Verifying the Incorrectness of Programs and Automata*

Scot Anderson and Peter Revesz

Department of Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln, NE 68588, USA
{scot,revesz}@cse.unl.edu

Abstract. Verification of the incorrectness of programs and automata needs to be taken as seriously as the verification of correctness. However, there are no good general methods that always terminate and prove incorrectness. We propose one general method based on a *lower bound* approximation of the semantics of programs and automata. Based on the lower-bound approximation, it becomes easy to check whether certain error states are reached. This is in contrast to various abstract interpretation techniques that make an *upper bound* approximation of the semantics and test that the error states are not reached. The precision of our lower bound approximation is controlled by a single parameter that can be adjusted by the user of the MLPQ system in which the approximation method is implemented. As the value of the parameter decreases the implementation results in a finer program semantics approximation but requires a longer evaluation time. However, for all input parameter values the program is guaranteed to terminate. We use the lower bound approximation to verify the incorrectness of a subway train control automaton. We also use the lower bound approximation for a problem regarding computer security via trust management programs. We propose a trust management policy language extending earlier work by Li and Mitchell. Although, our trust management programming language is Turing-complete, programs in this language have semantics that lend themselves naturally to a lower-bound approximation. Namely, the lower bound approximation is such that no unwarranted authorization is given at any time, although some legitimate access may be denied.

1 Introduction

Testing the correctness of a program or an automaton can be done by finding an *upper approximation* of its semantics. If the upper approximation *does not* contain the error states needed to be checked, then the automaton can be said to be correct. However, if the upper approximation *contains* the error states, then the actual program or automaton may still be correct.

Similarly, if the *lower bound* approximation of the semantics contains an error state, then we know that it is incorrect. If it does not, then the program may still be incorrect.

Hence an *upper bound* may be good to verify that a program is correct, while a *lower bound* may be good to verify that it is incorrect. The *verification of incorrectness* is just as important in practice as the verification of correctness, because many users are reluctant to change incorrect and expensive programs unless those are proven incorrect. For example, if a banking system allows invalid access to some bank accounts, then a lower bound approximation would be needed to verify the incorrectness.

* This research was supported in part by NSF grant EIA-0091530 and a NASA Space and EPSCoR grant.

Until recently, in the verification area the focus was in verifying correctness using *abstract interpretation* [8, 16, 22] or *model checking* [1, 5, 9, 30, 36]. In contrast, in this paper, we focus on verifying incorrectness.

Verifying incorrectness is needed when we suspect a program to be incorrect, and we want to prove that it is indeed incorrect. For example, if there is an accident with a space shuttle, then we need to find what caused it. Was it caused by an incorrect program?

There are many reasons that a program may be suspected to be incorrect. For example, a program that fails a verification for correctness using abstract interpretation or model checking would be suspicious.

There are some problems that naturally lend themselves to a lower-bound approximation. For example, the semantics of a computer security system would contain the facts that describe who gets access to which resource at what time. In this case a lower-bound approximation is meaningful, conservative, and safe to use. That is, it never gives unwarranted authorizations, although some legitimate access may be denied at certain time instances. For example, not being able to access one's own bank account at a particular time is frustrating, but it is certainly less frustrating than if someone else, who should not, can access it.

We use the above idea in proposing a Turing-complete extension of the *trust management* language RT [25–27], which is a recent approach to computer security in a distributed environment. The latest version of the RT language uses Datalog but with simpler constraints than we allow in this paper. We choose the RT trust management family of languages *as an example* of how to use constraint database approximation techniques in other areas beyond database systems where lower-bound approximations are meaningful. (See the survey [15] and the recent article [24] about trust management in general.)

The rest of this paper is organized as follows. Section 2 gives a brief review of constraint database approximation theory and its implementation in the MLPQ constraint database system [38]. Section 3 applies the approximation method to verify the incorrectness of an automaton. Section 4 applies the approximation method to find a safe evaluation of a trust management program. Section 5 discusses some related work. Finally, Section 6 gives some conclusions and future work.

