cones language with either a stack or bounded model pending structure. We believe that this system is expressive enough to be interesting for rule designers to use. Furthermore, it is theoretically interesting from a language/logic perspective.

Bearing this in mind, we would also emphasise the importance of the undecidability results presented. In particular, the undecidability of the unsafe one literal language is rather surprising, given its seemingly “simple” nature. Both this and the undecidability results for the various meta features, seem to suggest a cautious view of verification in active rule systems is also needed.

There is clearly scope for investigation of further variations in classes of rule systems. Example possibilities are to limit the number of rules or the number of updates per rule (rather like the use of *sirrups* in boundedness [2]). Other alternatives are to vary the semantics by e.g. using instance instead of set-oriented execution.

9 Appendix: Proof of the decidability of safe one-literal triggers

In this appendix we prove Lemma 3.5. This proof is long and involved, and will follow these steps:

1. Translate a given set of rules into an equivalent set of rules over a schema just containing one relation. This is done to simplify the arguments.

2. Define a (finite) set of relational calculus queries such that each database instance, which is constructible by some sequence of updates, is the union of the answers to some of these queries on the initial database instance.

3. Show that the behaviour of the rules on every possible database state can be described in terms of these queries. In fact, the behaviour is imitated by the rules on a corresponding database instance with a bounded number of constants: A query defined in (2) has a nonempty answer on this new database instance iff it has a nonempty answer on this original database instance.

9.1 Translation to One Relation

Without loss of generality, we simplify the arguments by considering a schema with just one relation.

Proposition 9.1 *Let $\mathcal{R}_1$ be a set of rules written in the safe one-literal language over a schema $\mathcal{S}_1$. Suppose $\mathcal{S}_1$ contains n relations with the maximal arity being m. Then it is possible to define another set of safe one-literal rules $\mathcal{R}_2$ over a schema $\mathcal{S}_2$, which contains just one relation of arity $(m + 2 + \lceil \log_2 n \rceil)$ and $\mathcal{R}_1$ is non terminating iff $\mathcal{R}_2$ is non terminating.*

Proof: The proof is in two steps. First, by padding, we can construct a set of rules $\mathcal{R}_1'$ defined on a schema $\mathcal{S}_1'$ containing only relations of arity m and $\mathcal{R}_1'$ terminates iff $\mathcal{R}_1$ terminates. More specifically, for each relation R in $\mathcal{S}_1$ whose arity is less than m, we create another relation R' of arity m. We translate old rules by replacing references to R with references to R' and duplicating the variable in the last column an appropriate number of times. For example, the formula $R(X, Y, Z)$ becomes $R'(X, Y, Z, Z, Z)$ if $m = 5$. The relations are also translated in the same manner. Second, we construct $\mathcal{R}_2$ and $\mathcal{S}_2$ from $\mathcal{R}_1'$ and $\mathcal{S}_1'$. There is just one relation, called T. The first m columns are used to hold data contained in the original relations, whereas the additional $2 + \lceil log_2 n \rceil$ columns are used for specifying names of the original relations. X_{m+1} and X_{m+2} are used to store two arbitrary but unequal constants (from any domain). The final $\lceil \log_2 n \rceil$ columns are used to specify a number between 1 and n in binary, using X_{m+1} and X_{m+2} as the 0 and 1 respectively. For example, for $n = 4$ and $m = 5$, the update

$$-R_1(X, X, Y, Y, Y) \leftarrow R_0(X, Z, Z, Y, Y)$$

is mapped to

$$-T(X, X, Y, Y, Y, A, B, A, B) \leftarrow T(X, Z, Z, Y, Y, A, B, A, A), A \neq B$$

Old events are changed into new events by replacing the old relational atom using the relational atom plus any required equality and inequality constraints. ■

In the rest of this section we assume T is the only relation in the database and it has arity m.

9.2 Weak Equivalence Class Definition

Roughly speaking, given a relation T, we will construct, independently of the triggers, a finite number of relational calculus queries, say $C_1, \ldots, C_N$, which satisfy the following properties:

For each initial database instance I_0, each possible database instance constructible due to any possible sequence of updates on I_0 is the union of some of the sets $C_1[I_0], \ldots, C_N[I_0]$, and, it is the case that $C_i[I_0] = C_j[I_0]$ whenever $C_i[I_0] \cap C_j[I_0] \neq \emptyset$. In fact, the tuples in any one $C_i[I_0]$ will always "travel" together during the execution of the triggers.

Thus $C_1[I_0], ..., C_N[I_0]$ are nearly equivalence classes, except that the disjointness property is not satisfied. We will refer to each $C_i[I_0]$ a weak equivalence class (abbreviated as *WEC*) with respect to I_0 and C_i a weak equivalence class description (abbreviated as *WECD*), and we will say that C_i is the description of $C_i[I_0]$.

Equivalence relations on tuples have also been considered in other contexts [5]. One example is that of automorphism classes of tuples, where two tuples u, v are in the same equivalence class iff there exists an automorphism f of I such that $v = f(u)$. Although this relation ensures that "equivalent" tuples travel together, the number of equivalence classes depends on the structure of the input instance and so these equivalence classes are not usable for static analysis. Another example is the equivalence of tuples relative to FO^k (i.e. first order logic with k variables). Two tuples u, v are in the same equivalence class relative to a set of FO^k formulas P, if they cannot be distinguished by any composition of the formulas in P. Unfortunately, it has been proven [23] that even if we are given a finite set of such classes, in general it is impossible to produce an example database instance satisfying them. In contrast, the equivalence relation we will introduce has the desirable property that such example databases can be generated effectively.

