Constraint-Based Termination Analysis for Cyclic Active Database Rules*

Saumya Debray and Timothy Hickey

Abstract. There are many situations where cyclic rule activations—where some set of active database rules may be activated repeatedly until the database satisfies some condition—arise naturally. However, most existing approaches to termination analysis of active rules, which typically rely on checking that the triggering graph for the rules is acyclic, cannot infer termination for such rules. We present a constraint-based approach to termination analysis that is able to handle such cyclic rule activations for a wide class of rules.

1 Introduction

Active databases, which are conventional databases extended with a mechanism to create and execute production rules that manipulate the state of the database, have attracted considerable interest in recent years. Such rules provide a general mechanism for a number of database features such as integrity constraint checking and view maintenance, and simplify building and reasoning about database applications.

In general, rule activations in active databases can "cascade," i.e., the execution of an active rule can cause a change in the database state that causes another rule to be executed; the resulting change can then cause the activation of a third rule; and so on. Ensuring that such cascaded rule activations do not go on forever therefore becomes of fundamental importance. Analyses that examine a set of active rules to determine whether rule activations will terminate are called *termination analyses*.

Almost all of the work to date on termination analysis for active databases (see Section 6) relies on checking that a directed graph called the "triggering graph" for a set of rules is acyclic. The essential intuition this expresses is that a rule should not be able to cause itself to be (re-)activated, either directly or indirectly. The differences between various proposals for such analyses lie in the sets of edges they are able to eliminate from the triggering graph before this acyclicity check. In most of these proposals, the underlying sets of rules being considered satisfy the property of not being self-activating in this manner, and the analyses themselves focus on identifying and eliminating edges that could introduce spurious cycles into the triggering graph.

* This work was supported in part by the National Science Foundation under grants CCR-9711166, CDA-9500991, and ASC-9720738. *Authors' addresses*: S. Debray, Department of Computer Science, The University of Arizona, Tucson, AZ 85721, USA, *E-mail*: `debray@cs.arizona.edu`; T. Hickey, Michtom School of Computer Science, Brandeis University, Waltham, MA 02454, USA, *E-mail*: `tim@cs.brandeis.edu`.

There are, however, many situations where it is natural to have a rule that can activate itself, but where such self-activations are guaranteed to eventually terminate. As an example, a company that suffers a revenue shortfall may impose budget cuts in all of its departments. It would make sense, in such a case, to impose larger cuts on departments with many employees, since they have larger budgets to work with. Thus, suppose that the budget reduction for each department is set at 1% of its total salary expenses, and expressed as an active rule that reduces the budget of each department by the appropriate amount. A single round of reductions may not suffice to meet the revenue shortfall, so the rule may be activated again. It is not hard to see, however, that eventually the budget will be reduced enough that the rule activation will stop.

Throughout this paper, we use the following example, taken from Chapter 2 of a text by Zaniolo *et. al.* [25].

Example 1. The following rule, defined by a budget-conscious manager, imposes a salary reduction of 10% on every employee in an organization whenever the average salary exceeds a threshold (in this case 100):

```
rule SalaryControl on Emp
when inserted, deleted, updated(Sal)
if (Select Avg(Sal) from Emp) > 100
then update Emp
     set Sal = 0.9*Sal
```

Notice that this rule can be activated if an employee is hired with a high initial salary; if the initial salary is high enough, one round of salary reductions may not suffice to satisfy the termination condition, so the rule may be activated again. Eventually, however, the average salary must fall below 100, causing the rule activations to terminate.

While this rule seems "obviously terminating," reasoning about such rules can be quite subtle. For example, consider a rule that is identical to that shown above, the only difference being that the activation condition is

```
if (Select Avg(Sal) from Emp) > 0 then ...
```

This rule is structurally very similar to that of Example 1, with a decreasing value for the average salary and a lower bound on how far it can decrease. It is, nevertheless, non-terminating. As another example, consider a rule that is identical to that of Example 1, with the only difference being that the action is to set each employee's salary to `0.9*Sal + Bonus`, where `Bonus` is some constant. In this case, it turns out that the rule is terminating if `Bonus` < 10, and nonterminating if `Bonus` ≥ 10. What these examples illustrate is that any termination analysis that aims to handle such cyclic rule activations must be able to analyze the effects of cyclic rule execution with a fairly high degree of precision.

2 Preliminaries

2.1 Active Rules

We consider *Event-Condition-Action* (ECA) rules: a rule is triggered when certain *events* specified in the rule occur, and provided that their (optional) *conditions* hold; when a rule so triggered is executed, the *actions* specified in the rule are carried out. For concreteness we use the syntax and semantics of the Starburst rule system [24]: a rule is assumed to have the structure

rule *RuleName* **on** *Relation*
when *EventList*
if C **then** *Action*

where *EventList* specifies a set of events that cause the rule to be triggered. The execution of a triggered rule involves the evaluation of the condition C, and if this evaluates to *true*, carrying out the actions specified in the list *Action*. The condition C, which determines when the rule is activated, is referred to as the *activation condition* of the rule; we sometimes also use the term *termination condition* for the rule to refer to the condition $\neg C$.