2 Review of Constraint Database Approximation Theory

The *constraint logic programming* languages proposed by Jaffar and Lassez [17], whose work led to CLP(R) [19], by Colmerauer [7] within Prolog III, and by Dincbas et al. [10] within CHIP, were Turing-complete. Kanellakis, Kuper, and Revesz [20, 21] considered those to be impractical for use in database systems and proposed less expressive *constraint query languages* that have nice properties in terms of guaranteed and efficient evaluations. Many researchers advocated extensions of those languages while trying to keep termination guaranteed. For example, the least fixed point semantics of Datalog (Prolog without function symbols and negation) with integer gap-order constraint programs can be always evaluated in a finite constraint database representation [33].[1]

With gap-order constraints many NP-complete problems can be expressed that cannot be expressed in Datalog without constraints. However, even Datalog with addition constraints, which seems only a slight extension, is already Turing-complete. Hence Revesz [35] introduced an approximate evaluation for Datalog with addition constraints.

[1] A gap-order is a constraint of the form $x - y \geq c$ or $\pm x \geq c$ where x and y are variables and c is a non-negative integer constant.

This approximation is different from *abstract interpretation* methods (for a recent review see [8]). The main difference is that, at least in theory, in [35] both a lower and an upper bound approximation of the least fixed point can be arbitrarily close to the actual least fixed point with the decrease of a single parameter towards $-\infty$. The decrease indirectly increases the running time.

Below we focus on the definitions that are relevant to approximations. The reader can find more details in the surveys [18, 34] and the books [23, 28, 37] about constraint logic programming and constraint databases.

Definition 1. *Addition constraints [37] have the form*

$$\pm x \pm y \ \theta \ b \quad or \quad \pm x \ \theta \ b$$

where x and y are integer variables and b is an integer constant, called a bound*, and θ is either $\geq$ or $>$.*

In the following we will also use $x = b$ as an abbreviation for the conjunction of $x \geq b$ and $-x \geq -b$. Similarly, we use $x + y = b$ as an abbreviation for the conjunction of $x + y \geq b$ and $-x - y \geq -b$.

Each *constraint database* is a finite set of *constraint tuples* of the form:

$$R(x_1, \ldots, x_k) \ :- \ C_1, \ldots, C_m.$$

where R is a k-ary relation symbol, each x_i for $1 \leq i \leq k$ is an integer variable or constant, and each C_j for $1 \leq j \leq m$ is an addition constraint over the variables. The meaning of a constraint tuple is that each substitution of the variables by integer constants that makes each C_j on the right hand side of $:-$ true is a k-tuple that is in relation R.

A *Datalog program* consists of a finite set of constraint tuples and rules of the form:

$$R_0(x_1, \ldots, x_k) \ :- \ R_1(x_{1,1}, \ldots, x_{1,k_1}), \ldots, R_n(x_{n,1}, \ldots, x_{n,k_n}), \ C_1, \ldots, C_m.$$

where each R_i is a relation name, and the xs are either integer variables or constants, and each C_j is an addition constraint over the xs. The meaning of the rule is that if for some substitution of the variables by integer constants each R_i and C_j on the right hand side of $:-$ is true, then the left hand side is also true.

A *model* of a Datalog program is an assignment to each k-arity relation symbol R within the program a subset of $\mathbb{Z}^k$ where $\mathbb{Z}$ is the set of integers such that each rule holds for each possible substitution. The *least fixed point* semantics of a Datalog program contains the intersection of all the models of the program.

It is easy to express in Datalog [37] with addition constraints a program that will not terminate using a standard bottom-up evaluation [37]. Consider the following Datalog with addition constraint program:

$$\begin{array}{lll} D(x, y, z) & :\!- & x - y = 0, \quad z = 0. \\ D(x', y, z') & :\!- & D(x, y, z), \quad x' - x = 1, \quad z' - z = 1. \end{array} \tag{1}$$

This expresses that the *Difference* of x and y is z. Further, based on (1) we can also express a *Multiplication* relation as follows:

$$\begin{array}{lll} M(x, y, z) & :\!- & x = 0, \quad y = 0, \quad z = 0. \\ M(x', y, z') & :\!- & M(x, y, z), \quad D(z', z, y), \quad x' - x = 1. \\ M(x, y', z') & :\!- & M(x, y, z), \quad D(z', z, x), \quad y' - y = 1. \end{array} \tag{2}$$

Intuitively, a standard bottom-up evaluation derives additional constraint tuples until a certain saturation is reached, and the saturation state represents in a constraint database form the least fixed point. We omit the precise definition of bottom-up evaluation of Datalog with constraint programs, because it is not needed for the rest of this paper. It is enough to note that the simple Datalog program that consists of the above two sets of rules never terminates in a standard bottom-up evaluation.

