Decidability and Undecidability Results for the Termination Problem of Active Database Rules

James Bailey

Department of Computer Science, King's College London, Strand, London WC2R 2LS

james@dcs.kcl.ac.uk

Guozhu Dong

Department of Computer Science, University of Melbourne, Parkville, 3052, Australia

dong@cs.mu.oz.au

Kotagiri Ramamohanarao

Department of Computer Science, University of Melbourne, Parkville, 3052, Australia

dong@cs.mu.oz.au

Abstract

Active database systems enhance the functionality of traditional databases through the use of active rules or 'triggers'. One of the principal 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 two families of rule languages, the *one literal* languages where each update is permitted to have just one atom in its body, and the *unary* languages where only unary relations may be updated, but higher arity relations may be accessed through views. Within each of these, we identify members close to the boundary of (un)decidability. Our context is similar to the *while* query language and the dynamics gives an interesting contrast to Datalog with negation; our results shed insights on the power of triggers as well as comparison of the termination problem to boundedness and query containment.

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 [18].

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. 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 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*.

1.1 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 two 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

To appear in the 17th ACM Symposium on Principles of Database Systems, Seattle, 1998

adopted by the commercial database system Oracle, where a sequence limit of 32 is used. While easy to implement, it has the defect that valid rule execution sequences may exceed this limit and be prematurely halted, an approach unsuitable for applications where correctness is paramount, such as mission critical systems and even banking systems.
In this paper, we examine the problem of deciding termination. Since it is obviously undecidable in general, other work which has considered the problem, has predominantly dealt with either developing sufficient conditions on rule sets to ensure they are terminating (*approximate termination analysis* [5, 8]), or on designing languages which cannot express non terminating programs [9, 14, 19]. 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 rules. We hope that this information can then be used to help make informed choices in rule language design.

1.2 Contributions

Our principal contribution is the identification of a number of active rule languages for which termination analysis lies close to the boundary of (un)decidability. They are divided into two principal families: one literal languages where each update rule is permitted to have just one atom in its body, and unary languages where only unary relations may be updated, but higher arity relations may be accessed through views. We also provide some complexity results for the decidable cases. To our knowledge, this is the first paper to systematically study decidability of termination for active rules in a relational database. The closest work being that of [17], where decidability of termination in N steps for an 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* [3], or partial fixpoint logic program 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. Indeed, the termination problem is closely related to decidability of boundedness problem (*e.g.* [12, 11]); we will see that, as a consequence of the dynamic ability through triggers and the presence of deletion updates (which can be used to simulate negation), different kinds of syntactic conditions are needed to guarantee decidability. Our results and techniques are also related to the query containment problem; our undecidability result for the unary conjunctive$^{\neg}$ view language can be used to show that query containment is undecidable for nonrecursive programs with unary intensional predicates having at most three strata. This is an improvement over a result of [16]. Another result of ours complements a containment result of [13].

1.3 Paper Outline

In section 2, we present the preliminaries needed in the paper; we also highlight some subtleties in the definition of termination. In section 3, we study the decidability of termination for the *one literal* languages. We identify two particular members, *safe one literal* and *unrestricted one literal*, which lie either side of the boundary of decidability. In section 4, we study the decidability of the *unary* languages. We again identify fragments close to the boundary, making use of the notion of *conjunctive views*. Finally, we present a summary and outline some directions for future work.
Due to space limitations, full proofs of results are not included; some details are given in the appendix; a complete presentation is contained in [6].

2 Notation and Terminology

Our active rules will use only updates which are *insertions* or *deletions*, given by such expressions

$$\pm R(u) \leftarrow body$$

where (i) R is a relation, (ii) u a tuple of appropriate arity, (iii) if $+$ is present then each variable x in u must occur in the body, and (iv) if $-$ is present then at least one variable from u must occur in the body and u must not contain constants. The *body* is a conjunction of literals and each variable in the *body* occurs in at least one positive literal (range restriction). An empty body is assumed to equal *true*. The update will first find the set δ of all tuples u satisfying the body and then it will insert δ to R if $+$ is present and delete δ from R if $-$ is present.
We consider rules that have the "ECA" form

on event *if* condition *then* action

which satisfy the following requirements (a–c).

(a) Events are either of the form Insert(R_i) or Delete(R_i). We understand this to mean that the relation R_i has had a tuple inserted/deleted and the relation has actually changed. The semantics is a *set* oriented one: an update can cause at most one event, even though a set of tuples are inserted/deleted. A rule is said to be *triggered* when its event occurs.

(b) A condition is a query over the current database state. It returns *true* if the answer relation is non empty and *false* otherwise.

