Reformulation and Approximation in Model Checking*

Peter Z. Revesz

University of Nebraska-Lincoln, Lincoln, NE 68588, USA
revesz@cse.unl.edu
http://cse.unl.edu/∼revesz

Abstract. Symbolic model checking of various important properties like reachability, containment and equivalence of constraint automata could be unsolvable problems in general. This paper identifies several classes of constraint automata for which these properties can be guaranteed to be solvable by reformulating them as the evaluation problem of solvable or approximately solvable classes of constraint logic problems. The paper also presents rewrite rules to simplify constraint automata and illustrates the techniques on several example control systems.

1 Introduction

Several types of constraint automata are used in a natural way to model the operation of systems and processes. Some of the early types of constraint automata include counter machines with increment and decrement by one operators and comparison operators as guard constraints [20, 21] and Petri nets that are equivalent to vector addition systems [22, 24]. Other types of constraint automata with more complex guard constraints are applied to the design of control systems [2, 6, 7, 8, 10, 15]. In this paper we use a particular type of constraint automata that contains read operators and existentially quantified variables in guard constraints (see the definition in Section 2.1).

While the ease of modeling by constraint automata is useful for the description of systems, symbolic model checking, i,e, answering several natural questions about constraint automata, is unsolvable in general [19]. In fact, even counter machines are theoretically as expressive as Turing machines [20, 21], which means that reachability, i.e., checking whether the system will ever reach some given configuration, is undecidable. For Petri nets the reachability problems is decidable [16, 18], but some other natural problems like the equivalence between two Petri nets is undecidable.

The potential of reformulating symbolic model checking problems as decidability problems of various questions about the model of constraint logic programs [12] or constraint query languages [13, 14] was noticed by many authors. However, the model of constraint logic programs is not computable in general,

* This research was supported in part by NSF grant IRI-9625055 and a Gallup Research Professorship.

hence many model checking proposals that use this approach yield possibly non-terminating procedures.

In contrast, in this paper, we aim at guarantees of termination. We do that in two steps, first, if possible, then we simplify the constraint automata by some rewriting rules. Second, we rewrite the simplified constraint automata into solvable classes of constraint logic programs, in particular, Datalog programs with the following classes of constraints: (1) gap-order constraints, (2) gap-order and positive linear inequality constraints, (3) gap-order and negative linear inequality constraints. In each of these cases, we can use the constraint logic program to find (in a constraint database form) the set of *reachable configurations* of the original constraint automata, that is, the set of states and state values that a constraint automaton can enter [27, 25]. This leads to a decidability of both the reachability and the containment and equivalence problems.

For constraint automata for which such reformulation of the original problem does not lead to a solution, some form of approximation can be used. For example, approximation methods for analyzing automata with linear constraints are presented in [15, 8]. Both of these approximation methods yield an upper bound on the set of state configurations by relaxing some of the constraints (in fact, [8] relaxes them to gap-order constraints). However, in many cases of these approaches the upper bound is quite loose. We present an approximation method that derives arbitrarily tight upper and lower bounds for those constraint automata that can be expressed as Datalog with difference constraint programs.

The rest of the paper is organized as follows. Section 2.1 defines and gives several examples for constraint automata. Section 2.2 defines Datalog programs with constraints and several main classes of constraints. Section 2.3 reviews approximate evaluation methods for Datalog with difference constraints. Section 3 presents some reduction rules that can be used to rewrite the constraint automata into equivalent constraint automata. Section 4 presents a method of analyzing the reachable configurations of constraint automata by expressing the constraint automata in Datalog with constraints. Section 5 discusses some more related work. Finally, Section 6 gives some conclusions and directions for further work.

2 Basic Concepts

2.1 Constraint Automata

A constraint automaton consists of a set of states, a set of state variables, transitions between states, an initial state and the domain and initial values of the state variables. Each transition consists of a set of constraints, called the *guard* constraints, followed by a set of assignment statements. The guard constraints of a constraint automaton can contain relations. In constraint automata the guards are followed by question marks, and the assignment statements are shown using the symbol :=.

A constraint automaton can move from one state to another state if there is a transition whose guard constraints are satisfied by the current values of the state variables. The transitions of a constraint automaton may contain variables in addition to the state variables. These variables are said to be *existentially quantified* variables. Their meaning is that some values for these variables can be found such that the guard constraints are satisfied.

A constraint automaton can interact with its environment by sensing the current value of a variable. This is expressed by a $read(x)$ command on a transition between states, where x is any variable. This command updates the value of x to a new value. The read command can appear either before or after the guard constraints.

Each constraint automaton can be drawn as a graph in which each vertex represents a state and each directed edge represents a transition.

Drawing a constraint automata can be a good way to design a control system. The next is a real-life example (from [10]) of a subway train control system.

Example 2.1 A subway train speed regulation system is defined as follows. Each train detects beacons that are placed along the track, and receives a "second" signal from a central clock.