Since the handling of aggregation operations is more or less orthogonal to the main focus of this paper, we make the simplifying assumption that any aggregation operations in active rules are applied to the entire relation, i.e., there is no aggregation over partitions of the relation computed using constructs such as the '`group by`' clause of SQL. The basic idea is that if a rule computes an aggregate operation f on an attribute A of a relation R, we handle this using a dummy relation R_A_f containing a single tuple that has a single attribute whose value is that of the operation f applied to attribute A of R. Changes to the relation R through *insert*, *delete*, or *update* operations—including those in active rules—are considered to also modify such dummy relations appropriately, albeit in a conservative manner: that is, unless the new value of the aggregate value can be predicted, its value is considered to be unknown and represented in the dummy relation using a null value.

2.2 Constraint Systems

For the purposes of this paper, a *constraint system* is a system for maintaining and manipulating constraints over a *constraint domain* $\mathcal{D}$, which is essentially a (first-order) structure, i.e., a universe D together with an appropriate assignment of functions and relations over D to the symbols of the constraint system. We assume that the constraint systems under consideration have the functionality encountered in typical constraint logic programming systems such as CLP($\mathcal{R}$) [16] and SICStus Prolog [21] (see also the survey by Jaffar and Maher [17]). More formally, *constraints* are first-order formulae over a signature Σ, such that: the binary predicate symbol '=' is in Σ; there are constraints that are identically true and identically false; and the class of constraints is closed under variable renaming, conjunction, and existential quantification. Operations on constraints supported by the constraint system are assumed to include [17]:

- A test for *consistency* or *satisfiability*: $\mathcal{D} \models (\exists)c$.
- A test for *implication* (i.e., *entailment*) of one constraint by another: $\mathcal{D} \models c_0 \Rightarrow c_1$.
- The *projection* of a constraint c_0 onto variables $\bar{x}$ to obtain a constraint c_1 such that $\mathcal{D} \models c_1 \Leftrightarrow (\exists\bar{y})c_0$, where $\bar{y} = vars(c_0) - \bar{x}$ is the set of variables in c_0 except for those mentioned in $\bar{x}$.

In particular, we focus on the CLP(F) constraint system [13], which is powerful enough for our needs and whose implementation is freely available. This is a constraint system over the reals that supports, in addition to the usual arithmetic and comparison operators, the functions $abs(x)$, $exp(x)$, $log(x)$, x^n, x^y, as well as the trigonometric functions $sin(x)$, $cos(x)$, $tan(x)$ and their inverses. Our implementation of this system uses interval constraints, and handles these functions using a reimplementation of the standard math library based on interval arithmetic. A fundamentally important aspect of the system is that the constraint system is provably sound [13, 14]. A detailed discussion of this system is omitted due to space constraints: for our purposes it suffices to note that the constraint solver is queried with a quantifier-free first-order conjunction $Q(x_1, \ldots, x_n)$, interpreted as the question "*do there exist any* $x_1, \ldots, x_n$ *such that* $Q(x_1, \ldots, x_n)$*?*" The solver responds either with a set of real intervals $I_1, \ldots, I_n$, interpreted as "*if there exist any* $x_1, \ldots, x_n$ *such that* $Q(x_1, \ldots, x_n)$*, then for all such values it must be the case that* $x_1 \in I_1$ *and ... and* $x_n \in I_n$*,*" or indicates that there are no values for the variables x_i that satisfy the formula $Q(\cdots)$. In other words, the answer returned by the solver contains the projection, on each of the variables $x_1, \ldots, x_n$, of the convex hull of the region containing all solutions to the question.

3 Annotated Triggering Graphs

Many of the termination analyses proposed in the literature use the notion of triggering graphs. A triggering graph is a directed graph where each vertex represents a rule and where there is an edge from vertex r_i to vertex r_j if the action of rule r_i can cause rule r_j to become triggered [8]. We generalize this notion to that of "annotated triggering graphs," which additionally incorporate information, at each vertex, about the change(s) resulting from the activation of the corresponding rule. To this end, we first define how such changes might be represented.

Definition 1. [Bounds function] *A* bounds function *over a schema S maps each attribute of S to a pair $\langle lo, hi\rangle$ where each of lo and hi is either $\perp$ or a linear expressions over (numerical) attributes in S.*

If a bounds function maps an attribute to $\langle lo, hi\rangle$, this indicates that lo is a lower bound on the values of that attribute while hi is an upper bound, with a value of $\perp$ indicating a complete lack of knowledge (i.e., indicating that we cannot say anything about the values for the corresponding attribute), and non-$\perp$ values indicating definite knowledge. Note that these bounds are not restricted to be numbers, but may in general be expressions that depend on the values for other attributes.

Definition 2. [Annotated Triggering Graph] *Given a set of rules R with schema S, an* annotated triggering graph *is a pair $(G, \mathcal{F})$ where $G = (V, E)$ is a conventional triggering graph, and $\mathcal{F}$ maps each vertex in V to a bounds function over S.*

Thus, at each vertex of an annotated triggering graph, each attribute is mapped to a pair of expressions that specify upper and lower bounds on the values of that attribute. As an example, the rule in Example 1 would associate, with the vertex corresponding to the rule, the bounds function $[Sal \mapsto \langle 0.9 * Sal, 0.9 * Sal\rangle]$ (attributes that are not explicitly mentioned are assumed to not have changed, and are therefore mapped to themselves). This indicates that the result of the activation of the rule is to update each tuple so that the value of its *Sal* attribute is $0.9\times$ its old *Sal* value, while the other attribute values are left unchanged.

