On the Foundations of Termination Analysis of Active Database Rules

James Alexander Bailey

Department of Computer Science

University of Melbourne

Parkville 3052

Australia

Submitted in total fulfilment of the requirements of

the degree of Doctor of Philosophy.

December 1997

Abstract

Traditional database systems provide a mechanism for storing large amounts of data and an interface for querying and manipulating this data. They are, however, passive in the sense that their state can only change as a result of outside influences. An active database system enhances a traditional database system by providing, through active rules or triggers, the capability of automatically reacting to state changes without user intervention.

The behaviour of active rules in such a system, however, can be difficult to predict and there is consequently a need for developing techniques to automatically analyse properties of a rule set.

The main purpose of this thesis is to examine the property of termination, *i.e.* given a collection of active rules, is there a database state in which, once activated, the rules could trigger each other indefinitely. This problem is clearly of practical importance, since the possibility of infinite looping by rules in a database system should certainly be prevented. It is also very interesting and challenging theoretically, since the dynamics of triggers give rise to many intricacies.

A major theme of our study is the investigation of the impact that language and control features can have upon the decision problem of termination. In contrast, earlier work has concentrated on developing sufficient criteria for termination in systems with complex rule languages. Similar to previous important research problems such as boundedness of datalog, we aim to draw the exact boundary between the decidable and the undecidable cases. We do this by considering a number of parameters of rule systems which can affect the expressiveness and hence the complexity of termination analysis. These can be broadly grouped into two categories. The first, rule syntax, includes relation arity, update rule form and input domain constraints. The second, rule control structure, includes the pending rule mechanism and the meta rule language. Using these parameters, we identify rule system configurations which lie near the boundary. Our results help clarify the relationships between different rule components. We also give some complexity results for the decidable cases and show how our techniques for the termination problem can be adapted to analyse the property of reachability.

Finally, rather than giving up when presented with an unpromising situation, we propose a method which allows some definitely terminating cases to proceed; we do this by obtaining a partial answer to termination based on abstract interpretation.

In terms of precision, the method compares favourably with previous approaches to the problem. This is because it investigates the flow of data, rather than just the syntax of conditions and actions.

Preface

This thesis contains seven chapters and one appendix. The first chapter is an introduction and the seventh a summary. Some of the material in Chapter one appeared in the 1995 Australasian Database Conference [11]. Chapters two, three, and four contain material which has not appeared previously. Chapters five and six are considerably expanded treatments of material which appeared in Proceedings of the 1997 International Conference on Database Theory [8, 9].

This is to certify that

1. *the thesis comprises only my original work,*

2. *due acknowledgement has been made in the text to all other material used,*

3. *the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies, appendices and footnotes*

Acknowledgements

I am greatly indebted to my supervisors, Kotagiri Ramamohanarao and Guozhu Dong. They have assisted me greatly throughout my studies and their enthusiasm and guidance helped make this thesis a reality.

I would like to extend my thanks to others with whom I have collaborated - to Lobel Crnogorac, with whom I have been fortunate to share an office and also to Harald Søndergaard and David Kemp. I am also grateful to my parents, who gave me the opportunity to pursue tertiary study.

This work was supported by funding from the Co-operative Research Centre for Intelligent Decision Systems and the Key Centre for Knowledge Based Systems.

Contents

Chapter 1

Introduction and Preliminaries

In this chapter, we introduce the field of active databases and the problem of termination. We then provide some notation and terminology and outline the structure of the rest of the thesis.

1.1 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 ([25, 32, 48, 53]). Work on the foundations of active database systems and their analysis, however, is still somewhat scarce. As a contribution in this direction, in this thesis we present some new foundational work.

The potential benefits of active databases are numerous. In contrast to distributed application programs which poll the database at defined intervals, the use of active rules centralises reactive functionality, ensuring timeliness of response to critical changes and enhancing modularity of code. Furthermore, it creates new opportunities for automatic analysis and optimisation. Active database applications range from simple alerters and integrity constraints ([18, 19]), to systems that perform complex reasoning and updates.

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 (*coupling modes*).

Major areas of research in active database systems include

- Rule specification.
- Rule execution models.
- System architectures.
- Optimisation of rule execution.
- Rule analysis and formal foundations.
- 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 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 and this is called *non termination.* Clearly, these 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 adopted by the commercial database system Oracle ([43]), 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. Also, while it may prevent non termination, it is difficult to use as an aid for identifying the underlying causes of anomalous behaviour.

In this thesis, we examine static methods for analysing termination. Other work which has considered this question, has predominantly dealt with either developing sufficient conditions on rule sets to ensure they are terminating (*approximate termination analysis* [7, 12, 15, 37, 49]), or on designing languages which cannot express non terminating programs ([22, 41, 55]). There has not to our knowledge, however, been a study of the general problem of decidability of the termination property for active rules in a relational database, the closest work being that of [49], where decidability of termination in N steps for an object oriented language is examined. This gap is somewhat surprising. Since it is often stated that termination is "undecidable in general", it is perhaps assumed that decidable languages are unlikely to be useful. 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.

An additional benefit, is that using some of the insight gained in our study of decidability, we are able to outline a promising new method for approximate termination analysis, based on abstract interpretation.

A Broader Picture

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* ([4]), 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. This forms a natural adjunct to research which has been conducted on optimisation and analysis of *inflationary* logic programs, where questions such as decidability of boundedness (*e.g.* [33, 30]) have been popular. The whole area of database dynamics is currently becoming increasingly important, especially with the advent of so-called agent systems, which maintain database like *beliefs* and use active rule like *plans*, in order to achieve goal directed behaviour in a changing world ([10]).

1.2 Notation and Terminology

We now present some necessary terminology for the rest of the thesis. We assume familiarity with relational databases and some knowledge of active databases. For further background see [2] and [54].
Formally, we assume the existence of three disjoint infinite sets: a set **rel** of *predicate* or *relation names*, each of which has an associated *arity* $n \geq 0$, a (universal) domain **dom** of *constants* and a set of variables **var**.

Let $\mathbb{N}$ denote the set of natural numbers. For each $n \in \mathbb{N}$, an *n-ary tuple* is a mapping from $\{i \mid 1 \leq i \leq n\}$ to **dom**, and an *n-ary relation* is a finite set of n-ary tuples. Observe that there is exactly one 0-ary tuple, which is denoted as $[\,]$. The *cardinality* of a relation r, denoted by $|r|$, is the number of tuples in r. 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* (also referred to as a *database state*) of a database schema $\mathcal{S}$ is a mapping I such that, for each relation name R in $\mathcal{S}$, if R has arity n, then $I(R)$ is an n-ary relation; the *cardinality* of I, denoted by $|I|$, is defined as the total number of tuples in I, *i.e.* $|I| = \sum_{R \in \mathcal{S}} |I(R)|$. We also denote by $inst(\mathcal{S})$ the set of all database instances of $\mathcal{S}$.

Let R be a k-ary relation ($k \in \mathbb{N}$). We define the *active domain* of R, denoted by $adom(R)$, to be the set of constants occurring in R, *i.e.* $adom(R) = \bigcup_{1 \leq i \leq k} \pi_i(R)$, where $\pi_i(R)$ ($1 \leq i \leq k$) is the projection of R onto the i-th column. For each database instance I of a schema $\mathcal{S}$, we define the *active domain* of I as $adom(I) = \bigcup_{R \in \mathcal{S}} adom(I(R))$. Finally, for each finite set $J = \{R_1, ..., R_k\}$ of relations, we define the *active domain* of J as $adom(J) = \bigcup_{1 \leq i \leq k} adom(R_i)$.

An *atom* is a formula of the form $R(X_1, \ldots, X_k)$ where R is a relation name and $X_1, \ldots, X_k$ are variables. A *literal* is a formula of the form $R(\ldots)$ or $\neg R(\ldots)$. A *variable pattern* is a sequence of variables, *e.g.* $\overline{X} = (A, B, B, C)$.

An update δ over a database schema $\mathcal{S}$ is a mapping from $inst(\mathcal{S})$ to $inst(\mathcal{S})$ which is computable and generic (insensitive to renaming of constants) and so for each permutation ρ of **dom**, $\rho(\delta(I)) = \delta(\rho(I))$.

Let $k \in \mathbb{N}$, $\mathcal{S}_{\text{in}}$ be a database schema, and R_Q (the answer relation) be a k-ary relation name not in $\mathcal{S}_{\text{in}}$. A *query from* $\mathcal{S}_{\text{in}}$ *to* R_Q is a computable and generic mapping Q from $inst(\mathcal{S}_{\text{in}})$ to relation instances of R_Q.

We will adhere to the following naming conventions throughout.

- Tuples are represented by the letters t, u (possibly subscripted).
- Relation names are represented by the letters Q, R, S, T (possibly subscripted).
- Constants are represented by the letters a, b, c, k, p, q (possibly subscripted or quoted).
- Variables are represented by the letters A, B, C, X, Y, Z (possibly subscripted).
- Variable patterns are represented by $\overline{X}, \overline{Y}, \overline{Z}$ (possibly subscripted).
- Events are represented by $e, e_1, e_2, \ldots$
- Rules are represented by $r, r_1, r_2, \ldots$
- Updates are represented by $\delta, \delta_1, \delta_2, \ldots$
- Schemas are represented by $\mathcal{S}, \mathcal{S}_1, \mathcal{S}_2, \ldots$
- Database instances/states are represented by I, d (possibly subscripted or superscripted)
- Sets of rules are represented by $\mathcal{R}, \mathcal{R}_1, \mathcal{R}_2, \ldots$
- $\mathcal{P}$ represents the powerset operator.

1.2.1 Update Languages

Our active rules will be using a limited form of updates. In fact, we only consider updates which are *insertions* or *deletions*. An update is an expression of the form

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

where R is a relation, u a tuple of the appropriate arity to be inserted or deleted into R and each variable x in u occurs in the body for insertions and at least one variable from x must occur in u for deletions on constant free u. The *body* is a conjunction of literals and each variable in the *body* occurs in at least one positive literal (referred to in the literature as range restriction). An empty body can be assumed to equal *true*. The semantics of evaluating the body of a rule is that of nonrecursive semi-positive datalog with negation ([2]). The values satisfied by the rule produce either a set of inserted tuples Δ_R or a set of deleted tuples ∇_R, depending on whether $+$ or $-$ is present.

Similar to [4], the notation $erase(R)$ represents the update $-R(u) \leftarrow R(u)$

1.3 Active Rules

1.3.1 Syntax

As mentioned earlier, we consider rules that have the "ECA" form.

on event *if* condition *then* action

- 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(*i.e.* the relation has actually changed). The semantics is a *set* oriented one, which means that an update can cause at most one event for each set of inserted/deleted tuples. A rule is said to be *triggered* when its event occurs.
- A condition is a query over the current database state. It returns *true* if the answer relation is non empty and *false* otherwise.
- An action is a finite sequence of updates.

Suppose a rule's update actions are $\delta_0, \delta_1, \ldots, \delta_n$ and it is chosen for execution in a database state d_0. Then the database state after the execution of the rule (providing its condition is *true*) is $d_{final} = \delta_n(\delta_{n-1}(\ldots \delta_0(d_0) \ldots))$.

We will sometimes omit the condition from rules. In such cases it can be assumed to be " if *true*".

We assume there exists a total ordering of rules for a given program, *i.e.* there is an injective and surjective function *priority* which maps a set of rules $\{r_1, \ldots, r_n\}$ to priorities $\{1, \ldots, n\}$.

One particular rule syntax we will be using is the *0-1 rule language*, which only operates on variables (rather like boolean relations). It was used in an example in [12]. Note that the form of events is slightly different from our generic definition.

Definition 1 0-1 Language

- Events are of the form U(X) which we understand to mean "the variable X has had a value assigned". We don't require that the new value be different from the old, but if we did, it wouldn't make any essential difference to the language
- A condition is a conjunction of simple conditions. A simple condition is a test of the form *Var=0* or *Var=1*
- An action is a sequence of simple actions. A simple action is an update of the form Var=y $\leftarrow$ condition, where $y \in \{0, 1\}$

■

Thus a typical 0-1 rule might be

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

1.3.2 Generic Execution Model

Suppose an event occurs in a transaction that triggers a rule (the semantics of this is not relevant to our discussion). This then initiates *rule processing* from the the current state, termed the *initial* database state, and the following steps are then performed.

1. If there are no triggered rules then exit rule processing and resume the transaction.
2. Select 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 (which become triggered only after all the parts of the action have been completed). These are added to the pending structure. Thus the steps 1-2-3-4 can loop forever.

The *pending rule structure* is used to store rules which are awaiting execution. In chapters 1-4 we will be using a *singleton* pending structure, which requires that there can only ever be one rule awaiting execution. If two or more rules are triggered simultaneously, then the one with highest priority is added to the structure and the other(s) discarded. Some generalisations of this will be defined in chapters 5 and 6.

Iteration

We will often be using rules to simulate the behaviour of 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 [44, 45].

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)

Rule $\mathbf{r_1}$	**Rule $\mathbf{r_2}$**	**Rule $\mathbf{r_3}$**	**Rule $\mathbf{r_4}$**
On e_1	On e_2	On e_3	On 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 and $priority(r_3) > priority(r_4)$. The notation $trigger(e)$ represents an update which causes the event e (*e.g.* an insertion/deletion pair such as $+T(a); -T(a)$, where T is the relation on which e is defined).

1.3.3 Termination

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

Definition 2 Local and Global Termination

- 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.*

■

Previous work (such as [7]) has only considered global termination. For reasons which we will explain in chapter 4, local termination is a more suitable notion for studying decidability. Henceforth, when we refer to termination without any qualification, we will mean local termination.

1.3.4 Some Simple Decidable/Undecidable Examples

Since much of this thesis contains proofs of (un)decidability of rule termination, we present some introductory examples to give the flavour of our techniques.

Decidability

If we examine the 0-1 language presented in definition 1, it is not too difficult to see that it is decidable. Firstly, we can enumerate all possible database states (which correspond to all the different combinations of variable values). Next, for each candidate state, we input it as the *initial* database state to the set of rules. Rule processing will either eventually terminate, in which case we will select another state as the initial state, or it will continue infinitely. It is possible to recognise this latter case by maintaining a history of all possible database states that have occurred at different program points (a program point corresponds to whichever rule is in the pending set) and if we detect a repetition at the same program point, then we must have entered an infinite loop. This technique is used by [13] in a dynamic context. Clearly, if rule processing terminates on all candidate initial database states, then the rule set is *terminating*, otherwise it is *non terminating.*
Now, suppose we alter the 0-1 language to use variable assignments instead of constant assignments and use an infinite domain instead of a finite one. The domain of each variable ranges over the integers and no constants are used inside rules. An example rule is

On U(A)
If $C = B \ \wedge D = E \ \wedge \ T \neq D$
Then $T = C \ ; \ B = D$

This language is also decidable, but the proof is not quite as straightforward. We would like to enumerate all possible database states, but there are an infinite number of them (since the domain is infinite). The solution is to collapse this infinite domain into a finite one. The key point to note is that any database state is completely characterised by the equality/inequality relationships among the variables, an example being

$$\begin{array}{l} A = B, A \neq C, A \neq D, \ldots, A = Y \\ B = A, B = C, B \neq D, \ldots, B \neq Y \\ \vdots \\ Y = A, Y \neq B, \ldots \end{array}$$

Obviously, there is a finite number of these sorts of state descriptions. Thus, we can enumerate all of them and use the same technique as for the 0-1 case, inputting each to the rule set and checking for repeating states.

This technique of domain compression is the basic idea that we will be using for proving decidability of more sophisticated languages. Further complications will come when we consider relations instead of variables.

Undecidability

Our proofs of undecidability will be based on the method of reduction. For example, it is known that satisfiability of first order queries is undecidable ([26]). This fact can be used to prove a result about rules whose conditions are first order. *e.g.* Consider the rule "on e if c then trigger(e)". This rule is non terminating iff c is satisfiable and consequently termination is undecidable.

Guaranteed Termination

Suppose we use a rule language which permits either only insertions or only deletions, then rule processing will always terminate, since execution will eventually reach a fixpoint where no event can be generated to trigger more rules.

1.4 Thesis Outline

In chapter 2, we study the decidability of a class of languages called 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 chapter 3, we study the decidability of a family of languages called *unary type* languages. We again identify fragments close to the boundary, making use of the notion of conjunctive views. In chapter 4, we tie together some of the results in chapters 2 and 3. We consider the complexity of our decision procedures and generalise the decidable cases under the heading of *bounded model* languages. We also identify some subtleties in the definition of termination and show that our results can be adapted to analyse the property of reachability in certain cases. In chapter 5, we study the impact that execution and control features can have upon termination decidability. In particular, we look at the influence of the *pending rule structure* and the *meta rule language*. In Chapter 6, we shift focus from decidability to approximate termination analysis and propose a new method based on abstract interpretation. Its novelty lies in its investigation of the flow of data, rather than just checking the syntax of conditions and actions. In chapter 7, we present a summary and outline some directions for future work.

Chapter 2

Decidability of One Literal Languages

2.1 Introduction

In this chapter, 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 chapter 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. The first half of this chapter is devoted to proving the former result and the second half devoted to proving the latter result. We then complete the picture by showing how an extension of safe one literal to two literals also results in undecidability. We end with a summary and some conclusions.

2.2 Safe One Literal Language

We begin with a definition of the safe one literal language.

Definition 3 Safe One Literal Language

- Conditions have the form
  $R(\overline{X}), Z_i \neq Z_j, \ldots, Z_m \neq Z_n$ where each Z_p appears in $\overline{X}$.
- 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
$-R_2(X, Y) \leftarrow R_3(X, X, Y, Y, A, B, B), A \neq X;$
$-R_4(B) \leftarrow R_2(A, B);$
$+R_7(A, B) \leftarrow R_6(B, A, B, A, X, X, Z, Y, Y), X \neq Z, A \neq Z;$
$-R_5(X, Y) \leftarrow R_2(Y, X), X \neq Y;$
$+R_8(X, X, Y, Y) \leftarrow R_9(X, Y, Z);$

Recall that 1) Variables appearing in a condition/action are local to that condition/action and 2) Updates are performed sequentially.

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

The rest of this section will be devoted to proving this theorem. The following steps will be used.

1. Translate the given rules into an equivalent set of rules over a schema just containing one relation.
2. Define a (finite) set of equivalence classes of tuples on these rules.
3. Show that every possible database state can be described in terms of these equivalence classes and that for each of them there is a corresponding database state with a bounded number of constants, which will behave the same way during rule execution .

2.2.1 Rule Translation

To simplify our arguments, we restrict ourselves to a schema with just one relation. The following proposition states that this does not lead to any loss of generality.

Proposition 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. We first show how to 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. We then show how to construct $\mathcal{R}_2$ and $\mathcal{S}_2$ from $\mathcal{R}'_1$ and $\mathcal{S}'_1$.

a) For each relation R in $\mathcal{S}_1$ whose arity is less than m, we create another relation R' of arity m. Wherever a rule references R, we replace it by R' and duplicate the variable in the last column an appropriate number of times. *e.g.* For $m = 5$, $R(X, Y, Z)$ becomes $R'(X, Y, Z, Z, Z)$. If $\mathcal{R}_1$ is non terminating on an instance I_1, then we could clearly construct I_2 for $\mathcal{R}'_1$ by padding each tuple from I_1. Conversely, if $\mathcal{R}'_1$ is non terminating for an instance I_2, then we can construct I_1 for $\mathcal{R}_1$ by restricting it to the tuples in I_2 which are padded correctly, since any tuples which are not padded correctly will be unable to participate in an update.

b) Let the relation in $\mathcal{S}_2$ be $T(X_1, X_2, \ldots, X_k)$, where $k = (m + 2 + \lceil \log_2 n \rceil)$. Conceptually, $\pi_{X_1,\ldots,X_m}(T)$ is used to store all the data contained in $R_1, R_2, \ldots, R_n$. $X_{m+1}, \ldots, X_k$ contains *control information.* X_{m+1} and X_{m+2} are used to store two unequal constants, say 0 and 1. The final $\lceil \log_2 n \rceil$ positions are used to specify a number between 1 and n in binary. Each rule update in $\mathcal{R}'_1$ is mapped to a new update in $\mathcal{R}_2$. This is best illustrated by an example. Suppose $n = 4$ and $m = 2$. The update

$$-R_1(X, X, Y, Y, Y) \leftarrow R_2(X, Z, Z, Y, Y)$$

is mapped to

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

A variable pattern of the form (A, B) is used to identify R_1 and a pattern of the form (A, A) identifies R_2. If $\mathcal{R}'_1$ is non terminating on an instance I'_1, then we can construct I_2 for $\mathcal{R}_2$ by inserting all the tuples into T from $R_1, \ldots, R_n$, supplemented with the appropriate control information. Conversely, if $\mathcal{R}_2$ is non terminating for an instance I_2 over T, then we can construct I'_1 for $\mathcal{R}_1$ by spreading the tuples with consistent control information over $R_1, \ldots, R_n$. Any tuples with inconsistent control information will be unable to participate in an update and can hence be ignored. Note that although we are using the final $\lceil \log_2 n \rceil$ bits to represent a number in binary, different tuples may use different constants for this representation. All that matters is that the two constants in positions $m+1$ and $m+2$ are different. For example the tuples $(\ldots, 5, 6, 5, 5, 6, 5)$ and $(\ldots, 7, 5, 7, 7, 5, 7)$ in T should both be mapped to the same relation in $\mathcal{S}_1$.