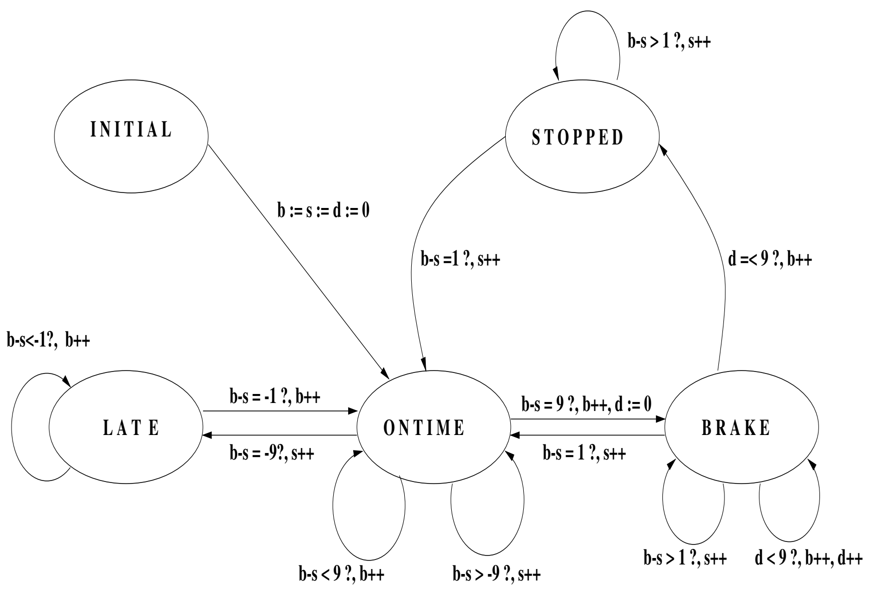

Fig. 1. The Subway Train Control System

Let b and s be counter variables for 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 its 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 1 where $x++$ and $x--$ are abbreviations for $x := x + 1$ and $x := x - 1$, respectively, for any variable x.

2.2 Datalog with Constraints

In this section we review some simple cases of constraint logic programs [12] and constraint query languages [13, 14] that are based on a simplified version of Prolog called Datalog, which is a common query language within database systems [1, 23].

Syntax: Each Datalog program with constraints consists of a finite set of rules of the form

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

where each R_i is a relation name or a constraint, and the xs are either variables or constants. The relation names among $R_0, \ldots, R_n$ are not necessarily distinct. The rule above is read "R_0 is true if R_1 and ... and R_n are all true".

If each R_i is a constraint, then we call R_0 a constraint *fact*. In this paper we will be interested in the following types of constraints:

Order Constraint: Order constraints are constraints of the form $u\theta v$ where u and v are variables or constants over a domain, and θ is one of the operators in $\{=, \leq, \geq, <, >\}$. If the domain is Z or Q we talk of integer order constraints or rational order constraints, respectively.

Gap-order Constraint: Gap-order constraints are constraints of the form $u - v \geq c$ where u, v are variables or constants and c is a non-negative constant over either the domain $\mathbf{Z}$ or $\mathbf{Q}$. Note that each order constraint is also a (conjunction of) gap-order constraints.

Difference Constraint: Difference constraints are constraints of the form $u - v \geq c$ where u, v are variables or constants and c is a constant over either the domain $\mathbf{Z}$ or $\mathbf{Q}$. Note that difference constraints are more general than gap-order constraints.

Linear Inequality Constraint: This constraint is of the form $c_1x_1 + \ldots + c_nx_n \geq b$ where each c_i and b is a constant and each x_i is a variable over some domain. We call b the *bound* of the linear constraint.

Negative Linear Inequality Constraint: We call linear inequality constraints in which each coefficient c_i is negative or zero a *negative linear inequality constraints.*

Positive Linear Inequality Constraint: We call linear inequality constraints in which each coefficient c_i is positive or zero we call them *positive linear inequality constraints.*

Example 2.2 The following Datalog program with gap-order constraints defines the $Travel(x, y, t)$ relation, which is true if it is possible to travel from city x to city y in time t. (Note that one can always travel slower than a maximum possible speed. For example, if the fastest possible travel within two cities is 60 minutes, then the actual time could be anything ≥ 60 minutes.)

$$\begin{array}{ll} Travel(x, y, t) & :\!\!-\; Go(x, 0, y, t). \\ Travel(x, y, t) & :\!\!-\; Travel(x, z, t_1), Go(z, t_1, y, t). \\ & \\ Go(\text{“}Omaha'', t_1, ''Lincoln'', t_2) & :\!\!-\; t_2 - t_1 \geq 60. \\ Go(\text{“}Lincoln'', t_1, ''KansasCity'', t_2) & :\!\!-\; t_2 - t_1 \geq 150. \end{array}$$

Semantics: The proof-based semantics of Datalog programs views the facts as a set of axioms and the rules of the program as a set of inference rules to prove that specific tuples are in some relation. We define this more precisely below.

We call an *instantiation* of a rule, the substitution of each variable in it by constants from the proper domain. (For example, the domain may be the set of character strings, the set of integers, or the set of rational numbers.)

Let Π be a program, $a_1, \ldots, a_k$ constants and R a relation name or a constraint. We say that $R(a_1, \ldots, a_k)$ has a proof using Π, written as $\vdash_\Pi R(a_1, \ldots, a_k)$, if and only if for some rule or fact in Π there is a rule instantiation

$$R(a_1, \ldots, a_k) :\!\!-\; R_1(a_{1,1}, \ldots, a_{1,k_1}), \ldots, R_n(a_{n,1}, \ldots, a_{n,k_n}).$$

where $R_i(a_{i,1}, \ldots, a_{i,k_i})$ is true if R_i is a constraint or $\vdash_\Pi R_i(a_{i,1}, \ldots, a_{i,k_i})$ for each $1 \leq i \leq n$.

The *proof-based* semantics of each Datalog program Π with constraints is a set of relation-name and relation pairs, namely for each relation name R the relation $\{(a_1, \ldots, a_k) \; : \; \vdash_\Pi R(a_1, \ldots, a_k)\}$.

Example 2.3 Let us prove using the query in Example 2.2 that one can travel from Omaha to Kansas City in 180 minutes. We only show the derived tuples without mentioning the instantiations used.