3.1 Constructing Annotated Triggering Graphs

This section describes a simple algorithm for constructing annotated triggering graphs. It is quite conservative in its treatment of bounds, and can almost certainly be improved to increase its precision. Consider a rule

```
rule RuleName on R
when ...
if ...
then update R'
      set x = φ(y1,...,yn) where Cond
```

The first question we consider is whether all of the tuples in R' will be modified. If the final `where` clause is absent, or *Cond* is identically true, then we know that all tuples in R' will have the value of attribute x set to $\varphi(y_1, \ldots, y_n)$. So in this case the bounds function maps x to $\langle \varphi(y_1, \ldots, y_n), \varphi(y_1, \ldots, y_n)\rangle$.

Otherwise, if it is possible that only some of the tuples will be updated, we attempt to determine the relationship of the value of the expression $\varphi(y_1, \ldots, y_n)$ with that of x. Let $\mathcal{C}_0$ be a conjunction of constraints on the possible values for various attributes, obtained from domain information for the database schema as well integrity constraints on the database ($\mathcal{C}_0$ is a global constraint and need be computed only once, at the beginning of the analysis). We then construct a constraint $\mathcal{C} \equiv \mathcal{C}_0 \wedge [z = \varphi(y_1, \ldots, y_n) - x]$, where z is a new variable not appearing in $\mathcal{C}_0$, and examine whether or not certain constraints on z are entailed by $\mathcal{C}$ (see Section 2.2 for our assumptions regarding entailment operations in constraint systems). We consider the following possibilities:

- $\mathcal{C}$ entails $z \geq 0$. This means that the value of x is non-decreasing as a result of the update. Since it is possible that only some of the tuples will be updated, an upper bound on the x attribute value is given by $\varphi(y_1, \ldots, y_n)$, while a lower bound is given by x. Thus, in this case we have the bounds function $[x \mapsto \langle x, \varphi(y_1, \ldots, y_n)\rangle]$.
- $\mathcal{C}$ entails $z \leq 0$. This means that the value of x is non-increasing. Reasoning as in the previous case, we get the bounds function $[x \mapsto \langle \varphi(y_1, \ldots, y_n), x\rangle]$.
- If neither of these previous two cases holds, we conclude that nothing can be said about the value of x after the update. The resulting bounds function is $[x \mapsto \langle \bot, \bot\rangle]$.

3.2 Reasoning About Annotated Triggering Graphs

Once we have constructed an annotated triggering graph for a set of rules, we examine any cycles in this graph to determine the net effect of going around the cycle once. Intuitively, what we need to do is to somehow compose the bounds

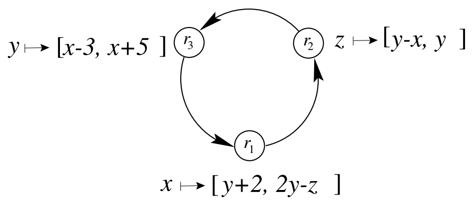

Fig. 1. An example of a cycle in an annotated triggering graph

functions at each of the vertices in the cycle. Before discussing the details of how this should be done, we consider an example. Consider the cycle consisting of three vertices, shown in Figure 1. Suppose we wish to determine an upper bound on the change in the value of x at vertex r_1 when we go around the cycle once. Let the upper bound on x be denoted by x_{max}: after the execution of r_1, we use the upper bound on x, from the bounds function at this vertex, to obtain the constraint $x_{max} = 2y - z$. After the execution of rule r_2, the new value of z has bounds $y - x \leq z \leq y$. Since z appears with a negative coefficient in the expression $2y - z$, we use the lower bound to obtain the constraint $z = y - x$. Composition (i.e., conjunction) of constraints then yields $x_{max} = 2y - z \wedge z = y - x$; projecting on x_{max} yields $x_{max} = y + x$. Finally, at vertex r_3, the bounds on y are $x - 3 \leq y \leq x + 5$; this time, since the coefficient of y is positive in the expression $y + x$, we use the upper bound $x + 5$. As before, we compose constraints to obtain $x_{max} = y + x \wedge y = x + 5$; this is then projected on x_{max} to yield $x_{max} = 2x + 5$. Thus, the change in the value of x when we go around this cycle once is (at most) $2x + 5$.

This example illustrates how we summarize the effects of a cycle. To estimate an upper bound on the value of x at a vertex r after going around the cycle (the case for lower bounds is analogous), we start with the constraint $x_{max} = E$, where E gives the upper bound on x after r's execution. We then work our way around the cycle: at each vertex we take the conjunction of the current constraint on x_{max} and constraints on the variables occuring in it, obtained from the bounds on these variables at that vertex given by the annotated triggering graph; the resulting constraint is then projected on x_{max}. During this process, we use the lower bound for a variable if it occurs negatively in the constraint associated with x_{max}, and the upper bound if it occurs positively.