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

If a rule's updates are $\delta_0, \delta_1, \ldots, \delta_n$ and it is chosen for execution in a database state d_0 where its condition is *true*, then the database state after the execution of the rule is $\delta_n(\delta_{n-1}(\ldots\delta_0(d_0)\ldots))$. We assume rules are totally ordered by some priority scheme.
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 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.
In 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.
The semantics that we have defined, although fairly general, does not incorporate certain sophisticated features of active rules such as delta relations, coupling modes and complex pending structures. We have chosen to do so because our purpose is to study the behaviour of rule termination with specific reference to *rule syntax* and this is best done with a simple semantics (the effect upon termination of varying some of these other features is discussed in [7]). Interestingly, as a consequence of this choice, our results are also applicable to fragments of the *while* language, since rule execution is essentially equivalent to iterating over sets of updates (this follows from results in [15]).
We now formally define the property of termination for active rules.

Definition 1 • A set of rules is said to be *globally terminating*, if for any initial database state and triggering event, rule processing terminates. Otherwise, the set of rules is said to be *globally non terminating*.

- A set of rules is said to be *locally terminating on event* e, if for any initial database state and a triggering event e, rule processing terminates. Otherwise, the set of rules is said to be *locally non terminating on event* e.

■

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* program. 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.

Theorem 2 *If local termination is decidable then global termination is decidable.*

Proof: Analysis is conducted for every possible initial triggering event to see whether the rules are locally terminating. If it is locally terminating for each one, then the system is globally terminating. If not, then the system is globally non terminating. ■

Surprisingly, the converse of this theorem is false. 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 may be unnecessary.

Proposition 1 *There exists a rule language for which global termination is decidable and local termination undecidable.*

Proof: We define a rule language L where rules are in one of two classes, *regular* rules or *special* rules. Regular rule conditions are first order queries and actions are an update of the form $+R(u) \leftarrow c$ where c is first order. Note that no deletions are permitted. Special rules have a fixed format "on e if *true* then $trigger(e)$" and have priority higher than any regular rule. Clearly, once a special rule is triggered, rule processing will not terminate. Consequently, a rule set is globally non terminating iff it contains one or more special rules. On the other hand, suppose a regular rule of the form "on $del(R)$ if c then $+R(X) \leftarrow dom(X)$" and a special rule whose event is "on $ins(R)$" are defined. Then this rule set is locally non terminating from (external) event $del(R)$ iff c is satisfiable. Hence local termination is undecidable. ■

Previous work (such as [5]) has only considered global termination. In light of the above two results, we believe that local termination is a more suitable notion for studying decidability. Henceforth, when we refer to termination without any qualification, we will mean local termination.

3 One Literal Languages

In this section, we begin our investigation of decidability of termination. Our goal is to define a rule language for which termination is decidable, but where a slight increase in expressiveness results in undecidability. The rule language we examine is called the *one literal* language (to our knowledge it hasn't been discussed in the literature before). It is distinguished by the simplicity of its updates, where each update is permitted to have just one atom in its body. We look at two varieties, *safe one literal* and *unrestricted one literal* . The difference between the two lies in the type of range restriction that is placed upon variables. The safe variety stipulates that *all* variables in the head of a deletion update must appear in the body, whereas the unrestricted variety only requires at least *one* of the head variables to appear in the body. The two major findings in this section are that termination is decidable for safe one literal, but not for unrestricted one literal and this difference is caused by the latter's ability to express *semi-joins*.

3.1 Safe One Literal

Definition 3 In the safe one literal language, (a) conditions have the form

$$R(\overline{X}), Z_i \neq Z_j, \ldots, Z_m \neq Z_n$$

where each Z_p appears in $\overline{X}$, and (b) updates have the form

$$\pm R(\overline{X}) \leftarrow S(\overline{Y}), Z_i \neq Z_j, \ldots, Z_m \neq Z_n$$

where all variables appearing in $\overline{X}$ also appear in $\overline{Y}$ and each Z_p appears in $\overline{Y}$. ■

An example safe one literal rule is

On insert(R_1)
If $R_1(X,X,Z), X \neq Z$ then
$-R_2(X,Y) \leftarrow R_3(X,X,Y,Y,A,B,B), A \neq X;$
$-R_4(B) \leftarrow R_2(A,B);$
$+R_7(A,B) \leftarrow R_6(A,B,X,Z), X \neq Z, A \neq Z;$
$-R_5(X,Y) \leftarrow R_2(Y,X), X \neq Y;$
$+R_8(X,X,Y,Y) \leftarrow R_9(X,Y,Z);$

Theorem 4 *Termination is decidable for a set of rules written in the safe one literal language.*

Proof (Sketch): We first translate the given set of rules into an equivalent set of rules over a schema containing just one relation (say T).
We next define a finite number of equivalence classes of tuples. Each equivalence class corresponds to the answer to a relational calculus query over the initial database state. These equivalence classes form a partition of all possible tuples that could appear in the relation T; and at any instant

of time between updates during rule execution, the contents of T is the union of some of these equivalence classes.
The structure of these classes is rather complicated, with each class essentially corresponding to the composition of a sequence of one literal updates – we will give more details of these after the main ideas of the proof. A crucial point to note is that such updates are quite limited in their expressiveness, since they can only perform projections and difference (plus minor generalisations) and they can't do join. In fact, there is only a finite number of ways they can alter the database.
Having defined these equivalence classes, we can then prove that the behaviour of rule execution is completely determined by the equivalence classes which are present in the initial database state. We can then show that for each such (satisfiable) combination of equivalence classes, there is another (condensed) database state using a bounded number (ρ) of constants which corresponds to it. Consequently, to decide termination, we just need to execute the rules on all possible databases with $\leq \rho$ constants and check for loops.