$\vdash_\Pi Go(\text{“}Omaha'', 0, ''Lincoln'', 60)$ using the first fact.
$\vdash_\Pi Go(\text{“}Lincoln'', 60, ''KansasCity'', 210)$ using the second fact.
$\vdash_\Pi Travel(\text{“}Omaha'', ''Lincoln'', 60)$ applying the first rule.
$\vdash_\Pi Travel(\text{“}Omaha'', ''KansasCity'', 210)$ applying the second rule.

Closed-Form Evaluation: If the semantics of Datalog programs with X-type of constraints can be always evaluated and described in a form such that each relation is a finite set of facts with the same X-type of constraints, then we say that the class of Datalog programs with X-type of constraints has a closed-form evaluation.

Theorem 2.1 The least fixed point model of the following types of constraint logic programs can be always evaluated in closed-form in finite time:

(1) Datalog with gap-order constraint programs [27].
(2) Datalog with gap-order and positive linear constraint programs [25].
(3) Datalog with gap-order and negative linear constraint programs [25]. □

In this paper we omit the details about how the evaluation can be done and only give a simple example of a closed-form.

Example 2.4 Since the program in Example 2.2 is a Datalog program with gap-order constraints, by Theorem 2.1 it has a closed-form evaluation. Indeed, one can give as a description of the semantics of the $Travel$ relation the following:

$$\begin{array}{ll} Travel(\text{“}Omaha\text{''},\text{''}Lincoln\text{''}, t) & :\!\!-\; t \geq 60. \\ Travel(\text{“}Lincoln\text{''},\text{''}KansasCity\text{''}, t) & :\!\!-\; t \geq 150. \\ Travel(\text{“}Omaha\text{''},\text{''}KansasCity\text{''}, t) & :\!\!-\; t \geq 210. \end{array}$$

2.3 Approximate Evaluation

The approximation of Datalog programs with difference constraints is studied in [26]. The following is a summary of the main results from [26].

Let us consider a constraint fact with a difference constraint of the form $x - y \geq c$. It may be that the value of c is so small that we may not care too much about it. This leads to the idea of placing a limit l on the allowed smallest bound. To avoid smaller bounds than l, we may do two different modifications.
Modification 1: Change in each constraint fact the value of any bound c to be $\max(c, l)$.
Modification 2: Delete from each constraint fact any constraint with a bound that is less than l.

No matter what evaluation strategy one chooses to derive constraint facts and add it to the database, one can always apply either of the above two modifications to any derived fact. In this way, we obtain modified rule evaluations.

Let $sem(\Pi)$ denote the proof-theoretic semantics of Datalog with difference constraints program Π. Given a fixed constant l, let $sem(\Pi)_l$ and $sem(\Pi)^l$ denote the output of the first and the second modified evaluation algorithms, respectively. We can show the following.

Theorem 2.2 For any Datalog with difference constraint program Π, input database D and constant l, the following is true:

$$sem(\Pi)_l \subseteq sem(\Pi) \subseteq sem(\Pi)^l$$

Further, $sem(\Pi)_l$ and $sem(\Pi)^l$ can be evaluated in finite time. □

We can also get better and better approximations using smaller and smaller values as bounds. In particular,

Theorem 2.3 For any Datalog with difference constraints program Π, input database D and constants l_1 and l_2 such that $l_1 \leq l_2$, the following hold.

$$sem(\Pi)_{l_2} \subseteq sem(\Pi)_{l_1} \text{ and } sem(\Pi)^{l_1} \subseteq sem(\Pi)^{l_2} \quad \Box$$

Example 2.5 Suppose that we want to find a lower approximation of the output of $Travel$ using $l = 100$, that is, when in the input program and after the derivation of any new constraint fact we change each bound c to be the maximum of 100 and c. The evaluation technique in [26] would yield in this case the following.

$$Travel(\text{"}Omaha\text{","}Lincoln\text{"}, t) \;{:}\!\!-\; t \geq 100.$$
$$Travel(\text{"}Lincoln\text{","}KansasCity\text{"}, t) \;{:}\!\!-\; t \geq 150.$$
$$Travel(\text{"}Omaha\text{","}KansasCity\text{"}, t) \;{:}\!\!-\; t \geq 250.$$

Note that the output will be a lower approximation of the semantics of $Travel$ because each possible solution of the returned constraint facts is in the semantics of the original program. It is also easy to see that the lower approximation will not contain for example $Travel(\text{"}Omaha\text{","}KansasCity\text{"}, 210)$, which as we saw in Example 2.3 is in the semantics of the original program.

3 Reformulation and Simplifications of Constraint Automata

For the constraint automaton in Figure 1 a correct design would require that $b - s$ is at least some constant c_1 and at most some constant c_2. The value of $b - s$ may be unbounded in case of an incorrect design. Testing whether $b - s$ is within $[c_1, c_2]$ or is unbounded is an example of a model checking problem. For this problem both [10, 8] give approximate solutions, which may not be correct for some values of c_1 and c_2. We will give a solution that finds all possible values of $b - s$ precisely.

Variable change: The constraint automaton in Figure 1 is more complex than necessary because we are only concerned with the difference of the two variables b and s instead of the exact values of these two. Therefore, the constraint automaton can be simplified for the purpose of our model checking problem. Let's rewrite Figure 1 by using variable x instead of the value $(b-s)-20$ and y instead of d. This change of variables yields the automaton shown in Figure 2.

Now we can make some observations of equivalences between automata. We call these equivalences *reduction rules*. Reduction rules allow us to either rewrite complex constraints into simpler ones (like rules one and two below) or eliminate some transitions from the constraint automaton (like rule three below).