We now present some intuition relating to the weak class descriptions. To ensure that tuples within a *WEC* cannot be separated, our *WECD*s will reflect the distribution of constants throughout the database. Consider the following initial database state

$$T = \{(1,1,1),(2,2,2),(8,8,8),(1,3,3),(8,9,9),(2,4,4),(1,3,5),(8,9,10),(2,6,7)\}$$

and suppose the following two updates have been performed sequentially

$\delta_0 : \quad -T(A,B,B) \leftarrow T(A,B,C), B \neq C$
$\delta_1 : \quad -T(A,A,A) \leftarrow T(A,B,B), A \neq B$

in the order of δ_0 followed by δ_1. T now contains $\{(1,1,1),(8,8,8),(2,4,4),(1,3,5),(8,9,10),(2,6,7)\}$. (Observe that $(1,3,3)$ is deleted by δ_0 but $(2,4,4)$ is not; and $(2,2,2)$ is deleted by δ_1 because of $(2,4,4)$.) Since $(1,1,1)$ and $(2,2,2)$ were both in the initial T, and since $(1,1,1)$ is still in the current T but $(2,2,2)$ is not, $(1,1,1)$ and $(2,2,2)$ should not be in the same *WEC*. Observe that $(1,1,1)$ and $(8,8,8)$ *should be* in the same *WEC*, since the constants 1 and 8 are distributed in a similar way throughout T. For this T, the *WEC* that $(1,1,1)$ and $(8,8,8)$ are in can be described by the relational calculus query (which does not depend on δ_0 or δ_1)

$$\{< A,A,A > | \exists B, C(T(A,A,A) \wedge T(A,B,B) \wedge T(A,B,C) \wedge A \neq B \wedge B \neq C)\}$$

and the class for $(2,2,2)$ can be described by

$$\{< A, A, A > | \exists B(T(A, A, A) \wedge T(A, B, B) \wedge A \neq B \wedge \neg\exists C(T(A, B, C) \wedge B \neq C))\}.$$

The actual *WECD*s are more involved than these two queries, although they are equivalent to these two for the example database.

The complexity of the *WECD*s is essentially determined by the number of ways updates can "chase" components of the m-tuple around the relation T. Such chasing is limited for the safe one-literal updates, since such an update can only perform selection, projection, difference, or union. Thus a tuple $< a_1, \ldots, a_m >$ will be affected only by those tuples containing a superset of its constants. This is a key intuition behind our construction below.

We need some symbolisms to help construct the *WECD*s.

Each update in rules will need to use at most m variables, and we assume these are drawn from the set $\{X_1, \ldots, X_m\}$. Some updates may only refer to a smaller number of variables. For each $1 \leq i \leq m$, let $U_i = \{X_j \mid 1 \leq j \leq i\}$; a *variable pattern* over U_i is an m-tuple $< X_{i_1}, X_{i_2}, \ldots, X_{i_m} >$ constructed using all the variables in U_i but no more (possibly with repetitions); and let $\mathcal{S}_i = \{T(V) \mid V$ is a variable pattern over $U_i\}$. Observe that $|\mathcal{S}_i| = \{{m \atop i}\} * i!$, where $\{{m \atop i}\}$ is the number of partitions of m elements into i nonempty sets (the Stirling number), and thus $|\mathcal{S}_i| \leq m^i * i!$. (View the m elements as the positions from 1 to m. Each partition consists of all positions for one of the i variables. Since each position has at most i choices of partitions to belong to, $\{{m \atop i}\} \leq m^i$.)

Some updates may have more variables in their bodies than in their heads, i.e. they use projections. We will see shortly that it necessary to enumerate all the ways a tuple can be projected by an update, when defining the *WECD*s. We would therefore like to capture how the variables in the body of an update are mapped to the variables in the head. To specify the space of all such possible mappings, we define injection mappings. For all $1 \leq i < j \leq m$, let $\mathcal{F}_{i,j}$ be the set of injection mappings[2] from U_i to U_j. Intuitively, U_j consists of the variables in the body and U_i those in the head. The injections then let us refer to all the possible ways (i.e. for all possible updates) that variables from a formula with i variables (the head) could be appear within a formula having $j > i$ variables (the body) For each $f \in \mathcal{F}_{i,j}$, let $\overline{Y_f} = U_j - U_i$, and let f^{-1} denote a permutation of U_j such that $f^{-1}(X_s) = X_t$ if $f(X_t) = X_s$. (For example, suppose $i = 2$, $j = 4$, $f(X_1) = X_3$ and $f(X_2) = X_1$. Then $U_4 - U_2 = \{X_3, X_4\}$; $f^{-1}(X_3) = X_1$, $f^{-1}(X_1) = X_2$, $f^{-1}(X_2) = X_3$ and $f^{-1}(X_4) = X_4$; f^{-1} has freedom on variables that are not in the range of f, e.g. we could let $f^{-1}(X_2) = X_4$ and $f^{-1}(X_4) = X_3$.) Intuitively, f^{-1} allows us to refer to all the possible ways variables within a formula (the body) can be re-arranged with respect to the head. We will be later be applying f^{-1} to formulas, e.g. $f^{-1}(C(\overline{X})$. The meaning here is that f is an injection mapping from some $\overline{Y}$ to $\overline{X}$ ($Y \subseteq X$) and so f^{-1} is a re-arrangement (permutation) of the variables in $\overline{X}$ with respect to $\overline{Y}$.