In fact, with these two relations we can express any integer polynomial equation (see Example 3). Since integer polynomial equations are unsolvable in general [29], no algorithm would be able to evaluate precisely the least fixed point semantics of the Datalog program. Hence the situation we face is not just a particular problem with the standard bottom-up evaluation, but a problem that is inherent to the least fixed point semantics of Datalog with addition constraints.

Revesz [35] introduced two methods for approximating the least fixed point evaluation by modifying the standard bottom-up evaluation.

Definition 2. *Let $l < 0$ be any fixed integer constant. We change in the constraint tuples the value of any bound b to be $max(b, l)$. Given a Datalog program P the result of a bottom-up evaluation of P using this modification is denoted P_l.*

Definition 3. *Let $l < 0$ be any fixed integer constant. We delete from each constraint tuple any constraint with a bound that is less than l. Given a Datalog program P the result of a bottom-up evaluation of P using this modification is denoted P^l.*

These modifications lead to the following approximation theorem.

Theorem 1. *[35] For any Datalog program P and constant $l < 0$ the following is true.*

$$P_l \subseteq lfp(P) \subseteq P^l$$

where $lfp(P)$ is the least fixed point of P. Further, P_l and P^l can be computed in finite time.

We can also get better and better approximations using smaller and smaller values of l. In particular, we have the following theorem.

Theorem 2. *[35] For any Datalog with addition constraints program P and constants l_1 and l_2 such that $l_1 \leq l_2 < 0$, the following hold:*

$$P_{l_2} \subseteq P_{l_1} \quad \text{and} \quad P^{l_1} \subseteq P^{l_2}$$

Because we are interested in evaluations that are lower bounds of the least fixed point $lfp(P)$, we implemented P_l as defined in Definition 2. The implementation was done within the MLPQ constraint database system [38], which is available from the website: `cse.unl.edu/~revesz`. The implementation is a new result that is not described in any other publication.

3 Verifying the Incorrectness of a Subway Automaton

We consider counter automata $\mathcal{A}$ which are tuples $(S, X, \tau, s_0, \overline{x}_0)$ where S is a finite set of states, X is a finite set of state counters $x_1, \ldots, x_k$ which are integer variables, τ is a finite set of transitions from S to S, s_0 is an initial state, and $\overline{x}_0$ is an initial assignment of the state variables. Each

transition has two parts, a *guard constraint* over the variables that needs to be satisfied before the transition takes place and a set of assignments to the variables that update their values as the automaton enters the new state. In this paper we allow only addition constraints in the guard constraints and assignments that can be expressed by addition constraints.

Counter machines are an example of such automata which allow only guard constraints that are comparisons between variables and constants and assignments that increment and decrement a variable by one or set a variable to a constant. They were studied by Minsky [31, 32], who showed that they have the same expressive power as Turing machines. Floyd and Beigel [11] is an introduction to automata theory that covers counter machines.

More complex guard constraints have been allowed in later extensions of counter machines and applied to the design of control systems in Boigelot and Wolper [4], Fribourg and Olson [12], Fribourg and Richardson [13], Halbwachs [16], and Kerbrat [22]. Boigelot et al. [3], Cobham [6], and Wolper and Boigelot [39] study automata and Presburger definability. For additional discussion and examples of various types of counter (constraint) automata see [37].

Let us consider the following subway train speed regulation system described by Halbwachs [16]. Each train detects beacons that are placed along the track and receives a "second" signal from a central clock.