Moving increment after self-loop: This reduction rule is shown in Figure 3. This rule can be applied when no other arcs are ending at state S. This rule says that if there is only one self-loop at S and it can decrement repeatedly a variable while it is greater than c, then the $x++$ before it can be brought after it, if we replace c by $c-1$ in the guard condition of the self-loop. It is easy to see that this is a valid transformation for any initial value of x. We give an example later of the use of this reduction rule.

Elimination of increment/decrement from self-loops: This reduction rule is shown in Figure 4. There are two variations of this rule shown on the top and

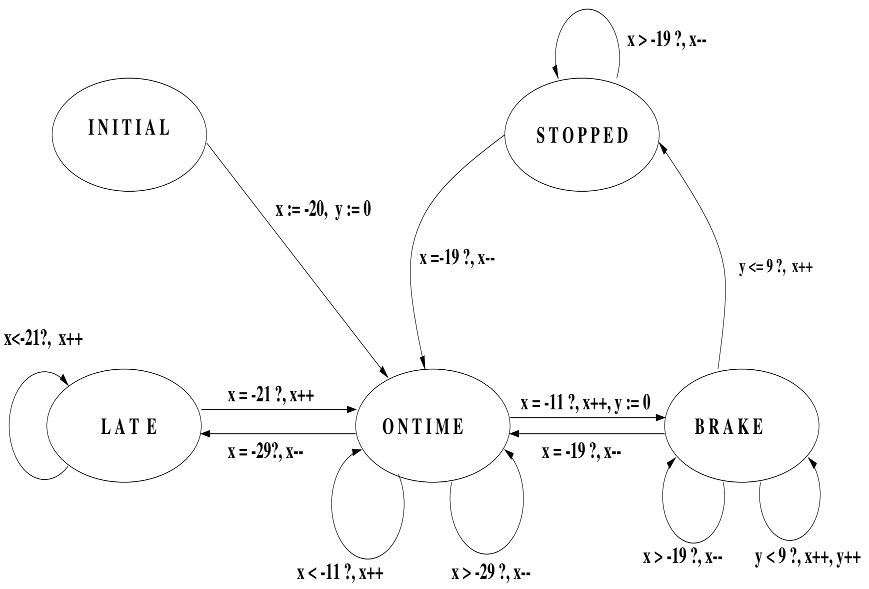

Fig. 2. The Subway System after Changing Variables

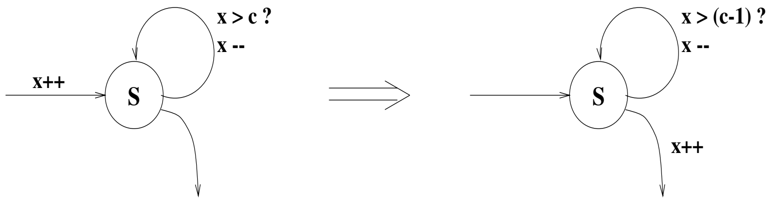

Fig. 3. Rule 1: Moving Increment after Self-Loop

the bottom, depending on whether the variable is incremented or decremented. The top variation says that if a variable is decremented one or more times using a self-loop until a guard condition $x > c$ is satisfied, then the repetition is equivalent to a self-loop which just picks some value x' greater than equal to c and less than the initial value of x and assigns x' to x. The bottom variation is explained similarly. Note that both reduction rules eliminate the need to repeatedly execute the transition. That is, any repetition of the transitions on the left hand side is equivalent to a single execution of the transition on the right hand side.

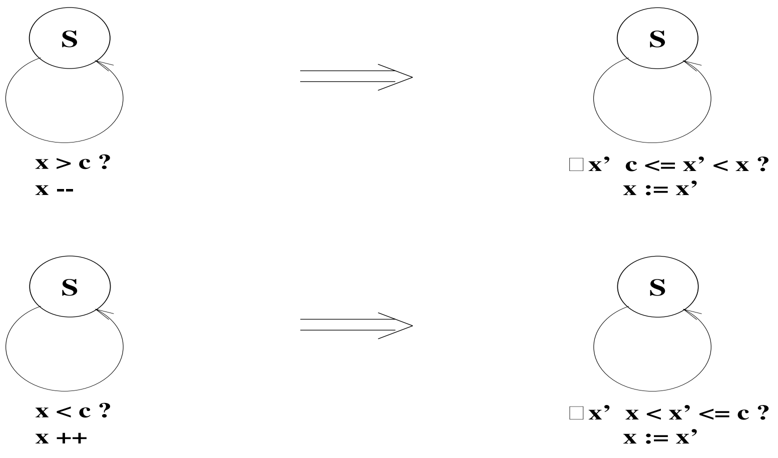

Fig. 4. Rule 2: Elimination of Increment/Decrement from Self-Loops

Elimination of increment/decrement from a pair of self-loops: This reduction rule is shown in Figure 5. This rule can be applied when $c_1 < a$ and $b < c_2$ and no other arcs end at S. Clearly the repetitions of the double increment loop alone, will keep $y - x = (b - a)$ because both y and x are incremented by the same amount. The incrementing applies between $c_2 \geq y \geq b$. However, the double increment loop may be interleaved with one or more single decrement rule that can decrease x down to c_1. The net effect will be that the condition $x' \geq c_1, c_2 \geq y' \geq b, y' - x' \geq (b - a)$ must be true after any sequence of the two self-loop transitions.