More Details of the Equivalence Classes: Between now and the end of the proof we give more details on the equivalence classes. Suppose the arity of our relation T is m. Then our equivalence classes can be separated into m groups, each containing a set of m-tuples. Suppose the number of distinct values in an m-tuple is p. Then the first group ($p = m$) describes tuples with m distinct values, the second group ($p = m - 1$) describes tuples with $m - 1$ distinct values, the third describes tuples with $m - 2$ distinct values etc. The definitions of groups with fewer values are dependent on the definitions of groups with more values. We now briefly give some examples.

Group $\mathbf{p = m}$: The first group of equivalence classes consists of m-tuples where each component (attribute value) is unequal, *i.e.* m different values. Each class corresponds to some permutation patterns of the m tuple in the initial database, specified by the (non) existence of certain other facts involving *the same* m values in T.

Example: For $m = p = 3$ the following specifies an equivalence class:
$T(X_1, X_2, X_3) \wedge T(X_3, X_1, X_2) \wedge \neg T(X_2, X_3, X_1)$
$\wedge T(X_2, X_1, X_3) \wedge T(X_3, X_2, X_1) \wedge T(X_1, X_3, X_2)$
$\wedge X_1 \neq X_2 \wedge X_1 \neq X_3 \wedge X_2 \neq X_3$
These are tuples (X_1, X_2, X_3) such that X_1, X_2, X_3 are all distinct and the positive literals all occur in T and $T(X_2, X_3, X_1)$ does not occur in T.

Group $\mathbf{p = m - 1}$: This contains equivalence classes where two of the components of an m-tuple are equal, and the rest unequal (*i.e.* $m-1$ different values). Once again, our equivalence classes describe the (non) existence of permutations of tuples in the initial database. There is an extra complication, however, due to the fact that not all of our values are unequal. This means we must also consider the distribution of the tuple's supersets in the initial database. For instance, 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 $< 2, 1, 4 >$. We can identify patterns which contain just values in $\overline{X}$ as in the example above, and identify the patterns involving extra values through existentially quantified equivalence classes in higher groups containing supersets of $\overline{X}$.

Example: For $m = 3$ and $p = 2$, the following specifies an equivalence class:
$\neg T(X_1, X_1, X_2) \wedge \neg T(X_1, X_2, X_1) \wedge T(X_2, X_1, X_1)$
$\wedge \exists Y_1 (Y_1 \neq X_1 \wedge Y_1 \neq X_2 \wedge \neg T(X_1, X_2, Y_1) \wedge$
$T(Y_1, X_1, X_2) \wedge \neg T(X_2, Y_1, X_1) \wedge T(X_2, X_1, Y_1)$
$\wedge T(Y_1, X_2, X_1) \wedge T(X_1, Y_1, X_2)) \wedge \ldots$
Observe that we gave only one existentially quantified pattern (corresponding to one of the higher classes) and we omitted many more (all the other higher classes). This class consists of tuples (X_1, X_1, X_2) satisfying the above query.

Group $p = k$: This group consists of tuples which have k distinct values. There will be conjuncts testing membership in tuples which contain all k values, conjuncts testing membership in a superset with $k+1$ distinct values, conjuncts for tuples in a superset with $k+2$ distinct values, ..., conjuncts for tuples in a superset with m distinct values. ■

The complexity of the termination problem is quite expensive in terms of the size of the database schema. Observe that this is expression complexity, not data complexity.

Theorem 5 *Deciding local termination for rules in the safe one literal language is in NON-ELEMENTARY, but PSPACE-hard, in terms of* $k = (m + 2 + \lceil \log_2 n \rceil)$*, where* n *is the number of relations in the database and* m *is their maximum arity.*

Proof (Sketch) The lower bound is by a reduction from QBF. The upper bound follows from the number of constants needed to realise (materialise) different combinations of equivalence classes. ■

It is easy to see that all datalog programs in the safe one literal language (which are sets of insertion rules) are bounded, and thus it is decidable whether a datalog program in this language is bounded.

A More General Form: We can actually allow updates in the safe one literal language to have a slightly more general form, without changing the expressiveness (or endangering decidability). Instead of restricting the right hand side to only single atom, it can instead be a conjunction of atoms, *provided* each atom contains the same variables and the head of the update is a subset of these variables. *e.g.* the update

$$-R(X, Y) \leftarrow S(X, X, Y, Z), T(Y, X, Z), Q(Y, Y, X, X, X)$$

is not allowed, since the Q atom doesn't contain the Z variable. Whereas

$$-R(X, Y) \leftarrow S(X, X, Y, Z), T(Y, X, Z), Q(Y, Y, X, X, Z)$$

is permitted, since $\{X, Y, Z\}$ appear in every atom on the right hand side. This can be simulated as a sequence of safe one literal updates, which use some extra temporary relations.