Given a constraint C and a set of variables[1] $X = \{x_1, \ldots, x_n\}$, we use the notation $\exists_X C$ to denote the constraint obtained by projecting away the variables in X, i.e., $\exists x_1 x_2 \cdots x_n C$. Let $vars(E)$ denote the variables appearing in an expression E. Given an annotated triggering graph $(G, \mathcal{F})$ with schema S and a vertex r in G, let $\mathcal{F}(r)(v) = \langle lo_v^{(r)}, hi_v^{(r)} \rangle$ for any variable $v \in S$. We can then formalize the procedure sketched above as follows.

[1] Since our approach uses constraints on attribute values, we treat attributes as the variables in such constraints. In the remainder of the paper, therefore, we will use the terms *attribute* and *variable* interchangeably.

– *Processing a single vertex r.* Given a constraint $C \equiv x_{max} = E$, let C' be the constraint

$$C' = \bigwedge_{v \in vars(E)} \{v = e \mid e = \begin{cases} lo_v^{(r)} & \text{if } v \text{ appears negatively in } E \\ hi_v^{(r)} & \text{if } v \text{ appears positively in } E \end{cases}\}.$$

Since bounds functions map variables to linear expressions (Definition 1), each variable occurs at most once in E. Thus, C' imposes a single (equality) constraint on any such variable, and therefore is satisfiable.
The result of propagating C through the vertex r is then given by

$$\mathsf{ProcVertex}(r, C) = \exists_{vars(E)}(C \wedge C').$$

– *Processing a sequence of vertices.* The result of propagating a constraint C through a sequence of vertices s is given by $\mathsf{ProcVertexSeq}(s, C)$, where:

$$\mathsf{ProcVertexSeq}(\varepsilon, C) = C$$
$$\mathsf{ProcVertexSeq}(rs', C) = \mathsf{ProcVertexSeq}(s', \mathsf{ProcVertex}(r, C)).$$

Here, ε denotes the empty sequence while rs' denotes the sequence whose first element is r and the remaining sequence is s'.

– *Processing a cycle.* Given a cycle $r_1 r_2 \cdots r_n r_1$, an upper bound on the change in the value of a variable x on going around the cycle once is given by

$$\mathsf{ProcCycle}(x, s) = \mathsf{ProcVertexSeq}(s, `x_{max} = x').$$

where $s = r_1 r_2 \cdots r_n$.

The determination of a lower bound on the change in the value of a variable is analogous.

Since bounds functions map variables to linear expressions (Definition 1), the procedure described above for computing $\mathsf{ProcCycle}(x, s)$ for a cycle s is essentially involves composing a sequence of linear functions, and therefore yields a linear expression of the form $ax+E$. The effect on the value of x of going around the cycle n times can then be expressed as a difference equation $x_n = ax_{n-1} + E$ where x_i represents the value of x after i iterations around the cycle.[2] In general, the procedure described can yield a system of simultaneous linear difference equations. However, it is always possible to reduce a system of linear difference equations to a single linear difference equation in one variable [19], so it suffices to consider the solution of a single linear difference equation in one variable.

3.3 Approximate Solution of Difference Equations

Having obtained a difference equation as discussed above, we consider how it may be solved. The automatic solution of general difference equations is a difficult problem, but there is a wide class of equations that can be solved automatically, using either characteristic equations or generating functions [9, 15, 20]. For the

[2] Again, note that this represents an upper bound on x, so strictly speaking we should write $x_n \leq ax_{n-1} + E$. However, since we are concerned with proving termination in the worst case, when this maximum is actually realized, we simply use the equality.

purpose of analysis of active database rules, however, we additionally require that the solution method used be efficient, even if this means sacrificing precision in some cases. For this reason, we use a table-driven method for computing an upper bound to the actual solution. Our approach is to use a "library" of difference equation templates together with a symbolic solution for each such template [12, 10, 11]. The idea is to use pattern matching to identify a template that matches the equation obtained from the analysis described in Section 3.2. Once a match is obtained against a template, the solution to the equation can then be obtained by substituting into the symbolic solution for that template. In general, the library of difference equation templates will contain many different entries, and the pattern matching process will try to match a given equation against these templates in increasing order of the "size" of their solutions. If the equation cannot be matched against any template in the library, we attempt to use simplifying approximations, as described below. If no match can be obtained even after any applicable simplifying approximations, we give up and return the value $\perp$, indicating that we cannot say anything about the solution.

The idea can be illustrated by an example. Suppose that the difference equation library has the template:

$$x_n = Ax_{n-k} + B$$

together with the symbolic solution

$$x_n = (x_0 + \tfrac{B}{A-1})A^{n/k} - \tfrac{B}{A-1}, \qquad \text{where } x_0 \text{ is the initial value of } x.$$

Given an equation $x_n = 0.9x_{n-1}$, pattern matching against this template succeeds with $A = 0.9$, $B = 0$, $k = 1$; substituting these values into the symbolic solution yields the solution $x_n = 0.9^n x_0$.