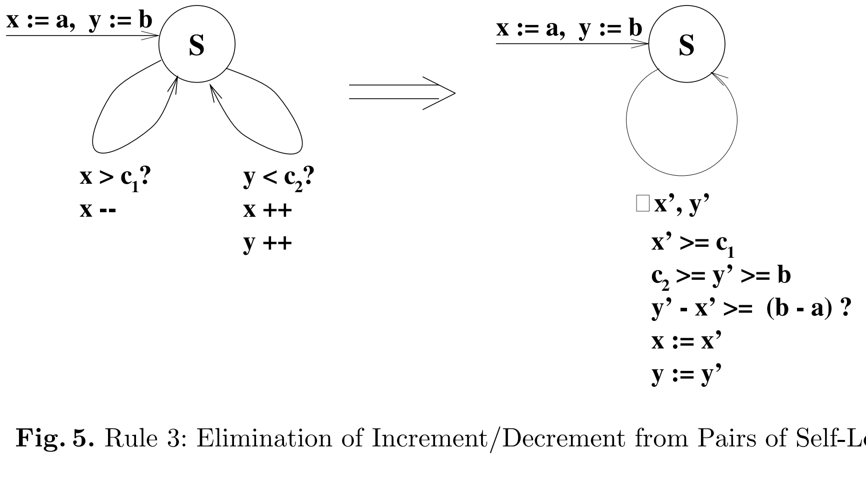

Fig. 5. Rule 3: Elimination of Increment/Decrement from Pairs of Self-Loops

Now let's see how the above reduction rules can be applied to the constraint automaton in Figure 2. Applying the first rule brings $x++$ after the self-loop over state *Stopped*. Then it is trivial to note that "increment x, test whether it is -19 and if yes decrement x" is the same as "test whether $x = -20$". Hence we can further simplify the constraint automaton as shown in the top of Figure 6.

Now after applying the second rule with the self-loops over the states *Late, Ontime* and *Stopped* and the third rule over the state *Brake* we obtain the constraint automaton shown in the bottom of Figure 6.

4 Analysis of Reachable Configurations

Each combination of a state name with values for the state variables is a *configuration.* Often, it is important to know what is the set of configurations that a constraint automaton may move to. This set is called the set of *reachable* configurations.

The set of reachable configurations can be found by translating the constraint automaton into a Datalog program. The Datalog program will use a separate relation for representing each state. Each relation will have the set of state variables as its attributes. Each transition of the constraint automaton will be translated to a Datalog rule. We give a few examples of translations.

Analysis of Example 1: We saw in Section 3 that the constraint automaton of Example 1 can be simplified to the one shown in Figure 6. The set of reachable configurations of the constraint automaton shown in Figure 6 can be expressed in Datalog as follows.

$$\begin{array}{ll}
\mathit{Brake}(-10,0) & :\!\!-\; \mathit{Ontime}(-11,y). \\
\mathit{Brake}(x',y') & :\!\!-\; \mathit{Brake}(x,y),\ x' \geq -19,\ 9 \geq y' \geq 0,\ y'-x' \geq 10. \\
\\
\mathit{Initial}(-20,0). & \\
\\
\mathit{Late}(-30,y) & :\!\!-\; \mathit{Ontime}(-29,y). \\
\mathit{Late}(x',y) & :\!\!-\; \mathit{Late}(x,y),\ x \leq x' \leq -21. \\
\\
\mathit{Ontime}(x,y) & :\!\!-\; \mathit{Initial}(x,y). \\
\mathit{Ontime}(-20,y) & :\!\!-\; \mathit{Late}(-21,y). \\
\mathit{Ontime}(-20,y) & :\!\!-\; \mathit{Brake}(-19,y). \\
\mathit{Ontime}(-20,y) & :\!\!-\; \mathit{Stopped}(-20,y). \\
\mathit{Ontime}(x',y) & :\!\!-\; \mathit{Ontime}(x,y),\ x \leq x' < -11. \\
\mathit{Ontime}(x',y) & :\!\!-\; \mathit{Ontime}(x,y),\ -29 \leq x' < x. \\
\\
\mathit{Stopped}(x',y) & :\!\!-\; \mathit{Stopped}(x,y),\ -20 \leq x' < x. \\
\mathit{Stopped}(x,y) & :\!\!-\; \mathit{Brake}(x,y),\ y \leq 9.
\end{array}$$

This Datalog program contains only gap-order constraints. Therefore by Theorem 2.1 its least fixpoint model can be found in finite time. In fact, we evaluated this Datalog program using the DISCO constraint database system [3], which