3.2 Unrestricted One Literal

We now define an extension of the one literal language we have just examined. We call it *unrestricted one literal.* Deletion updates no longer require all variables of the head to appear in the body. This gives it the ability to express semijoins.

Definition 6 In the unrestricted one literal language, (a) conditions are expressions of the form

$$R(\overline{X}),$$

and (b) updates have one of the following forms

$$+R(\overline{X}) \leftarrow S(\overline{Y})$$
$$-R(\overline{U}) \leftarrow S(\overline{V})$$

where every variable that appears in $\overline{X}$ also appears in $\overline{Y}$, and $\overline{U} \cap \overline{V} \neq \emptyset$. ■

We will use the don't care symbol _ as used in logic programming, to mean a unique variable not occurring elsewhere. An example unrestricted one literal rule is

On ins(R)
If M(X,Y) Then
$-Q(X, \_) \leftarrow S(X, X, Y, Y, Z)$
$-U(B) \leftarrow R(A, B)$
$-T(X, Y) \leftarrow R(Y, X)$
$+Q(A, B) \leftarrow G(B, A, B, A, X, X, Z, Y, Y)$
$-G(X, \_, X, \_, Y, Y, \_, \_, \_) \leftarrow T(Y, X)$

The undecidability of the unrestricted one literal language is as result of its ability to evaluate semi-joins. For example, the statement $R(\overline{X}) \ltimes S(\overline{Y}) = \Pi_{\overline{X}}(R(\overline{X}) \bowtie S(\overline{Y}))$ can be simulated as unrestricted one literal updates as follows:[1]

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

where *tmp* and *result* are workspace relations having the same arity as R, and *result* contains the tuples in $R(\overline{X}) \ltimes S(\overline{Y})$ at the end of the computation.

Theorem 7 *Termination is undecidable for a set of rules written in the unrestricted one literal language.*

Proof (Sketch): The proof is by a reduction from the halting problem of two-counter machines (2CM's) starting with zero in the counters. Given any description of a 2CM and its computation, we can 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 unknown at the time the first rule is triggered, the rules first have to perform checking to see whether the relations contain a consistent description of the 2CM. The simulation is similar to one in [13, 10], but with some important differences regarding the interpretation of relations. In that work, they make use of a relation *succ* (the successor relation) to provide a total order on a group of constants. In our case, however, we cannot express inequality. Therefore *succ* has to represent a partial order on groups of constants. Some further details are in the appendix. ■

[1] We will use the notation $erase(R)$ as shorthand for the update $-R(u) \leftarrow R(u)$.

3.3 One Literal Invent and Two Literal

We have seen how allowing non safe deletions results in undecidability. Another way to extend one literal would be to allow two atoms on the right hand side, while still retaining safety. We can show that this new language is at least as powerful as the unrestricted one literal language and hence termination is undecidable. Consider the statement

$$-R(A, \_) \leftarrow S(A, C)$$

This can be rewritten in the two literal language as

$$erase(tmp);$$
$$+tmp(A, B) \leftarrow R(A, B), S(A, C)$$
$$-R(A, B) \leftarrow tmp(A, B)$$

In a similar vein, we can define yet another extension of safe one literal called *one literal invent.* Deletions remain the same, but insertions can now have don't care symbols in the head of the rule. *e.g.*

$$+R(X, \_) \leftarrow S(X, Y, Z)$$

where the semantics is that the don't care variable is matched with all constants in the active domain. We can show that termination is undecidable for this new language by showing that it can simulate the unrestricted one literal language. For example,

$$-P(X, \_) : -Q(X, Y, Z)$$

is equivalent to

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

The intuition being that this new operator allows us to perform a kind of cross product, which can then be utilised to express semi joins.

Theorem 8 *Termination is undecidable for a set of rules written in either the safe two literal rule language or the one literal invent language.*

An obvious question to ask, is whether safe one literal can have its expressiveness changed in some alternative way, without sacrificing decidability. The answer is yes, provided we are willing to accept a reduction in the arity of our relations. This forms the topic of our next section.

4 Unary Languages

Since termination is undecidable for even a simple rule language such as unrestricted one literal, in order to regain decidability, a natural restriction is to limit the arity of relations. To this end, we now examine a class of languages called *unary languages.* These are languages whose expressiveness is in some way intimately linked to the use of unary relations. The languages we examine are

1. *Arbitrary Updates and Unary Initialisation* (AUUI): Updates are arbitrary, but the only non empty relations in the initial database state must be unary ones.
2. *Unary Conjunctive View language*: Only unary relations may be updated, but higher arity relations may be accessed through unary conjunctive views.
3. *Unary Conjunctive$^\neg$ View Language*: Same as 2), but with negation allowed in view bodies.