If the difference equation at hand cannot be matched against any template in the library, we attempt to approximate it in a way that is conservative, i.e., termination inferred from the approximating equation (as discussed in the next section) must imply termination of the original equation. Space constraints preclude a detailed discussion of such approximations: we outline the general ideas and illustrate them with an example. Suppose we have the difference equation $x_n = 0.8x_{n-1} - 0.15x_{n-2}$, and are trying to simplify it so as to match against the template shown above. To do this, we use the activation condition on x (see Section 2.1), obtained from the rule(s) involved in the cycle, to determine (i) whether the values of x are bounded above or below; and (ii) whether the values of the x_i are positive or negative. This is done using the entailment operation of the constraint system, in a manner very similar to that discussed in Section 3.1. For example, suppose that for this particular case we have the activation condition $x > 100$. Then, we have the following:

(i) The termination condition for the rule is $x \leq 100$, i.e., we have a lower bound on the value of x below which the rule will not be activated. This implies that we can use an upper bound on the actual difference equation for termination analysis. In other words, if we can construct a difference equation $y_n = f(\cdots)$ such that $y_n \geq x_n$ for all $n \geq 0$, and can use this equation to determine that eventually the values of y_n will satisfy the termination condition for the rule, then we can conclude that the original variable x_n would eventually satisfy the termination condition of the rule as well. If the termination condition implied an upper bound for x, then we would, analogously, construct an approximation that is a lower bound on x_n.

(ii) The activation condition $x > 100$ implies that x is positive. This, in turn, implies that the expression $0.8x_{n-1}$ is an upper bound on the expression $0.8x_{n-1} - 0.15x_{n-2}$.

We therefore use the equation $x_n = 0.8x_{n-1}$ to approximate (from above) the original equation. The approximating equation can now be successfully matched against the template shown above.

3.4 Handling Non-Numeric Attributes

While the discussion thus far has focused on numerical attributes, the approach described can also be used to handle rules that manipulate non-numerical attributes. Our approach will be to formulate and reason about difference equations involving aggregate values such as the number of tuples in a relation.

In the absence of any additional information, we can, at the very least, use the dummy relation R_`Count`, for handling the aggregation operation `Count` on a relation R (see Section 2), to monitor the number of tuples in R: the insertion of a tuple into R causes this value to increase by 1, the deletion of a tuple causes it to decrease by 1, and updates leave it unchanged. As an example, this can be used to infer termination of a cycle along which two tuples are deleted from a relation and one tuple inserted: we would obtain a difference equation stating that there is a net reduction of 1 tuple in the size of the relation each time around the cycle, and use this to determine that the deletions must eventually stop. This approach can be improved further using additional semantic information about the database, e.g., from integrity constraints.

4 Static Termination Analysis

The approach described in the previous section allows us to obtain an (upper bound) solution to a difference equation describing the effects of a cycle in the triggering graph. Ultimately, however, we are interested not so much in the solutions to these equations, but rather in determining whether or not the rule activations eventually terminate. Suppose that the activation condition for the rule under consideration is $C(x)$. We use the constraint solver determine an interval within which all of the values of n for which $C(x_n)$ is true, i.e., for which the rule will be activated, must lie; termination can then be inferred by examining this interval. This is done as follows:

1. We add constraints expressing upper and lower bounds on the value of x_0, denoted by `MAXVAL` and `MINVAL`, obtained from domain information for the database schema as well as any applicable integrity constraints; if there are no applicable constraints, these can simply be the largest and smallest numerical values representable on the system. Moreover, if $C(x_0)$ is false the cycle of active rules will not be initially triggered (see Section 2.1), so we may assume that $C(x_0)$ holds: this provides additional constraints on the

values of x_0. Let the conjunction of these constraints on the possible values of x_0 be denoted by $Bounds(x_0)$.
2. Suppose the difference equation library associates, with the equation template we have matched, the solution $x_n = \mathcal{E}(n, x_0)$, where $\mathcal{E}(\cdots)$ is some expression involving n and x_0. We then solve the following constraint for n:

$$(\exists x_0, x_n, n)[Bounds(x_0) \wedge x_n = \mathcal{E}(n, x_0) \wedge n \geq 0 \wedge C(x_n)]. \qquad (1)$$

If the activation cycle is non-terminating then the constraint (1) will be true for all $n \geq 0$. The constraint solver will return an interval $I \subset [0, \infty]$ (or, if metalevel solvers are used [14], a union I of intervals) and soundness of the solver implies that all n which satisfy this constraint must lie in I. If I is a *proper* subset of $[0, \infty]$ which omits some positive integer m, then termination (in at most m steps) has been proved. If, on the other hand, $I = [0, \infty]$, then nothing has been proved, and rule activation may indeed be nonterminating.