A technical point to note is that we only described how to alter conditions and updates, but not events. Since only one relation is being used, we need a slight change in syntax. Instead of having the event "on $ins(R)$", we define the event "on $ins(T^c(\overline{X}))$", where $\overline{X}$ is a variable pattern and c is a conjunction of inequalities on $\overline{X}$. The semantics for this is straightforward, since our translation scheme will ensure that only one pattern can ever apply. ■

Throughout this section we will assume, unless stated otherwise, that T is the only relation in the database and has arity m.

2.2.2 Equivalence Class Definition

The purpose of this subsection is to describe the equivalence classes of tuples.

Roughly speaking, given a set of rules, we will construct 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, the sets $C_1(I_0), \ldots, C_N(I_0)$ form a partition of all possible tuples that could appear in T, due to a sequence of updates on I_0. Furthermore, at any instant of time between updates during rule execution, the contents of T is the union of some of these $C_i(I_0)$.

We will call each of these $C_i(I_0)$ an equivalence class and C_i its description. The following properties of these equivalence classes will be relied upon and their correctness later verified:

- Each possible tuple that could be constructed during rule execution belongs to exactly one equivalence class.
- Tuples that are in the same equivalence class cannot become separated during rule execution.
- The number of equivalence classes is bounded by a constant depending only on the schema and it is possible to produce an example database instance for each satisfiable set of equivalence classes.

Equivalence relations on tuples have also been considered in other contexts ([5]). One example is that of automorphism classes of tuples. Here, 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 it is not usable for static analysis. Another example is equivalence of tuples relative to FO^k (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 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. The explanation for this will depend in a critical way on the "safety" of one literal rules.

We will shortly show how the class descriptions are defined. But first we present some intuition relating to their construction.

Intuition behind Equivalence Classes

Since we need to ensure that tuples within an equivalence class cannot be separated, our class structure reflects 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 are performed sequentially

$$\begin{array}{ll} \delta_0 : & -T(A,B,B) \leftarrow T(A,B,C), B \neq C \\ \delta_1 : & -T(A,A,A) \leftarrow T(A,B,B), A \neq B \end{array}$$

T now contains $\{(1,1,1),(8,8,8),(2,4,4),(1,3,5),(8,9,10),(2,6,7)\}$. Thus $(1,1,1)$ and $(2,2,2)$ should not be in the same equivalence class. Observe that $(1,1,1)$ and $(8,8,8)$ *are* in the same equivalence class, since the constants 1 and 8 are distributed in a similar way throughout T and thus the two tuples will always "travel together" .

For this T, the equivalence class that $(1,1,1)$ and $(8,8,8)$ are in can be described by the relational calculus query (which doesn't depend on δ_0 or δ_1)

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

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

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

The actual class descriptions are more involved than these two queries, although they are equivalent to these two for the example database. We will also show that there is a limit to how complicated the descriptions may become. This is essentially determined by the number of ways updates can "chase" components of the m-tuple around the relation T. Such chasing is restricted in complexity, since an update can only really 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. These are the key intuitions behind our construction below.

Definitions of the Class Descriptions

We first give some definitions to help in constructing the class descriptions.

The updates in rules use certain variables. This pool of variables is represented by $V_r = \{X_1, \ldots, X_m\}$.

Let $V_p =< X_{i_1}, X_{i_2}, \ldots, X_{i_m} >$, where each $X_{i_k} \in V_r$, be some variable pattern constructed using (possibly repeated) elements from V_r.

Let $S(V_p) = \{T(\overline{u}) \,|\overline{u}$ is a permutation of $V_p\}$. Note that $|S(V_p)| \leq m!$.
This set contains all positive atoms that may be used by updates for the V_p pattern (without projections).

Let $s_1^{V_p}, \ldots, s_{n_1}^{V_p}$ be an enumeration of the elements of $\mathcal{P}(S(V_p))$. We have $n_1 \leq 2^{m!}$.

We now define some formulas which are conjunctions of literals and will be used as building blocks for equivalence classes.

For each $1 \leq i \leq n_1$, let $E_i^{V_p} = (\bigwedge_{a \in s_i^{V_p}} a) \,\wedge\, (\bigwedge_{a \in S(V_p) - s_i^{V_p}} \neg a)$

Example: For $m = 2$ and $V_p =< X_1, X_2, X_3 >$, we have 64 such $E_i^{V_p}$.
$E_1^{(X_1,X_2,X_3)} = T(X_1,X_2,X_3) \wedge T(X_3,X_1,X_2) \wedge T(X_2,X_3,X_1)$
$\wedge\, T(X_2,X_1,X_3) \wedge T(X_3,X_2,X_1) \wedge T(X_1,X_3,X_2)$
$E_2^{(X_1,X_2,X_3)} = \neg T(X_1,X_2,X_3) \wedge T(X_3,X_1,X_2) \wedge T(X_2,X_3,X_1)$
$\wedge\, T(X_2,X_1,X_3) \wedge T(X_3,X_2,X_1) \wedge T(X_1,X_3,X_2)$
$E_3^{(X_1,X_2,X_3)} = T(X_1,X_2,X_3) \wedge \neg T(X_3,X_1,X_2) \wedge T(X_2,X_3,X_1)$
$\wedge\, T(X_2,X_1,X_3) \wedge T(X_3,X_2,X_1) \wedge T(X_1,X_3,X_2)$
$E_4^{(X_1,X_2,X_3)} = T(X_1,X_2,X_3) \wedge T(X_3,X_1,X_2) \wedge \neg T(X_2,X_3,X_1)$
$\wedge\, T(X_2,X_1,X_3) \wedge T(X_3,X_2,X_1) \wedge T(X_1,X_3,X_2)$
$\vdots$
$E_{64}^{(X_1,X_2,X_3)} = \neg T(X_1,X_2,X_3) \wedge \neg T(X_3,X_1,X_2) \wedge \neg T(X_2,X_3,X_1)$
$\wedge\, \neg T(X_2,X_1,X_3) \wedge \neg T(X_3,X_2,X_1) \wedge \neg T(X_1,X_3,X_2)$

For $m = 2$ and $V_p =< X_1, X_1, X_2 >$, we have 8 $E_i^{V_p}$.
$E_1^{(X_1,X_1,X_2)} = T(X_1,X_1,X_2) \wedge T(X_1,X_2,X_1), T(X_2,X_1,X_1)$
$E_2^{(X_1,X_1,X_2)} = \neg T(X_1,X_1,X_2) \wedge T(X_1,X_2,X_1), T(X_2,X_1,X_1)$
$E_3^{(X_1,X_1,X_2)} = T(X_1,X_1,X_2) \wedge \neg T(X_1,X_2,X_1), T(X_2,X_1,X_1)$
$E_4^{(X_1,X_1,X_2)} = T(X_1,X_1,X_2) \wedge T(X_1,X_2,X_1), \neg T(X_2,X_1,X_1)$
$E_5^{(X_1,X_1,X_2)} = \neg T(X_1,X_1,X_2) \wedge \neg T(X_1,X_2,X_1), T(X_2,X_1,X_1)$
$E_6^{(X_1,X_1,X_2)} = T(X_1,X_1,X_2) \wedge \neg T(X_1,X_2,X_1), \neg T(X_2,X_1,X_1)$
$E_7^{(X_1,X_1,X_2)} = \neg T(X_1,X_1,X_2) \wedge T(X_1,X_2,X_1), \neg T(X_2,X_1,X_1)$

$E_8^{(X_1,X_1,X_2)} = \neg T(X_1, X_1, X_2) \wedge \neg T(X_1, X_2, X_1), \neg T(X_2, X_1, X_1)$

Let the isomorphism relation on variable patterns be $\simeq$, where $V^1 \simeq V^2$ if there exists a permutation ρ of V_r such that $\rho(V^1) = V^2$.

Let $\mathcal{V}$ be a maximal set of non isomorphic variable patterns constructed from V_r. *e.g.* For $m = 3$ and $V_r = \{X_1, X_2, X_3\}$, one $\mathcal{V}$ can be

$$\mathcal{V} = \{(X_1, X_2, X_3), (X_1, X_1, X_2), (X_2, X_1, X_1), (X_1, X_2, X_1), (X_1, X_1, X_1)\}$$

Note that (X_1, X_2, X_2) is isomorphic to (X_2, X_1, X_1).

We will need the function *replace* :

$\{\text{Integer}\} \times \{\text{Var. Pattern}\} \to \mathcal{P}(\{\text{Var. Pattern}\})$

for modifying variable patterns. $replace(c, \overline{X})$ replaces c repeated variables in $\overline{X}$ with new variables $Y_1, \ldots, Y_c$ and returns a set of modified variable patterns, each containing at least one instance of each of the original variables.

Example: Consider $replace(2, (A, A, B, B, C, D, A))$. The variable A has three occurrences and the variable B two occurrences. Because the first argument of the function is 2, we create two new variables Y_1 and Y_2. A new variable pattern is then produced by replacing an A or a B by Y_1 or Y_2 and then assembling all the variables in some order. There are obviously several ways this can be done.

$(Y_1, Y_2, B, B, C, D, A)$
$(A, D, C, B, B, Y_2, Y_1)$
$(Y_1, A, B, B, C, D, Y_2)$
$(Y_1, D, B, B, C, A, Y_2)$
$(Y_1, A, Y_2, B, C, D, A)$
$(Y_1, A, B, Y_2, C, D, A)$
$(A, Y_1, B, Y_2, C, D, A)$
$\vdots$

Note that each new variable pattern *must* contain at least one instance of each of A,B,C and D. We will use the notation $replace^i(\ldots)$ to denote the i'th value (determined according to some fixed scheme) of *replace*. Clearly, for each $\overline{X}$, we can enumerate these as $replace^1$,$replace^2$,...,$replace^{n_2^{\overline{X}}}$, where $n_2^{\overline{X}}$ is bounded by the number of ways we can choose the variables for replacement. Observe that $n_2^{\overline{X}} \leq \begin{pmatrix} m \\ c \end{pmatrix} \times m!$. Let n_2 be the maximum such $n_2^{\overline{X}}$. ∎

Equivalence Class Construction

Figure 2.1 presents an algorithm for generating equivalence classes. This algorithm also serves as a definition of their structure. The base case generates classes with m distinct variables and then the inductive step is used to generate classes with $m-1$ distinct variables, $m-2$ distinct variables etc. Thus we can partition the classes that are output into m groups, each corresponding to a particular choice of p (the number of distinct variables). The definitions of groups with lower p values are dependent on the definitions of groups with higher p values. The formula for an equivalence class has the form $C_i^{\overline{X}}$ and the tuples in the class are produced by evaluating this formula on the initial database state *i.e.* $C_i^{\overline{X}}(I_0)$. We now briefly discuss some of the cases.

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.

Example: For $m = 3$ and for some i
$C_i^{X_1,X_2,X_3} = (E_i^{(X_1,\ldots,X_m)}$ and no two X_i's are equal)
$= (X_1 \neq X_2 \wedge X_1 \neq X_3 \wedge X_2 \neq X_3 \wedge$
$T(X_1,X_2,X_3) \wedge T(X_3,X_1,X_2) \wedge \neg T(X_2,X_3,X_1) \wedge T(X_2,X_1,X_3) \wedge T(X_3,X_2,X_1) \wedge$
$T(X_1,X_3,X_2))$

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 where permutations of a tuple are distributed throughout 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. *i.e.* For a tuple $< 1,2,2 >$, we need to look for other tuples containing the values 1 and

Base Case:

1 Choose the $\overline{X} \in \mathcal{V}$ with m distinct variables
2 For each $r \in [1, n_1]$
3 $C_r^{\overline{X}} = (E_r^{\overline{X}} \wedge \neq\text{'s})$
4 End For
5 End For

Inductive Case:

6 For each $\overline{X} \in \mathcal{V}$ with p distinct variables
7 Let $S^h = \{C_i^{\overline{Z}} \mid \overline{Z}$ has $> p$ distinct variables$\}$
8 Let $k = |S^h|$
9 Let $S^t = \{\overline{Y} \mid \exists i \in [1, n_2], c \in [1, m-p]\ (\overline{Y} = replace^i(c, \overline{X})\}$
10 Let $S^a = \{C_j^{\overline{X}'} \mid \overline{X}' \in S_t, \exists \overline{Z}(C_j^{\overline{Z}} \in S^h \wedge \overline{Z} \simeq \overline{X})\}$
11 Let $S^q = \{\exists Y_1, \ldots Y_e(C_j^{\overline{X}'} \wedge \neq\text{'s}) \mid C_j^{\overline{X}'} \in S_a$ and $Y_1, \ldots, Y_e$ are variables in $\overline{X}'$, but not $\overline{X}\}$
12 Let $a_0, a_1, \ldots, a_{n_3}$ be an enumeration of the elements of $\mathcal{P}(S^q)$.
13 Let $r = k + 1$;
14 For each $0 \leq i \leq n_3$
15 For each $j \in [1, n_1]$
16 $C_r^{\overline{X}} = (E_j^{\overline{X}} \wedge \bigwedge_{b \in a_i} b \wedge \bigwedge_{b \in S^q - a_i} \neg b \wedge \neq\text{'s})$
17 r=r+1
18 End For
19 End For
20 End For

Explanation:

Line 3 - $\neq$'s means include an inequality statement $X_i \neq X_j$, for each X_i and X_j in the formula.
Line 7- S^h is the set of all classes in higher groups (*i.e.* previously generated).
Line 9 - S^t is the set of all variable patterns which result from applying *replace* to $\overline{X}$.
Line 10 - S^a is the set of formulas in S^h which have their original variable pattern instantiated by one in S^t.
Line 11 - Puts some quantifiers in front of each element of S^a.
Line 16 - A class consists of some $E_j^{\overline{X}}$ followed by some conjunction of all the elements of S^q.
Line 17 - r is used to number the classes produced.

Figure 2.1: Procedure for generating an equivalence class $C_r^{\overline{X}}$

2 such as $< 1,3,2 >$ and $< 2,1,4 >$. The first conjuncts ($E_i^{\overline{X}}$) identify relations which contain just values in $\overline{X}$ and the other conjuncts (preceded by existential quantifiers) identify equivalence classes in higher groups containing supersets of $\overline{X}$.

Example: For $m = 3$ and for some i
$C_i^{(}X_1, X_1, X_2) = \neg T(X_1, X_1, X_2) \wedge \neg T(X_1, X_2, X_1) \wedge T(X_2, X_1, X_1)) \wedge \exists Y_1(Y_1 \neq X_1$
$\wedge\, Y_1 \neq X_2 \wedge \neg T(X_1, X_2, X_3) \wedge T(X_3, X_1, X_2) \wedge \neg T(X_2, X_3, X_1) \wedge T(X_2, X_1, X_3) \wedge$
$T(X_3, X_2, X_1) \wedge T(X_1, X_3, X_2)) \wedge \ldots$

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.

2.2.3 Equivalence Class Properties

We now prove some of the properties of equivalence classes stated earlier.

Lemma 5

1) *Each of the classes defined is distinct (*i.e. *a tuple cannot belong to two different equivalence classes).*

2) *The set of equivalence classes defined is complete (*i.e. *every possible tuple that may appear in a relation during rule execution belongs to an equivalence class).*

3) *The number of equivalence classes is bounded by a constant depending only on the schema.*

Proof:
1) Suppose we have a tuple belonging to distinct classes $C_a^{\overline{X}}$ and $C_b^{\overline{Y}}$. Then $\overline{X}$ and $\overline{Y}$ satisfy the same equality/inequality relationships among their components and thus they are are isomorphic. So, by our choice of $\mathcal{V}$, $\overline{X} = \overline{Y}$. But if two classes have the same variable pattern, then they must be the boolean AND of some common set $\{e_1, \ldots, e_s\}$ of formulas. Therefore, for them to be different, there must

be a conjunct e_i which is negated in one class and not in the other. But this would produce a contradiction since e_i and $\neg e_i$ can't both be true for the same tuple.

2) This should be clear from the construction.

3) Let N_i = Number of classes in group i

$N_m \leq m^m \times n_1$ where
number of non isomorphic variable patterns $\leq m^m$
n_1 = number of choices for E_r's.

$N_{m-1} \leq m^m \times n_1 \times 2^{N_m \times n_2}$
$\leq m^m \times n_1 \times 2^{m^m \times n_1 \times n_2}$

$N_{m-2} \leq m^m \times n_1 \times 2^{(N_m + N_{m-1}) \times n_2}$
$\leq m^m \times n_1 \times 2^{(m^m \times n_1 + m^m \times n_1 \times 2^{m^m \times n_1 \times n_2}) \times n_2}$

$\vdots$

$N_1 \leq m^m \times n_1 \times 2^{(N_m + N_{m-1} + \ldots + N_3 + N_2) \times n_2}$

Total number of equivalence classes is
$N = N_1 + N_2 + \ldots N_m$
$\leq m \times N_1$

Since N is a very complicated number, we won't attempt to write it out. Suffice to say it has $m+1$ levels of exponentials ! ■

2.2.4 Instance Mappings

We now study how the equivalence classes just defined can be used for deciding termination. We sketch the main ideas first and then provide proofs of our claims in a later section.

$$\begin{tikzcd}
\boxed{I_0} \arrow[r, "\psi"] \arrow[d, "\delta"'] & \boxed{J_0} \arrow[r, shift left, "\tau"] \arrow[d, "\delta"'] & \boxed{I_0^\tau} \arrow[l, shift left, "\psi"] \arrow[d, "\delta"'] \\
\boxed{I_1} \arrow[r, "\psi"] & \boxed{J_1} \arrow[r, shift left, "\tau"] & \boxed{I_1^\tau} \arrow[l, shift left, "\psi"]
\end{tikzcd}$$

Figure 2.2: Mapping and Realising Equivalence Classes

Using our equivalence class definition, any database instance I over the relation T can be thought of as consisting of a union of equivalence classes (see lemma 7). The function $\psi : inst(T) \rightarrow \mathcal{P}(\{C_1, C_2, \ldots, C_N\})$ is used to map database instances to such descriptions. Given a description $\psi(I_0)$, it is possible to produce some example *realisation* of it using a bounded number of constants k (see lemma 6). The function $\tau : \mathcal{P}(\{C_1, C_2, \ldots, C_N\}) \rightarrow inst(T)$ is used to do this. The new instance $I_0^\tau = \tau(\psi(I_0))$, can be thought of as a "compressed" version of I_0. It only contains enough tuples to realise each equivalence class (but no more); whereas I itself may contain many "unnecessary" tuples, ones which travel together with the realising tuples. If an update δ is applied to an instance I_0, then the new instance $I_1 = \delta(I_0)$, has the same equivalence class description as the result of applying δ to $\tau(\psi(I_0))$ (see lemma 9). Figure 2.2 summarises this situation. Thus, the behaviour of our set of rules is completely characterised by the set of instances $\tau(\psi(inst(T)))$. Since these contain no more than k constants, we can now describe a procedure to decide termination (see lemma 10).

2.2.5 Termination Decision Procedure

Our algorithm for deciding termination is as follows

1. Generate all possible database states with number of constants $\leq k$, where k is the number obtained in the proof of lemma 6.

2. Select the first state

3. Run rule set on this state. If a repeating state is reached, return *non termination* and exit.

4. If there is a next database state, then select it and goto 3. If all database states have been used then return *termination* and exit.

2.2.6 Correctness Proofs

We now provide proofs of the results used in section 2.2.4.

Lemma 6 *There is an upper bound $k < \infty$, such that for every satisfiable instance $s \in \mathcal{P}(\{C_1, C_2, \ldots, C_N\})$, there exists $d \in \tau(s)$ such that the number of constants used in d is $\leq k$.*

Proof: The no. of possibilities for s is finite, since the domain $\mathcal{P}(\{C_1, C_2, \ldots, C_N\})$ is finite. Our procedure is to examine every possible s, produce an example realisation $\tau(s)$ if it is satisfiable (*i.e.* can be realised) and let k be the maximum number of constants needed for generating any of these realisations. The detail for realising any particular s now follows. The equivalence classes in s are generated bottom-up from group $p = 1$ to group $p = m$. For each class, we use a new set of constants to create a representative m-tuple, respecting the appropriate equality patterns. We insert tuples into the T atoms which are not negated in the conjuncts of the equivalence class. 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. The use of new constants for each class eliminates "crosstalk" between them. The number of constants needed for any particular equivalence class is bounded by

$m\times$ (number of atoms in the class definition)
$\leq m \times N^m$ where N is the number of equivalence classes.

and so the total number of constants is bounded by $m \times N^m \times N$.

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 s, *i.e.* it would be realised by every instance which realises s. We use the notion of a *class dependency tree*. Each class C_x can be associated with such a tree, comprised of nodes which are a $(Class, Variable\ list)$ pair, where the variable list contains the set of free variables in the definition of the class (ones which aren't existentially quantified). An arc from node 1 to node 2 indicates that the definition of the class in node 1 (say C_1) requires the existence of the class in node 2 (say C_2). This means that in order to realise node 1, we will also have to realise node 2, using some extra variables. These variables are bound in C_1, but free in C_2. The root node contains the class C_x. We will refer to this kind of tree as a *free tree.*

Let C_b be some class that has been realised "accidentally" and let U be the set of tuples that cause C_b to be true. Then it must be the case that all the tuples in U were generated in the process of making some class non empty, since different constants were used for different classes; call this C_a and let the tuples used to realise C_a be S_a. Suppose $t_a \in C_a(S_a)$ and $t_b \in C_b(S_a)$.

Let M be a map, which given a free tree and a database instance, produces a new kind of tree, a *bound tree.* This has nodes of the form $(Class, BindingList)$, where the binding list describes the bindings of the free variables in the class for the given instance. Let h be a map of type M, which takes C_a and S_a as arguments and produces t_a. Similarly, let g be a map of type M, taking C_b and S_a as arguments and producing t_b. Suppose q is the smallest (*i.e.* closest to the root) node in $h(C_a, S_a)$ containing all constants in t_b. Let q_a be the counterpart of this node in C_a.