4. *Binary Conjunctive View language*: Same as 2), but with binary conjunctive views.

Case 1 is decidable and cases 3 and 4 are undecidable. The decidability of case 2 is currently unknown, but we conjecture that it is decidable.
The updates used in rules can be thought of as operating on three different types of relations.

- *Views:* These are essentially derived predicates defined over a set of *hidden* relations.
- *Hidden Relations:* These are relations which may be used to define views, but cannot be used directly in rule updates or conditions.
- *Working Relations:* These relations may appear in the head or body of updates and so can be read and modified in the usual way.

Throughout rule execution, the contents of views and hidden relations do not alter and only the contents of the working relations may change. Consequently, to test for repeating states, we need only compare the states of the working relations.
Note that these restrictions only apply to updates inside rules. An external user is still able to issue arbitrary updates inside transactions and these may change the hidden relations.

4.1 AUUI

Suppose the only relations which are initially non empty are unary ones. Then even for rules which use complex updates with the expressiveness of Datalog$^{\neg\neg}$ [2], termination is decidable. The general form of an update is

$$\pm R(\overline{Y}) \leftarrow^{(\neg)} S_1(\overline{X_1}), \cdots,^{(\neg)} S_n(\overline{X_m})$$

and all relations in the database are working relations.

Theorem 9 *Termination is decidable for a rule system using Datalog$^{\neg\neg}$ updates provided that when the initial triggering event occurs, all non unary relations are empty.*

Proof (Sketch) We can again define a finite number of equivalence classes of tuples over relations. Let the unary relations be $U_1, U_2, \ldots, U_n$. Unary equivalence classes are then specified as answers to queries of the following form

$$^{(\neg)}U_1(X) \wedge^{(\neg)} U_2(X) \wedge \cdots \wedge^{(\neg)} U_n(X)$$

It is easy to construct any equivalence class instance using a single constant for each equivalence class; the constant is in U_i iff U_i occurs positively in the equivalence class. Termination is then decided by generating all possible databases with $\leq 2^n$ constants in the unary relations and checking for repetitions during rule execution. The proof of correctness for this approach is similar to that of the safe one literal case. At any given time, any relation of arity greater than one consists of the union of cross products of m unary equivalence classes. ■

Theorem 10 *Let n be the number of unary relations and let r be the number of non unary relations of maximum arity m. Deciding local termination for rules in the AUUI language is in EXPSPACE, but NP-hard, in terms $n \times m$.*

Proof (Sketch) The lower bound follows from a reduction to the 3-SAT problem. The upper bound follows since an exponential number of constants is needed to realise the equivalence classes and the amount of space needed is polynomial in the number of constants needed. ■

Remark: It is interesting to compare this complexity result with the situation where a 0-1 rule language is used (every relation in the database has arity zero). Deciding termination for such a language is also NP-hard, but is in PSPACE.

4.2 Unary Conjunctive Views

The condition that only unary relations can be non empty initially is rather restrictive. We now define a generalisation of it called the *unary conjunctive view* rule language. All working relations are unary and we allow unary arity views and arbitrary arity hidden relations. Updates can have the form

$$\pm U_1(X) \leftarrow^{(\neg)} U_2(X_1), \cdots,^{(\neg)} U_m(X_m),$$
$$^{(\neg)}V_1(Y_1), \cdots,^{(\neg)} V_n(Y_n)$$

where each U_i is a unary relation and each V_j is a unary view of an arbitrary conjunctive query. No U_i can appear in view definitions, which have the form:

$$V_i(X) \leftarrow R_1(\overline{X_1}), R_2(\overline{X_2}), \ldots, R_k(\overline{X_k})$$

Example: Two views are

$$V_1(X) \leftarrow R_1(X,Y), R_2(Y,Z,Z,X), R_4(X,X,Z)$$
$$V_2(X) \leftarrow R_1(X,X), R_3(Y,Y)$$

and an update is

$$-U_2(X) \leftarrow U_1(X),^{\neg} U_3(X), U_2(Y),^{\neg} V_1(X)$$

It should be pointed out that a view can be defined by two or more rules.

Open Question: Is termination decidable for rules in the unary conjunctive view language ?

We conjecture that the answer to this question is positive. Although it is relatively easy to construct equivalence classes for this rule language, the chief technical difficulty lies in determining whether certain combinations of class states are satisfiable.

4.3 Unary Conjunctive Views with Negation

Since the status of pure unary views is unknown, we explore an intermediate case between unary and binary. In this, we permit the incorporation (safe) negation inside a unary view. We call this the unary conjunctive$^{\neg}$ view language. The view form is

$$V_i(X) \leftarrow^{(\neg)} R_1(\overline{X_1}), \cdots,^{(\neg)} R_n(\overline{X_n})$$

An example is

$$V_1(X) \leftarrow R_1(Y,Z,Z,X),^{\neg} R_2(X,Y)$$

Theorem 11 *Termination is undecidable for a rule system in the unary conjunctive$^{\neg}$ view language.*

Proof (Sketch) To show that it is undecidable whether a rule system in the unary conjunctive$^\neg$ view language terminates, we again simulate the test of whether a two-counter machine (2CM) terminates when started with zero in its counters: We will define some unary and nullary views in the conjunctive$^\neg$ view language and a set of rules over these views so that the rules is nonterminating iff the 2CM halts. Much of the simulation is about ensuring that all the configurations are reachable through *succ*. This is made hard because we do not have recursion and we do not have binary views.