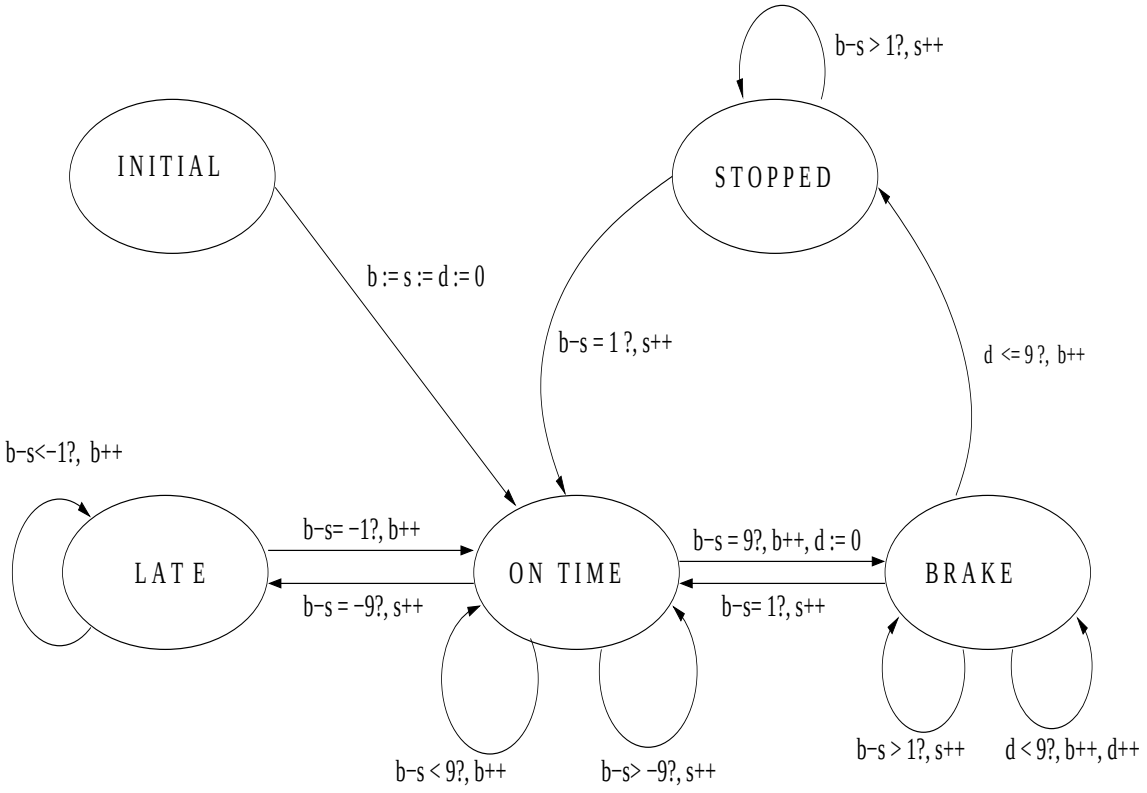

Fig. 1. The subway train control system.

Let b and s be counter variables for the number of beacons and second signals received. Further, let d be a counter variable that describes how long the train is applying its brake. The goal of the speed regulation system is to keep $\mid b - s \mid$ small while the train is running.

The speed of the train is adjusted as follows. When $s + 10 \leq b$, then the train notices it is early and applies the brake as long as $b > s$. Continuously braking causes the train to stop before encountering 10 beacons.

When $b + 10 \leq s$ the train is late and will be considered late as long as $b < s$. As long as any train is late, the central clock will not emit the second signal.

The subway speed regulation system can be drawn as a constraint automaton shown in Figure 3, where the guard constraints are followed by question marks, and $x{+}{+}$ and $x{-}{-}$ are abbreviations for the assignments $x := x + 1$ and $x := x - 1$, respectively, for any variable x.

The set of reachable configurations (combinations of states and state variable values) of the automaton shown in Figure 3 can be expressed in Datalog with addition constraints by creating a new ternary relation for each state with the order of variables (b, d, s) and writing the following Datalog with addition constraints rules:

$$
\begin{array}{lcl}
Brake(b, s', d) & :\!- & Brake(b, s, d),\quad b - s > 1,\quad s' - s = 1. \\
Brake(b', s, d') & :\!- & Brake(b, s, d),\quad -d > -9,\quad b' - b = 1,\quad d' - d = 1. \\
Brake(b', s, d') & :\!- & Ontime(b, s, d),\quad b - s = 9,\quad b' - b = 1,\quad d' = 0. \\
 & & \\
Initial(b, s, d) & :\!- & b = 0,\quad s = 0,\quad d = 0. \\
 & & \\
Late(b', s, d) & :\!- & Late(b, s, d),\quad -b + s > 1,\quad b' - b = 1. \\
Late(b, s', d) & :\!- & Ontime(b, s, d),\quad b - s = -9,\quad s' - s = 1. \\
 & & \\
Ontime(b, s', d) & :\!- & Brake(b, s, d),\quad b - s = 1,\quad s' - s = 1. \\
Ontime(b, s, d) & :\!- & Initial(b, s, d). \\
Ontime(b', s, d) & :\!- & Late(b, s, d),\quad b - s = -1,\quad b' - b = 1. \\
Ontime(b', s, d) & :\!- & Ontime(b, s, d),\quad -b + s > -9,\quad b' - b = 1. \\
Ontime(b, s', d) & :\!- & Ontime(b, s, d),\quad b - s > -9,\quad s' - s = 1. \\
Ontime(b, s', d) & :\!- & Stopped(b.s.d),\quad b - s = 1,\quad s' - s = 1. \\
 & & \\
Stopped(b', s, d) & :\!- & Brake(b, d, s),\quad -d \geq -9,\quad b' - b = 1. \\
Stopped(b, s', d) & :\!- & Stopped(b, s, d),\quad b - s > 1,\quad s' - s = 1.
\end{array}
$$