Suppose we are given some database instance I, such that $C_a(I) \neq \emptyset$. Then there exists $t' \in C_a(I)$. Therefore, we can define another non empty map f of type M, which takes C_a and I as arguments and produces t'. Let q_I be the counterpart of q in this tree. Then there is a one to one correspondence between the constant pattern at node q_I and the constant pattern at node q. Let t_c be the tuple extracted from those positions of q_I which correspond to those positions where t_b is extracted. Each child (which is an instantiated equivalence class description) of q_I has the same truth value as the corresponding child of q. Therefore t_c belongs to $C_b(I)$. Hence $C_b(I)$ is non empty and so C_b will always be realised if C_a is. ■

Description of Computation: Suppose we have a set of rules $\mathcal{R}$ and a rule is triggered in state d_0. This gives rise to a (possibly infinite) sequence of updates $\delta_0, \delta_1, \delta_2, \ldots$ which produce a sequence of database states $d_1, d_2, \ldots$ where $d_1 = \delta_0(d_0), d_2 = \delta_1(d_1), \ldots, d_k = \delta_{k-1}(d_{k-1})$.

Lemma 7 *Let $\delta = \delta_0, \delta_1, \ldots$ be an update sequence which is applied to the initial database state d_0. For any database state d_j, each relation consists of a union of equivalence classes.*

Proof: The proof is by induction on j.

Base case j=0: Since an equivalence class is essentially a query defined with respect to the initial database, this follows straightforwardly.

Induction Step: Assume statement is true $\forall d_i,\ i \leq j$

Deletion Case: First suppose the update that takes us to d_{j+1} is a deletion. *e.g.*

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

Let $S_i = \{C_1^{\overline{X_1}}, C_2^{\overline{X_2}}, \ldots\}$ be the set of all non empty equivalence classes for d_j and let $S_j = \{a \mid C_a^{\overline{X_a}} \in S_i\}$. Then

$$d_j = \bigcup_{a \in S_j} C_a^{\overline{X_a}}(d_0)$$

Let $(\pi_{\overline{X}}\overline{Z})$ be a function which replaces the Z's with the appropriate X's, according to the update we are considering.

Now d_{j+1}

$$= \bigcup_{a \in S_j} C_a^{\overline{X_a}}(d_0) - (\pi_{\overline{X}}\overline{Z})(\bigcup_{a \in S_j} C_a^{\overline{X_a}}(d_0))$$
$$= \bigcup_{a \in S_j} C_a^{\overline{X_a}}(d_0) - \bigcup_{a \in S_j^{\overline{Z}}} (\pi_{\overline{X}}\overline{Z})(C_a^{\overline{X_a}}(d_0))$$
$$= \bigcup_{a \in S_j} C_a^{\overline{X_a}}(d_0) - \bigcup_{a \in S_j^{\overline{Z}}} (\pi_{\overline{X}}\overline{Z})(C_a^{\overline{Z}}(d_0))$$
$$= \bigcup_{a \in S_j} (C_a^{\overline{X_a}} \wedge \neg C_{a_1}^{\overline{X}} \wedge \ldots \wedge \neg C_{a_1}^{\overline{X}})(d_0), \text{ since } (A \cup B) - (C \cup D) = (A - C - D) \cup (B - C - D))$$

This last line is a union of equivalence classes and so we are done.

Insertion Case: This can be handled similarly to the deletion case. ∎

Lemma 8 *Let d_0 and d'_0 be initial database states with no constants in common and $\psi(d_0) = \psi(d'_0)$. Also let $d''_0 = d_0 \cup d'_0$. If a sequence of updates $\delta_0, \delta_1, \ldots, \delta_{i-1}$ is applied separately to each, then $\psi(d''_i) = \psi(d_i)$.*

Proof: Consider an equivalence class $C^i_a(d_0) \neq \emptyset$. Then also $C^i_a(d'_0) \neq \emptyset$ and $C^i_a(d''_0) \neq \emptyset$. In fact, $C^i_a(d''_0)$ consists of the union of $C^i_a(d_0)$ and $C^i_a(d'_0)$. This is because the equivalence class a tuple is assigned to depends only on the distribution of (at least one of) its constants throughout the initial database. Since there are no constants in common between d_0 and d'_0, their equivalence classes cannot affect each other. Thus $\psi(d''_0) = \psi(d_0) = \psi(d'_0)$. Now the same set of updates is applied to d''_0 as is applied to d_0. Since d''_0 contains all the tuples that d_0 does, all update instances that occurred for d will also occur for d''. But as shown in lemma 7, tuples within equivalence classes are constrained to travel together and so the d'' tuples follow the appropriate d tuples. Thus $\psi(d''_i) = \psi(d_i)$. ■

Lemma 9 *Let d_0 and d'_0 be initial database states with $\psi(d_0) = \psi(d'_0)$. If a sequence of updates $\delta_0, \delta_1, \ldots, \delta_{i-1}$ is applied separately to each, then $\psi(d_i) = \psi(d'_i)$*

Proof: The proof is by induction on i.

Base case: $\psi(d_0) = \psi(d'_0)$ is given already.

Induction Step: Consider $\psi(d''_i)$ as defined in lemma 8. Using that result, we can deduce that for each update instance on an equivalence class containing just constants from d_i, there will be a corresponding update on the same equivalence class involving just constants from d'_i. Hence, equivalence classes will be added or deleted exactly the same in d'_i as they are in d_i. Thus $\psi(d_i) = \psi(d'_i)$. ■

Lemma 10 *Consider an initial database state d_0. There exists another initial database state d'_0, where the number of constants in d'_0 is bounded and the rule set terminates on d'_0 if and only if it terminates on d_0.*

Proof: Choose a d'_0 such that $d'_0 = \tau(\psi(d_0))$. Using lemma 6, this can be done in $\leq k$ constants. We need to be able to check for a repeating state condition in order to decide termination. Suppose $d_{j+m} = \delta_{j+m-1}(\delta_{m+j-2}(\ldots(\delta_j(d_j))\ldots))$

We want to prove $d_j = d_{j+m} \Leftrightarrow d'_j = d'_{j+m}$

First prove $d_j = d_{j+m} \Rightarrow d'_j = d'_{j+m}$

Now $d_j = d_{j+m} \Rightarrow \psi(d_j) = \psi(d_{j+m})$, but $\psi(d_j) = \psi(d_{j+m}) \Rightarrow \psi(d'_j) = \psi(d'_{j+m})$ due to lemma 9 and $\psi(d'_j) = \psi(d'_{j+m}) \Rightarrow d'_j = d'_{j+m}$ since equivalence classes will always contain the same tuples if they relate back to the same original state. Thus $d_j = d_{j+m} \Rightarrow d'_j = d'_{j+m}$. We can use a similar argument to show that $d_j = d_{j+m} \Leftarrow d'j = d'_{j+m}$.

Using this fact and the repeating state technique, our computation starting from d'_0 will be non terminating iff the computation starting from d_0 is non terminating, and this can be detected by looking for a repeating state. ■

Theorem 4 now follows from the steps of the decision procedure outlined in section 2.2.5. Correctness follows from lemmas 7 and 10. We can obtain an analogous result for *while* programs also. Define a safe one literal *while* program to correspond to the language *while* restricted to performing safe one literal updates. Such a program could look like

while $(R \neq \emptyset)$
  $+R(X,Y) \leftarrow S(Y,Z,X), Y \neq Z$
  $+R(X,Y) \leftarrow Q(A,Y,Z,X)$
  while $(S = \emptyset)$
    $-Q(X,X,X,X) \leftarrow R(X,X)$
    $+S(X,Z,Y) \leftarrow Q(Y,Z,X,X), Y \neq Z, X \neq Z$
  end while
end while

Corollary 11 *Termination is decidable for safe one literal* while *programs.*

Proof: Execute the program on all databases with $\leq k$ constants and check for repeating states at program points. Program points can be defined to occur immediately after each *while* statement. ■

2.2.7 A More General Form

We can actually allow updates written in the safe one literal language to have a slightly more general form, without endangering decidability. Instead of restricting the right hand side to only a single atom, it can instead be a conjunction of atoms, *provided* each atom contains the same 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 is equivalent to the following rules in the old form

$$\begin{array}{l}
erase(tmp_1); erase(tmp_2); erase(tmp_3); erase(tmp_4) \\
+tmp_1(X,X,Y,Z) \leftarrow S(X,X,Y,Z) \\
+tmp_2(X,X,Y,Z) \leftarrow S(X,X,Y,Z) \\
-tmp_1(X,X,Y,Z) \leftarrow T(Y,X,Z) \\
-tmp_2(X,X,Y,Z) \leftarrow tmp_1(X,X,Y,Z)
\end{array}$$

/* tmp_2 contains the "join" of S and T. */

$$\begin{array}{l}
+tmp_3(Y,Y,X,X,Z) \leftarrow Q(Y,Y,X,X,Z) \\
+tmp_4(Y,Y,X,X,Z) \leftarrow Q(Y,Y,X,X,Z) \\
-tmp_3(Y,Y,X,X,Z) \leftarrow tmp_2(X,X,Y,Z) \\
-tmp_4(Y,Y,X,X,Z) \leftarrow tmp_3(Y,Y,X,X,Z)
\end{array}$$

/* tmp_4 contains the "join" of S, T and Q. */

$$-R(X,Y) \leftarrow tmp_4(Y,Y,X,X,Z)$$

where $tmp_1, \ldots, tmp_4$ are workspace relations.

2.3 Unrestricted One Literal Language

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 and can now use don't care symbols _ (whose usage is like that of logic programming). This gives it the ability to express semi-joins.

Definition 12 Unrestricted One Literal Language

- Conditions have the form
  $R(\overline{X})$
- Updates have the form
  $+R(\overline{X}) \leftarrow S(\overline{Y})$ where every variable that appears in $\overline{X}$ also appears in $\overline{Y}$.
  $-R(\overline{X}) \leftarrow S(\overline{Y})$ where $\overline{X} \cap \overline{Y} \neq \emptyset$ and $\overline{X}$ may contain don't care symbols.

■

An example unrestricted one literal rule is

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

Semi-Join Notation: The unrestricted one literal language is unable to do full joins, since updates are limited to one atom on their right hand side. It can, however, evaluate semi-joins. *e.g.* The statement $R(\overline{X}) \ltimes S(\overline{Y}) = \Pi_{\overline{X}}(R(\overline{X}) \bowtie S(\overline{Y}))$. The unrestricted one literal version of this is

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

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. We will henceforth use semi-join notation wherever possible to improve clarity.

Other operators: Some other operations are also expressible by one literal unrestricted. Indeed, $T \leftarrow R \cup S$ is equivalent to

$$\begin{aligned}
&erase(T)\\
&+T(\overline{X}) \leftarrow R(\overline{X})\\
&+T(\overline{X}) \leftarrow S(\overline{X})
\end{aligned}$$

and $T \leftarrow R \cap S$ is equivalent to

$$T \leftarrow R(\overline{X}) \ltimes S(\overline{X})$$

Emptiness of a relation R is also testable by trying to insert the contents of R into some temporary relation and seeing whether this triggers a rule of the form "on $ins(tmp)\ldots$".

Basic Idea

We will prove that termination is undecidable for the unrestricted one literal language. The proof makes use of two counter machines (2CM's), whose halting problem is undecidable. Given any description of a 2CM and its computation, 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 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. This accounts for most of the detail.

Details

The simulation is similar to one in [40, 29], but with some important differences regarding the interpretation of relations. 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. It is known that the halting problem for two counter machines is undecidable for the situation where the counters are set to zero in the initial state ([34]).

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. 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 the unrestricted one literal language is that it cannot express inequality. This means we can't do simple things like testing whether a relation contains more than one constant. Since the simulation in [40, 29] depends on this feature for checking the goodness of the $succ$ relation (amongst other things), we need to devise 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 rather than the usual *total* order, we can instead think of $succ$ as representing a kind of *partial* order on constants. Suppose we are trying to represent the constants $0, 1, \ldots, k$. Let f be the function mapping each of these numbers to every possible representation they may have in $succ$.

$$
\begin{aligned}
&f(0) = \{a_0^1, a_0^2, \ldots, a_0^{n_0}\} \\
&f(1) = \{a_1^1, a_1^2, \ldots, a_1^{n_1}\} \\
&f(2) = \{a_2^1, a_2^2, \ldots, a_2^{n_2}\} \\
&\vdots \\
&f(k) = \{a_k^1, a_k^2, \ldots, a_k^{n_k}\}
\end{aligned}
$$

where each $n_i < \infty$ and each $f(i)$ can be thought of as a stratum of constants.

We require the function f to have certain properties. These are
1) $\forall i, j \in [0, k]$ $(i \neq j \Leftrightarrow f(i) \cap f(j) = \emptyset)$ (no cycles in strata)
2) $\forall i \in [0, k-1]$ $\forall n(n \in f(i+1) \Leftrightarrow (\exists m \in f(i) \wedge succ(m, n)))$ (each "$i + 1$" is reachable from zero)

We need some more relations in our simulation, all 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 $last\_time$.
$last\_C_1$: contains *all* constants representing the value of the first counter at $last\_time$.
$last\_C_2$: contains *all* constants representing the value of the second 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 one.
$nonzero$: contains all constants in $succ$ which are not in R_0.
bad : has arity zero and is used to indicate whether the database has a valid computation or not.

Figure 2.3 outlines the procedure we use for checking the correctness of the various relations. It can be expressed using a set of unrestricted one literal rules. We next examine each of its components in turn.

Initialise 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. Relations to be emptied include $bad, R_0, \ldots, R_h$, $sofar$, $nonzero, reach$. We also have a number of copies of the $config$ and $succ$ relation, called $succ^1$,$succ^2$, $\ldots$,$config^1$,$config^2$, $\ldots$. These are created by statements such as $+succ^i(\overline{X}) \leftarrow succ(\overline{X})$.

```
Initialise Relations;
Construct R_0, ... , R_h;
Check that there are no cycles in succ;
Check goodness of config at time zero;
If      bad ≠ true
        reach = R_0;
last_time(X) = R_0(X);
current_time = succ(R_0);
While (current_time ≠ ∅ ∧ bad ≠ true)
        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
                EndFor
        EndFor
        If all tuples correct then
                reach = reach ∪ current_time
        Else {bad = true; reach = ∅};
        last_time = current_time
        current_time = succ(current_time)
End While
If (bad ≠ true and there is a time in reach for which config is in the halting state)
        Loop infinitely;
Else End;
```

Figure 2.3: Checking Algorithm

Construction of $\mathbf{R_0, \ldots, R_h}$: We wish to put each group of constants corresponding to one of the states $[0, h]$ in its own relation.

We 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 the ones which have a predecessor */

We now construct the R_1 relation, *i.e.* find the successor of '0'.

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

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. 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

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).

$$+config^1(T, S, C_1, C_2) \leftarrow config(T, S, C_1, C_2) \ltimes R_0(T)$$

$config^1$ now contains only the tuples from $config$ with '0' as a first argument.

The relation *nonzero* contains all the constants which are not in R_0 (easily expressible).

$-config^1(T, S, C_1, C_2) \leftarrow nonzero(S)$

If this deletion succeeds (*i.e.* if the state of $config^1$ is altered), then make *bad=true*. Similarly, we also do

$-config^1(T, S, C_1, C_2) \leftarrow nonzero(C_1)$
$-config^1(T, S, C_1, C_2) \leftarrow nonzero(C_2)$

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

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* is 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 $\quad R_i \cap candidate\_state \neq \emptyset$
        $last\_state = R_i; \quad exit;$
End For

Similar rules 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*, which can be done using a while loop similar to that used in the cycle checking section.

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 $\delta(j, >, =) = (j', pop, push)$. The following correspondences hold

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

Using these equivalences, we can check whether the condition of the transition is satisfied.

If the transition isn't applicable, then we ignore it and select another one, but if it is, then we need to check 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.

$$+current\_state(S) \leftarrow config^8(T, S, C_1, C_2) \ltimes current\_time(T)$$

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.

Halt Check: After exiting the main *while* loop in Figure 2.3, we need to check whether there is a time in reach for which $config$ is in the halting state.

$$erase(config^{11});$$
$$+config^{11}(T, S, C_1, C_2) \leftarrow (config^9(T, S, C_1, C_2) \ltimes reach(T)) \ltimes R_h(S)$$

If $config^{11}$ is not empty, then the halting state is reachable from the initial state.

Lemma 13 *If reach is non empty, then it contains representations of the integers from* 0 *to some* m *and for every* i $(1 \leq i \leq m)$*, the correct configuration of the machine at time* i *is in* $config$*.*

Proof: The proof is by induction on i.

Base Case: If reach isn't empty then it must contain '0' and we have already checked that the correct configuration at time '0' is in $config$.

Induction Step: Suppose we have the representations for the integers 0 to $k-1$ in $reach$. Now if there is no $k = succ(k-1)$ in $succ$, or no configuration for time k in $config$, then we are finished. If there does exist such a k for both $succ$ and $config$, then it will be in $reach$. We just need to show that the configuration is correct. But since $reach$ isn't empty, bad can't be true and so the transitions between configurations in $config$ are correct. ■

Theorem 14 *Termination is undecidable for a set of rules written in the unrestricted one literal language.*

Proof: We will prove that our rule system is non terminating iff the two counter machine halts.

1) 2CM halts $\Rightarrow$ rules non terminating

If the 2CM halts, then clearly there is an initial database state such that $bad \neq true$ and the halting state is in $config$ for some time t which is reachable. Thus the rules will be non terminating.

2) 2CM halts $\Leftarrow$ rules non terminating

If the rules are non terminating, then this must be due to the 'Loop infinitely' statement at the bottom of Figure 2.3. It can't be due to the main While loop, since there are no cycles in $succ$. Since the configurations in $config$ describe a correct halting computation of M (by lemma 13), the halting state must be reachable by the 2CM. ■

2.4 Safe Two Literal Language

Let us examine a two literal language, where don't care symbols are prohibited and the variables on the left hand side must all appear on the right hand side, but there may be two literals on the right hand side. We show that this is at least as powerful as the unrestricted one literal language and hence termination is undecidable.

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

Table 2.1: Summary of Decidability Results

Consider the statement

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

This can be rewritten in the two literal language as

$$\begin{aligned} & erase(tmp); \\ & +tmp(\overline{A}, \overline{B}) \leftarrow R(\overline{A}, \overline{B}), S(\overline{A}, \overline{C}) \\ & -R(\overline{A}, \overline{B}) \leftarrow tmp(\overline{A}, \overline{B}) \end{aligned}$$

Theorem 15 *Termination is undecidable for a set of rules written in the safe two literal rule language.*

2.5 Discussion

Table 2.1 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 $\bowtie$ and $\times$ 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.

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 chapter.

Chapter 3

Decidability of Unary Languages

3.1 Introduction

In the previous chapter, we saw that 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 type 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 may be complex, 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*$^{\neq,\neg}$ *View Language*: Same as 2), but with negation and inequality allowed in view bodies.
4. *Binary Conjunctive View language*: Same as 2), but with binary conjunctive views.
5. *Variable Arithmetic*: Consists of updates which may create new domain values for variables.

Case 1 is decidable and cases 3,4 and 5 are undecidable. The decidability of case 2 is currently unknown, but we conjecture that it is decidable.

The structure of this chapter is as follows - we first discuss some preliminary concepts concerning the structure of database relations. Thereafter, in each subsequent section, an update syntax is defined and a decidability theorem presented. In the last section, we discuss the results obtained, make a comparison with related work, and outline some directions for future research.

3.2 Relation Classification

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 in the body of view definitions, 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.

Note that throughout rule execution, the contents of views and hidden relations do not alter and only the contents of the working relations may change. Consequently, if we wish to test for repeating states, we need only compare the states of the working relations.

Also note that these descriptions 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.

3.3 Arbitrary Updates and Unary Initialisation (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 16 *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: Our proof of decidability is similar to that used for the safe one literal language (theorem 4) in its use of equivalence classes. In the present case though, their structure is much less complex. The technique used is also similar to ones used in [5, 40] and we therefore only sketch the details.

Let the unary relations be $U_1, U_2, \ldots, U_n$. Unary equivalence classes then have the form

$$C^1(X) = \{< X > |^{(\neg)} U_1(X) \wedge^{(\neg)} U_2(X) \wedge \cdots \wedge^{(\neg)} U_n(X)\}$$

They can be enumerated $C_1^1, C_2^1, \ldots, C_{n_1}^1$, where $n_1 = 2^n$

It is easy to construct any equivalence class instance using a single constant for each equivalence class. *e.g.* $C_2^1 = k_2$, where k_2 is in U_i iff U_i occurs positively in C_2^1. Termination is then decided by generating all possible databases with $\leq n_1$ 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.
i.e.

$$\{< X_1, \ldots, X_m > |C_{i_1}^1(X_1) \wedge \cdots \wedge C_{i_m}^1(X_m)\} \text{ where } i_k \in [1, n_1]$$

This can be shown by considering the relational operations $\times, \sigma, \pi$ and demonstrating closure. ■

Clearly, if a schema contains only unary relations, then the conditions of the above theorem will be satisfied.

Corollary 17 *Termination is decidable for a rule system using Datalog$^{\neg\neg}$ updates over a schema containing only unary relations.*

3.4 Unary Updates and 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 one of the view definitions.

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

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

$$-U_2(X) \leftarrow U_1(X),^{\neg} U_3(X), V_2(X),^{\neg} V_1(X), U_3(Y), U_4(Y), U_2(Y)$$

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

Throughout this subsection we will assume that

- The hidden relations are $R_1, \ldots , R_p$.
- The views are $V_1, \ldots , V_n$.
- The working relations are $U_1, \ldots , U_\ell$.