We will use some relations to encode the computation of 2CMs starting from with zero in the counters. These are:

- $S_0, \ldots, S_h$: each contains a constant which represents that particular state.
- *succ*: the successor relation.
- *config*: contains computation of the 2CM.
- *tc*: a transitive relation (but not necessarily the transitive closure of *succ*).
- *zero*: contains the first constant in *succ*. This constant is also used as the number zero.
- *eq*: this is the equality relation, $eq(A, B)$ has the same interpretation as $A = B$. Thus, $\neg eq(A, B)$ is interpreted as $A \neq B$.

The nullary and unary views are:

- *halt*: true if the machine halts.
- *bad*: true if the database doesn't correctly describe the computation of the 2CM.
- *dsucc*: contains all constants in *succ*.
- *dtc*: contains all constants in *tc*.
- *dT*: contains all time stamps in *config*.
- *dP*: contains all constants in *succ* with predecessors.

The "domain" views are easy to define. The other views are defined as follows:

- The relation *eq* needs to be reflexive, symmetric and transitive.

$$\begin{aligned} bad &\leftarrow succ(x, y), \neg eq(x, x) \\ bad &\leftarrow succ(y, x), \neg eq(x, x) \\ bad &\leftarrow eq(x, y), \neg eq(y, x) \\ bad &\leftarrow eq(x, y), eq(y, z), \neg eq(x, z) \end{aligned}$$

  The statement $\neg eq(X, Y)$ has the same meaning as $X \neq Y$. Each constant is represented by its equivalence class, so we need other rules to make sure that all relations are closed under these equivalence classes, *e.g.*

$$bad \leftarrow tc(x, y), eq(x, z), \neg tc(z, y)$$

- Check that *tc* is transitive and contains no cycle.

$$\begin{aligned} bad &\leftarrow tc(X, Y), tc(Y, Z), \neg tc(X, Z) \\ bad &\leftarrow tc(X, X) \end{aligned}$$

- Check that *succ* is contained in *tc*.

$$bad \leftarrow succ(X, Y), \neg tc(X, Y)$$

- Check that each constant in *succ* has at most one successor and at most one predecessor.

$$\begin{aligned} bad &\leftarrow succ(X, Y), succ(X, Z), Y \neq Z \\ bad &\leftarrow succ(Y, X), succ(Z, X), Y \neq Z \end{aligned}$$

- Check that zero has no predecessor.

$$bad \leftarrow zero(X), succ(A, X)$$

- Check that each S_i contains $\leq$ one element.

$$bad \leftarrow S_i(X), S_i(Y), X \neq Y$$

- Check that S_i, S_j are disjoint ($0 \leq i < j \leq h$):

$$bad \leftarrow S_i(X), S_j(X)$$

- Check that the timestamp is the key for *config*. There are three rules, one for the state and two for the two counters; the one for the state is:

$$\begin{aligned} bad \leftarrow\ & config(T, S, C_1, C_2), \\ & config(T, S', C_1', C_2'), S \neq S' \end{aligned}$$

- Check the configuration of the 2CM at time zero.

$$\begin{aligned} bad &\leftarrow zero(T), config(T, S, \_, \_), \neg S_0(S) \\ bad &\leftarrow zero(T), config(T, \_, C, \_), \neg zero(C) \\ bad &\leftarrow zero(T), config(T, \_, \_, C), \neg zero(C) \end{aligned}$$

- Check that the transitions of the 2CM are followed. For each transition $\delta(j, >, =) = (k, pop, push)$, include the following rule:

$$\begin{aligned} bad \leftarrow\ & config(T, S, C_1, C_2), \\ & succ(T, T'), S_j(S), S_k(S'), succ(X, C_1), \\ & zero(C_2), succ(C_1', C_1), succ(C_2, C_2') \\ & \neg config(T', S', C_1', C_2') \end{aligned}$$

  The second line in the rule checks the condition of the transition, whereas the third line sets the values for the action.
- Check that halting state is in *config*.

$$halt \leftarrow config(T, S, C_1, C_2), S_h(S)$$

Given these views, we design a set of rules which will be nonterminating iff $\psi = \neg bad \wedge hasPred \wedge sameDom \wedge halt \wedge goodzero \wedge nempty$, is true, where

- *hasPred* says "each nonzero constant in *succ* has a predecessor" and is given by

$$\neg \exists X (dsucc(X) \wedge \neg zero(X) \wedge \neg dP(X))$$

- *sameDom* says "the constants used in *succ* and the timestamps in *config* are the same set":
  $\neg \exists X, Y((dsucc(X) \wedge \neg dT(X))$
  $\wedge \neg(dT(Y) \wedge \neg dsucc(Y)))$
- *goodzero* says "the zero occurs in *succ*":

$$\neg \exists x(zero(x) \wedge \neg dsucc(x))$$

- *nempty* says: each of the domains and each of the unary base relations is not empty.