Error Condition: Suppose that this automaton is correct if $|b - s| < 20$ in all states at all times. Then this automaton is incorrect if $|b - s| \geq 20$ at least in one state at one time. The table below shows the result of the lower bound approximation using the MLPQ constraint database system.

MLPQ Lower-Bound

Brake	Late	Ontime	Stopped
$1 \leq b - s \leq 19$	$-10 \leq b - s \leq -1$	$-9 \leq b - s \leq 9$	$1 \leq b - s \leq 20$
$10 \leq b \leq 19$	$10 \leq s \leq 19$	$0 \leq b \leq 9$	$11 \leq b \leq 20$
$0 \leq s \leq 18$	$0 \leq d \leq 9$	$0 \leq s \leq 18$	$0 \leq s \leq 9$
$0 \leq d \leq 9$		$0 \leq d \leq 9$	$0 \leq d \leq 9$

The above was obtained by using $l = -30$ as in Definition 2. If l is decreased, then the upper bounds of b and s increase in the above table. Therefore, in the limit those upper bounds can be dropped.

Further, for any value of u, since the above is a lower bound, any possible integer solution of the constraints below the state names *must occur* at some time. For example, the *Stopped* state must contain the case $b - s = 20$ at some time. Therefore, this automaton is incorrect by our earlier assumption.

3.1 Comparison with Verimag

The Verimag laboratory has software for testing program correctness using abstract interpretation. Halbwachs [16] gave the following upper bound derived using Verimag's software for the subway automaton.

Verimag Upper-Bound

Brake	Late	Ontime	Stopped
$1 \leq b - s \leq d + 10$	$-10 \leq b - s \leq -1$	$-9 \leq b - s \leq 9$	$1 \leq b - s \leq 19$
$d + 10 \leq b$	$s \geq 10$	$b \geq 0$	$19 \leq 9s + b$
$0 \leq d \leq 9$		$s \geq 0$	$b \geq 10$

Surprisingly, this result does not match our result. In particular, the upper bound for the *Stopped* state contains the constraint $b - s \leq 19$, which says that the value of $b - s$ cannot be 20, but our lower bound says that 20 must be one of the cases. To resolve this apparent contradiction, we need to look more closely at the automaton. We can see that the following is a valid sequence of transitions, where $S(b, s, d)$ represents the values of b, s, and d is each state $S \in \{Brake, Initial, Late, Ontime, Stopped\}$.

$Initial(0,0,0) \longrightarrow Ontime(0,0,0) \longrightarrow Ontime(1,0,0). \longrightarrow Ontime(2,0,0) \longrightarrow Ontime(3,0,0) \longrightarrow$
$Ontime(4,0,0) \longrightarrow Ontime(5,0,0) \longrightarrow Ontime(6,0,0) \longrightarrow Ontime(7,0,0) \longrightarrow Ontime(8,0,0) \longrightarrow$
$Ontime(9,0,0) \longrightarrow Brake(10,0,0) \longrightarrow Brake(11,0,1) \longrightarrow Brake(12,0,2) \longrightarrow Brake(13,0,3) \longrightarrow$
$Brake(14,0,4) \longrightarrow Brake(15,0,5) \longrightarrow Brake(16,0,6) \longrightarrow Brake(17,0,7) \longrightarrow Brake(18,0,8) \longrightarrow$
$Brake(19,0,9) \longrightarrow Stopped(20,0,9)$

Note that $Stopped(20,0,9)$ contradicts the first constraint in the Verimag upper bound for the *Stopped* state. Hence we suspect that the Verimag software contains some bug or there was some problem in data entry. We suggest that its incorrectness be tested using other examples and our lower-bound method.