We now inductively define the set of *WECD*s, which are partitioned into m groups. A *WECD* belongs to group i iff each tuple satisfying this description contains exactly i distinct constants. Descriptions in group $i - 1$ are defined using those in groups $i, i + 1, ..., m$. Each description C is associated with a variable pattern V, and it will be referred to as $C(V)$ if we wish to refer to the variable pattern, and simply as C otherwise.

[2] An injection mapping from S to S' is a total 1-to-1 mapping but not necessarily onto.

Group m: This set of *WECD*s consists of all queries specified by formulas of the form:

$$\{< X_1, X_2, \ldots, X_m >| \; (\bigwedge_{\psi \in \mathcal{S}} \psi) \; \wedge \; (\bigwedge_{\psi \in \mathcal{S}_m - \mathcal{S}} \neg\psi) \wedge \phi_m\}$$

where ϕ_m is a formula that says that all variables in U_m are unequal, and $\mathcal{S}$ is any nonempty subset of $\mathcal{S}_m$. Thus a *WECD* in this group completely describes, for each m-tuple t in the corresponding *WEC*, the distribution of m-tuples containing exactly the constants in t. Observe that there are $2^{m!} - 1$ *WECD*s in this group.

Example 9.2 The following is a *WECD* in group 3 for $m = 3$:

$T(X_1, X_2, X_3) \wedge T(X_1, X_3, X_2) \wedge T(X_2, X_1, X_3) \wedge \neg T(X_2, X_3, X_1)$
$\wedge T(X_3, X_1, X_2) \wedge T(X_3, X_2, X_1) \wedge X_1 \neq X_2 \wedge X_1 \neq X_3 \wedge X_2 \neq X_3.$

Observe that $\mathcal{S}_3$ has 6 elements. The corresponding $\mathcal{S}$ for this *WECD* contains the five positive relational atoms, and $\mathcal{S}_3 - \mathcal{S} = \{T(X_2, X_3, X_1)\}$. ■

Group $m - j$: This is the induction step. Suppose $j < m$ is a positive integer and let $\mathcal{C}_i$ denote the set of *WECD*s of group i for each $m - j < i \leq m$. We wish to define *WECD*s of group $m - j$.

We first explain in this and the next paragraphs the intuition and intricacies behind the definitions. To simplify the argument, we first consider the simplest of all these groups, namely group $m - 1$. Tuples in each *WEC* in this group are m-tuples where two of the components are equal, and the rest unequal (*i.e.* each of these tuples contains exactly $m - 1$ different values). For each tuple t in this group, the *header* of the *WEC* formula checks for the existence or non existence of all possible permutations of t in the initial database (as was done above for group m). There is an extra complication, however, due to the fact that not all of the values are unequal. It means that we must also consider the distribution of supersets of the tuple's constants in the initial database. This is because an update might be applied to a tuple in a higher group and a resulting new tuple could then be part of group $m - 1$ (due to projections) and thus be a permutation of t. However, since this new tuple for group $m - 1$ didn't appear in the initial database, it thus wasn't included as a possible permutation of t in the initial database. It therefore didn't appear in the header of the equivalence formula for t.

For example, for a tuple $< 1, 2, 2 >$, we need to look for other tuples containing the values 1 and 2 such as $< 1, 3, 2 >$ and $< 2, 1, 4 >$. These will be taken care of using *WECD*s of the higher groups, with the aid of the injection mappings. In summary, we need to identify existence of tuples with exactly some $m - 1$ distinct values, and for each *WEC* in group m and for each possible projection, whether these $m - 1$ values occur in that *WEC* (together with some additional value).

For the general case, each tuple in some *WEC* in this group has exactly $m - j$ distinct values. To specify such a description, we need to identify existence of tuples with exactly some $m - j$ distinct values, and we need to identify, for each *WEC* in groups $m - j + 1, ..., m$ and for each possible projection, whether these $m - j$ values occur in that *WEC* (together with some additional values).

Technically, let $\mathcal{FQ}_{m-j}$ be the set $\bigcup_{i=m-j+1}^{m} (\mathcal{F}_{m-j,i} \times \mathcal{C}_i)$. $\mathcal{FQ}$ represents all possible pairs (a, b) (the cross product) of an injection mapping a from U_{m-j} to U_i ($i > m - j$) and a *WECD* b from $\mathcal{C}_i$. It is used so we can generate all possible $m - j$ projections of equivalences classes in groups $i > m - j$. Such projections could occur when updates are applied to the database. Projected equivalence classes in groups $i > m-j$ are then combined with possible permutations of tuples in group $m-j$ to yield the entire equivalence class description for group $m-j$.