It is easy to see that, if ψ is satisfied by a database D, then *succ* contains no cycles.

Claim: Let $TCsucc$ be the transitive closure of *succ*. If ψ is satisfied in D, then for each pair of distinct constants a and b in *dsucc*, either (a, b) is in $TCsucc$ or (b, a) is. See appendix for proof.

Suppose ψ is satisfied in D. Because the unary base relations are all nonempty, the first constant exists. Because *goodzero* is true, the first constant occurs in *succ*. By Claim, the first constant reaches through the *succ* relation all constants in *dsucc*. Because *dsucc* and dT are equal, all times in $config$ are reachable from the first constant through the *succ* relation. So the halting state must be reachable if it is in the $config$ relation. ■

The simulation is done without recursion, without views whose arity > 1, and without using views in defining the views. To our knowledge, this is the first such simulation. All the simulations using unary intensional predicates we know of make use recursion. The above proof gives us this result:

Proposition 2 *The satisfiability of a query and the query containment problem are both undecidable for nonrecursive datalog¬ using unary intensional relations with three strata.*

This result is stronger than one of [16], which says that the satisfiability of a query is undecidable for nonrecursive datalog¬ with three strata; their construction used the negation of nonunary views.
In [13] is was shown that the containment of datalog¬ programs using only unary recursive intensional predicates is undecidable. We can use (the proof is given in the Appendix) the idea in our proof above to give the following:

Proposition 3 *It is undecidable, given safe nonrecursive datalog¬ programs* P_1, P_2, P_3 *and* Q_1, Q_2, Q_3 *which negate only base relations and which use only unary intensional predicates, whether* P_i *is contained in* Q_i *for all* i.

In comparison, it was shown [11] that boundedness of unary recursive datalog programs is undecidable using recursion through unary relations.

4.4 Binary Conjunctive Views

It is natural to extend the definition of unary views to binary ones. Now the left hand side of an update is a unary relation and the right hand side is a conjunction of unary relations and binary conjunctive views. No negation is permitted. We call this the *binary conjunctive view* rule language. *e.g.*

$$V_1(X,Y) \leftarrow R_1(X,Y), R_2(Y,Z,Z,X)$$
$$-U_2(X) \leftarrow U_1(X), U_3(X), V_1(X,Y), U_3(Y)$$

Theorem 12 *Termination is undecidable for a rule system written in the binary conjunctive view language.*

Proof (Sketch) The proof is quite similar to that of undecidability of the one literal unrestricted language, mainly because the only relations needed for communication in that simulation are binary. We can essentially regard *succ* as a binary view and use iteration to cycle through it and also define some binary views of the $config$ relation. ■

5 Summary and Further Work

The techniques used to prove decidability of termination in sections 3 and 4 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 [4], 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 4 can be seen as a generalisation of this, since for the safe one literal 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.

Safe One Literal	Decidable
Unrestricted One Literal	Undecidable
Safe Two Literal	Undecidable
One Literal Invent	Undecidable

Table 1: Summary of Decidability Results for One Literal Languages

AUUI	Decidable
Unary conjunctive views	Open
Unary conjunctive¬ views	Undecidable
Binary conjunctive views	Undecidable

Table 2: Decidability Results for Unary Languages

Table 1 summarises our results on the one literal language. It is perhaps not surprising that the safe one literal language is decidable, considering its limited expressiveness. Indeed its updates are like the fragment of relational algebra where the operations ⋈ and × are not allowed. This makes it rather like a natural form of update language for enforcing/checking *inclusion dependencies*. The undecidability of the unrestricted one literal language is quite surprising though. This language cannot perform a proper join of two relations, but it can mimic a *semi join* operation. This results in undecidability. The undecidability of the safe two literal language completes the picture by showing that an extra literal cannot be added, even for the safe case, without loss of decidability.
Table 2 summarises the decidability results presented for the unary rule languages. The key features which cause undecidability are either allowing views to have binary arity or allowing negation in view bodies. The first is essentially because a binary view provides a "sufficiently large" window onto the base relations, for the updates to check their structure by iterating through them. The second is because negation effectively makes it impossible to determine realisability of equivalence class states. Unary conjunctive views do not possess any of these features, but it is unknown what impact this has on decidability.
It is of interest to ask for what kinds of applications rules written in the AUUI language are useful. An important example is the checking of class hierarchies, where unary views can define relationships amongst classes. This itself is an instance of the more general problem of checking certain types of path expressions for graphs.
Unary type languages seem to share similarities with the area of description logics. Some cross fertilisation of results may be possible here and we intend to pursue this as a line of further research.