The decidability of termination for this class of rules is closely tied to the following problem.

Definition 18 The Realisability of a Combination of Unary Views Problem
A *view-state combination* is a formula ϕ of the form

$$(\neg)U_1(X) \wedge \ldots \wedge (\neg)U_\ell(X) \wedge (\neg)V_1(X) \wedge \ldots \wedge (\neg)V_n(X)$$

where at least one $U_i(X)$ or one $V_j(X)$ is not negated. *Question:* Given a set S of view-state combinations, is it possible to realise *just* the elements of S ? *i.e.* Is there an instance I over $U_1, \ldots, U_\ell, R_1, \ldots, R_p$ in which every view-state combination from the set is satisfied and every view state combination not in the set is not satisfied ? ■

Note that there are $2^{n+\ell}$ view-state combinations. The formula where each $U_i(X)$ and each $V_j(X)$ is negated is not included as a view-state combination because it will not contribute tuples to working relations.

Observe that each view state combination corresponds to the definition of an equivalence class for rules in the unary conjunctive view language. If we can realise a set of view states, then we will be able to realise any set of equivalence classes. Unfortunately, we do not know whether the problem is decidable.

Open Question: Is the realisability of a combination of unary views problem decidable ?

Since the view states correspond to equivalence classes, we can prove a conditional decidability theorem for the unary conjunctive view language.

Theorem 19 *Termination is decidable for a rule system using the unary conjunctive view language iff the realisability of a combination of unary views problem is decidable.*

Proof: Suppose the view realisability problem is decidable, then the same decision procedure as that used for the case of arbitrary updates and unary initialisation can be employed: We just generate and test for termination over all realisable sets of view-state combinations. This can be done using $2^{n+\ell} \times (n + \ell)$ constants using similar reasoning to that in the proof of theorem 16. It returns non termination iff one of these tests says so. If the view realisability is undecidable, then it is straightforward to write a set of rules which terminate iff a particular view state is reachable. Hence termination will also be undecidable. ■

3.5 Unary Updates and Binary Conjunctive Views

It is natural to extend the definition in section 3.4 by allowing binary views. 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.*

$$V_1(X,Y) \leftarrow R_1(X,Y), R_2(Y,Z,Z,X)$$
$$-U_2(X) \leftarrow U_1(X), U_3(X), V_1(X,Y), U_3(Y), U_4(Y)$$

For this type of view, we are able to prove undecidability.

Theorem 20 *Termination is undecidable for a rule system written in the binary conjunctive view language.*

Proof: We show how to define a set of rules which is non terminating iff a two counter machine halts. The proof is quite similar to that used for the undecidability of the unrestricted one literal language in theorem 14, mainly because the only relations that are needed for "communication" between rules in that simulation are unary. Therefore, we only sketch some of the important parts.

Checking the succ relation: It is necessary to ensure that *succ* does not contain any cycles. This can be done using a while loop. We assume that *succ* is a binary view. Notice how we are able to iterate through the elements of *succ*, something which could not be done if it was hidden inside a unary view.

$+reach(X) \leftarrow R_0(X)$
$+current(X) \leftarrow R_0(Y), succ(Y,X)$
while ($current \neq \emptyset$)
    if $current \cap reach \neq \emptyset$
        $bad = true; erase(current)$
    else
        $erase(tmp);$
        $+reach \leftarrow current(X);$
        $+tmp(X) \leftarrow current(X);$
        $erase(current);$
        $+current(X) \leftarrow tmp(Y), succ(Y,X);$
end while

Check the goodness of $config$: Define a view of the form

$$V_s(S) \leftarrow R_0(T), config(T, S, C_1, C_2)$$

and then set bad if $V_s(X) \not\subseteq R_0(X)$. Similarly define V_{c_1} and V_{c_2} and do the same checking.

Checking Transitions: Suppose the transition is $\delta(j, >, =) = (j', pop, push)$. We define a view V_1 for checking the state j' and check the equality of two relations tmp_1 and tmp_2

$$\begin{aligned}
V_1(S') &\leftarrow config(T, S, C_1, C_2), config(T', S', C_1', C_2'),\\
&\qquad succ(T, T'), S = j, succ(X, C_1), zero(C_2)\\
+tmp_1(X) &\leftarrow V_1(X)\\
+tmp_2(X) &\leftarrow R_{j'}(X)\\
-tmp_1(X) &\leftarrow tmp_2(X)\\
bad &\leftarrow tmp_1(X)\\
-tmp_2(X) &\leftarrow V_1(X)\\
bad &\leftarrow tmp_2(X)
\end{aligned}$$

Similar rules are needed for checking the new values of C_1' and C_2'.

Halt Check: Define a binary view of the time and state components of $config$ and then iterate through it, starting from R_0, to see whether R_h is present. ■

3.6 Extended Unary Views

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

$$V_i(X) \leftarrow^{(\neg)} R_1(\overline{X_1}), \cdots, ^{(\neg)} R_n(\overline{X_n}), Y_i \neq Y_j, \ldots, Y_p \neq Y_q$$

where the Y's are contained in the $\overline{X}$'s. An example is

$$V_1(X) \leftarrow R_1(X, Y), R_2(Y, Z, Z, X), ^{\neg} R_3(X, Y), Y \neq X$$

Theorem 21 *Termination is undecidable for a rule system in the unary conjunctive$^{\neg,\neq}$ view language.*

Proof: Once again we can simulate a two counter machine (2CM). The proof is quite similar to previous ones and so we leave most of the details to the appendix. The main idea is that we do not need to use the capability of rules to iterate through the $config$ relation and find reachable states. Instead, we use a transitive closure relation tc, which we require to have certain reachability properties. These can be enforced by using negation in the body of unary views. The successor relation is then defined to be an appropriate subset of tc and so we ensure that all constants in $succ$ are reachable. Similarly, we can ensure that all times in $config$ are reachable and so the machine will halt iff the halt state appears anywhere in $config$. ■

3.7 Arithmetic

Although not technically a unary type language, we briefly look at the implications of allowing constant invention for the 0-1 language. The database consists of variables which can be tested, incremented and decremented by updates. *e.g.*

$$++Y_2 \leftarrow^{\neg} X_2, Y_1$$
$$--X_3 \leftarrow X_2, ^{\neg} X_5$$

where a variable appearing positively on the right hand side means it is not equal to zero and appearing negatively means it equals zero.
As one might expect, termination is undecidable for this language, since we can directly simulate a two counter machine.

Theorem 22 *Termination is undecidable for a rule system written in the 0-1 language with arithmetic.*

3.8 Discussion

Table 3.1 summarises the decidability results presented in this chapter. The key features which cause undecidability are either allowing views to have binary arity or allowing inequality and 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

AUUI	Decidable
Unary conjunctive views	Open
Unary conjunctive$^{\neq,\neg}$ views	Undecidable
Binary conjunctive views	Undecidable
Variable Arithmetic	Undecidable

Table 3.1: Decidability Results for Unary type Languages

negation and inequality effectively make 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. This will be the object of future work.

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.

The combination of unary views realisability problem is similar to the *view update problem* ([28]) and the area conjunctive query containment ([39, 47]). The emphasis is somewhat different, however, since we require 1) some set of view formulas to be satisfied and 2) some set of formulas *not* to be satisfied for a given instance. The above areas are oriented towards 1).

Unary type languages seem to share similarities with the area of description logics ([16]). Some cross fertilisation of results may be possible here and we intend to pursue this as a line of further research.

We have now concluded our presentation of rule languages for which termination is (un)decidable. In the next chapter, we will tie together some of the results and study their complexity.

Chapter 4

Complexity and Further Extensions

4.1 Introduction

Previous chapters have presented decidability results about local termination for some specific rule languages. We now expand on these results in a number of ways. First, we present complexity bounds on termination analysis for the decidable cases, indicating the intractability of the problem, which is NP-hard even for just the 0-1 language. Next, these cases are unified under the general heading of *bounded model languages* and the properties of local and global termination are formally compared. Lastly, we show how our termination analysis techniques can be used to analyse the property of *reachability*.

4.2 Complexity Bounds

As observed in chapter 1, termination analysis is an important practical problem and it is therefore of interest to provide bounds on the complexity of decision procedures. Unfortunately, as we now show, compile time detection of this property is potentially expensive, even for relatively simple languages.

4.2.1 0-1 Language

The complexity of the 0-1 language is examined to act as a reference point for that of more complex languages.

Theorem 23 *Let k be the number of variables in the program. Deciding local termination for rules in the 0-1 language is in PSPACE, but NP-hard, in terms of k.*

Proof:

Upper Bound: We generate every possible state of the 0-1 variables and check whether there is one which results in non termination when we input the state to the rule program. Let I be a particular input consisting of k variables all having some value. The rules are executed until they either terminate or we determine that we're in a state that's already occurred. The amount of space this requires at first seems exponential, because we want to maintain a history of all states. This can be reduced to a polynomial amount, however, by using an argument similar to that in [3]. Suppose we wish to test whether a rule program P is non terminating, then we run two copies of it "simultaneously"- P_1 and P_2. For every rule execution of P_1, we execute two rules of P_2. Now if P is non terminating, then there exist K and L such that P enters a loop of length L at step K. Thus $\forall i \geq K$ and $j \geq 1$ $P^i(I) = P^{i+jL}(I)$, where I is an initial database instance. Also, for $i = LK$ and $j = K$, we have $P^{LK}(I) = P^{2LK}(I)$. Thus, at iteration LK, P_1 and P_2 have the same (dbstate,rule) pair iff the system is non terminating and a fixpoint has not been reached. Hence the total amount of space used for checking an instance is that required for storage of two rule systems, which is $\mathcal{O}(k)$. This space can be reused for all possible instances if necessary and so the total amount of space used is polynomial.

Lower Bound: We can construct a set of rules for any instance of the 3-SAT problem. *e.g.* $C_1 \wedge C_2 \wedge \ldots \wedge C_q$ where each C_i is a disjunct of three positive or negative boolean variables. Suppose $C_i = x_1 \vee \neg x_2 \vee x_3$ and $C_{i+1} = x_1 \vee x_3 \vee \neg x_1$. We define a set of 0-1 rules to check whether a database state satisfies the constraint. Note that we abuse our notation slightly and use statements such as $a \leftarrow \neg b$ to mean "if $b = 0$ then $a = 1$". We also assume all variables have either a 0 or 1 value in the initial state.

Rule C_i	Rule C_{i+1}
on U(A_i)	on U(A_{i+1})
if *true*	if *true*
$\neg c_i \leftarrow true;$	$\neg c_{i+1} \leftarrow true;$
$c_i \leftarrow x_1;$	$c_{i+1} \leftarrow x_1;$
$c_i \leftarrow \neg x_2;$	$c_{i+1} \leftarrow x_3;$
$c_i \leftarrow x_3;$	$c_{i+1} \leftarrow \neg x_1;$
$A_{i+1} \leftarrow c_i;$	$A_{i+2} \leftarrow c_{i+1};$

Rule C_q will be defined so that an infinite loop occurs if c_k is *true*, otherwise execution will halt. The rule system will then be locally non terminating when rule C_1 is initially triggered iff the 3-SAT formula is satisfiable. ■

4.2.2 Safe One Literal Language

Theorem 24 *Let n be the number of relations in the database and let m be their maximum arity. Deciding local termination for rules in the safe one literal language is in NON-ELEMENTARY SPACE, but PSPACE-hard, in terms of $k = (m + 2 + \lceil \log_2 n \rceil)$.*

Proof:

Upper Bound: Recall that in proposition 1, we collapsed the n database relations into a new one of arity $k = (m + 2 + \lceil \log_2 n \rceil)$. We showed in lemma 5 that the formula for the number of equivalence classes (N) requires $k + 1$ levels of exponentials. The number of constants needed for realising classes is $\mathcal{O}(N^k)$, from lemma 6. Executing the rules requires space polynomial in the number of constants and we can reuse space with the method of the proof for theorem 23. Thus space with $k + 2$ levels of exponentials overall, which is non elementary.

Lower Bound: We demonstrate how to construct a safe one literal rule program which checks whether an instance of a quantified boolean formula (QBF) is satisfiable (a PSPACE-complete problem). The proof is similar to one in [27]. The language we use includes constants, which are used purely for convenience. The one literal equivalence class definition could easily be extended to incorporate them

QBF instances are well-formed quantified boolean formulas of form $\mathcal{Q}_1 x_1 \cdots \mathcal{Q}_n x_n \phi$, where each $\mathcal{Q}_i$ is either $\forall$ or $\exists$ and ϕ is a quantifier free boolean formula in conjunctive normal form over variables $x_1, \ldots, x_n$. Let $\Phi = \mathcal{Q}_1 x_1 \cdots \mathcal{Q}_n x_n \phi$ be an instance of QBF where $\phi = \bigwedge_{j=1}^{m} C_j$ is conjunctive. By considering $\forall y \Phi$, where y is a new variable if necessary, we may assume $\mathcal{Q}_1 = \forall$.
We use an input schema having the following relations

- *bad* of arity 0
- T_i and p_i of arity i for $1 \leq i \leq n$
- S_j of arity n for $1 \leq j \leq m$

T_i holds all possible truth assignments for $x_1, \ldots, x_i$. S_j holds assignments under which $\bigwedge_{l=1}^{j} C_l$ is *true*. p_i holds all those assignments such that the formula $\mathcal{Q}_i x_i \cdots \mathcal{Q}_n x_n \phi$ is *true*. All relations are made empty initially using *erase* updates.

Initialisation rules: We first generate all possible database states.

$$
\begin{aligned}
&+T_1(1)\\
&+T_1(0)\\
&+T_2(X_1, 0) \leftarrow T_1(X_1)\\
&+T_2(X_1, 1) \leftarrow T_1(X_1)\\
&\vdots\\
&+T_i(X_1, X_2, \ldots, X_{i-1}, 0) \leftarrow T_{i-1}(X_1, X_2, \ldots, X_{i-1})\\
&+T_i(X_1, X_2, \ldots, X_{i-1}, 1) \leftarrow T_{i-1}(X_1, X_2, \ldots, X_{i-1})
\end{aligned}
$$

We next store all the assignments that make $\bigwedge_{l=1}^{j} C_l$ *true* . A rule for S_j selects assignments from S_{j-1} (T_n if $j = 1$) that make at least one literal in C_j true.
e.g. Suppose $C_2 = x_1 \vee \neg x_4 \vee x_5$, then we have

$$
\begin{aligned}
&+S_2(1, X_2, \ldots X_n) \leftarrow S_1(1, X_2, \ldots X_n)\\
&+S_2(X_1, \ldots, X_3, 0, X_5, \ldots, X_n) \leftarrow S_1(X_1, \ldots, X_3, 0, X_5, \ldots, X_n)\\
&+S_2(X_1, \ldots, X_4, 1, X_6, \ldots, X_n) \leftarrow S_1(X_1, \ldots, X_4, 1, X_6, \ldots, X_n)
\end{aligned}
$$

We check that S_m is non empty and if so, copy assignments from S_m to p_n.

Validity Checking Rules: We now propagate assignments where all conjunctions are true to the p_i's. The propagation depends on the quantifiers and the QBF formula will be satisfiable iff p_1 is non empty.
If the quantifier for $Q_{i+1} = \exists$ then

$$+p_i(X_1, \dots, X_i) \leftarrow p_{i+1}(X_1, \dots, X_i, X_{i+1})$$

If the quantifier for $Q_{i+1} = \forall$ then

$$\begin{aligned}
+t_1(X_1, \dots, X_i) &\leftarrow p_{i+1}(X_1, \dots, X_i, 0) \\
+t_2(X_1, \dots, X_i) &\leftarrow p_{i+1}(X_1, \dots, X_i, 1) \\
+t_3(X_1, \dots, X_i) &\leftarrow t_1(X_1, \dots, X_i) \\
-t_3(X_1, \dots, X_i) &\leftarrow t_2(X_1, \dots, X_i) \\
-t_1(X_1, \dots, X_i) &\leftarrow t_3(X_1, \dots, X_i)
\end{aligned}$$

t_1 now contains the intersection of t_1 and t_2.

$$+p_i(X_1, \dots, X_i) \leftarrow t_1(X_1, \dots, X_i)$$

If p_1 is non empty then go into an infinite loop. If not, the program will terminate.
■

4.2.3 Arbitrary Updates and Unary Initialisation

Theorem 25 *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:
Upper Bound: We showed that there were $k = 2^n$ equivalence classes and that realising a class in the database requires one constant. The total amount of space used by rule execution is then bounded by $(r + n) \times k^m = \mathcal{O}(2^{n \times m})$.

Lower Bound: This language is a generalisation of the 0-1 language and so NP-hardness follows from theorem 23. ■

4.3 Bounded Model Languages

The techniques used to prove decidability of termination in chapters 2 and 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 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, we showed that we could construct a representative for each satisfiable combination of equivalence classes in an instance using a bounded number of constants. Henceforth, we will refer to languages having this property as *bounded model* languages.

Definition 26 Bounded Model Language
A rule language L is a bounded model language if there is an effectively computable $c < \infty$ (depending only on the schema) such that for every rule program P written in L and for every instance I, there is another instance I' using $\leq c$ constants such that $P(I)$ terminates iff $P(I')$ terminates. ■

It is easy to show that termination is decidable for this class of languages.

Theorem 27 *Termination is decidable for any bounded model language L.*

Proof: Generate all realisable database instances with $\leq c$ constants and use them as input to the rule system. Check for loops using state comparison. ■

Remark 28 It would be tempting to define a bounded model language as one which has a bounded number of equivalence classes. Unfortunately, this would mean that the proof for theorem 27 would no longer work, since it has been proven that even if we have some description of an equivalence class, such as an L^k axiomatisation (L^k signifies the fragment of first order logic using at most k variables), there is no automated procedure in general to produce an example structure of that equivalence class ([31]).

Remark 29 Although the property of *confluence* is guaranteed by our imposition of a total order upon rules, if we altered it to a partial order, then confluence would also be decidable for bounded model rule languages.

4.4 Local and Global Termination

Recall that in chapter 1, we distinguished between two types of termination, *local* and *global*. Local termination analysis is performed on a rule set with a well defined starting point, and so the execution of rules can be essentially regarded as a deterministic *while* program. Global termination analysis corresponds to analysing a program 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 30 *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 not true. Intuitively, this is because in order to decide local termination, we may need to conduct a kind of 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 not be necessary. The following proposition demonstrates this.

Proposition 2 *There exists a rule language for which global termination is decidable and local termination undecidable.*

Proof: We define a rule language L with such a property. 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 event $del(R)$ iff c is satisfiable. Hence local termination is undecidable. ■

4.5 Reachability

As observed in section 4.4, local termination analysis is similar to reachability analysis. We now explore this relationship using a property called *reflection.* The notion of reachability might at first seem to be vacuous, since any rule can be reached by having its event triggered. A more reasonable question, however, is whether a rule is reachable as a consequence of *some other* rule being triggered. Reachability can be thought of as a generalised kind of *satisfiability*, since a rule is reachable only if certain conditions can be satisfied.

Definition 31 A rule r_b is reachable due to an event e, if there exists an instance I such that for some terminating rule execution sequence triggered by e, r_b is in the pending structure. ■

We now define a property which we will use for reasoning about reachability.

Definition 32 Reflective Rule languages
Let $\mathcal{R}_1$ be a set of rules written in a language L using predicates in a schema $\mathbf{S}$. Let $\mathcal{R}_2$ be a set of rules written in L on a schema $\mathbf{S}'$ which is a superset of $\mathbf{S}$ and in particular includes a special 0-ary predicate (not in $\mathbf{S}$), say *halt.*
We say $\mathcal{R}_2$ is a *reflective version* of $\mathcal{R}_1$, if on each input I over $\mathbf{S}$, $\mathcal{R}_2$ always terminates and:

- $\mathcal{R}_1$ is locally non terminating on input I triggered by event e iff $\mathcal{R}_2$ locally terminates on input I triggered by event e with *halt* false.

- $\mathcal{R}_1$ locally terminates on input I triggered by event e with I' as final state iff $\mathcal{R}_2$ locally terminates on input I triggered by event e with *halt* true and the output restricted to $\mathbf{S}$ is I'

- For each r_i in $\mathcal{R}_1$ and triggering event e, there is an r_i' in $\mathcal{R}_2$ such that r_i is reachable due to event e iff r_i' is reachable due to event e.

We say a rule language L is *reflective*, if for any set of rules $\mathcal{R}_1$ written in L on $\mathbf{S}$ and triggering event e, there exists a set of rules $\mathcal{R}_2$ on $\mathbf{S}'$ such that $\mathcal{R}_2$ is a reflective version of $\mathcal{R}_1$. ■

Essentially, this definition says that a language is reflective, if for any given rule set, we can ensure that it is locally terminating by adding extra rules which will "reflect" upon the execution. These rules break any loops without interfering with the original database contents. Reflection is similar to the notion of loop free simulation defined in [3].

An obvious question is what types of rule languages are reflective. The two principal groups we've already examined, the one literal and unary type languages are both reflective. To see this, we can use the method of [3] to construct an $\mathcal{R}_2$ which runs two program copies of $\mathcal{R}_1$ "simultaneously" and which was sketched in part a) of theorem 23. It is straightforward to write rules which achieve this kind of sequencing, by defining some control rules with higher priority than the existing rules, which are invoked after each "standard" iteration/rule execution and these will do appropriate checking to ensure any loops are broken. Updates just need to have the ability to copy one relation to another, to test whether the intersection of two relations is non empty and to (unconditionally) trigger other rules.

By using the property of reflection, theorem 33 now states that it is possible to give a tight relationship between reachability and local termination.