*WECD*s of group $m-j$, $\mathcal{C}_{m-j}$, are formulas of the following form:

$$\{< X_{i_1}, ..., X_{i_m} >| \, (\bigwedge_{\psi \in \mathcal{S}} \psi) \ \wedge \ (\bigwedge_{\psi \in \mathcal{S}_{m-j} - \mathcal{S}} \neg\psi) \ \wedge \ \phi_{m-j}$$
$$\wedge \, (\bigwedge_{(f,Q) \in \mathcal{FQ}} \exists \overline{Y_f} f^{-1}(Q)) \ \wedge \ (\bigwedge_{(f,Q) \in \mathcal{FQ}_{m-j} - \mathcal{FQ}} \nexists \overline{Y_f} f^{-1}(Q))\}$$

where $< X_{i1}, ..., X_{i_m} >$ is a variable pattern over U_{m-j}, ϕ_{m-j} is a formula that says that all variables are unequal, $\mathcal{S}$ is any subset of $\mathcal{S}_{m-j}$, and $\mathcal{FQ}$ is any subset of $\mathcal{FQ}_{m-j}$ such that at least one of $\mathcal{S}$ and $\mathcal{FQ}$ is not empty. Thus a *WECD* in this group completely describes, for each m-tuple t in the *WEC*, the distribution of m-tuples containing exactly the set of or a superset of the constants in t. The use of f allows us to describe different ways projections (of tuples in groups $i > m-j$) can be done in updates, by "passing" constants between an atom in the body and the atom in the head.

We do not use unsafe queries as *WECD*s since tuples in such classes cannot be generated during rule execution.

Example 9.3 We now describe the construction of group $m-j$, indicating the ingredients. We consider the construction of *WECD*s of group 2 for $m = 3$.

- Observe that $\mathcal{S}_2 = \{T(X_1, X_1, X_2), T(X_1, X_2, X_1), T(X_2, X_1, X_1), T(X_1, X_2, X_2), T(X_2, X_1, X_2), T(X_2, X_2, X_1)\}$. In each *WECD* of group 2, each of these 6 elements can occur either positively or negatively but not both.
- There are 6 injections from $\{X_1, X_2\}$ to $\{X_1, X_2, X_3\}$. Let $f_1, ..., f_6$ be an enumeration of them.

  Let $C_1(X_1, X_2, X_3)$, ..., $C_{63}(X_1, X_2, X_3)$ be an enumeration of the *WECD*s of group 3.

  In each *WECD* of group 2, each of $\exists \overline{Y_{f_i}} f_i^{-1}(C_j)$ can occur either positively or negatively but not both. For example, suppose f_1 is defined such that $f_1(X_1) = X_2$ and $f_1(X_2) = X_3$. Then $\exists \overline{Y_{f_1}} f_1^{-1}(C_1(X_1, X_2, X_3))$, namely $\exists X_3 C_1(X_3, X_1, X_2)$, can occur either positively or negatively but not both in each *WECD* of group 2.

So there are $6 \times 2^{6+6\times 63} - 1$ *WECD*s in group 2. Note: The first and second occurrences of 6 actually corresponds to the fact that $|\mathcal{S}_2| = 6$.

■

Example: This follows on from the previous example. We give a *WECD* in group 2 for $m = 3$

$$\{< X_1, X_1, X_2 >| \, T(X_1, X_1, X_2) \wedge T(X_1, X_2, X_1) \wedge \neg T(X_2, X_1, X_1) \wedge \neg T(X_1, X_2, X_2) \wedge$$
$$T(X_2, X_1, X_2) \wedge T(X_2, X_2, X_1) \wedge \bigwedge_{i \in 1...63} \exists X_3 C_i(X_1, X_2, X_3) \wedge \bigwedge_{i \in 1...63} \exists X_3 C_i(X_1, X_3, X_2) \wedge$$
$$\bigwedge_{i \in 1...63} \exists X_3 C_i(X_3, X_1, X_2) \wedge \bigwedge_{i \in 1...63} \exists X_3 C_i(X_2, X_1, X_3) \wedge \bigwedge_{i \in 1...63} \exists X_3 C_i(X_2, X_3, X_1) \wedge$$
$$\bigwedge_{i \in 1...63} \exists X_3 C_i(X_3, X_2, X_1) \wedge X_1 \neq X_2 \wedge X_1 \neq X_3 \wedge X_2 \neq X_3\}$$

Where $C_1, \ldots, C_{63}$ are all the equivalences classes for group 3 (Example 9.2 lists one of them). Other equivalence classes in group 2 could be generated by either a) Using a different 'output' variable pattern instead of $< X_1, X_1, X_2 >$ (e.g. $< X_2, X_1, X_1 >$) and/or b) adding/removing negations from either the $T(\ldots)$'s or negating some of the $\exists X_3 C(\ldots)$ formulas. ■

The *WECD*s constructed in this way do not guarantee disjointness. We have chosen to do so to simplify the arguments, as this suffices for our purpose. A more involved construction could be given to guarantee disjointness.

We conclude this subsection by proving that the number of *WECD*s is bounded.