Finally, the results we have presented can also be related to other properties of interest for active rules:

- The techniques used in theorems 4 and 9 can be used to prove *confluence* is decidable for the safe one literal and AUUI rule languages (assuming the total order on rules is relaxed). This follows from their characterisation via equivalence classes.
- All our decidability and undecidability results apply to both termination and *satisfiability* for (the appropriate fragments of) *while* languages.
- 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 are applicable to it.

As part of future work, we intend to a) further investigate the open question of whether the unary conjunctive view language is decidable, b) try to narrow the complexity results stated in theorem 5, c) try to derive further (un)decidability results by examining new restrictions, such as limiting the number of rules or the number of updates per rule (rather like the use of *sirrups* in boundedness [1]), and d) try to characterise the computational power of the trigger languages we studied.

Acknowledgements

Guozhu Dong acknowledges support from the Australian Research Council through research grants. We thank Jianwen Su for pointing out the inequality simulation in the proof of theorem 11.

References

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

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

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

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

[5] 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.

[6] J. Bailey. *On the Foundations of Termination Analysis of Active Database Rules*. PhD thesis, Department of Computer Science, University of Melbourne, September 1997.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

[15] 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.

[16] R. van der Meyden. *The complexity of querying indefinite information: Defined relations, recursion and linear order*. PhD thesis, Rutgers, The State University of New Jersey, October 1992.

[17] 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.

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

[19] 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.

Appendix

Proof Details of Theorem 7

A two-counter machine is a deterministic finite state machine with two non-negative counters. The machine can test whether a particular counter is empty or non-empty. The transition function has the form

$\delta : S \times \{=,>\} \times \{=,>\} \rightarrow S \times \{pop, push\} \times \{pop, push\}$
e.g. The statement $\delta(4,=,>) = (2, push, pop)$ means that if we are 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.
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 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 in correspondence with the transition function. $succ(X, Y)$ is used to represent the successor relation and $R_0(X)$ contains a representation for the constant 0.
As stated earlier, the active rules need to check whether the description of the two counter machine is good. We provide a couple of examples of the rules for illustration.
Construction of $\mathbf{R_0}, \ldots, \mathbf{R_h}$: We wish to put each group of constants corresponding to one of the states $[0, h]$ in its own relation.
We begin with the R_0 relation
$+R_0(X) \leftarrow succ(X, Y)$
/* insert all candidate constants */
$-R_0(X) \leftarrow succ(Y, X)$
/* remove those with a predecessor */
We now construct the R_1 relation, *i.e.* find the successor of '0'.

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

Recall that the unrestricted one literal language can simulate semijoins.
R_1 now contains all the successors of '0' - *i.e.* '1'. We can similarly construct $R_2, R_3 \ldots, R_h$. If any of $R_1, \ldots, R_h$ is empty, then we make $bad = true$.
Cycle Check: We check that the $succ$ relation does not contain any cycles. This can be done using a while loop (which can easily be rewritten in terms of active rules triggering each other). The relation $sofar$ is used to record constants already examined.

$+sofar(X) \leftarrow R_0(X);$
$+current(Y) \leftarrow succ(X, Y) \ltimes R_0(X);$
while $(current \neq \emptyset)$
    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

Proof of Claim of Theorem 11

We assume that neither a nor b is the first constant; the case where one of them is the first constant is similar and easier. Because each element which is not the first has a predecessor, we can find the predecessor of a, and the predecessor of the predecessor, etc. Because $succ$ contains no cycles so this process will terminate at some point; the node where we terminate must be the first constant. Let the nodes found be $a_1, \ldots, a_n$ such that (a_i, a_{i+1}) is in $succ$ for each $i < n$ and a_1 is the first constant and $a_n = a$. Similarly we can find $b_1, \ldots, b_m$ such that (b_i, b_{i+1}) is in $succ$ for each $i < m$ and b_1 is the first constant and $b_m = b$. If b is some a_i or a is some b_j then we are done. Otherwise, the largest common ancestor (which exists since the first constant is a common ancestor) of a and b has two successors; this contradicts the condition that bad is false.

Proof (sketch) of Proposition 3

If bad is true then we flood the answer predicates of all P_i's and Q_i's with the same set of domain elements. We use the containment of P_1 by Q_1 to simulate $hasPred$ (when $halt \wedge nempty$ is true) as follows:

$$P_1 \quad p_1(x) \leftarrow dsucc(x), halt, dtc(\_), \ldots, S_h(\_)$$

$$Q_1 \quad \begin{array}{l} q_1(x) \leftarrow zero(x), halt, dtc(\_), \ldots, S_h(\_) \\ q_1(x) \leftarrow dp(x), halt, dtc(\_), \ldots, S_h(\_) \end{array}$$

Similarly we use "P_2 contained in Q_2" to simulate $sameDom$ and "P_3 contained in Q_3" to simulate $goodzero$. ■