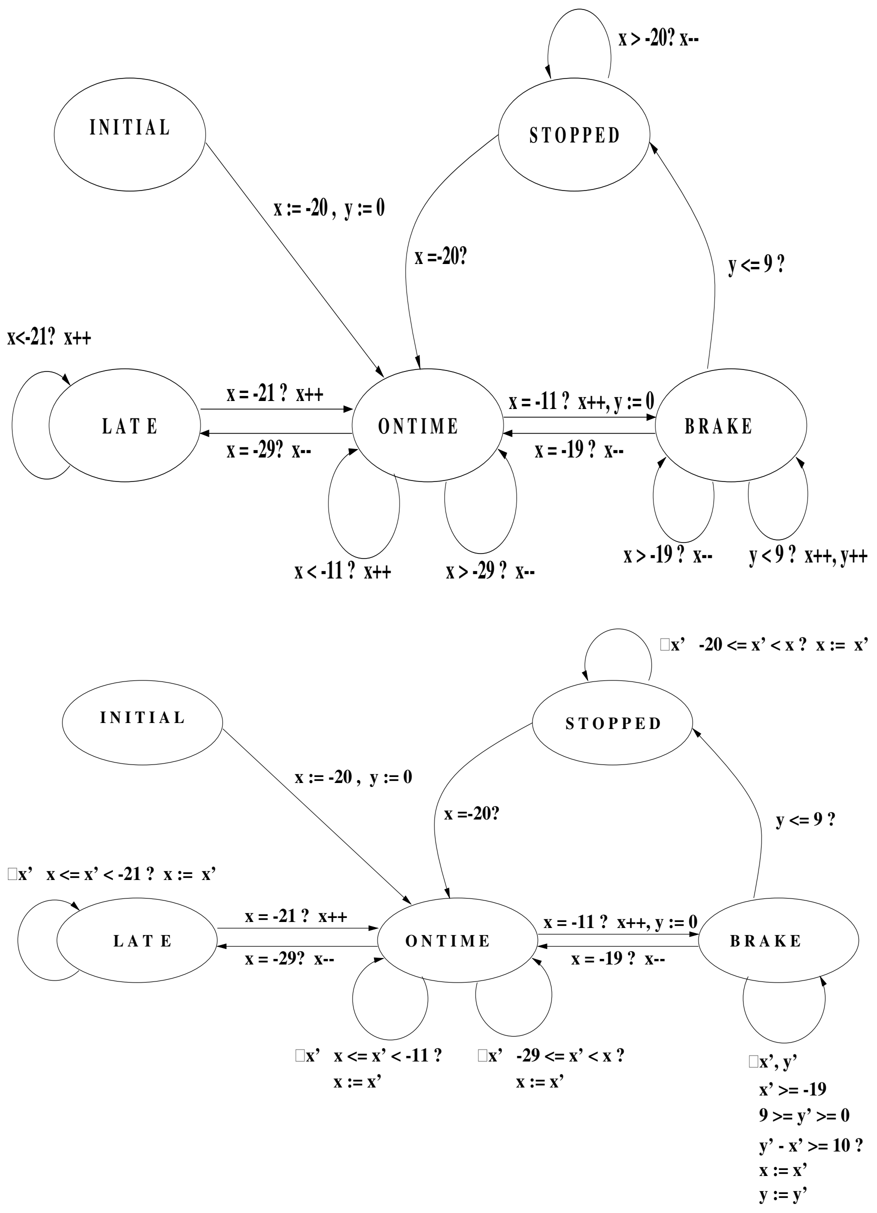

Fig. 6. The Subway System after Rules 1 (above) 1-3 (below)

includes an implementation of integer gap-order constraints, and found that in each tuple x is within -30 and 0. Therefore, $s - d$ is always within -10 and 20.

4.1 The Cafeteria Constraint Automaton

The following is an example of a constraint automaton in which the guards contain relations and negative linear inequality constraints.

Example 4.1 A cafeteria has three queues where choices for salad, main dishes, and drinks can be made. A customer has a coupon for \$10. He first picks a selection. His selection must include a main dish and a salad, but drink may be skipped if the salad costs more than \$3. If the total cost of the selection is less than \$8 then he may go back to make a new choice for salad or drink.

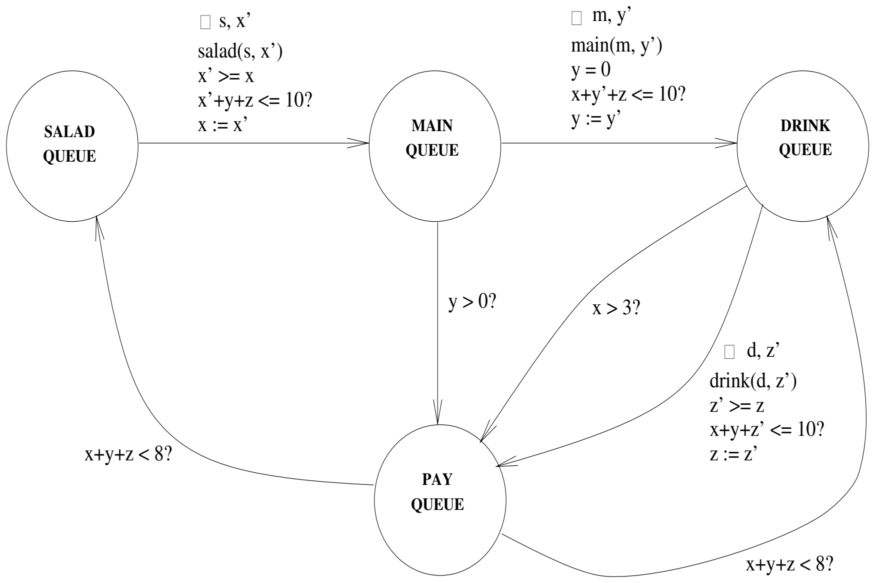

Fig. 7. The Cafeteria Constraint Automaton

Let *salad*, *main* and *drink* be three binary relations in which the first argument is the name of a salad, main dish, or drink, and the second argument is its price. The constraint automaton in Figure 7 expresses this problem.

Analysis of Example 4.1: Assume that in Example 4.1 each main dish costs between five and nine dollars, and each salad and drink costs between two and four dollars. The set of reachable configurations of the constraint automaton shown in Figure 7 can be expressed in Datalog as follows.

$$\begin{array}{lll}
Drink\_Queue(x, y', z) & :\!- & Main\_Queue(x, y, z),\ 5 \leq y' \leq 9, \\
 & & y = 0,\ -x - y' - z \geq -10. \\
Drink\_Queue(x, y, z) & :\!- & Pay\_Queue(x, y, z),\ -x - y - z \geq -7. \\
 & & \\
Main\_Queue(x', y, z) & :\!- & Salad\_Queue(x, y, z),\ 2 \leq x' \leq 4, \\
 & & x' \geq x,\ -x' - y - z \geq -10. \\
 & & \\
Pay\_Queue(x, y, z') & :\!- & Drink\_Queue(x, y, z),\ 2 \leq z' \leq 4, \\
 & & z' \geq z,\ -x - y - z' \geq -10. \\
Pay\_Queue(x, y, z) & :\!- & Drink\_Queue(x, y, z),\ x > 3. \\
 & & \\
Pay\_Queue(x, y, z) & :\!- & Main\_Queue(x, y, z),\ y > 0. \\
 & & \\
Salad\_Queue(x, y, z) & :\!- & Pay\_Queue(x, y, z),\ -x - y - z \geq -7. \\
Salad\_Queue(0, 0, 0). & &
\end{array}$$

This Datalog program contains only gap-order and negative linear inequality constraints. Therefore by Theorem 2.1 its least fixpoint model can be found in finite time. We ran this also in the DISCO constraint database system, which returned the least fixpoint model represented as 20 different constraint facts.