Returning to the rule in Example 1, the difference equation we obtain is $x_n = 0.9x_{n-1}$. For the rule under consideration, we have $C(x) \equiv x > 100$. Suppose that in the system under consideration, $\texttt{MAXVAL} = 10^{100}$, which means $Bounds(x_0) \equiv$ '$x_0 > 100 \wedge x_0 \leq 10^{100}$'. We therefore solve the constraint

$$x_0 > 100 \wedge x_0 \leq 10^{100} \wedge x_n = 0.9^n * x_0 \wedge n \geq 0 \wedge x_n > 100.$$

In this case, the constraint solver yields the solution $n < 2142$.[3]

Notice that the constraint solver gives much more information than simply whether or not the cycle will terminate: it tells us that termination will occur after at most 2142 iterations of the cycle. This may seem high, but it is a result of the very large bound on the initial value x_0: it corresponds to starting out with an average salary of 10^{100}. If tighter bounds are available on the value of x_0, then the bound on the maximum number of iterations of the cycle can be correspondingly tightened. For example, suppose we know, from the integrity constraints on the database, that the maximum (and hence the average) salary cannot exceed 100,000: the bound we get in this case is $n < 66$. Information about the maximum number of iterations of a cycle can be a useful design and/or debugging tool for the database designer, e.g., for detecting inadvertantly omitted integrity constraints. It can also be useful, as discussed at the end of the next section, in application areas such as soft real-time systems, where we may be interested not just in whether a rule activation terminates, but also the maximum number of iterations it may execute.

Recall that a cycle in a directed graph is *simple* if no vertex in the cycle is also part of a different cycle. The following theorem gives the soundness of our termination analysis:

Theorem 1. *The procedure described for termination analysis is sound provided that all cycles in the triggering graph are simple. In other words, if the analysis infers that a cycle terminates, then it in fact terminates (equivalently, any cycle that may not terminate is inferred to be non-terminating).*

[3] Actually, the solution it returns is the interval [0,2141.7258420247217145515 60581]. In our current implementation the execution time for this is about 10 ms.

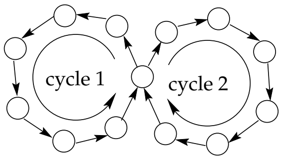

Fig. 2. An example of a non-simple cycle

The reason for the qualification that cycles should be simple is shown in Figure 2. The vertex in the center is part of two different cycles, so it is possible to have an execution where we go around cycle 1 some number of times, then around cycle 2 some number of times, then back around cycle 1, and so on. It may happen that each of the two cycles shown, considered on its own, can be shown to be terminating, but the two taken together do not terminate: this can happen, for example, if cycle 1 inserts some tuples into a relation until a maximum count is reached, while cycle 2 deletes tuples from that relation until a minimum count is reached. We believe that our results can be extended to non-simple cycles provided that the cycles don't "interfere" with each other, in the sense that one of them increases a value that is decreased by the other. We are currently looking into how our ideas may be extended to deal with arbitrary cycles.

Finally, the discussion of the way in which the solution to a difference equation is used for termination analysis can be used to guide the construction of the difference equation library. In particular, it makes no sense to have a very precise solution to a particular equation template if the constraint system is not powerful enough to handle that solution. Thus, knowing the capabilities of the constraint system, we may choose to associate "approximate solutions"—i.e., upper and lower bounds, intended to be used as discussed at the end of Section 3.3—that we know can be handled by by the constraint system, if the exact solution cannot be handled by it.

5 Dynamic Termination Analysis

There may be situations where the approach described in the previous section does not work, i.e., we are unable to prove, statically, that a cyclic rule activation will necessarily terminate. This may happen either because the cycle is, in fact, potentially non-terminating, or because the constraint system is not powerful enough to solve the constraint (1) sufficiently precisely. Conventional static termination analyses would then reject the rule set for not being provably terminating. An alternative, however, would be to use dynamic termination analysis [6], where we insert code into the appropriate active rules to determine, when the rule is activated, whether that particular activation of the rules can be guaranteed to terminate. The latter approach gives us greater flexibility in handling rules, by allowing us to work with rules that may not be provably terminating via static termination analyses, but nevertheless guarantee that at runtime there will not be any nonterminating executions.

The idea can be illustrated by the following variation to Example 1 mentioned at the end of Section 1:

```
if (Select Avg(Sal) from Emp) > 100
then update Emp
     set Sal = 0.9*Sal + Bonus
```

This rule will terminate if Bonus < 10; for values of Bonus ≥ 10 the rule is nonterminating. Thus, if the value of Bonus is not known statically (e.g., if it is computed dynamically based on other values in the database), it will not be possible to prove the termination of this rule statically. Instead of rejecting the rule, however, we can introduce code into it to carry out dynamic termination analysis: the result would be to allow rule activation for situations where the value B of Bonus guarantees termination and rejecting it for values that do not. Suppose that at runtime, this rule is activated with $x_0 = 10,000$ and Bonus $= 9$. Then, given the static solution (see Section 3.3) $x_n = (x_0 - \frac{B}{0.1}) * 0.9^n + \frac{B}{0.1}$ for the difference equation for the corresponding cycle, and the value $B = 9$, this runtime check would use the constraint solve to solve for n in the constraint

$$x_n = (x_0 - \frac{B}{0.1}) * 0.9^n + \frac{B}{0.1} \wedge x_0 = 10000 \wedge B = 9 \wedge n \geq 0 \wedge x_n > 100.$$

In this case, the CLP(F) constraint solver infers the bound $n < 66$, which means that termination (in at most 66 steps) can be guaranteed. In this case, therefore, the dynamic termination test succeeds and the rule execution is allowed to proceed. On the other hand, if at runtime we have Bonus $= 10$, the constraint solver infers the bound $n \in [0, +\infty]$, correctly indicating that the rule activation may not terminate.