Lemma 9.4 *The number of WECDs is bounded by a constant depending only on m, the arity of T.*

Proof: Let N_i denote the number of *WECD*s in group i. From the construction of Group m, it is easily seen that $N_m = 2^{m!} - 1 \leq 2^{m!}$. In general, $N_i \leq |\mathcal{S}_i| \times 2^{|\mathcal{S}_i| + \Sigma_{i<j\leq m}(\mathcal{I}_{ij} \times N_j)}$, where $\mathcal{I}_{ij}$ is the number of injections from i elements to j elements; the components in the right hand side of the inequality are in direct correspondence of the construction of the *WECD*s in such groups: $|\mathcal{S}_i|$ is the number of ways to choose the variables to the left of the "|" for rearranging the i variables that occur to the right of the "|", $2^{|\mathcal{S}_i|}$ corresponds to the number of choices of $\mathcal{S}$, and $2^{\Sigma_{i<j\leq m}(\mathcal{I}_{ij} \times N_j)}$ corresponds to the number of choices of $\mathcal{FQ}$. Recall that, to make the argument simpler, we consider *WECD*'s and some of them are equivalent to each other.

The number $\mathcal{I}_{ij}$ of injections from U_i to U_j is $\binom{j}{i} * i!$, where $\binom{j}{i}$ is the combinatorial number of choosing i things from j things. So $\mathcal{I}_{ij} \leq m!$. $|\mathcal{S}_i|$ is the number of partitions of m positions into i nonempty sets (the Stirling number $\left\{{m \atop i}\right\}$) times $i!$; as noted earlier, $|\mathcal{S}_i| \leq m^i * i!$, and so $|\mathcal{S}_i| \leq m^{2m}$. Hence N_i is bounded above by $m^{2m} \times 2^{2*m^{2m}*N_{i+1}}$, which is bounded above by $O(2^{N_{i+1}^2})$. So the number of *WECD*s is bounded above by some nonelementary number with m levels of exponents! ■

The bound given in the above lemma is a worst case bound. In practical situations, this can be a lot better. Indeed, it appears that the number of m levels of exponentials can be replaced by ρ levels of exponentials, where ρ is the number of arities among $1, \cdots, m$ that are referred to by the conditions or the updates.

9.3 Termination Decision Procedure

Our algorithm for deciding termination is as follows, with a safe one-literal trigger set as input. In the following proof we assume the existence of priorities which can guarantee a unique next trigger to execute (recall that we are considering the singleton pending rule structure in this section). The general case where this is not true can also be handled by adapting the algorithm to iteratively branch whenever it has to choose an ordering.

1. Repeat the following steps, 2 and 3, *for each possible* starting state $(I, \mathcal{R})$, where the initial database instance is I and the initial set of triggered rules is $\mathcal{R}$, and
   - I is an instance using (not necessarily all of) some k fixed constants, and k is the number obtained in lemma 9.6;
   - $\mathcal{R}$ is the set of all rules that could be triggered by some $Insert(T(\overline{X}), \theta)$ event or the set triggered by some $Delete(T(\overline{X}), \theta)$ event, where $\overline{X}$ is a variable pattern over U_m and θ is a conjunction of

inequalities over U_m. $\mathcal{R}$ thus corresponds to a combination of rules that could have been initially triggered by a single statement in the host transaction.

2. Run the rule set on an arbitrary initial state specified in Step 1. If a state repeats, then report *non termination* and exit.
3. If the execution in 2 terminates for all possible initial states, then report *termination*.

The correctness proof of this procedure will depend on two key lemmas. Together they ensure that the termination behaviour of the triggers on arbitrary databases is simulated by the termination behaviour of the triggers on small databases using no more than a fixed number of constants, and that fixed number can be effectively constructed.

Let $\mathcal{C}$ denote the set of all *WECD*s defined for a fixed relation T. A subset $\mathcal{C}'$ of $\mathcal{C}$ is said to be *realised* by a database instance I if $C \in \mathcal{C}'$ iff $C[I]$ is not empty; and $\mathcal{C}'$ is *realisable* if there exists a database instance I such that $\mathcal{C}'$ is realised by I. Intuitively, the following lemma demonstrates that the equivalence classes present in the initial database state cannot be split by any sequence of updates. This is because they were initially designed by taking into account all possible compositions of updates.

Lemma 9.5 *Let I_0 be a database instance with a single relation T, $\delta = \delta_0, \ldots, \delta_{n-1}$ a sequence of updates and $I_1, ..., I_n$ the database instances such that $I_{i+1} = \delta_i(I_i)$ for each i. Let $\mathcal{C}'_i = \{C \mid I_i \cap C[I_0] \neq \emptyset\}$, for each $0 \leq i \leq n$.* [3] *Then $I_i = \bigcup_{C \in \mathcal{C}'_i} C[I_0]$, and $\mathcal{C}'_i$ depends only on the updates and $\mathcal{C}'_0$.*

Proof: This lemma says that after applying some updates to a database instance, the resulting instance consists only of tuples which are in equivalence classes that appeared in the initial instance. i.e. All tuples are in an equivalence class and no equivalence class now exists which wasn't also present in the initial database instance.