4.2 The Account Balances Constraint Automaton

Let's look at a case of a constraint automaton that can be expressed as a Datalog program whose least fixpoint can be evaluated approximately.

Example 4.2 Three accounts have initially the same balance amounts. Only deposits are made to the first account and only withdrawals are made to the second account, while neither withdrawal nor deposit is made to the third account. Transactions always come in pairs, namely, each time a deposit is made to the first account, a withdrawal is made from the second account. Each deposit is at most \$200, and each withdrawal is at least \$300. What are the possible values of the three accounts?

Let x, y, z denote the amounts on the three accounts, respectively. The constraint automaton in this case is shown in Figure 8.

Here the transition rule says that if at some time the current values of the three accounts are x, y and z, then after a sequence of transactions the new account balances are x', which greater than or equal to x but is less than or equal to $x + 200$ because at most \$200 is deposited, y', which is less than or equal to $y - 300$ because at least \$300 is withdrawn, and z which does not change. The initialization which sets the initial balances on the three accounts to be the same is not shown.

Analysis of Example 4.2: The set of reachable configurations of the constraint automaton shown in Figure 8 can be expressed in Datalog with difference constraints as follows. (In the Datalog program we rewrote some of the constraints to make clear that they are difference constraints.)

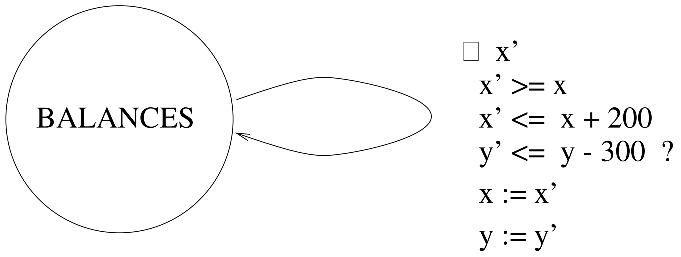

Fig. 8. The Account Balances Constraint Automaton

$$\begin{aligned} &Balance(x,y,z) &&:\!\!- \; x = y, y = z. \\ &Balance(x',y',z) &&:\!\!- \; Balance(x,y,z),\ x' - x \geq 0, \\ & && \quad x - x' \geq -200,\ y - y' \geq 300. \end{aligned}$$

This Datalog program contains only difference constraints. Therefore by Theorem 2.2 its least fixpoint model can be found in finite time approximately. Before discussing the approximation, let's note that in this case the least fixpoint of the Datalog program can be expressed as the relation

$$\{(x,y,z) \ : \ \exists k \ \ x \geq z, z - x \geq -200k, z - y \geq 300k\}$$

This relation is not expressible as a finite set of gap-order constraint facts. However, we can express for each fixed $l < -200$ the relation

$$\{(x,y,z) \ : \ \exists k \ \ x \geq z, z - x \geq max(l, -200k), z - y \geq 300k\}$$

as a finite set of gap-order constraint facts. This would be a lower bound of the semantics of the *Balance* relation. We can also express the relation

$$\{(x,y,z) \ : \ \exists k \ \ \begin{Bmatrix} x \geq z, z - x \geq -200k, z - y \geq 300k & \text{if -200}k \geq l \\ x \geq z, z - y \geq 300k & \text{otherwise} \end{Bmatrix}\}$$

as a finite set of gap-order constraint facts. This would be an upper bound of the semantics of the *Balance* relation.

The approximation could be used for example to decide some reachability problems. For example, consider the question: Is it possible that the account balances are at any time $x = 1500, y = 200$ and $z = 1000$? When we use an approximate evaluation with $l = -1000$, we see that is it not in the upper bound of the semantics of the *Balance* relation. Hence it cannot be possible.

5 Related Work

Most of the model checkers operate only on bounded models, not unbounded ones like we did in this paper. For example, the representation of binary decision diagrams or BDDs [19] captures a finite set of states of boolean variables and cannot be used to represent the set of states of our constraint automata. The **HyTech** system developed by Henzinger et al. [11] is a model checker for hybrid automata that model both discrete and continuous change and represent infinite states. Although this system has been useful for some applications, the system cannot be proven to always yield an answer, that is, it may not terminate unlike our system.

An interesting problem that occurs in debugging is to automate the generation of abstract models from software programs. Lowry and Subramaniam [17] extend program slicing techniques to abstract state-based programs for the purpose of model checking. Program slicing is a software engineering technique that extracts a partial program equivalent to an original program over a subset of the program variables [5]. Program slicing algorithms work backwards from the program end-point by keeping all statements that effect in any way the designated variables and removing all other statements. Lowry and Subramanian [17] propose semantic slicing which is done with respect to state predicates instead of state variables. That is, programs are sliced with respect to state predicates starting from a statement that contains the operations which are required to only be executed in particular states. [17] and [9] also propose performing data abstractions using weakest preconditions to compute an abstract model. The property of the abstract model is that whenever an invariant property is true in the abstract model then it is true in the concrete model, but not vice versa. In other words, the abstract model could lead to false negatives except in special cases when the property is expressible in restricted temporal logics like CTL. Therefore in general an abstract model counterexample to the property verified has to be still checked on the concrete model. Note that all our reformulations of constraint automata preserve all properties being verified, because we are interested in the computing in a finite representation the exact set models for each state.