Theorem 33 *Let L be a reflective rule language. Local termination is decidable for L if and only if reachability is decidable for L.*

Proof:

i) Local termination decidable $\Rightarrow$ reachability decidable.

Intuitively, we will show that we can test reachability by inserting and removing rules which cause loops or falsify conditions. Let $\mathcal{R}$ be a rule set. An event e causes a rule to be put in the pending structure (recall that it can't be more than one due to the singleton pending set semantics). Suppose r_b is the rule we're trying to determine reachability of. There are two cases to be considered.

Case 1: The rule set $\mathcal{R} \setminus \{r_b\}$ is locally terminating when e is triggered initially. We define a new rule called r_{nt}, whose priority is higher than any other rule and whose condition and action are such that it infinitely triggers itself. Modify r_b such that it always triggers r_{nt}. This rule set is now locally non terminating due to e iff r_b is reachable due to event e.

	Lower Bound	Upper Bound
0-1	NP	PSPACE
Safe One Literal	PSPACE	NON ELEMENTARY
AUUI	NP	EXPSPACE

Table 4.1: The Complexity of Local Termination Analysis

Case 2: The rule set $\mathcal{R} \setminus \{r_b\}$ is locally non terminating when e is triggered initially. We create a new rule set $\mathcal{R}_2$ such that $\mathcal{R}_2$ is a reflective version of $\mathcal{R}$. We note that by definition 32, the property of reachability is essentially invariant under reflection. Next define r_{nt} as above. This new rule set is now locally non terminating due to e iff r_b is reachable due to event e.

ii) Local termination decidable $\Leftarrow$ reachability decidable.

First create a reflective version of the rule set. We add a rule which is reached iff an infinite loop is occurring. For each rule r_i in the reflective version, we create the rule r'_i whose priority is higher than that of r_i, which is triggered on the same event as r_i and which itself triggers the rule r_{reach} if *halt=false*. Otherwise, it behaves like rule r_i. We now have the situation that r_{reach} is reachable iff the system is non terminating. ■

Corollary 34 *Reachability is decidable for the safe one literal and AUUI languages and undecidable for the unrestricted one literal, unary conjunctive*$^{\neq,\neg}$ *view and binary conjunctive view languages.*

This follows because all these languages are reflective and the first two have the bounded model property.

4.6 Discussion

Our complexity results are summarised in table 4.1. The bounds for the 0-1 language are quite tight, those for the safe one literal language are not. This is not of great concern, since our main aim has been to establish decidability and show the difficulty of the problem. We do, however, intend to investigate these bounds further in the future.

Reachability has been discussed in [12, 36]. In [12], it is used in a limited way by arguments about triggering graph analysis for rules with priorities. In [36], reachability (there termed liveness) in the presence of meta rules is analysed. The analysis is rule language independent though and so is in some sense orthogonal to our results.

As noted earlier, our definition of reflection is like the notion of loop freeness used in [3]. The principal difference is that we model a program as a set of *distinct* rules which must each have counterparts in the reflective version. This kind of work is also similar to work in [38], which discusses how to transform three valued programs into two valued programs. Exploring this connection will be a topic of further research.

Finally, we note that other properties of active rules, such as containment, can be defined and treated in a similar fashion to reachability. Since the techniques needed are largely the same, we do not pursue them here. Instead, in the next chapter we focus on a feature of active rules not commonly found in update languages, that of *control mechanisms*.

Chapter 5

Control Features

5.1 Introduction

Previous chapters have examined the effect variations in rule syntax can have upon the decision problem of termination. We now consider how other features of a rule system, namely methods for controlling execution, can affect this property. These are the *pending rule structure* and the *meta rule language*. The former represents the method used to keep track of multiple rules awaiting execution and can be modelled by structures such as a *set*, *stack*, *multiset* or *queue*. The latter is the language used to specify high level interactions between rules. We specifically consider *complex events*, *coupling modes* and *dynamic priorities*. Although these kinds of control mechanisms are present in active database prototypes, previous works on rule analysis have rarely considered sophisticated control structures. This is probably a reflection of both the difficulty of the problem and a lack of consensus on what the core features of an active database system should be.

We investigate a number of choices of control features and the impact which they can have upon termination decidability. The methods we use draw heavily upon formal language theory, partly because our focus is now less on manipulation of the database, than it is on manipulation of rules. As was the case for language syntax, termination is prone to become undecidable even for seemingly simple choices of functionality.

5.1.1 Choice of Rule Syntax

We will often be using the 0-1 rule language for our study of control structures, particularly the undecidable cases. This is because for these situations, we wish to limit the expressiveness of the rule syntax, in order to gauge the effect of other features.

5.2 Pending Rule Structure

The first control feature we examine is the pending rule structure. This is essentially a repository for rules which are 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 for a pending rule structure are

- *Singleton* - This has hitherto been our default choice for execution. The structure only ever contains a single rule. If a new rule is *put* onto the structure, it replaces any rule of lower priority already there. Otherwise, the structure remains the same and the new rule is discarded.

- *Set* - This is like the mathematical notion of a 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.

- *Multiset* - This allows multiple instances of a rule may be retained. Selection is done by removing an instance of a rule with highest priority.

- *Stack* - This may contain multiple instances of rules. New rules are placed on top of the stack in order of highest priority. Rule selection is done by removing the rule on top of the stack. The active database prototype NAOS [21] uses this approach.

- *Queue* - This may contain multiple instances of rules. New rules are placed on the bottom of the queue in order of priority. Rule selection is done by removing the rule at the top of the queue. The active database prototype HiPAC [25] uses this approach.

In order to analyse how pending structures can affect the termination behaviour of a rule system, we first need to define what we understand by the term system state.

Definition 35 System State
A *system state* is a pair *(dbstate,pstate)*, where dbstate is a database instance and pstate is an instance of the pending rule structure. ∎

5.2.1 Bounded Pending Structures

In previous chapters, we assumed an execution model using a singleton pending structure. This made it straightforward to detect termination by loop checking, since one only needed to look for repetitions of a (dbstate, single rule) pair. We now extend this by showing that termination is in fact decidable whenever the pending structure is bounded and we have a bounded model rule language. Examples of bounded structures are the singleton and the set.

Theorem 36 *Repeating System States*
Suppose the pending structure is bounded (contains $\leq f(n)$ rule instances for some f, where n is the number of defined rules). A rule computation will not terminate iff some system state occurs twice.

Proof: A straightforward generalisation of a theorem in [13]. ∎

A consequence of theorem 36 is that we can use our techniques for bounded model syntaxes in order to decide termination.

Corollary 37 *Bounded Rule Structure*
Termination is decidable for a 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 just look for repeating system states. ∎

5.2.2 Stack Pending Structure

Unlike a singleton or set, the stack is not a bounded structure. It is not necessarily a cause of undecidability though, as the following theorem shows.

Theorem 38 *Termination is decidable for a bounded model rule language and a stack pending structure.*

The rest of this section is devoted to proving this theorem. We begin with an example using the 0-1 rule language.

Example 39
r_1: On U(X_1) if true then null
r_2: On U(X_2) if true then X_1=1;X_3=1
r_3: On U(X_3) if true then X_4=1
r_4: On U(X_4) if true then $X_3 = 1$;$X_5 = 1$
r_5: On U(X_5) if true then X_3=1

priority(r_1)>priority(r_3)>priority(r_5)

The stack itself just contains a sequence of rules. After the execution of a rule, it is removed from the stack and possibly other rules added. Note that when a group of rules is added, because highest priority rules are put onto the stack first, the rule that ends up on top of the stack will be the one in the group with lowest priority. So suppose rule r_2 was triggered first, then the evolution of the stack would appear as follows (top of stack is at left) and iteration number on the right

r_2 **1**
r_3 r_1 **2**
r_4 r_1 **3**
r_5 r_3 r_1 **4**
r_3 r_3 r_1 **5**
r_4 r_3 r_1 **6**
r_5 r_3 r_3 r_1 **7**
r_3 r_3 r_3 r_1 **8**
r_4 r_3 r_3 r_1 **9**
r_5 r_3 r_3 r_3 r_1 **10**
$\vdots$

Note that the dbstate remains the same throughout. Looking at this example, it is clear that the stack is going to keep growing in an unbounded manner. Now, let us assign each rule on the stack a *global sequence number*. This describes the iteration at which this instance of the rule entered the stack. The evolution now becomes

r_{2_1} **1**

r_{3_2} r_{1_2} **2** **

r_{4_3} r_{1_2} **3**

r_{5_4} r_{3_4} r_{1_2} **4**

r_{3_5} r_{3_4} r_{1_2} **5** **

r_{4_6} r_{3_4} r_{1_2} **6**

r_{5_7} r_{3_7} r_{3_4} r_{1_2} **7**

r_{3_8} r_{3_7} r_{3_4} r_{1_2} **8** **

r_{4_9} r_{3_7} r_{3_4} r_{1_2} **9**

$r_{5_{10}}$ $r_{3_{10}}$ r_{3_7} r_{3_4} r_{1_2} **11**

$\vdots$

Consider the stacks labelled with a **. By examining the database state, the ordering of rules, their labels and their prefix and suffix structure, it is possible to deduce that a non terminating computation is occurring. To formalise and analyse this notion, we present some definitions. ■

Definition 40 Stack pstate
A stack *pstate* is a sequence of labelled rules (according to global sequence number) $x_1\ x_2\ \ldots x_n$ (*e.g.* $x_3 = r_{2_7}$). The rule on top of the stack is the first rule in the sequence. ■

Definition 41 Equivalence and Weak Equivalence
Suppose a_i and a_j are sequence elements, then we say $a_i = a_j$ if a_i and a_j both represent the same rule with the same global sequence number. We say $a_i \cong a_j$ if a_i and a_j both represent the same rule with possibly different global sequence numbers. If $s_1 = x_1\ \ldots\ x_n$ and $s_2 = y_1\ \ldots\ y_m$ are sequences, then $s_1 = s_2$ if $m = n$ and $x_i = y_i$. We can similarly define $s_1 \cong s_2$. ■

Example: $r_{4_2} = r_{4_2}$, $r_{4_2} \cong r_{4_2}$, $r_{4_2} \cong r_{4_4}$, $r_{4_2} \neq r_{4_4}$, $r_{4_2} \ncong r_{5_2}$

Definition 42 Strong Suffix Match
Let s_1 and s_2 be two stack pstates. The strong suffix match of s_1 and s_2 (denoted $ssm(s_1, s_2)$), is a new sequence s_3, which is the longest common suffix shared by s_1 and s_2. ■

Example: $ssm([r_{3_5}\ r_{3_4}\ r_{1_2}],\ [r_{3_8}\ r_{3_7}\ r_{3_4}\ r_{1_2}])=[r_{3_4}\ r_{1_2}]$.

Definition 43 Weak Prefix Match
Let s_1 and s_2 be two stack pstates. The weak prefix match of s_1 and s_2 (denoted $wpm(s_1, s_2)$), is a new sequence s_3, which is the longest possible weakly equivalent prefix that s_1 shares with s_2. ■

Example: $wpm([r_{3_5}\ r_{3_4}\ r_{1_2}],\ [r_{3_8}\ r_{3_7}\ r_{3_4}\ r_{1_2}])=[r_{3_5}\ r_{3_4}]$

Definition 44 Subtraction
Let s_1 and s_2 be two stack pstates where s_1=$x_1\ x_2\ \ldots x_n$ and s_2=$y_1\ y_2\ \ldots y_m$. Then s_3=$s_1 - s_2$ is the sequence $\perp$ if $ssm(s_1,s_2) \neq s_2$ else
$s_3 = x_1\ x_2\ \ldots x_k$ where
$k = n - m$ ■

Example: $[r_{3_5}]$ - $[r_{3_4}] = \perp$, $[r_{3_8}\ r_{3_7}\ r_{3_4}\ r_{1_2}]$ - $[r_{3_4}\ r_{1_2}] = [r_{3_8}\ r_{3_7}]$.

Definition 45 Containment
Suppose s_1 and s_2 are stack pstates. Then we say $s_1 \subseteq s_2$ if $s_1 - s_3 = wpm(s_1 - s_3, s_2 - s_3)$ where $s_3 = ssm(s_1,s_2)$. ■

Example: $[r_{3_2}\ r_{3_4}\ r_{3_6}\ r_{3_7}] \subseteq [r_{3_5}\ r_{3_7}\ r_{4_5}\ r_{3_6}\ r_{3_7}]$, $[r_{3_6}\ r_{3_7}] \not\subseteq [r_{4_6}\ r_{4_7}]$, $[r_{3_6}\ r_{3_7}] \not\subseteq [r_{3_6}\ r_{4_7}]$, $[r_{3_6}\ r_{3_7}\ r_{4_7}] \not\subseteq [r_{3_6}\ r_{4_5}\ r_{4_7}]$.

Definition 46 Let $t_1 = (db_1, ps_1)$ and $t_2 = (db_2, ps_2)$ be two system states for a stack rule system with t_2 occurring after t_1. We say that $t_1 \equiv t_2$ if $db_1 = db_2$ and $ps_1 \subseteq ps_2$. ■

The following lemma shows that the equivalence relation in definition 46 is a necessary and sufficient condition for repetition of stacks.

Lemma 47 *A rule computation will not terminate iff there exist two system states t_1 and t_2 such that $t_1 \equiv t_2$.*

Proof:

Let $t_1 = (db_1, ps_1)$ and $t_2 = (db_2, ps_2)$.

i) Show that $t_1 \equiv t_2 \Rightarrow$ non termination.

Executing the same rule in either db_1 or db_2 has equivalent effect since $db_1 = db_2$. Since $ps_1 \subseteq ps_2$ we can say $ps_1 = w_1.q$ and $ps_2 = w_2.w_3.q$ where $q = ssm(ps_1,ps_2)$ and $w_1 = wpm(ps_1,ps_2)$ and $w_2 \cong w_1$. Next, w_2 is executed and we will eventually get the stack pstate $w_5.w_4.w_3.q$ in db_1 where $w_5 \cong w_1$. The process then repeats again and we eventually get the stack pstate $w_7.w_6.w_4.w_3.q$ in db_1 where $w_7 \cong w_1$. This behaviour of having a $w_x \cong w_1$ executing in db_1 at each step is clearly going to cause infinite growth and we will get non termination.

ii) Show that $t_1 \equiv t_2 \Leftarrow$ non termination.

If non termination occurs, there must be some database state which occurs an infinite number of times. Call this db_a. Consider the stack *pstates* each time we reach db_a. If there are only a finite number of them, then we are done, since there will exist a pair ps_i and ps_j where $ps_i \subseteq ps_j$. Otherwise, suppose there are an infinite number of different stacks for db_a. Now for each occurring stack of length n, there exists a point in the execution sequence such that thereafter the stack never has length $\leq n$ (otherwise we would get stack repetition). Suppose we choose $n = k$, then there exists a last system state (db_a,ps_k) after which the stack is never smaller than k elements. This state will have the form $B_1.Q_1$ where $|Q_1| = k$ and $|B_1| \geq 0$. Since there is no shrinking, the sequence Q_1 will always remain on the stack. Also $|B_1| \leq max$ - 1 where max is the maximum number of rules which can be triggered by a single rule (easily obtainable by inspection of the rule set). Since the system is non terminating and the stack is growing infinitely, we can do this for an infinitely increasing n and we will obtain stacks of the form

$B_2.Q_2$

$B_3.Q_3$

$B_4.Q_4$

$\vdots$

where $Q_r \subseteq Q_{r+1}$. But there is a finite number of B_r's that can occur (its size is bounded by the value max). Thus at some point it must be the case there will be two stack pstates such that $ps_1 \subseteq ps_2$. Thus there exist t_1 and t_2 such that $t_1 \equiv t_2$. ■

Theorem 38 now follows from this lemma, since we can use the bounded model property to generate a finite number of databases for our simulation. For each database instance, we run our rule set on it and check whether the condition $t_1 \equiv t_2$ occurs. If it does, then report *non termination.* Otherwise we choose another instance. If none are left, then report *termination.*

5.2.3 Multiset Pending Structure

We now consider the multiset pending rule structure. This is somewhat similar to the stack structure we have just considered. We are again able to develop a necessary and sufficient condition for infinite behaviour. Termination is consequently decidable for bounded model rule languages using multisets.

Theorem 48 *Termination is decidable for a bounded model rule language and a multiset pending rule structure.*

The proof is similar to that of theorem 38. We begin with some terminology

Definition 49 Multiset pstate
A multiset *pstate* is a sequence $s_1, s_2, \ldots, s_n$ where each s_i is a sequence of instances of the same rule, labelled according to a global sequence number identifying the iteration when this rule first appeared and priority(s_{i+1})>priority(s_i) (abusing our notation slightly), *e.g.* $s_5 = r_{3_1}\ r_{3_2}\ \ldots\ r_{3_m}$. Rule selection is done by taking the highest priority rule with largest global sequence number. ■

We note that strictly speaking, rules should be selected only according to priority. However, imposing global sequence number criteria does not affect execution in any way, because different instances of the same rule always execute the same actions. It is done to allow us to treat each priority level in the multiset as a stack.

Definition 50 Containment
Suppose $m_1 = s_1, \ldots, s_n$ and $m_2 = t_1, \ldots, t_p$ are multiset pstates. Then we say $m_1 \sqsubseteq m_2$ if $s_1 \subseteq t_1$ and $\forall i \geq 2\ ssm(s_i,\ t_i)=s_i$. ■

Definition 51 Let $t_1 = (db_1, ps_1)$ and $t_2 = (db_2, ps_2)$ be two system states for a multiset rule system and t_2 occurs later than t_1. We say that $t_1 \equiv t_2$ if $db_1 = db_2$ and $ps_1 \sqsubseteq ps_2$. ■

The above definition is a necessary and sufficient condition for infinite execution in multiset rule systems.

Theorem 52 *A rule computation will not terminate iff there exist two system states t_1 and t_2 such that $t_1 \equiv t_2$.*

Proof (sketch): Let $t_1 = (db_1, m_1)$ and $t_2 = (db_2, m_2)$.
i) Show $t_1 \equiv t_2 \Rightarrow$ non termination

We can focus our attention on the top layer of the multiset, which contains instances of the highest priority rules. This is because $m_1 \sqsubseteq m_2$ implies that layers below this have not changed in the transition from t_1 to t_2. Using a similar argument to that in lemma 47, we can then argue that non termination must occur.

ii) Show $t_1 \equiv t_2 \Leftarrow$ non termination

If non termination occurs, there must be some database state which occurs an infinite number of times. Call this db_a. Consider the multiset pstates each time db_a occurs - if there is only a finite number of them, then we can find a pair m_i and m_j such that $m_i \sqsubseteq m_j$ and we are done. Otherwise, there must be an infinite number of different multiset pstates and there must be a priority level which grows infinitely. It follows that we must then be able to find a level which is infinitely increasing and also has highest priority in the multiset infinitely (because if we couldn't then our condition would be fulfilled). Call this level s_k. Now there must come a point at which everything below this level is never executed again and if not, we can find an s_p where priority(s_p)<priority(s_k) and choose this instead. Using this level, we can then apply a similar argument to that of lemma 47. ■

Decidability then follows in an analogous way to that in theorem 38.

5.2.4 Queue Pending Structure

We now examine the queue pending structure. This is more powerful than any of the possibilities we have seen so far. In fact, it is powerful enough to make termination undecidable.

Theorem 53 *Termination is undecidable for rules written in the 0-1 language using a queue pending structure.*

Proof: We will show that we can build a set of 0-1 triggers with queue to simulate any Post machine [46], a device which is as computationally powerful as the Turing machine. It 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 state. In each state one can move to another state after optionally reading and removing a symbol from the front of the queue and/or possibly adding an element to the end of the queue. The machine doesn't have a separate input tape unit, but rather the input string is initially loaded into the queue before execution. The machine halts when it 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 transition is of the form *(state, symbol, state', symbol')*

- *state* is the machine's current state
- *symbol* is the symbol on top of the queue
- *state'* is the new state the machine will go to
- *symbol'* is the symbol to place on the bottom of the queue

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	If V_p=1 and V_{flag}=1	If V_q=1 and V_{flag}=1
V_{flag}=1	Then V_p=0 ; V_{p_1}=1 ;	Then V_q=0 ; V_{q_1}=1 ;
	$V_{w_p} = 1$; V_{flag}=0	$V_{w_q} = 1$; V_{flag}=0

The variable V_{flag} ensures that only one of r_{a_p} and r_{a_q} is executed. Rule r_a resets 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}.
We also need a rule to empty the queue when an accepting state is entered. Its priority is less than that of r_a and larger than that of r_{a_p}.

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 rule triggered needs to have the form:

Rule $r_{a_{init}}$
On U($V_{initial}$)
If true
Then V_{accept}=0 ;V_s=1; $\forall i \in S$ $(V_i = 0)$;

where S={The set of all states other than START}
This ensures that we begin in the starting state. A 0-1 rule computation halts once the queue is empty. ■

This concludes our study of pending structures. In the next section, we consider some new mechanisms for controlling execution, that of *meta rules*.

5.3 Meta Rules

Meta rules are used for managing the behaviour of the set of active rules. We will consider three types of meta rules in this section.

- Complex events
- Coupling modes
- Dynamic priorities

5.3.1 Complex Events

Many active rule languages have a facility for specifying complex events. 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 [20].
The operator we will consider is the sequence operator. An event E=e1;e2 occurs if the event e1 followed by the event e2 occurs. The event consumption semantics we choose is a cumulative one and is intuitively 'match an e2 with each unconsumed e1 before it' (in real life this could correspond to tracking all deposits preceding a big withdrawal). Figure 5.1 illustrates this with six different occurrences of the event E. The numeric labels on the arcs indicate the complex event ordering *i.e.* 1 occurs before 2, 2 occurs before 3 etc. When multiple rules triggered by such an event are activated simultaneously (*e.g.* 1,2 and 3 are fired when the first e2 occurs), they are pushed onto the stack in order of most recent e1 firing (i.e 3 then 2 then 1).
Suppose we assume that our rule system has the power to recognise a complex event of the type just discussed. 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.