4 Approximating Trust Management Program Semantics

Trust management languages allow the expression of high-level rules about which principal can get access to which resource at what time in a distributed environment. The *Keynote* trust management system [2, 25] allowed integer polynomial constraints. However, later trust management systems do not allow such constraints, because allowing them leads to undecidability [29].

We argue that this restriction unnecessarily limits the expressibility of trust management languages. Our lower bound method can be used in most cases to verify the correctness of an access even when the rules contain integer polynomial constraints.

Note that an upper bound approximation may allow some access which is not specified by the trust management rules. Hence it is not appropriate for trust management, while a lower bound technique can be safely used. In a computer security system it leads to much less harm if a legitimate access request is denied (which can happen with a lower bound approximation) than if an illegitimate access is allowed (which can happen with an upper bound approximation).

Integer polynomial constraints arise naturally in security applications, as shown by the following example.

Example 1. Suppose an e-mail sender or server organization C needs to assign a level of trust to an individual based on the trust levels assigned by organizations A and B. Suppose C considers B's information much more valuable. Then C may use the following integer polynomial constraint to assign a trust level of its own:

$$3Level_C \ \geq \ 4(Level_A)^2 \ + \ 2Level_B \tag{3}$$

4.1 Extended RT Syntax

RT is a trust management policy language introduced by Li et al. [27]. The parameters in each of the different kinds of policy statements in RT define the relationships between *principal owners*, the *roles* they own and the *members* of the roles.

The following *simple member* rule defines the principal K_D to be a member of role R owned by K_A:

$$K_A.R(p_1, ..., p_n) \ \leftarrow \ K_D \tag{4}$$

where the role takes the form $R(p_1, ..., p_n)$, and R is a role name and each p_j is a variable in an order constraint.

Extended RT: We extend the RT language by allowing each p_j to be an integer variable within an integer polynomial constraint. The *extended simple member* rules have the syntax:

$$K_A.R(p(x_1, ..., x_l)) \ \leftarrow \ K_D \tag{5}$$

where K_A defines a role R to contain member K_D, if the integer polynomial constraint $p(x_1, ..., x_l)$ holds.

Example 2. The extended *RT* statement

$$Email.Permit(3Level_C - 4(Level_A)^2 - 2Level_B \geq 0) \leftarrow \text{"}Charlie\text{"} \tag{6}$$

allows Charlie access to e-mail, if the ratings obtained satisfy constraint (3).

4.2 Extended RT Semantics

The semantics of an extended RT program can be found by translating the extended RT rules into logically equivalent Datalog with addition constraints rules and then taking the least fixed point semantics of the resulting Datalog program.

Extended simple member rules of form (4) are translated into the following Datalog rule:

$$R(K_A, K_D, x_1, ..., x_m) \ :- \ p_1, ..., p_n. \tag{7}$$

where $x_1, ..., x_m$ are the integer constants and variables that may be used within the polynomial constraints $p_1, ..., p_n$.

We can translate integer polynomial constraints with k number of $+$ and $\times$ operations into a conjunction of at most $2k$ difference D and multiplication M relations defined in Section 2.

Example 3. The extended *RT* statement (6) can be translated into the following Datalog with addition constraints rule:

$$\begin{aligned} Permit(Email, Charlie, Level_A, Level_B, Level_C) :- \ & M(t_1,\ 3,\ Level_C), \\ & M(t_2,\ Level_A,\ Level_A), \\ & M(t_3,\ 4,\ t_2), \\ & M(t_4,\ 2,\ Level_B), \\ & D(e_1,\ t_1,\ t_3), \\ & D(e_2,\ e_1,\ t_4), \\ & e_2 \geq 0 \end{aligned}$$

where $Level_A, Level_B$ and $Level_C$ are either integer variables or constants and are the important parameters in this problem, while each t_i and e_1 and e_2 are additional integer variables that are introduced only for the sake of expressing the polynomial equation. Finally, *Email* and *Charlie* are integer constants that represent the strings "Email" and "Charlie" in the RT statement (6).

Since the difference D and multiplication M relations have already been defined in Section 2 using Datalog with addition constraints, the entire Datalog program can be evaluated using the lower-bound approximation of its least fixed point by a modified bottom-up evaluation. By Theorem 1 this evaluation terminates, giving a lower bound of the semantics of the extended RT program.