Without loss of generality, we can assume that the inequality conditions of the updates state that all the variables used in the updates are unequal. Indeed, we can transform the updates to satisfy this requirement such that the final database instance produced by the new updates is identical to the one produced by the old updates. We do this by appropriately duplicating the updates, by using all homomorphic images of the updates which respect the original inequality constraints and by adding inequality constraints to the bodies of the updates. For example, we replace the following update

$-T(X_1, X_2, X_1) \leftarrow T(X_1, X_2, X_3), X_1 \neq X_3$

by the following three new updates (which can be applied in any order):

$-T(X_1, X_1, X_1) \leftarrow T(X_1, X_1, X_3), X_1 \neq X_3$
$-T(X_1, X_2, X_1) \leftarrow T(X_1, X_2, X_2), X_1 \neq X_2$
$-T(X_1, X_2, X_1) \leftarrow T(X_1, X_2, X_3), X_1 \neq X_2, X_1 \neq X_3, X_2 \neq X_3$

We will verify the lemma by induction on i. For the base case of $i = 0$, it is clear that $\mathcal{C}'_0$ depends only on $\mathcal{C}'_0$. Let $t =< t_1, ..., t_m >$ be a tuple. We now prove that $t \in I_0 \Leftrightarrow t \in \bigcup_{C \in \mathcal{C}'_0} C[I_0]$.

[3] Observe that $\mathcal{C}'_0$ is the subset of $\mathcal{C}$ which is realised by I_0.

($\Rightarrow$) Let $t = < t_1, ..., t_m >$ be a tuple in I_0. We can construct a *WECD* C such that $t \in C[I_0]$, where the positive literals of C correspond to elements of the following set $\mathcal{S}_t = \{T(X_{i_1}, ..., X_{i_m}) \mid < t_{i_1}, ..., t_{i_m} > \in I_0$ and it is a permutation of $t\}$; if t has less than m distinct values, then C must also describe how projections of t occur in I_0 via choices on $\mathcal{FQ}$ (see the construction). Since the disjunction of these choices is $true$ (i.e. at least one of the combinations is guaranteed to be true for any I_0 due to the exhaustive nature of their construction), it is clear that $t \in C[I_0]$ for at least one of these choices. So I_0 is contained in $\bigcup_{C \in \mathcal{C}'_0} C[I_0]$.

($\Leftarrow$) Let $C(\overline{X}) \in \mathcal{C}'_0$ be fixed and let t be a tuple in $C[I_0]$. By the definition of $\mathcal{C}'_0$, there exists some tuple $t' \in I_0 \cap C[I_0]$. The definition of C must contain $T(\overline{X})$ as a positive literal, since t' cannot be in $C[I_0]$ otherwise. This implies that t is in $I_0(T)$. Thus $\bigcup_{C \in \mathcal{C}'_0} C[I_0]$ is contained in I_0, and therefore they are equal.

For the induction step, suppose the lemma is true for some $i \geq 0$. We will only need to specify how to derive $\mathcal{C}'_{i+1}$ from $\mathcal{C}'_i$. $I_{i+1} = \bigcup_{C \in \mathcal{C}'_{i+1}} C[I_0]$ follows easily because of the fact that all first-order queries are generic [3], and the fact that each *WECD* is an exhaustive construction of ways to project, select and intersect any tuple - the only operations one literal updates can perform.

The update δ_i is either an insertion or a deletion. We consider the deletion case, the insertion case being similar (and omitted). Suppose the update δ_i is the following:

$-T(\overline{X}) \leftarrow T(\overline{Z}), \theta$

where θ states that all variables in $\overline{Z}$ are unequal. Let $C_1(\overline{Z}_1), ..., C_\rho(\overline{Z}_\rho)$ be an enumeration of elements in $\mathcal{C}'_i$ whose variable pattern $\overline{Z}_j$ can be renamed to $\overline{Z}$; these will be used to identify those tuples that might lead to instantiated updates. Let $C'_1(\overline{X}_1), ..., C'_{\rho'}(\overline{X}_{\rho'})$ be an enumeration of elements in $\mathcal{C}'_i$ whose variable pattern $\overline{X}_j$ can be renamed to $\overline{X}$, say using σ_j; these will be useful to identify those tuples that might be updated (deleted here). Let $C''_1(\overline{Y}_1), ..., C''_{\rho''}(\overline{Y}_{\rho''})$ be an enumeration of elements in $\mathcal{C}'_i$ whose variable pattern $\overline{Y}_j$ cannot be renamed to $\overline{X}$; these will be used to identify those tuples that will definitely not be affected by this update.

By renaming variables if necessary, we can assume that $\overline{X}$ is a variable pattern over some U_κ, and that $\overline{Z}$ is a variable pattern over $U_{\kappa'}$, where κ' is the number of variables in $\overline{Z}$. Clearly, $\kappa' \geq \kappa$.

For each $1 \leq s \leq \rho$, let $\mathcal{F}_s$ be the set of injections from variables in $\overline{X}$ to variables in $\overline{Z}_s$. For each $f \in \mathcal{F}_s$, we will use f^{-1} (see Section 3.3.2) as a projection mapping from relations to relations.

We consider the case when $\kappa' > \kappa$, the case when $\kappa' = \kappa$ being similar but simpler (since we do not need projections in specifying the classes). We will use the result of the following rewriting process to specify the *WECD*s in $\mathcal{C}'_{i+1}$.