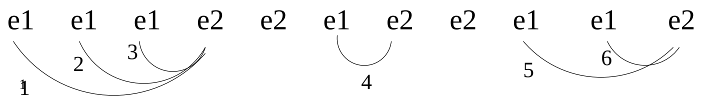

Figure 5.1: Cumulative Consumption Semantics

Theorem 54 *Termination is undecidable for a trigger system with 0-1 rule language, a stack pending structure and the cumulative event sequence operator.*

Proof (sketch): We show how it is possible to to use the complex event capability to simulate a second (virtual) stack. These two stacks are then sufficient to simulate the power of a queue.
Assume we define our rules as in theorem 53. For each rule in the rule set, we associate with it an identifier event. So for rule r_1 we have E_{r_1}, rule r_2 we have E_{r_2} etc. We also have a special event called E_{bottom}. Using these events, we create another set of rules that are activated on complex events of the type just discussed. For rule r_1 we would create another rule r_1' of the form

Rule r_1'
On $E_{r_1}; E_{bottom}$
if C_1'
then A_1'

where E_{r_1} is the triggering event for r_1 and C_1' and A_1' are the same as r_1's condition and action respectively.
Now suppose we wish to place rule r_5 on the bottom of the stack (to make it behave like a queue), but the stack already has a number of rules on it. We first set a variable called the consumption flag. It makes sure the condition of each rule in the stack is false and also causes an event to be fired when that rule is removed from the stack. This can be easily achieved by including a new rule for each package of rules as in theorem 53. *e.g.* There we defined the package $\{r_a, r_{a_{accept}}, r_{a_p}, r_{a_q}\}$ (let $r_{a_{empty}} = r_{a_{accept}}$). To this we add the rule $r_{a_{cons}}$ with priority less than $r_{a_{empty}}$ and greater than r_{a_p}. It will make the flag equal to zero and cause $E_{r_a}, E_{r_{a_{empty}}}, E_{r_{a_p}}, E_{r_{a_q}}$ to be fired. Similar events will be fired for all other packages on the stack. Once the bottom of the stack is reached (detectable by an appropriate marker rule which can be added in the START state), we do

several things. Firstly we fire the event E_{r_5} (since we want to put rule r_5 on the bottom of the stack), then we fire the event for the marker rule followed by event E_{bottom} and lastly we turn consumption mode off. The firing of E_{bottom} causes all the rules with complex events defined above to be placed back on the stack in reverse order of the firing of their E_{r_k} event. Thus the stack is the same as it was originally, but with rule r_5 on the bottom. Consequently we can simulate a queue and by theorem 53, undecidability then follows. ■

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 [25]. These can be thought of as another type of meta rule. Each rule can be triggered in either of two modes.

- *Immediate:* The rule is placed into the pending rule structure immediately after the event occurs.
- *Deferred:* If a rule is triggered, then it is placed into the pending rule structure only after the structure has become empty (until which time the rule can be thought of as occupying a separate "deferred" pending structure). Deferred mode is used in actual systems to postpone rule execution until the end of a transaction, just before the commit phase.

In our semantics described in Chapter 1, we effectively assumed immediate coupling mode for all rules. If we relax this restriction, then we can get increased power which results in undecidability.

Theorem 55 *Termination is undecidable for a trigger system with 0-1 rule language, a stack pending structure and the option of immediate and deferred coupling modes for all rules.*

Proof (sketch): As in Theorem 54, we will show how a queue can be simulated by demonstrating how to place a rule at the bottom of the stack. First, suppose that for every rule we define two variants, one in immediate mode and one in deferred mode. Assume also, that we have a flag indicating whether we currently want to activate rules in deferred mode or in immediate mode. Suppose we want to place rule r_5 at the bottom of the stack, then we set the flag to deferred, which will ensure that every rule on the stack is reactivated in deferred mode and then removed. Thus the stack will be emptied and conceptually we'll have a new stack containing all the rules activated in deferred mode, but with their original order reversed. We now activate rule r_5 in deferred mode (we can set a flag to remember to do this as soon as we reach a marker rule indicating the bottom of the stack) and then carry out the deferred activation process once again, for each element on the stack. We then reset the flag to indicate immediate mode. We now have a new stack with the order as it was initially and rule r_5 on the bottom. ■

5.3.3 Dynamic Priorities

We now examine the effect which the priority mechanism can have upon the termination behaviour of a rule system. The first thing to note is that using a multiset structure is somewhat like having unary relations with bag semantics. The critical difference is, however, that we have only a restricted access to these unary bags. If we now slightly increase the access power to these bags using dynamic priorities, termination will become undecidable.

The specific extension we make is to allow rule actions to alter the total order on priorities. *i.e.* A rule can execute the statement $r_i > r_j > r_k \ldots$ to enforce the condition that henceforth the priority of r_i is greater than that of r_j, which in turn is greater than that of r_k etc. Note that such a statement must include all defined rules.

Theorem 56 *Termination is undecidable for a 0-1 rule language with multiset pending rule structure and dynamic priorities.*

Proof(sketch): The proof is similar to that in theorem 53, though this time we are simulating a two counter machine instead of a Post machine. We can simulate the states of the machine by 0-1 variables. A counter is simulated by instances of a particular rule.

The multiset structure we will use consists of three distinct strata, with rules in higher strata having greater priority than those in lower strata.

1. *Transition Strata* - contains rules which examine the current state and decide whether to move to a new state or whether to inspect or change a counter. These rules are $r_{t_1}, \ldots, r_{t_n} = \overline{r_t}$.

2. *First Counter* - contains as many instances of r_{c_1} as the value of the first counter

3. *Second Counter* - contains as many instances of r_{c_2} as the value of the second counter

Initially priority($\overline{r_t}$) > priority(r_{c_1}) > priority(r_{c_2}) and the state of the multiset is

$$\begin{array}{c} M_0 \\ \overline{r_t}\ (8) \\ r_{c_1}\ (7) \\ r_{c_2}\ (5) \end{array}$$

where the rightmost number indicates priority level. Recall that a two counter machine has the following form

$$\delta :\ S \times \{=,>\} \times \{=,>\} \rightarrow S \times \{pop, push\} \times \{pop, push\}$$

Transition rules can be in the following modes

1. Getting the current state.
2. Testing whether counter 1 is greater than zero.
3. Testing whether counter 2 is greater than zero.
4. Incrementing counter 1
5. Incrementing counter 2
6. Decrementing counter 1
7. Decrementing counter 2
8. Moving to a new state

Accomplishing 1 and 8 is easy. We now sketch how to do 2-7.

M_1 :	M_2	M_3	M_4
r_{c_1} (7)	$\overline{r_t}$ (6)	r_{c_2} (5)	$\overline{r_t}$ (3)
$\overline{r_t}$ (6)	r_{c_2} (5)	$\overline{r_t}$ (3)	r_{c_1} (2)
r_{c_2} (5)	r_{c_1} (2)	r_{c_1} (2)	r_{c_2} (1)

Step 1: To check the value of the first counter requires access to the r_{c_1} layer. This is done by reactivating copies of the transition rules at a priority lower than the first counter. See M_1.

Step 2: The first counter is now on top. It is then straightforward to check whether it is greater than zero by executing all the r_{c_1} rules and have them reactivate copies of themselves at a lower priority. An appropriate flag can be set indicating whether an r_{c_1} actually executed, and this information can be used to guide our state transition. See M_2.

Step 3: Now if we wish to check the value of the second counter, we can again reactivate copies of the transition rules at a lower priority. See M_3.

Step 4: Since the second counter is now on top, we can now execute these rules and reactivate them at lower priority. See M_4.

Step 5: We have now returned to the same relative ordering of priority strata we had initially, the difference now being that the absolute priority levels are lower. Obviously we could not keep on repeating this process indefinitely, since we do not have an infinite number of rules/priorities we can use. The solution is to use our dynamic priority modification ability to return to the priority levels we were using initially. The transition rules now activate two reordering rules r_{o_1} and r_{o_2} to accomplish this (see M_5).

M_5	M_6	M_7	M_8
r_{o_1} (10)	r_{c_2}(1)	r_{o_2} (4)	$\overline{r_t}$ (8)
r_{o_2} (4)	r_{c_1} (2)	r_{c_2} (5)	r_{c_1} (7)
$\overline{r_t}$ (3)	$\overline{r_t}$ (3)	r_{c_1} (7)	r_{c_2} (5)
r_{c_1} (2)	r_{o_2} (4)	$\overline{r_t}$ (8)	
r_{c_2} (1)			

Step 6: Now $r_{o_1} = 1 > 2 > 3 > 4 > 5 > 6 > 7 > 8 > 9 > 10$ executes, inverting all the priorities (see M_6)

Step 7: We then get r_{c_2}, r_{c_1} and r_{t_i} to reactivate copies of themselves at their original priority level. See M_7.

Step 8: Rule $r_{o_2} = 1 > 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10$ then executes and we return to $M_8 = M_0$ with the original priority ordering.

We have thus shown how checking counters can be accomplished. Incrementing the value of a counter is done by adding another instance of the particular counting rule to the multiset. Decrementing is more complicated, since we need to remove one instance of a rule and keep all the others. It is possible, though, to use a similar technique to that we have shown for checking the value of a counter.
We have not outlined the rules which need to appear in $\overline{r_t}$, but using the same kind of approach as in theorem 53, we can effectively write a statement such as

if c_1 then a_1 else
if c_2 then a_2 else
if c_3 then a_3 else
$\vdots$

with the use of rule packages and this will be sufficient for the simulation. ■

5.4 Related Work

The work in this chapter has been based upon that in [9]. The emphasis, however, is somewhat different. In [9], 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've examined.

In [44, 45], 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 [35], 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 [36], methods for specifying meta rules to manage execution of the rule set as a whole are presented. Although we also consider meta rules, 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 [50], where the property of confluence in the presence of meta rules is examined.

5.5 Future Work

There is clearly a great deal of scope for future work in this area. Many more kinds of control features can be considered, but the challenge is to find ones which are interesting and relevant to active databases. Some possibilities are to investigate

- The effect of changing our semantics to instance based instead of set based.
- The effect of allowing rules to inspect to delta relations.
- The effect of further types of "useful" meta rules.
- The relationship between the systems presented and various "exotic" grammars, such as ordered grammars, timed grammars and grammars with control rules.
- The language expressiveness for rules using stacks or multisets.

5.6 Summary

Our results are summarised in table 5.1. All the decidable results are for rule languages with the bounded model property. All the undecidability results are for the 0-1 rule language and so consequently hold for more expressive languages.

Bounded Structure	Decidable
Stack	Decidable
Multiset	Decidable
Stack + Complex Events	Undecidable
Stack + Coupling Modes	Undecidable
Multiset + Changing Priorities	Undecidable
Queue	Undecidable

Table 5.1: Decidable and Undecidable Configurations

Looking at the table, it is clear that control structures play a vital part in the decidability of termination. Indeed, our results suggest that even approximate termination analysis is likely to become far more difficult to do precisely for complex configurations. Consequently, the use of simple features may need to be mandatory, if rule analysis is to be a useful tool for system builders and administrators.

Chapter 6

Abstract Interpretation

6.1 Introduction

In our presentation so far, we have focused on the question of whether termination is decidable for a particular rule system. The decision procedures we have developed, however, have a very high complexity and thus are not likely to be suitable for situations with large parameters (such as relation arity). Furthermore, there is also the problem of what to do with the undecidable cases. Previous work on static analysis [7, 15, 49, 22, 55, 12, 37] has approached these difficulties by developing sufficient conditions for a rule set to be non-terminating, but has often suffered from a lack of precision. This motivates our presentation of a new algorithm for approximate termination analysis, based on abstract interpretation, which goes beyond the checking of syntax. Abstract interpretation is a well-understood, semantics based method for static analysis. Once the underlying semantics has been captured formally, a variety of analyses can be derived, almost for free, as *approximations* to the semantics. The analysis propagates sophisticated dataflow information about calling patterns and how different rules interact. An important advantage of the technique is that it can be used for examining a wide variety of rule properties. We apply our method to the particular problem of rule termination and are able to obtain more precise answers for many types of rule sets.

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

6.2 Approximate Termination Analysis

Since previous work has considered only *global* termination, we will restrict the discussion to this from now on. Also, since we are now concerned with approximate analysis, we need some new terminology.

Definition 57 Termination Algorithm Precision
A termination algorithm can return three types of results.

- Definitely terminating
- Definitely non terminating
- Don't know

The *precision* of a termination algorithm, is defined as the value "1 - (the probability that a 'Don't know' result is returned)". ■

Ideally, we would like any algorithm we use to have best possible precision. For decidable rule languages, this equals one and for undecidable rule language it is less than one. We would also like it to have low complexity. But as we have shown, even for basic decidable rule languages, the problem is NP-hard (theorem 23). Partly because of this difficulty, previous algorithms have tended to compromise on precision, but have low complexity. They have typically been based on syntactic analysis [7], local semantic analysis [15], or a combination of both [12]. One of the first studies of active rule termination analysis was due to Aiken, Widom and Hellerstein [7] who introduced an abstraction called the *triggering graph*. In a triggering graph, the rules are represented by vertices and the possibility that execution of one rule can cause the execution of another rule is captured by a directed arc between two vertices. An acyclic triggering graph implies definite termination of a set of rules. This idea was refined by Karadimce and Urban [37]. Baralis and Widom [15] suggested another type of graph, the *activation graph*. Again rules are represented by vertices, but this time an arc indicates that the action of one rule may change the condition of another rule from *false* to *true*. Here too, acyclicity means definite termination, but generating the graph is non-effective. The precision of this method is heavily dependent upon an assumption that most rules will be *self disactivating* (have their action falsify their condition). As an extension, Baralis *et al.* [12] presented the *rule reduction method*, combining the use of triggering and activation graphs. Any vertex which does not have both an incoming triggering and activation arc, can be removed from the graph (along with its outgoing arcs). If this procedure results in all vertices being removed, then the rule set must terminate.

Van der Voort and Siebes [49] consider the termination problem in an object-oriented rule system. The focus is on deciding termination in a fixed number of steps. The proofs make use of typical database states and the reasoning has some resemblance to a dataflow style of analysis. The general termination problem is not discussed, however, and details of the algorithms are unclear.

In [56], an algorithm based on Petri nets is presented for analysing termination. It is essentially a generalisation of [12], since it relies on activation and triggering notions, but can also accurately take into account priorities among rules. Being based upon the notion of reachability markings, it's worst case complexity is high.

In [51], an analysis method for OO rules is presented. It's essentially a refinement of the triggering graph method, except with an additional optimisation that candidate cycles are examined more closely and if the net action of a cycle is a monotonic one and there is a bound on the domain, it may be possible to eliminate it.

Another paper by Baralis *et al.* [13], describes a method for conducting termination analysis at run time. A history of states is maintained during rule execution and loop detection is done by looking for repeated histories. The algorithm presented in this chapter, although static, does have features in common with this dynamic method, since we are effectively "running" the rule set, albeit in an approximate manner.

A higher level approach to analysis is described in [14], where methods for combining termination knowledge about modules of rules is outlined. The technique still requires, however, lower level algorithms (such as triggering/activation graphs), to verify the termination of the individual modules.

An alternative approach to rule termination is that of designing languages and semantics which enforce termination upon all rule executions. This is discussed in [41, 55, 22].

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

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

Table 6.1: Approximating multiplication (left) and the *even* test (right)

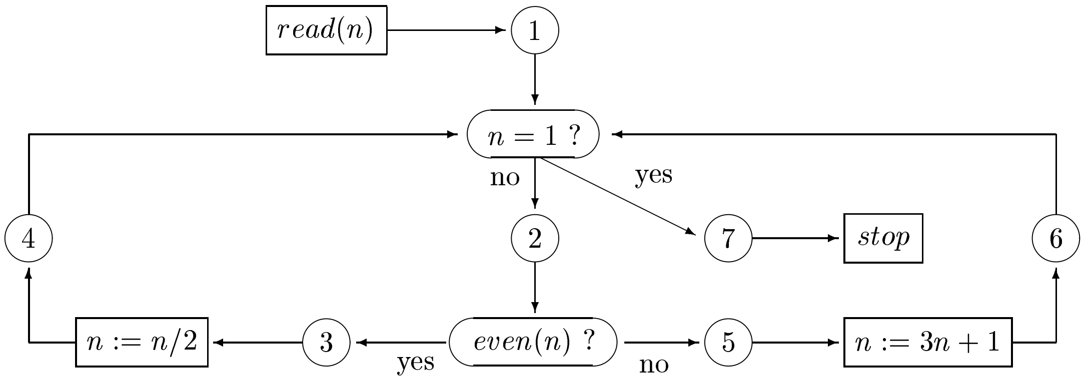

Figure 6.1: A flow diagram for the $3n + 1$ example

6.3 Abstract Interpretation

Abstract interpretation has proven a useful tool for analysis of imperative, functional and logic programs [6, 24, 42]. It has numerous applications in compiler optimisation and program transformation. The idea is to infer and collect information about a program's run time properties and its operational aspects. This information is known as the collecting semantics of a program.

The semantic domain of a program is approximated by an "abstract" domain, essentially a coarse grained version of the semantic domain. The operations of a program are approximated by abstract operations. Consider a program to be analysed whose semantic domain is $\mathcal{Z}$, the set of integers, and whose operations are the arithmetic operations $\{+, -, *, /\}$, the comparison functions, $\{<, >, =\}$, as well as tests $\{odd(\cdot), even(\cdot)\}$. The semantic domain could be approximated by descriptions $\{none, even, odd, any\}$ and the arithmetic operations could be approximated according to the well-known "rules of parities". For example, Table 6.1 shows how multiplication is approximated (left table), as well as the approximation of the test *even* (right table).

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

Table 6.2: Approximate computation to a fixpoint from an initial state

Now consider the approximate computation of the flow diagram in Figure 6.1. Each arc of the diagram corresponds to a program point, and is labelled with a number. The computation involves iterating towards a fixpoint. The iteration proceeds by following arcs from the goal and updating program points until the values at each point have stabilised. Table 6.2 shows how the states iterate towards a fixpoint assuming n is *any* at point 1. The last step yields a fixpoint. The result of the analysis justifies transformation of the program so as to avoid a number of tests; for example, the statement $n := 3n + 1$ can be replaced by $n := (3n + 1)/2$ by noticing that the value of n at point 6 is always *even*.

Even though the actual computation may never terminate, the approximate computation always terminates.[1] This is because the analysis is monotone with respect to the natural ordering of the descriptions, namely $d \leq d'$ iff $(d = none) \vee (d = d') \vee (d' = any)$. Annotations are only allowed to "grow" with respect to that ordering. In particular, where two or more arcs run together, the result is taken to be their least upper bound. We return to this point in Section 6.5.

6.4 A Collecting Semantics for Active Rules

We use denotational semantics to formally specify the execution of our active database model as a fixpoint computation. Analysis then consists of finding an approximation to the least fixpoint.

In our description, we have made the choice that the pending rule structure should be a set, this allows quite complex behaviour, without making the analysis too complicated. We now give a denotational definition of the execution model. It is

[1]Uniform termination of the flow diagram is the famous $3n + 1$ problem, or Collatz's problem, still unsolved.

similar in spirit to those of Widom [52] and Coupaye and Collet [23].

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

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

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

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

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

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

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

For analysis purposes we are interested not only in the result of the computation, but also in all the database states that are obtained at intermediate points in the execution. For this reason, the denotational definition is changed into a so-called *collecting semantics* [24, 42]. The idea is to use a history to record all the possible combinations (set of pending rules, database state) that occur. The domain $Hist = \mathcal{P}(Rule) \to \mathcal{P}(Dbs)$ is used for this. For example, if at some stage we encounter a set R of pending rules in a current database state d, then the entry for R will be updated to include d. Notice that the rule r which would have been activated next is the rule in R which has the highest priority. This leads to the following definition of a collecting semantics: $\mathcal{C} : \mathcal{P}(Rule) \to Dbs \to Hist$ is defined by

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

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

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

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

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

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

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

6.5 Approximating the Collecting Semantics

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

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

In the $3n + 1$ example in Section 6.1, we dealt informally with the notion of approximation. More formally we can specify the meaning of *none*, *even*, *odd*, and *any* by mapping each to the set of integers it stands for:

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

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

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

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

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

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

$$u \ appr \ v \iff v \leq \gamma(u).$$

There are standard ways of extending this notion to composite domains, such as Cartesian products and function spaces [42].

We can now employ a powerful correctness theorem [24]. Assume we have two chains, $\{u_i\}_{i \in I}$ and $\{v_i\}_{i \in I}$, such that u_i *appr* v_i for each $i \in I$. Then $\sqcup_{i \in I} u_i$ *appr* $\sqcup_{i \in I}$ v_i. In this sense, local correctness automatically implies global correctness. If we guarantee correct manipulation of approximations by all basic operations, then we automatically approximate least fixpoints correctly. Moreover, the design of approximate operations does not need to be *ad hoc*. For each operation $f : V \to V$ on the concrete domain, the Galois connection determines a unique best approximation $f' : U \to U$, namely $f' = \alpha \circ f \circ \gamma$.

We now make these points more specific, by considering example concrete and abstract domains. The rule language we consider is similar to the 0-1 language that we considered earlier, but with a slightly different action structure. Variables can range over $\mathcal{Z}$.