5 Related Work

There are few papers on lower bounds for automata and programs. Godefroid et al. [14] gives a lower-bound approximation of the automaton by simplifying its states according to some predicates that hold in each state. The state transitions considered in the simplification are must-transitions, that is, if the condition in the previous state holds, then only one subsequent state can be reached. Unfortunately, this is very limited, because in fact most transitions among states are not must-transitions. In general, predicate abstraction methods, such as [14], can yield more precise approximations with the introduction of additional predicates, making the automata structures increasingly more complex.

In contrast, our approximation is radically different and does not change at all the automata structure, rather it indirectly changes the algebra in which (polynomial) constraints are interpreted and solved. Essentially, the simplified algebra relies on modified addition and multiplication relations. These relations are smoothly and naturally extended as l decreases. Hence our method may yield more precise approximations without increasing the size of the automaton.

6 Conclusions and Future Work

We have seen that in general decreasing the bound l toward $-\infty$ leads to tighter lower and upper bound approximations, P_l and P^l, respectively. If the lower and upper bound approximations agree (i.e, $P_l = P^l$), then we know that we have found the precise least fixed point. However, even if they do not agree, but seem to converge to the same value –and that may be apparent from only a few examples of l values, then we still can give the limit of convergence as the precise least fixed point. The subtle point is that the series of lower (upper) bound approximations themselves show a convergence and hence their limits can be approximated. It is an approximation of approximations, but it may work beautifully in many cases.

Open Problem: Determine the precise conditions under which the least fixed point can be predicted as described above.

We have to be cautious not to overclaim the potential of the above approach, for it is easy to see that the above method may fail sometimes. For example, consider any query that requires a polynomial integer equation that is build using the `Diff` and `Mult` relations. Clearly, the solutions of the polynomial equation can be found using the `Diff` and `Mult` relations built using lower or upper bound approximation. However, decreasing l does not guarantee finding tighter lower and upper bound approximations for the polynomial equation. It may be impossible to tell when all the solutions will be found. Since integer polynomial equations are undecidable in general, there always will be some cases when the convergence is unpredictable.

Considering a general difference constraint, if the upper bound approximation is infinite, then it may not be possible to predict how the two bounds approach one another. In fact the lower bound may grow without bound as l approaches $-\infty$.

Hence in general we conclude by the examples above that there exists a class of queries that are not stable. We also conjecture that when $|P^l \backslash P_l| = \infty$ and the expected solution is finite the query is not stable.

In the future, we would like to determine which classes of queries are stable and implement routines that predict an accurate solution to improve the approximation process when enough evidence is collected to make a precise prediction of convergence. That would result in a kind of approximation that is neither a simple lower nor a simple upper bound approximation but is something much more sophisticated.

In conclusion,

References

1. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., and Yovine, S. The algorithmic analysis of hybrid systems. *Theoretical Computer Science 138*, 1 (1995), 3–34.
2. Blaze, M., Feigenbaum, J., and Lacy, J. Decentralized trust management. Tech. Rep. 96-17, AT and T Research, 1996.
3. Boigelot, B., Rassart, S., and Wolper, P. On the expressiveness of real and integer arithmetic automata. In *International Colloquium on Automata, Languages and Programming* (1998), vol. 1443 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 152–63.
4. Boigelot, B., and Wolper, P. Symbolic verification with periodic sets. In *Proc. Conference on Computer-Aided Verification* (1994), pp. 55–67.