Another interesting use of model checking occurs in planning. Cimatti at al. [4] observe that a planning search problem can be articulated as the problem of achieving a goal state starting from some initial state through a sequence of operator applications, where each operator is applicable on certain preconditions and specifies a change in the environment (variables), and this search can be easily expressed as a model checking problem. However, the difficulty is again lies in the abstractions that many model checkers use. The abstract plan generated using model checking may not be refinable to a correct ground solution, i.e., executable by a concrete sequence of operator applications.

6 Conclusion and Further Work

As our examples illustrate, constraint automata can be often expressed in Datalog programs that contain only specific types of constraints (gap-order, positive or negative linear inequality, difference). These Datalog programs that define the set of reachable configurations of the constraint automata, can be always evaluate or approximately evaluated with any desired precision. That leads to solutions to model checking problems like reachability, containment and equivalence. One can also test any of a number of more complex conditions on the model as we have done for the subway train control system.

It remains as an interesting further work to find other classes of constraints for which at least an approximate closed-form evaluation can be guaranteed. We have given rewrite rules for the case of gap-order and increment/decrement constraints. It is also an interesting task to find rewrite rules in the case of other types of constraints.

It is also an important challenge to see whether constraint automata could be applied to some of the software debugging problems [9, 17], especially for concurrent software algorithms, and for other applications like planning [4].

References

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

[2] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *Proc. Conf. on Computer-Aided Verification*, pages 55–67, 1994.

[3] J.-H. Byon and P.Z. Revesz. DISCO: A constraint database system with sets. In *Proc. Workshop on Constraint Databases and Applications*, number 1034 in LNCS, pages 68–83. Springer-Verlag, September 1995.

[4] A. Cimatti, F. Giunchiglia, and M. Roveri. Abstraction in planning via model checking. In *Proc. Symposium on Abstraction, Reformulation and Approximation*, pages 37–41, 1998.

[5] J. J. Comuzzi and J. M. Hart. Program slicing using weakest precondition. In *Proc. Industrial Benefit and Advances in Formal Methods*, number 1051 in LNCS. Springer-Verlag, 1996.

[6] G. Delzanno and A. Podelski. Model checking in clp. In *Second International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer LNCS, 1999.

[7] L. Fribourg and H. Olsén. A decompositional approach for computing least fixed-points of datalog programs with **z**-counters. *Constraints*, 3–4:305–336, 1997.

[8] L. Fribourg and J.D.C. Richardson. Symbolic verification with gap-order constraints. In *Prof. LOPSTR*, 1996.

[9] S. Graf and H. Saidi. Constructing abstract graphs using pvs. In *Proc. Computer Aided Verification*, number 1102 in LNCS. Springer-Verlag, 1996.

[10] N. Halbwachs. Delay analysis in synchronous programs. In *Proc. Conf. on Computer-Aided Verification*, pages 333–346, 1993.

[11] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. Hytech: A model checker for hybrid systems. In *Proc. Computer Aided Verification*, number 1254 in LNCS, pages 460–463. Springer-Verlag, 1997.

[12] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proc. 14th ACM POPL*, pages 111–119, 1987.

[13] P.C. Kanellakis, G.M. Kuper, and P.Z. Revesz. Constraint query languages. In *Proc. of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 299–313, New York, 1990. ACM Press.

[14] P.C. Kanellakis, G.M. Kuper, and P.Z. Revesz. Constraint query languages. *Journal of Computer and System Sciences*, 51:26–52, 1995.

[15] A. Kerbrat. Reachable state space analysis of lotos specifications. In *Proc. 7th International Conference on Formal Description Techniques*, pages 161–176, 1994.

[16] R. Kosaraju. Decidability of reachability in vector addition systems. In *Proc. of the 14th Annual ACM Symposium on Theory of Computing*, pages 267–280, 1982.

[17] M. Lowry and M. Subramaniam. Abstraction for analytic verification of concurrent software systems. In *Proc. Symposium on Abstraction, Reformulation and Approximation*, pages 85–94, 1998.

[18] E. Mayr. An algorithm for the general petri net reachability problem. In *Proc. of the 13th Annual ACM Symposium on Theory of Computing*, pages 238–246, 1981.

[19] K. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[20] M. L. Minsky. Recursive unsolvability of post's problem of 'tag' and other topics in the theory of turing machines. *Annals of Mathematics*, 74(3):437–455, 1961.

[21] M. L. Minsky. *Computation: Finite and Infinite Machines*. Prentice Hall, 1967.

[22] J. Peterson. *Petri Net Theory and Modeling of Systems*. Prentice-Hall,Inc., 1981.

[23] R. Ramakrishnan. *Database Management Systems*. McGraw-Hill, 1998.

[24] W. Reisig. *Petri Nets: an Introduction*. Springer, 1985.

[25] P. Z. Revesz. Safe datalog queries with linear constraints. In M. Maher and J.-F. Puget, editors, *Proc. Fourth International Conference on Principles and Practice of Constraint Programming*, number 1520 in LNCS. Springer-Verlag, 1998.

[26] P. Z. Revesz. Datalog programs with difference constraints. In *Proc. Twelfth International Conference on Applications of Prolog*, pages 69–76, September 1999.

[27] P.Z. Revesz. A closed-form evaluation for Datalog queries with integer (gap)-order constraints. *Theoretical Computer Science*, 116:117–149, 1993.