Definition 58 N Variable Language

- An event is of the form $U(X)$ which means "update variable X." It is triggered by any assignment to X.
- A condition is a conjunction of simple conditions. A simple condition is a test of the form $X = n$ or $X = Y$, where X and Y are variables and n is an integer.
- An action is a sequence of simple actions. A simple action is an assignment of the form $X{:=}Y$ or $X{:=}n$.

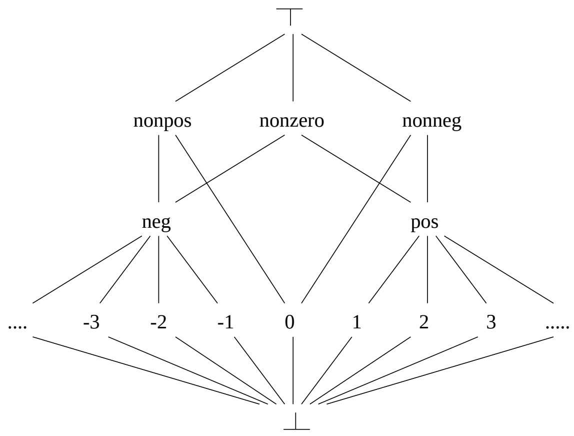

Figure 6.2: The abstract domain $Adom$

■

As an example abstract domain let us use the set

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

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

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

gives the meaning of elements of $Adom$. The domain's Hasse diagram is shown in Figure 6.2. The ordering is induced by the subset ordering on $\mathcal{P}(\mathcal{Z})$.
The collecting semantics is approximated in two steps, as indicated in Figure 6.3. Assuming a finite set Var of database variables, a database state is a finitely based function in $Dbs = Var \to \mathcal{Z}$. First, the domain $\mathcal{P}(Dbs)$ used for collecting histories is replaced by $Var \to \mathcal{P}(\mathcal{Z})$. This involves an approximation by which the set of functions F is approximated by the single function f' defined by

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

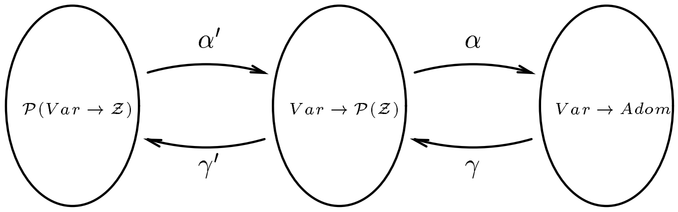

Figure 6.3: The two steps needed to approximate sets of database states

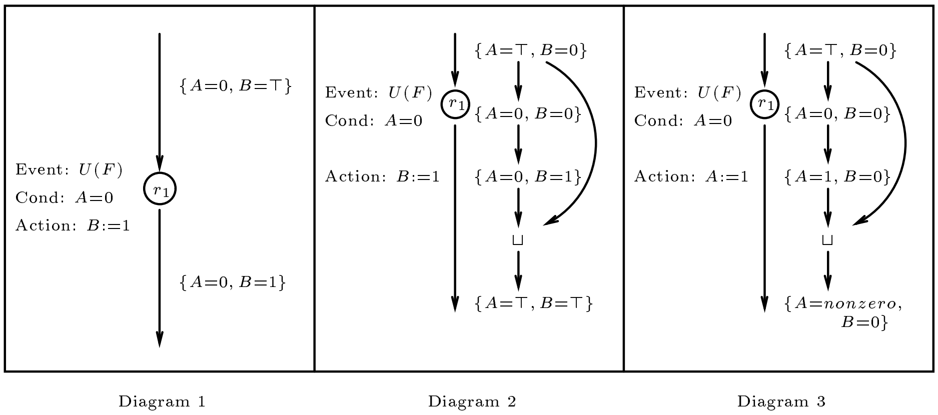

Figure 6.4: Approximate execution of rules

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

Example 59 Let us consider the approximate execution of rules. Recall that a rule is seen as a database state transformer. The meaning of a rule here has type $(Var \to Adom) \to (Var \to Adom)$ and a denotational definition is straightforward. We omit the details, giving instead, in Figure 6.4, some examples of "approximate" rule behaviour.

Diagram 1 shows the result of executing a rule. In the initial database state, nothing is known about B, while A is 0. This is sufficient to guarantee that the condition must be *true* and the action $B{:=}1$ is executed, with the resulting database state being $A = 0$ and $B = 1$.

Diagram 2 considers the case where a rule's condition involves an indefinite variable

A and so the condition could be either *true* or *false*. We proceed by case analysis. In case $A = 0$, action B:=1 results in a database state which has $A = 0$ and $B = 1$. Alternatively, $A \neq 0$ and no action is taken, so $A = nonzero$ and $B = 0$. The least upper bound of these two (the junction labelled '$\sqcup$' represents this operation) is the state $A = \top$ and $B = \top$.

Diagram 3 shows an example of a *self-disactivating* rule. The rule r_1 makes its own condition *false*. The resulting approximate database state is sufficiently precise that we can detect self-disactivation in this case. It is obtained as the least upper bound of $\{A = 1, B = 0\}$ which is the state resulting if the rule does execute and $\{A = nonzero, B = 0\}$ which is the state if it doesn't, The latter is a *sharpening* of the original $\{A = \top, B = 0\}$. Sharpening is an improvement used to incorporate the extra knowledge that a condition (in this case $A = 0$) was false. ■

The above examples illustrated the approximate execution of rules. Cases where we perform tests or assignments of the kind $A = B$ are handled in a similar way. The result of executing an entire set of rules is determined simply by approximating the collecting semantics. In particular, the function that updates the "history" remains the same except that its type is $\mathcal{P}(Rule) \to Dbs' \to Hist' \to Hist'$, where $Dbs' = Var \to Adom$ and $Hist' = \mathcal{P}(Rule) \to \mathcal{P}(Var \to Adom)$.

6.6 Termination Analysis

As mentioned in definition 57, a termination detection algorithm when given a rule set to analyse, will typically return either an answer that it *definitely* terminates or an answer that it doesn't know. The approximate semantics developed so far can be used for this type of analysis. Essentially, if, during approximate execution, we never encounter the same (pending rule set, approximate database state) combination twice, then the rules must terminate. Only finitely many such combinations are possible, in spite of *Adom* being infinite, since rules are unable to create new constants. Ways to relax this restriction are examined in section 6.8.

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

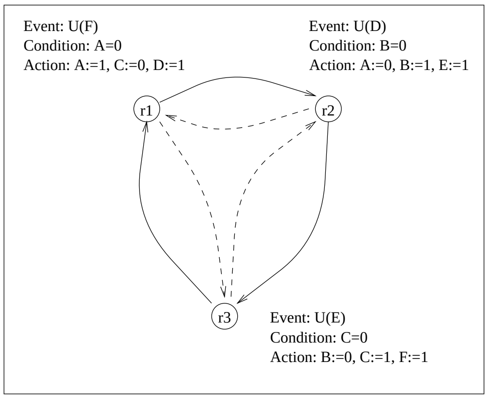

Figure 6.5: Triggering/activation graph

Example 60 This example of non-terminating behaviour is from [12]. Figure 6.5 shows the rule system. A solid arc from one rule to another indicates that the first rule's action can trigger the event of the second rule (the *triggering graph* [7]). For example, there is a solid arc from r_3 to r_1 because r_3 contains the action F:=1 which causes the event of r_1 to trigger. The dashed arcs represent the *activation graph* [15]. Thus, there is a dashed arc from rule r_2 to rule r_1 because r_2 contains the action A:=0 which can make the condition of r_1 true. The triggering and activation graphs are used in the *rule reduction algorithm* [12]. In this example the rule reduction algorithm cannot eliminate any rules and so the rule set is deemed possibly non-terminating. This is a precise statement since this rule set is non-terminating in general (for example, if A and B are both 0 initially and r_1 is triggered).
In Table 6.3, trace 1 gives the trace of execution for a method based on abstract interpretation. After six steps our algorithm detects a repeating state (3 and 6) and possible non-termination is signalled. ■

Example 61 Now change r_2 slightly, adding C:=1 to its actions. Trace 2 shows the trace of the resulting computation. Our algorithm signals termination after two steps because in step 2, $C \neq 0$. This is more precise than the rule reduction algorithm, which still returns possible non termination due to the triggering/activation graph remaining the same as in Figure 6.5. Our algorithm yields a better result because it is able to mimic the dataflow of the real execution. Note that we only show traces assuming that r_1 was the first rule activated. In practice it would be necessary to give additional traces for when r_2 or r_3 is initially activated. ■

trace 1

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

trace 2

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

Table 6.3: Traces of execution

Note that our traces have assumed an initial pending set. This means that given a set of rules, we would conceptually have to analyse them multiple times, giving a different rule as the initial rule triggered each time. This is not a problem, since in practice, it is usually obvious which rules are the "interesting" rules. Also, if we are aware of constraints on the initial state, then these can be easily integrated into the algorithm and we can expect more precise answers than the graph based methods.

Figure 6.6 describes our algorithm. It follows the outline provided by the collecting semantics, but additional features have been added to obtain a precise termination analysis.

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

1. The function *must_satisfy* takes an approximate database state d and a condition c and returns true iff c holds for *every* database state approximated by d and the selected rule is not marked. If it returns true then the function *eval* is called to yield the state that results after the rule's action has been executed.

2. The function *may_satisfy* takes the same arguments and returns true iff c holds for *some* database state approximated by d or a rule is marked. In general we should then take the least upper bound of two possible results: d itself (if r is not activated) and $eval(action(r), d)$ (in case r is activated). However, we can improve on this: If r was the only rule pending, then we can

```
For each rule r_0 do
 For each set R do h[R] := ∅
 For each variable V do d[V] := ⊤
 R = {r_0}
 While R ≠ ∅
  if d ∈ h[R] then { (* we have met this state before *)
   report "rules may not terminate"; exit }
  else { h[R] := h[R] ∪ {d} }
  r := select(R)
  R' := triggered(r) ∪ (R \ {r})
  if must_satisfy(d, cond(r)) then { d := eval(action(r), d) }
  else if may_satisfy(d, cond(r)) then {
   if R = {r} then { d := eval(action(r), eval(condition(r), d));
                R' := triggered(r) }
   else {d := sharpen(cond(r), d) ⊔ eval(action(r), d);
     R' := mark(triggered(r)) ∪ (R \ {r}) } }
  else { R' := R \ {r} } (* must not satisfy *)
  R := R'
 end While
end For
report "rules must terminate"
```

Figure 6.6: Abstract Interpretation Algorithm

assume r's condition must be true, since if it wasn't then we would terminate. This means we need only apply the action to the state that would result if the condition was true (*i.e.* $eval(action(r), eval(condition(r), d)))$ Furthermore, knowing that d does not satisfy the condition of r can sometimes allow a *sharpening* of d before taking its least upper bound with $eval(action(r), d)$. The function *sharpen* is for this purpose. For example, with $d(A) = \top$ and the condition $A = 0$, we know that $d(A) = nonzero$ is correct when the test fails. For correctness reasons, sharpening is constrained to only have an effect on rules which are not marked, otherwise it behaves like an identity function. A rule is marked if triggered as the result of a *may_satisfy* rule.

3. In the remaining case, the rule will definitely not be activated, so it is simply removed from the set of pending rules. No new rules can be triggered.

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

Theorem 62 *The abstract interpretation algorithm always terminates.*

Proof: The only way our algorithm could be non terminating is if the While loop was executed infinitely. This couldn't happen though, since execution uses only a finite number of constants and thus at some iteration there would a repeated state detected and the loop would be exited. ■

Theorem 63 *If our algorithm signals that the rule set is terminating, then the rule set terminates for any initial database state and triggering event.*

Proof: See Appendix.

6.6.1 Precision of the Algorithm

Simple Graphs

The precision of the algorithm is dependent upon the structure of the triggering graph. Since we take least upper bounds when merging alternative execution paths, the higher the branching factor, the less precise the results. This suggests that for *simple* graphs (ones where a rule can trigger at most one other rule), our precision should be higher. In fact, if we excluded $A = B$ type actions and conditions, then our algorithm would be a decision procedure for these kinds of graphs.

In the case of non-simple graphs, it is possible to achieve greater precision by modifying our algorithm to keep a disjunction of variable dependencies. The reason our precision is good for simple graphs is partly due to the nature of our rule actions. Recall definition 58 which limits us to performing no more than one condition test inside a rule. This contrasts with some of the languages we have already considered, where a rule can contain multiple updates and thus could effectively be testing several conditions (*i.e.* the right side of an update). If we were to make it possible for a rule to have several *if-then* statements, then this would mean that least upper bounds might have to be taken multiple times inside rules and so precision could be degraded even for simple graphs.

Rule Reduction Algorithm Comparison

We now compare our algorithm with the rule reduction method for the N variable language We first state a lemma.

Lemma 64 *For a rule set written in the N variable language with simple conditions and having a simple triggering graph, every rule that is eliminated by the rule reduction algorithm is executed a finite number of times and without state repetition by the abstract interpretation algorithm.*

Proof: For the rule reduction method to remove a rule, either it has no incoming triggering arc or no incoming activation arc. In case of no incoming triggering arc, no other rule can trigger it and thus our algorithm will never be able to consider it for execution. In case of no incoming activation arc, the rule must be self-disactivating and no other rule's action can make it true. Since the conditions are simple, sharpening will detect self-disactivation and hence our algorithm executes it a finite number of times and without state repetition. ■

We now claim that the abstract-interpretation-based algorithm is at least as precise as the rule reduction method for simple conditions (recall definition 58) and simple triggering graph.

Proposition 3 *The abstract interpretation algorithm, applied to the N variable rule language, is uniformly at least as precise as the rule reduction method provided the rule set only contains simple conditions and has a simple triggering graph.*

Proof: To prove this, we only need to show that it signals termination whenever the rule reduction algorithm reports termination. Suppose the rule reduction algorithm reports termination. This implies it is able to reduce the input set of rules to the empty set. This means that every rule in the rule set is eventually eliminated. By Lemma 64, the abstract interpretation algorithm will eventually be unable to execute any rule and thus it signals termination. ∎

6.7 Complexity

We now consider the complexity of our algorithm. The input consists of k variables all having the value $\top$ and suppose the rules refer to q constants. The rules are run until they either terminate or we detect a loop. The number of rule executions this may take is $\mathcal{O}(q^k)$, since it is possible we visit every possible state before detecting a loop. Using the upper bound result from theorem 23, the amount of space used is polynomial in k.
A brute force decidability algorithm which runs the rules on all possible database instances would in the worst case take time $\mathcal{O}(q^k \times q^k) = \mathcal{O}(q^2k)$, which is longer than abstract interpretation, but still in the same complexity class. It can also be done in polynomial space.

The rule reduction method is essentially based around triggering graph cycle checking and this is linear in the number of rules. Since our algorithm has the same complexity order as the brute force approach and is much greater than that of the rule reduction algorithm, in the next section we investigate ways in which it can be reduced, provided we are willing to sacrifice some precision.

6.7.1 Inflationary Semantics

The extension necessitates we alter the way we update our history of database states. Instead of simply adding a new state to the set of previous states, we only record one entry for each pending set, and incrementally refine its value by taking the least upper bound with new states as they arise[2]. This will cause loss of precision in our analysis, but guarantees termination. The advantage is that termination and correctness of this algorithm come almost for free, due to the

[2]We will need to regard the first state as $\perp$ instead of $\top$ for this to be useful. We also need to alter our semantics so that an update is only registered if the value of a variable is changed by a rule

nature of the framework (recall section 6.5 where local correctness implies global correctness). Also, by more complex choices of abstract domain, we can regain precision at the expense of efficiency. Indeed this alteration can be thought of as enforcing a weak kind of *inflationary semantics* upon rule execution, since a variable's value can only change monotonically and a finite number of times. We benefit in terms of complexity from this since the algorithm now takes polynomial time. *i.e.* Given n variables and l levels in abstract domain lattice. We can only alter a variable's value up to l times. Therefore, our program reaches a fixpoint in $\leq l \times n$ iterations. For constant l , this is $\mathcal{O}(n)$.

6.8 Extending the Rule Language

We now look at ways our algorithm can be extended to more complex rule languages. The first of these, is one for which termination is undecidable.

6.8.1 Arithmetic

We extend the N variable language (definition 58) such that it is now is possible to increment variables and use inequality tests. The value of a variable X is approximated by an interval $[a, b]$ where $a \leq b$. Assignment statements are treated using interval arithmetic (*e.g.* $[i, j] + [k, l] = [i + k, j + l]$). Similarly, tests are done using an interval logic. For example, if our approximation to X is $[a, b]$ and the condition is $X \geq 100$, then we take the intersection $[a, b] \cap [101, \infty]$. If this intersection is empty we know that the condition is definitely *false*. If $[a, b]$ is a subset of $[101, \infty]$ then the condition is definitely *true*. Otherwise the condition is possibly *true*. We define the ordering on the domain of intervals to be the obvious subset ordering ([a,b] $\leq$ [c,d] iff c $\leq$ a $\wedge$ b $\leq$ d).

In this case, we have to be careful as to how we detect termination, since the same state may never be repeated. This is because a variable may be associated with an ever increasing interval. Thus, we need to limit the domain somehow to make the number of states finite. For instance, we could say that a variable whose interval is greater than [5,∞] is treated as the top value [-∞, ∞]. The difficulty is how to choose this 'magic' value of [5,∞]. If it is too small, then we can precipitately signal possible non-termination, since we may prematurely reach a fixpoint. One possible way would be to examine all constants which appear in the rules and take the maximum. It is possible, though, the rules may be constant free. In such a situation, we would need to arbitrarily select a value. Another possibility is to use techniques such as *widening* [24] and dynamic partitioning [17]. The power of these approaches is the subject of future research.

6.8.2 Relational Syntax Extensions

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

Theorem 65 *Consider a rule set R written in a bounded model rule language. We can construct a rule set R' of rules written in the 0-1 rule language which is non-terminating iff R is.*

Proof(sketch): It is possible to finitely enumerate the tuples that could occur. The presence of a tuple can be signified by '1' and its absence by '0'. The truth value of a relational condition can be determined by testing the values of the appropriate 0-1 variables, and the effect of a relational action can be captured by appropriate assignments. ■

Therefore, in principle, if we have a bounded model rule language, we can use our abstract interpretation method to analyse properties in a rule system which uses relational algebra. Unfortunately, the complexity of this procedure will be prohibitive, since a huge number of variables(constants) could be required. Therefore, we now sketch ways in which the complexity might be reduced. The first thing to note is that we only need to simulate relations which are used by rules, the rest can be ignored. Second, if we are analysing for termination, then as a first pass one of the graph based methods such as the rule reduction method can be used to highlight which rules *may* be part of cycles, all other rules can be disregarded. The abstract interpretation method can then be used to conduct a more refined termination analysis of the relevant rules. Third, if we have information on some semantic properties of the data, such as functional dependencies and constraints, this can be used to eliminate further potential database states.

6.9 Other Types of Analysis

Although we have been looking specifically at sufficient conditions for a rule set to terminate, because we are doing dataflow analysis, there are other situations in which we can employ our abstract interpretation style of analysis.

6.9.1 State Invariance

Since rule priorities have a total order, all rule sets are guaranteed to be confluent. There is a similar property, though, which we call *state invariance*. A rule set is state invariant, if no matter which rule is triggered first, the final database state is always the same. Our algorithm can determine that a rule set is definitely state invariant, if all variables in the final iteration of the abstract interpretation algorithm have integer values. This is somewhat unrealistic, however, since the algorithm cannot guarantee this level of precision. Therefore it is more practical to examine partial state invariance. This states that some subset of the database must always have the same state after rule execution terminates. *e.g.* Are the values of A and B always the same after rule execution terminates ? A refinement of this could be whether A and B always equal zero after rule execution terminates. Our algorithm can state that a rule set is definitely partially state invariant if all the relevant variables have integer values in the final iteration and these values are related in the appropriate way. These kinds of properties could be useful for optimising transaction processing, since if we know certain parts of the database to be invariant under rule firings, then this knowledge could be used to eliminate tests in the transaction after rule triggerings.

6.9.2 Definite Non Termination

Similar to testing for confluence, we can state that a rule set is definitely non terminating if we find a repeating state in which all variables have precise integer values. The graph-based methods can never make this kind of statement.

6.9.3 Termination in N steps

For many applications, particularly real time ones, the property of termination in N steps is more interesting than just termination. It is straightforward to check for this using our method, we just need to see whether N+1 abstract executions of rules can occur and if not, then the rule set must terminate in N steps.

6.10 Discussion and Future Work

6.10.1 Types of Approximation

We have seen two types of abstract approximation in this chapter so far.

1 Approximation due to alternative execution paths, as outlined in the algorithm.

2 Approximation on the size of the domain as sketched in section 6.8.1.

These are not the only possibilities. We can also conduct

3 Approximation of complex pending structures such as a stack or queue.

4 Approximating complex event sequences/histories.

Both of these can be used to make either infinite sequences finite or finite sequences shorter and thus less complicated. They highlight one of the features of an abstract interpretation approach, namely that it can be employed to deal with a whole variety of execution model extensions.

6.10.2 Choices of Abstract Domain

It should be clear that the precision of our algorithm is dependent upon the choice of abstract domain. Obviously having more levels in the lattice will produce better results. This can be compared with the notion of a *hardwired limit* as discussed in chapter 1. There, trigger execution is always halted after a certain number of rule firings. (*i.e.* "premature" termination within N steps). Obviously, both approaches depend upon some N being chosen for them to proceed. One possibility would be to impose a hardwired limit (N) upon trigger execution, but choose this limit by executing the abstract interpretation algorithm and letting N equal the maximum number of iterations that it runs for.