An interesting aspect of this kind of dynamic termination analysis is that it allows runtime decisions based not just on whether or not a cycle terminates, but, if we wish, the maximum number of iterations that may be executed. This can be used for controlling rule activation in active databases within soft real-time systems. For example, suppose that based on runtime monitoring of rule activations, we decide that a particular cycle can be allowed to iterate at most 50 times if the timing constraints are to be satisfied. Using our approach, we can test for this before the rule is activated: this allows more flexible systems (cyclic activations are permitted) but at the same time improves resource utilization ("bad" rule activations are rejected ahead of time, instead of having to be aborted if they are found to be running too long).

The overhead of dynamic termination analysis for cycles can be reduced significantly by observing that, once we have verified that a sequence of cyclic rule activations will eventually terminate, it is not necessary to test it again and again as we go around the cycle during that sequence of rule activations. The dynamic termination check can therefore be moved out of the cycle, in a manner similar to the optimization of invariant code motion out of loops commonly carried out in compilers [1].

6 Related Work

There is a significant body of literature on termination analysis for active database rules. Among the earliest of these is the work of Aiken *et al.* [2], who proposed using triggering graphs to reason about termination; this approach has subsequently been refined and improved by various authors [4, 5, 7, 18, 22, 23]. The general idea here is to use acyclicity of the triggering graph to infer termination; the relative precision of different analyses depend on their use of different techniques to remove edges from the triggering graph prior to the acyclicity test.

Weik and Heuer describe an approach to identify terminating cycles in triggering graphs [23]. They consider lattice-structured domains: a cycle is then inferred to be terminating if it represents an increasing operation in the lattice (i.e., values get mapped to "higher" values according to the lattice ordering) with a non-decreasing step size, and there is an upper bound on the resulting values (and dually with decreasing operations). While the goals of this work are similar to ours, the details are very different. Their approach is unable to infer termination for rules such as that in Example 1, since the step size of the operation in this example does not satisfy their criterion for being non-increasing.

Bailey *et al.* use abstract interpretation for termination analysis of active rules [3]. The idea is to reason about sequences of database states using an "approximate semantics," and use fixpoint computation (over a lattice) to handle cycles. The algorithm described by these authors does not have any knowledge of arithmetic operations, and so cannot infer termination of rules such as that in Example 1. A more fundamental problem is the issue of termination of the termination analysis itself. The usual approach taken in the abstract interpretation literature for proving termination of analyses is to assume that the abstract domain is Noetherian, i.e., does not contain any infinite ascending chains; such an assumption, while not explicitly stated, seems necessary for the work of Bailey *et al.* as well. This requirement restricts the structure of the abstract domains they are able to use. The restriction seems especially problematic for situations such as those considered here, where we have numeric domains such as the integers and reals, and where it may not be *a priori* obvious which subsets of these domains may be relevant for a particular rule set. This problem does not arise with our approach because we do not attempt to construct fixpoints iteratively. For this reason, we believe that the approach described in this paper is more precise than that of Bailey *et al.*.

Baralis *et al.* discuss the problem of dynamic termination analysis [6]. Their approach is based on the idea of monitoring rule activations at runtime to detect situations where a database state is repeated during execution, thereby indicating nontermination. This is a sufficient condition for nontermination in general, and is necessary and sufficient for "function-free" rules, which do not introduce any new values into the database. The runtime monitoring of database states can be quite expensive, and Baralis *et al.* propose a number of optimizations to their basic scheme to reduce this cost. Their approach differs from ours in two important ways. First, our approach does not involve keeping track of (representations or encodings of) previously encountered database states, and so can be made more efficient. Second, cyclic activations involving real numbers, as illustrated by the examples considered in this paper, may introduce new values into the database (e.g., the series of values 0.9, 0.9^2, 0.9^3, $\ldots$), and so are not function-free; for such rules, the technique of Baralis *et al.* give a sufficient condition for nontermination but not a necessary one. This means that, at least in principle, there may be nonterminating executions that will not be detected as nonterminating by their analysis; however, such executions will be detected as nonterminating by the approach described in this paper.

The table-driven approach described here for approximate solution of difference equations was developed by us in the context of optimized execution of parallel logic programs [10]. We have subsequently used it for query size analysis for recursive rules in deductive databases [11] and for estimating the computational cost of recursive logic programs [12]. Caslog, a system for cost analysis of logic programs that is based on this work, is available via anonymous FTP from `ftp.cs.arizona.edu/caslog`, and is part of the CIAO-Prolog distribution available at `www.clip.dia.fi.upm.es`. Our implementation of the CLP(F) constraint system is freely available at `www.cs.brandeis.edu/~tim/clip`.

7 Conclusions

Most existing approaches to termination analysis of active database rules rely on verifying that the triggering graph for those rules is acyclic. Because of this, they are unable to handle rules whose triggering graphs are inherently cyclic. Such rules can, nevertheless, be useful because they allow us to express, in a straightforward and natural way, situations that involve the repeated application of a set of active rules until some desired state is reached. This paper describes a constraint-based approach that can be used for termination analysis in such cases. The basic idea is to use a notion of *annotated triggering graphs* to capture the effect of going around a cycle in the triggering graph once, use this to estimate what happens after n executions of the cycle, and verify from this that the cyclic rule activation will eventually terminate. The idea can be readily generalized to allow dynamic termination testing, thereby allowing the analysis to cope with both proof systems that are not sufficiently powerful, and with rules that terminate sometimes but not necessarily always.