Intuitively, I_{j+1} consists of those tuples that are unaffected by the update plus those tuples which 'match' the head pattern $\overline{X}$ minus those tuples which 'match' the body pattern $\overline{Z}$.

$$\begin{aligned} I_{j+1} &= \textstyle\bigcup_{1 \leq j \leq \rho''} C''_j[I_0] \cup (\bigcup_{1 \leq j \leq \rho'} C'_j(\overline{X}_j)[I_0] - \bigcup_{1 \leq s \leq \rho} \bigcup_{f \in \mathcal{F}_s} f^{-1}(C_s(\overline{Z}_s)[I_0])) \\ &= \textstyle\bigcup_{1 \leq j \leq \rho''} C''_j[I_0] \cup (\bigcup_{1 \leq j \leq \rho'} \sigma_j(C'_j(\overline{X}_j)) \wedge \bigwedge_{1 \leq s \leq \rho} \bigwedge_{f \in \mathcal{F}_s} \neg \exists \overline{Y}_f f^{-1}(C_s(\overline{Z}_s)))[I_0]. \end{aligned}$$

Clearly, each C''_j belongs to $\mathcal{C}'_{i+1}$, since $\overline{X}_j$ and $\overline{X}$ are not renamings of each other. The other *WECD*s in $\mathcal{C}'_{i+1}$ can now be given by considering each formula

(*) $\sigma_j(C'_j(\overline{X}_j)) \wedge \bigwedge_{1\leq s\leq \rho} \bigwedge_{f\in\mathcal{F}_s} \neg\exists \overline{Y}_f f^{-1}(C_s(\overline{Z}_s))$

in the result of the rewriting above. We check $\neg\exists\overline{Y}_f f^{-1}(C_s(\overline{Z}_s))$ against subformulas in $\sigma_j(C'_j(\overline{X}_j))$: We eliminate duplicates (upto renamings) in (*), and include (*) in $\mathcal{C}'_{i+1}$ precisely when there is no inconsistency in (*), i.e. it is not the case that a formula and its negation (up to renaming) are both present. ■

The above lemma has a very important implication: The termination behaviour of the triggers on one particular database instance can be simulated by any other database instance, as long as they realise the same set of *WEC*s. Therefore, all we need now is to show the following:

Lemma 9.6 *There is an integer k such that, every realisable subset $\mathcal{C}'$ of $\mathcal{C}$ can be realised by a database instance I using at most k constants.*

Proof: The *WEC*s in $\mathcal{C}'$ are realised bottom-up from group $p = 1$ to group $p = m$. For each *WEC* in $\mathcal{C}'$, we use a new set of constants to create a representative m-tuple, respecting the appropriate equality patterns. We insert tuples which correspond to the T atoms which are not negated in the conjuncts of the *WECD*. Where a conjunct has an existential quantifier with some new variable, we create a new constant for it and then proceed in the same way. We assign different new constants for different occurrences of the existential quantifier. The use of new constants for each class eliminates "crosstalk" between them. The number of constants needed for any particular *WEC* is bounded by

$$1 + N + N^2 + ... + N^m \leq (m + 1)N^m,$$

where N is the total number of *WECD*s. Thus the total number of constants for all classes is bounded by $(m + 1) \times N^m \times N$.

To verify that this procedure is correct, we need to show that this instance cannot generate any unwanted classes. More precisely, we will show that if it does realise a class, then this class would be in $\mathcal{C}'$, *i.e.* it would be realised by every instance which realises $\mathcal{C}'$.

Suppose $\mathcal{C}'$ is realised by a given database instance I. Let J be our realisation of $\mathcal{C}'$ constructed above. Let C_b be some *WEC* that has been realised by J. It is easy to show that C_b must be in $\mathcal{C}'$.

Indeed, let t be an element in $C_b[J]$. Let S_b be the set of all tuples t' in J such that the constants in t intersects with the set of constants contained in t'. All these must have been generated in the process of making some *WECD* non empty, since different constants were used for different classes; call this C_a. The truth of each subformula of $C_a[J]$ on t and the truth of each subformula of $C_b[J]$ on t must coincide. Hence C_a and C_b are the same. ■

Acknowledgements

Part of Guozhu Dong's work was supported by research grants from the Australian Research Council (ARC) and from Ohio Board of Regents. Part of Rao Kotagiri's work was supported by the ARC.

References

[1] Levy. A., Mendelzon. A., Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proceedings of the ACM International Conference on Principles of Database Systems*, pages 95–104, San Jose, California, 1995.

[2] S. Abiteboul. Boundedness is undecidable for datalog programs with a single recursive rule. *Information Processing Letters*, 32:281–288, 1989.

[3] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[4] S. Abiteboul and V. Vianu. Procedural languages for database queries and updates. *Journal of Computer and System Sciences*, 41:181–229, 1990.

[5] S. Abiteboul and V. Vianu. Computing with first-order logic. *Journal of Computer and System Sciences*, 50(2):309–335, 1995.

[6] A. Aiken, J. Widom, and J. M. Hellerstein. Behaviour 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.

[7] J. Bailey and G. Dong. Decidability of first order logic queries over views. In *Proceedings of the Seventh International Conference on Database Theory, LNCS 1540*, pages 83–99, Jerusalem, Israel, 1999.

[8] J. Bailey, G. Dong, and K. Ramamohanarao. Structural issues in active rule systems. In *Proceedings of the Sixth International Conference on Database Theory, LNCS 1186*, pages 203–214, Delphi, Greece, 1997.