6.11 Future Work

There are several other avenues for future work:

- Improving the precision of the existing algorithm without a blowout in complexity. Storage of auxiliary information about variable dependencies may help here.

- Exploring various abstract domains for rules with arithmetic and investigating how they work in practice.

- Experimentally comparing the precision of termination analysis using abstract interpretation with that of current syntactic methods.

- Exploring the effect of non-determinism.

- Exploring the effect of different semantics on rule precision.

- Analysing complex update languages. We have noted that bounded model rule languages can be simulated by a 0-1 variable rule language. This is possible because a bounded model rule language is associated with a finite number of equivalence classes defined upon it. We would like to be able to extend this idea to languages where there are an infinite number of equivalence classes. For this, it would be necessary to construct an abstract domain of equivalence classes which is finite and then define the behaviour of abstract operations upon it. Work in [5] may provide a starting point for exploring the structure of these classes.

- Applying abstract interpretation to other problems in active databases, such as condition evaluation.

6.12 Conclusion

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

- The approach is generic, so that the same formal machinery can help solve a variety of problems.

- It is well-understood and based on a solid mathematical foundation.

- Correctness proofs are simplified, since global correctness follows automatically whenever basic operations are approximated correctly.

- The technique allows/forces one to think in semantic terms and conveniently separate two concerns: approximation and implementation [42].
- Because it is concerned with dataflow, extra knowledge about a problem can easily be integrated into the algorithm (*e.g.* state constraints).
- The technique can be used for analysis of other properties in a rule system such as the detection of redundant rules or determination of the reachability of particular database states as a result of trigger activation.

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

Chapter 7

Conclusions

In this chapter, we summarise the contributions we have made and discuss some possible extensions.

7.1 Contributions

Active databases are unlikely to gain widespread acceptance until it can be demonstrated that reasonable methods exist for analysing the behaviour of rules. An important practical issue is that of ensuring termination of a set of rules. In this thesis, we have presented some new foundational work on the property of termination for sets of active database rules.

In chapters 2 and 3, we identified some important decidable/undecidable groups of rule languages.

- The *safe one literal* versus the *unrestricted one literal* language. The transition from decidability to undecidability is caused by the latter being able to perform a semi-join.
- The *AUUI* language versus the *unary conjunctive*$^{\neq,\neg}$ and *binary conjunctive* view languages. The transition from decidability to undecidability is caused by being able to either iterate through binary relations or use negation and inequality inside unary views.

We gave complexity bounds for the decidable cases and unified them under the heading of *bounded model* languages. We next highlighted some subtleties in the definition of termination, distinguishing between two varieties - *local* and *global*. We then related the decidability of *reachability* to decidability of local termination.

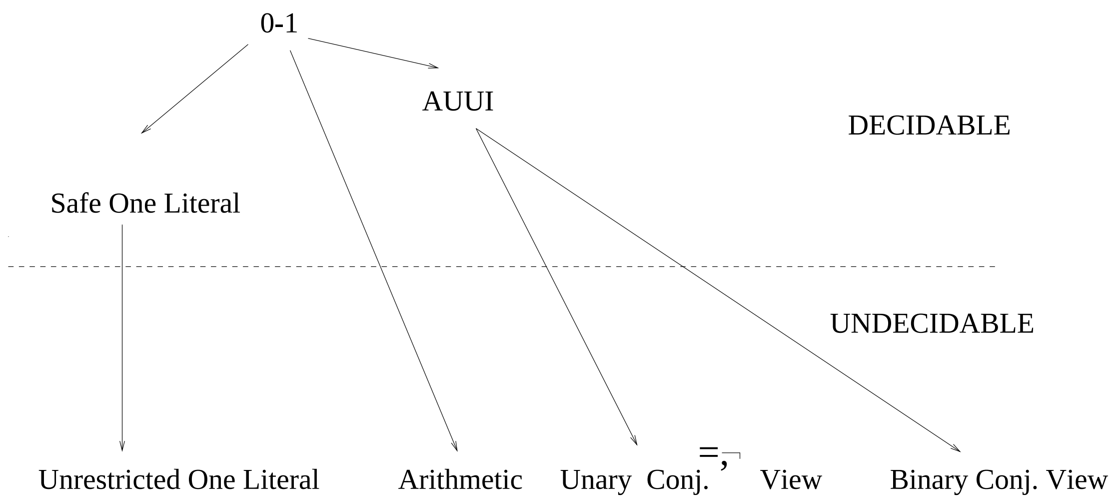

Figure 7.1: Decidability Results for Rule Languages

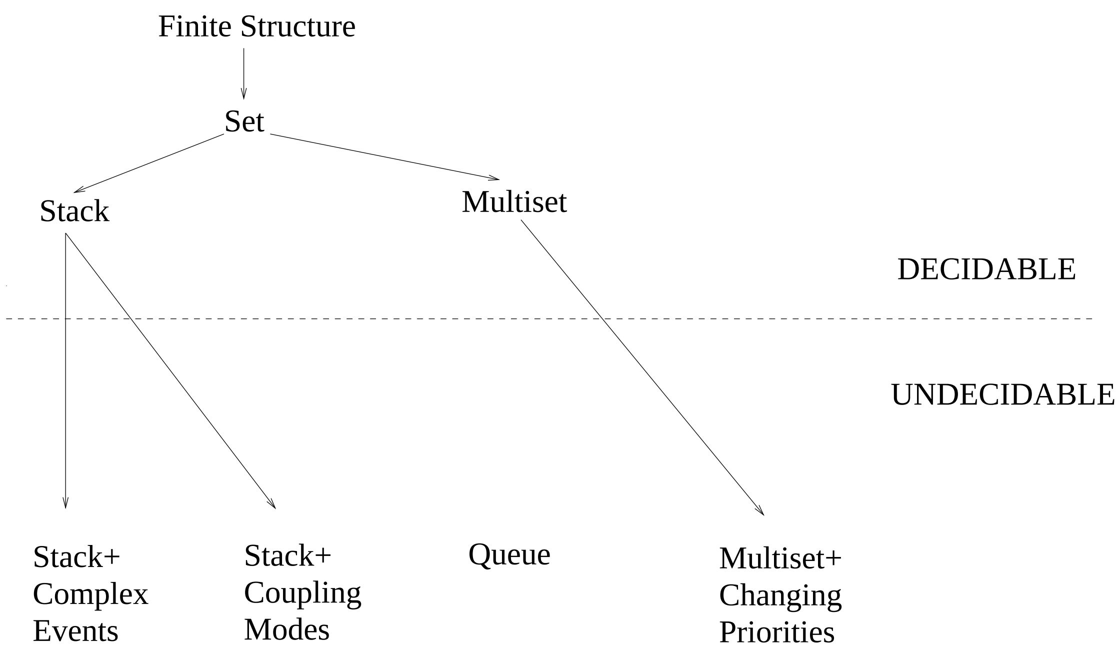

Figure 7.2: Decidability Results for Control Features

The foregoing work is relevant to update languages, since certain types of rule executions can be rewritten as *while* programs. A distinguishing characteristic of active rules, however, is their ability to use sophisticated control mechanisms. In chapter 6, we examined some of these features, namely the *pending rule structure* (set, stack, queue, multiset) and *meta rule language* (complex events, rule priorities and coupling modes). Here again, we identified decidable and undecidable configurations. Figures 7.1 and 7.2 summarise these results.

Finally, in chapter 6, we presented a promising new algorithm for approximate termination analysis, based on the idea of abstract interpretation.

7.2 Perspectives

In the introduction, we stated that rule termination analysis is both theoretically interesting and important practically. We now provide some perspectives of our work in this context.

On the practical side, our results help support the view that rule termination analysis is a difficult task and the problem is liable to become undecidable, even for seemingly easy rule languages. Indeed, given the complexity of analysis, it is apparent that the use of decision algorithms is unlikely to be a practical option for rule designers. This lends justification to the many methods that can be employed for approximate termination analysis, of which the abstract interpretation method is just one. Ultimately though, the rule designer may still be forced to perform manual checking, in the cases where these algorithms can provide no information. This suggests that research into the design of useful languages where termination is guaranteed, is likely to be an important direction of research (see [22, 41, 55] for results in this area).

On the theoretical side, we have helped elucidate features of active rules which can cause undecidability and we have shown how an abstract framework can be employed to derive useful termination information. This is interesting, not just for active databases and update languages, but for the area of transition systems generally, since active rules have much in common with other reactive methodologies such as statecharts and Petri nets.

7.3 Future Work

Some directions for further work that continue the line of investigation of this thesis include the following

- Further work on resolving the decidability of the unary conjunctive view language.
- Further analysis of the complexity bounds we have given for the analysis of the decidable languages is needed, to see if the gaps can be narrowed.
- It should be possible to derive further decidability results by examining new dimensions of active behaviour such as the use of further syntax/semantics like the use of delta relations, non determinism, instance oriented execution and other types of meta rules. Other possibilities are to limit the number of rules or the number of updates per rule (rather like the use of *sirrups* in boundedness [1]).
- It may be possible to identify languages for which termination is both decidable and analysis takes polynomial time on average.
- We intend to explore extensions of the abstract interpretation method to rules with a more sophisticated syntax.

Appendix A

A.1 Sketch Proof of Theorem 21

Recall that we are using rules in the unary conjunctive$^{\neq,\neg}$ view language to check the computation of a two counter machine. The relations we use are

$S_0, \ldots, S_h$ - contain a constant which represents that particular state.
$succ$ - the successor relation
$config$ - contains the computation of the 2CM.
tc - the transitive closure relation
$zero$ - contains the constant zero
$last$ - contains the highest constant
$nonzero$ - contains all constants in $succ$ which are not zero
bad - will be true if the database doesn't correctly describe the 2CM and its computation.

We will show how to define a set of rules in the unary conjunctive$^{\neq,\neg}$ view language which are non terminating iff the 2CM halts. The rules are

- Make sure there is only one zero.

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

- Do this also for $S_0, \ldots, S_h, last$. Then check that $S_i \cap S_j = \emptyset$.

- Make sure zero has no predecessor

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

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

$$bad \leftarrow succ(X,Y), succ(X,Z), Y \neq Z$$
$$bad \leftarrow succ(Y,X), succ(Z,X), Y \neq Z$$

- Check that *last* has no successor.

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

- Make sure that *succ* is contained in *tc*.

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

- Make sure that *succ* doesn't take any shortcuts.

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

- Check that each constant in *succ* has a predecessor

$$V_4(X) \leftarrow succ(X,Y), \neg zero(X)$$
$$V_5(X) \leftarrow succ(Z,X)$$
$$bad \leftarrow V_4(X), \neg V_5(X)$$

- Check that *tc* is transitive.

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

- Make sure there transitive closure contains no loops.

$$bad \leftarrow tc(X,X)$$

- Check that each element in *tc* is reachable. Define more rules for the other combinations.

$$bad \leftarrow tc(X,U), tc(Y,V), X \neq Y, \neg tc(X,Y), \neg tc(Y,X)$$

- Check that every time in *config* is in *succ*.

$$
\begin{aligned}
&V_1(X) \leftarrow succ(X,Y)\\
&V_2(X) \leftarrow succ(Y,X)\\
&V_3(T) \leftarrow config(T,S,C_1,C_2)\\
&bad \leftarrow V_3(X), \neg V_1(X)\\
&bad \leftarrow V_3(X), \neg V_2(X)
\end{aligned}
$$

- Check that each time in *succ* is in *config*.

- Check configuration of the machine at time zero and transitions of machine. This can be done similar to theorem 20.

- Check that halt state is in config. We will make the rules loop iff $halt(X) \wedge \neg bad$ is satisfiable.

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

All times in *config* are reachable because all constants in *succ* are reachable. All constants in *succ* are reachable because everything in *tc* is reachable. Also note that the counter values can never be higher than the current time. ■

A.2 Proof of Theorem 63

We need to prove that we signal termination only if rule execution actually terminates.

Similar to real execution, we can conceptually think of the execution of our algorithm as a sequence of *(state approximation, pending set)* pairs. Let s be the sequence of real execution, and let t be the sequence of our algorithm's execution. We would like the execution sequence of our algorithm to correspond to an execution of the rule set in some way. Clearly, our execution sequence cannot exactly mimic the real execution sequence, because we have to account for computations starting with all possible initial states. Therefore, we have to relax the requirements that we impose on our approximate execution sequence. Firstly, we define our approximation relation.

Definition 66 Approximations

Suppose we have a database with variables $v_1, v_2, \ldots, v_n$. Now suppose d_a is an abstract database state and d_c a concrete (actual) database state. Then we say $d_a\ appr_{dbs}\ d_c$ iff $\forall i\ \alpha(d_c(v_i)) \leq d_a(v_i)$

Intuitively, , we say d_a approximates d_c if the values of all its variables are higher in the lattice. ■

Definition 67 appr$_{pset}$
Let $pset_1, pset_2 \in \mathcal{P}(R)$

$$pset_1\ appr_{pset}\ pset_2 \Longleftrightarrow pset_2 \subseteq pset_1 \wedge select\_first\_element(pset_1) = select\_first\_element(pset_2).$$

The function select_first_element returns the highest priority rule in the set (if the set is non empty). ■

Definition 68 Execution States
We say that $(dbs_1, pset_1)\ appr\ (dbs_2, pset_2)$ iff $dbs_1\ appr_{dbs}\ dbs_2$ and $pset_1\ appr_{pset}\ pset_2$. Let t=$(dbs_i, pset_i)$, then $t \downarrow 1$ refers to dbs_i and $t \downarrow 2$ refers to $pset_i$. ■

Lemma 69 *Approximation of execution sequences Let the sequence $s : s_1, s_2, \ldots, s_i, \ldots$ and the sequence $t : t_1, t_2, \ldots, t_k, \ldots$ be given.*
Pick two elements t_k and s_i such that $k \geq i$. If $t_k\ appr\ s_i$ then
$\exists l : l > k\ \wedge\ t_l \downarrow 2\ appr_{pset}\ s_{i+1} \downarrow 2 \Longrightarrow t_l\ appr\ s_{i+1}$.

Proof: We let the pending sets be represented by ordered lists with highest priority first. Let $s_i \downarrow 2 = [r_1, \ldots, r_n]$. Assuming s_{i+1} exists, we have $n \geq 1$.
case1: The condition is actually true for s_i. In our approximation, then the condition is either a *must_satisfy* or a *may_satisfy*. If it is *must_satisfy* in $t_k \downarrow 1$ then we execute the same actions as the real execution, we trigger the same rules. The pending sets of s_{i+1} and t_{k+1} still have the same first element and our approximation still approximates s_{i+1}
A similar result is produced if the condition is *may_satisfy* in $t_k \downarrow 1$ except the algorithm may also perform a least upper bound. Clearly this may make our approximations less precise, but preserves correctness.
case2: The condition is actually false for s_i. In our approximation the condition is either *may_satisfy* or *cannot_satisfy*.
If the latter, then we know that the condition definitely fails in all possible executions and we can remove r_1 from the pending set. Hence, $l = k + 1$. If the former,

then our approximation would treat the condition as *true* even though it is always *false* in all real executions. In this case, if there's only one rule in the pending set, then s_{i+1} doesn't exist because execution has terminated. Otherwise, we do a least upper bound, and t_{k+1} appr s_{i+1} Note that $t_{k+1} \downarrow 1$ still approximates $s_{i+1} \downarrow 1$, but the pending sets of t_{k+1} and s_{i+1} are very different. When the algorithm reaches t_l any rules it executed must have been marked which means least upper bounds will have been taken and so the current approximation $t_l \downarrow 2 \geq t_{k+1} \downarrow 2$. Hence, our approximation approximates s_{i+1}. ■

We are ready to prove Theorem 63.

If the algorithm signals termination then the approximate execution sequence is finite and ends with an empty pending set. We prove that the sequence t is strictly longer than the sequence s. Therefore, since the sequence t is finite (because our algorithm always halts), the sequence s must also be finite. Let t_k *appr* s_i. This is true for the two initial elements t_1 and s_1. We show that if s_{i+1} exists $\exists l : l > k \ \wedge \ t_l \ appr \ s_{i+1}$. If this holds then the sequence t is strictly longer than s. Using lemma 69, all we have to show is that $\exists l : l > k \ \wedge \ t_l \downarrow 2 \ appr_{pset} s_{i+1} \downarrow 2$. If the condition evaluates to *true*, then $l = k + 1$ by lemma 69. Similarly, if the condition evaluates to *false* and our approximation gives a *must_not_satisfy* then $l = k + 1$ also.

If the condition evaluates to *false* and our approximation gives *may_satisfy* then $s_{i+1} \downarrow 2 = [r_2, \ldots, r_n]$ and $t_{k+1} \downarrow 2 = [\ldots, r_2, \ldots, r_n, \ldots]$. If n was 1 then the pending set at s_{i+1} is empty and s is finite. Otherwise, we know that the last element of t is an empty set. The element r_2 must have been removed from the pending set at between t_{k+1} and the last element of t. Hence, the element r_2 must have been the first element of the pending set at some point between t_{k+1} and the last element of t. Thus the sequence $s_1, s_2, \ldots$ is finite and we never signal termination when we shouldn't. Thus our algorithm is correct. ■

Bibliography

[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 E. Simon. Fundamental properties of deterministic and non-deterministic extensions of datalog. *Theoretical Computer Science*, 78(1):137–158, 1991.

[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] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[7] 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.

[8] J. Bailey, L. Crnogorac, K. Ramamohanarao, and H. Søndergaard. Abstract interpretation of active rules and its use in termination analysis. In *Proceedings of the Sixth International Conference on Database Theory, LNCS 1186*, pages 188–202, Delphi, Greece, 1997.

[9] 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.

[10] J. Bailey, M. Georgeff, D. Kemp, D. Kinny, and K. Ramamohanarao. Active databases and agent systems - a comparison. In *Proceedings of the Second International Workshop on Rules in Database Systems, LNCS 985*, pages 342–356, Athens, Greece, 1995.

[11] J. A. Bailey and K. Ramamohanarao. Issues in active databases. In *Proceedings of the 6th Australasian Database Conference*, pages 27–35, 1995.

[12] E. Baralis, S. Ceri, and S. Paraboschi. Improved rule analysis by means of triggering and activation graphs. In T. Sellis, editor, *Rules in Database Systems*, Lecture Notes in Computer Science 985, pages 165–181. Springer-Verlag, 1995.

[13] E. Baralis, S. Ceri, and S. Paraboschi. Run-time detection of non terminating active rule systems. In *Proceedings of the Fourth International Conference on Deductive and Object Oriented Databases*, pages 38–54, Singapore, 1995.

[14] E. Baralis, S. Ceri, and S. Paraboschi. Modularization techniques for active rules design. *ACM Transaction on Database Systems*, 21(1), 1996.

[15] 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.

[16] A. Borgida. Description logics in data management. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):671–682, 1995.

[17] F. Bourdoncle. Dynamic partitioning: A method for interprocedural abstract interpretation. *Journal of Functional Programming*, 2(4):407–423, 1992.

[18] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Constraint enforcement through production rules: Putting active databases to work. *IEEE Data Engineering Bulletin*, 15(1–4):10–14, 1992.

[19] 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.

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

[21] 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.

[22] 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.

[23] T. Coupaye and C. Collet. Denotational semantics for an active rule execution model. In *Proceedings of the Second International Workshop on Rules in Database Systems*, pages 36–50, Athens, Greece, 1995.

[24] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic Programming*, 13(2&3):103–179, 1992.

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

[26] R. A. DiPaola. The recursive unsolvability of the decision problem for a class of definite formulas. *Journal of the ACM*, 16(2):324–327, 1969.

[27] G. Dong and J. Su. Conjunctive query containment with respect to views and constraints. *Information Processing Letters*, 57:95–102, 1996.

[28] A. L. Furtado and M. A. Casanova. Updating relational views. In *Query Processing in Database Systems*, 1985.

[29] 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.

[30] 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.

[31] 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.

[32] E. Hanson. Rule condition testing and action execution in Ariel. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 49–58, San Diego, 1992.

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

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

[35] 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.

[36] 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.

[37] A. Karadimce and S. Urban. Refined triggering graphs: A logic based approach to termination analysis in an active object-oriented database. In *Proceedings of the 12th International Conference on Data Engineering*, pages 384–391, New Orleans, Louisiana, 1996.

[38] D. B. Kemp, P. J. Stuckey, and K. Ramamohanarao. Els-programs and the efficient evaluation of non-stratified programs by transformation to els. In *Proceedings of the Fourth International Conference on Deductive and Object Oriented Databases*, pages 91–108, Singapore, 1995.

[39] A. Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, 35(1):146–160, 1988.

[40] 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.

[41] 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.

[42] K. Marriott, H. Søndergaard, and N. D. Jones. Denotational abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 16(3):607–648, 1994.

[43] Oracle 7 server concepts manual, December 1992.

[44] 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.

[45] 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.

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

[47] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM*, 27(4):633–655, 1980.

[48] M. Stonebraker and G. Kemnitz. The POSTGRES next-generation database management system. *Communications of the ACM*, 34(10):78–92, 1991.

[49] 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.

[50] 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.

[51] T. Weik and A. Heuer. An algorithm for the analysis of termination of large trigger sets in an OODBMS. In *Proceedings of the International Workshop on Active and Real-Time Databases*, pages 170–189, Skövde, Sweden, 1995.

[52] J. Widom. A denotational semantics for the Starburst production rule language. *ACM SIGMOD Record*, 21(3):4–9, 1992.

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

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

[55] 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.

[56] D. Zimmer, A. Meckenstock, and R. Unland. Rule termination analysis based on Petri nets. Technical report, C-LAB Paderborn, 1996.