Acknowledgements

We are grateful to Elena Baralis for pointers to related work.

References

1. A. V. Aho, R. Sethi and J. D. Ullman, *Compilers – Principles, Techniques and Tools*, Addison-Wesley, 1986.
2. A. Aiken, J. M. Hellerstein, and J. Widom, "Static Analysis Techniques for Predicting the Behavior of Active Database Rules", *ACM Transactions on Database Systems*, vol. 20 no. 1, pp. 63–84, March 1995.
3. J. Bailey, L. Crnogorac, K. Ramamohanarao, and H. Søndergaard, "Abstract Interpretation of Active Rules and Its Use in Termination Analysis", *Proc. 6th. International Conference on Database Theory*, 1997.
4. E. Baralis, S. Ceri, and J. Widom, "Better Termination Analysis for Active Databases", *Proc. First International Workshop on Rules in Database Systems*, Aug. 1993, pp. 163–179.

5. E. Baralis, S. Ceri, and S. Paraboschi, "Improved Rule Analysis by means of Triggering and Activation Graphs", *Proc. 2nd. International Workshop on Rules in Database Systems* (RIDS), Sept. 1995.
6. E. Baralis, S. Ceri and S. Paraboschi, "Compile-Time and Runtime Analysis of Active Behaviors", *IEEE Transactions on Knowledge and Data Engineering* vol. 10 no. 3, May/June 1998, pp. 353–370.
7. E. Baralis and J. Widom, "Better Static Rule Analysis for Active Database Systems", *ACM Transactions on Database Systems*, 2000 (to appear).
8. S. Ceri and J. Widom, "Deriving Production Rules for Constraint Maintenance", *Proc. 16th. VLDB Conference*, Aug. 1990, pp. 566–577.
9. J. Cohen and J. Katcoff, "Symbolic Solution of Finite-Difference Equations," *ACM Transactions on Mathematical Software* 3, 3 (Sept. 1977), pp. 261–271.
10. S. K. Debray, N. Lin and M. Hermenegildo, "Task Granularity Analysis in Logic Programs," *Proc. ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, June 1990, pp. 174–188.
11. S. K. Debray and N. Lin, "Static Estimation of Query Sizes in Horn Programs," *Proc. Third International Conference on Database Theory*, Paris, France, December 1990, pp. 514–528.
12. S. K. Debray and N.-W. Lin, "Cost Analysis of Logic Programs", *ACM Transactions on Programming Languages and Systems*, vol. 15 no. 5, Nov. 1993, pp. 826–875.
13. T. J. Hickey, "Analytic Constraint Solving and Interval Arithmetic", *Proc. 27th. ACM Symposium on Principles of Programming Languages*, Jan. 2000, pp. 338–351.
14. T. J. Hickey, "CLIP: A CLP(Intervals) Dialect for Metaleve Constraint Solving", *Proc. PADL'00*, LNCS vol 173, Jan. 2000, pp. 200-214.
15. J. Ivie, "Some MACSYMA Programs for Solving Recurrence Relations," *ACM Transactions on Mathematical Software* 4, 1 (March 1978), pp. 24–33.
16. J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap, "The CLP($\mathcal{R}$) Language and System", *ACM Transactions on Programming Languages and Systems* vol. 14 no. 3, July 1992, pp. 339–395.
17. J. Jaffar and M. J. Maher, "Constraint Logic Programming: A Survey", *J. Logic Programming* vol. 19.20, May/July 1994, pp. 503–581.
18. A. Karadimce and S. Urban, "Refined Triggering Graphs: A Logic Based Approach to Termination Analysis in an Active Object-Oriented Database", *Proc. 12th. International Conference on Data Engineering*, 1996.
19. H. Levy and F. Lessman, *Finite Difference Equations*, Sir Isaac Pitman & Sons, London, 1959.
20. M. Petkovsek, *Finding Closed-Form Solutions of Difference Equations by Symbolic Methods*, PhD Thesis, Carnegie Mellon University, 1991.
21. Swedish Institute of Computer Science, *SICStus Prolog User Manual*, Release 3.8, Oct. 1999.
22. A. Vaduva, S. Gatziu, and K. R. Dittrich, "Investigating Termination in Active Database Systems with Expressive Rule Languages", *Proc. 3rd International Workshop on Rules in Database Systems*, June 1997.
23. T. Weik and A. Heuer, "An Algorithm for the Analysis of Termination of Large Trigger Sets in an OODBMS", *Proc. International Workshop on Active and Real-Time Database Systems*, June 1995.
24. J. Widom, "The Starburst Active Database Rule System", *IEEE Transactions on Knowledge and Data Engineering*, 8(4):583-595, August 1996.
25. C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subramanian, and R. Zicari, *Advanced Database Systems*, Morgan Kaufman, 1997.