[9] J. Bailey, G. Dong, and K. Ramamohanarao. Decidability and undecidability results for the termination problem of active database rules. In *Proceedings of the 17th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 264–273, Seattle, Washington, 1998.

[10] E. Baralis, Ceri. S., and S. Paraboschi. Compile-time and runtime analysis of active behaviors. *IEEE Transactions on Knowledge and Data Engineering*, 10(3):353–370, 1998.

[11] 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.

[12] S. Ceri, P. Fraternali, S. Paraboschi, and L. Branca. Active rule management in chimera. In J. Widom and S. Ceri, editors, *Active Database Systems*. Morgan-Kaufmann, San Mateo, California, 1995.

[13] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proceedings of the 16th International Conference on Very Large Databases*, pages 566–577, Brisbane, Australia, 1990.

[14] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data and Knowledge Engineering*, 14(1):1–26, 1994.

[15] A. Cherubini, C. Citrini, S. C. Reghizzi, and D. Mandrioli. Qrt fifo automata, breadth-first grammars and their relations. *Theoretical Computer Science*, 85, 1991.

[16] C. Collet, T. Coupaye, and T. Svensen. Naos: Efficient and modular reactive capabilities in an object oriented database system. In *Proceedings of the 20th International Conference on Very Large Data bases*, pages 132–143, Santiago, Chile, 1994.

[17] S. Comai and L. Tanca. Using the properties of datalog to prove termination and confluence in active databases. In *Proceedings of the Third International Workshop on Rules in Database Systems*, Skövde, Sweden, 1997.

[18] U. Dayal et al. The HiPAC project: Combining active databases and timing constraints. *ACM SIGMOD Record*, 17(1):51–70, 1988.

[19] H. Gaifman, H. Mairson, Y. Sagiv, and M. Vardi. Undecidable optimisation problems for database logic programs. In *Proceedings of the Second IEEE Symposium on Logic in Computer Science*, pages 106–115, Ithaca, NY, 1987.

[20] H. Gaifman, H. Mairson, Y. Sagiv, and M. Vardi. Undecidable optimisation problems for database logic programs. *Journal of the ACM*, 40(3):683–713, 1993.

[21] S. Gatziu and K. Dittrich. Detecting composite events in active database systems using Petri nets. In *Proceedings of the 4th International Workshop on Research Issues in Data Engineering*, pages 2–9, Houston, Texas, 1994.

[22] N. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model and implementation. In *VLDB'92*, pages 327–338, 1992.

[23] M. Grohe. Large finite structures with few L^k-types. In *Proceedings of the 12th IEEE Symposium on Logic in Computer Science*, pages 216–227, 1997.

[24] G. Hillebrand, P. Kanellakis, H. Mairson, and M. Vardi. Undecidable boundedness problems for datalog programs. *Journal of Logic Programming*, 25(2):163–190, 1995.

[25] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[26] R. Hull and D. Jacobs. Language constructs for programming active databases. In *Proceedings of the 17th International Conference on Very Large Databases*, pages 455–468, Barcelona, Spain, 1991.

[27] H. V. Jagadish, A. O. Mendelzon, and I. S. Mumick. Managing rule conflicts in an active database. In *Proceedings of the 14th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 192–201, Montreal, Canada, 1996.

[28] K. Kulkarni, N. Mattos, and R. Cochrane. Active database features in SQL3. In N. Paton, editor, *Active Rules in Database Systems*, pages 197–219. Springer–Verlag, 1999.

[29] A. Levy, I. S. Mumick, Y. Sagiv, and O. Shmueli. Equivalence, query reachability, and satisfiability in datalog extensions. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 109–122, Washington D.C., 1993.

[30] B. Ludäscher, U. Hamann, and G. Lausen. A logical framework for active rules. In *Proceedings of the seventh International Conference on Management of Data*, Pune, India, 1995.

[31] P. Picouet and V. Vianu. Semantics and expressiveness issues in active databases. In *Proceedings of the 14th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 126–138, San Jose, California, 1995.

[32] P. Picouet and V. Vianu. Expressiveness and complexity of active databases. In *Proceedings of the Sixth International Conference on Database Theory, LNCS 1186*, pages 155–172, Delphi, Greece, 1997.

[33] P. Picouet and V. Vianu. Semantics and expressiveness issues in active databases. *Journal of Computer and System Sciences*, pages 325–355, 1998.

[34] E. Post. Finite combinatory processes-formulation I. *Journal of Symbolic Logic*, 1:103–105, 1936.

[35] 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.

[36] X. Wang, J. You, and L. Yuan. On confluent property of active database with meta-rules. In *Proceedings of the Third International Workshop on Rules in Database Systems*, Skövde, Sweden, 1997.

[37] J. Widom. The Starburst rule system: Language design, implementation and applications. *IEEE Data Engineering Bulletin*, 15(1–4):15–18, 1992.

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

[39] C. Zaniolo. Active database rules with transaction-conscious stable-model semantics. In *Proceedings of DOOD'95*, pages 55–72, 1995.

[40] Y. Zhou and M. Hsu. A theory of rule triggering systems. In *Proceedings of the International Conference on Extending Database Technology*, pages 407–422, Venice, Italy, 1990.