5. Clarke, E. M., Grumberg, O., and Peled, D. A. *Model Checking*. MIT Press, 1999.
6. Cobham, A. On the base-dependence of sets of numbers recognizable by finite automata. *Mathematical Systems Theory 3* (1969), 186–92.
7. Colmerauer, A. Note sur Prolog III. In *Proc. Séminaire Programmation en Logique* (1986), pp. 159–174.
8. Cousot, P. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *Sixth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)* (Paris, France, LNCS 3385, Jan. 17–19 2005), Springer, Berlin, pp. 1–24.
9. Delzanno, G., and Podelski, A. Model checking in CLP. In *2nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (1999), vol. 1579 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 74–88.
10. Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T., and Berthier, F. The constraint logic programming language chip. In *Proc. Fifth Generation Computer Systems* (Tokyo, Japan, 1988), pp. 693–702.
11. Floyd, R. B., and Beigel, R. *The Language of Machines: An Introduction to Computability and Formal Languages*. Computer Science Press, 1994.
12. Fribourg, L., and Olsén, H. A decompositional approach for computing least fixed-points of datalog programs with Z-counters. *Constraints 2*, 3–4 (1997), 305–36.
13. Fribourg, L., and Richardson, J. D. C. Symbolic verification with gap-order constraints. In *Proc. Logic Program Synthesis and Transformation* (1996), vol. 1207 of *Lecture Notes in Computer Science*, pp. 20–37.
14. Godefroid, P., Huth, M., and Jagadeesan, R. Abstraction-based model checking using modal transition systems. In *12th International Conference on Concurrency Theory* (2001), pp. 426–440.
15. Grandison, T., and Sloman, M. A survey of trust in internet application. *IEEE Communications Surveys and Tutorials 3*, Fourth Quarter (2000).
16. Halbwachs, N. Delay analysis in synchronous programs. In *Proc. Conference on Computer-Aided Verification* (1993), pp. 333–46.
17. Jaffar, J., and Lassez, J. L. Constraint logic programming. In *Proc. 14th ACM Symposium on Principles of Programming Languages* (1987), pp. 111–9.
18. Jaffar, J., and Maher, M. Constraint logic programming: A survey. *J. Logic Programming 19/20* (1994), 503–581.
19. Jaffar, J., Michaylov, S., Stuckey, P. J., and Yap, R. H. The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems 14*, 3 (1992), 339–95.
20. Kanellakis, P. C., Kuper, G. M., and Revesz, P. Constraint query languages. In *Proc. ACM Symposium on Principles of Database Systems* (1990), pp. 299–313.
21. Kanellakis, P. C., Kuper, G. M., and Revesz, P. Constraint query languages. *Journal of Computer and System Sciences 51*, 1 (1995), 26–52.
22. Kerbrat, A. Reachable state space analysis of lotos specifications. In *Proc. 7th International Conference on Formal Description Techniques* (1994), pp. 161–76.
23. Kuper, G. M., Libkin, L., and Paredaens, J., Eds. *Constraint Databases*. Springer-Verlag, 2000.
24. Li, N., and Mitchell, J. Understanding SPKI/SDSI using first-order logic. In *Proc. IEEE Computer Security Foundations Workshop* (2003), pp. 89–108.
25. Li, N., and Mitchell, J. C. Datalog with constraints: A foundation for trust management languages. In *Proceedings of the Fifth International Symposium on Practical Aspects of Declarative Languages* (Jan. 2003), pp. 58–73.
26. Li, N., and Mitchell, J. C. RT: A role-based trust-management framework, April 2003.
27. Li, N., Mitchell, J. C., and Winsborough, W. H. Design of a role-based trust management framework. In *Proc. IEEE Symposium on Security and Privacy, Oakland* (May 2002).
28. Marriott, K., and Stuckey, P. J. *Programming with Constraints: An Introduction*. MIT Press, 1998.

29. Matiyasevich, Y. Enumerable sets are diophantine. *Doklady Akademii Nauk SSR 191* (1970), 279–82.
30. McMillan, K. *Symbolic Model Checking*. Kluwer, 1993.
31. Minsky, M. L. Recursive unsolvability of Post's problem of "tag" and other topics in the theory of Turing machines. *Annals of Mathematics 74*, 3 (1961), 437–55.
32. Minsky, M. L. *Computation: Finite and Infinite Machines*. Prentice Hall, 1967.
33. Revesz, P. A closed-form evaluation for Datalog queries with integer (gap)-order constraints. *Theoretical Computer Science 116*, 1 (1993), 117–49.
34. Revesz, P. Constraint databases: A survey. In *Semantics in Databases*, L. Libkin and B. Thalheim, Eds., vol. 1358 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998, pp. 209–46.
35. Revesz, P. Datalog programs with difference constraints. In *Proc. 12th International Conference on Applications of Prolog* (1999), pp. 69–76.
36. Revesz, P. Reformulation and approximation in model checking. In *Proc. 4th International Symposium on Abstraction, Reformulation, and Approximation* (2000), B. Choueiry and T. Walsh, Eds., vol. 1864 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 124–43.
37. Revesz, P. *Introduction to Constraint Databases*. Springer-Verlag, 2002.
38. Revesz, P., Chen, R., Kanjamala, P., Li, Y., Liu, Y., and Wang, Y. The MLPQ/GIS constraint database system. In *ACM SIGMOD International Conference on Management of Data* (2000).
39. Wolper, P., and Boigelot, B. An automata-theoretic approach to Presburger arithmetic constraints. In *Proc. Static Analysis Symposium* (1995), vol. 983 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 21–32.