CDB-PV: A Constraint Database-Based Program Verifier*

Scot Anderson[1] and Peter Revesz[2]

[1] Southern Adventist University, Collegedale, TN 37315, USA
scot@southern.edu
[2] University of Nebraska-Lincoln, NE 68588, USA
revesz@cse.unl.edu

Abstract. In this paper we present a new system called CDB-PV that uses constraint databases (CDBs) for program verification (PV). The CDB-PV system was implemented in C++ and tested on several sample programs that are difficult to verify using other methods. The CDB-PV system also runs efficiently for the sample programs. The CDB-PV approach is similar to abstract interpretation but it allows non-convex approximations.

1 Introduction

Programs increasingly control many aspects of our daily lives such as air traffic control (ATC) systems. Failures such as the computer controlled Airbus A320 crash on June 26, 1988 [1] show that programs need thorough debugging and verification before risking lives. We propose a new constraint database approach to program debugging and verification.

Verifying the correctness of programs is undecidable in general. That is easy to see by looking at the well-known halting problem, which is the problem of deciding whether a given program with a given input will terminate. Since the halting problem is undecidable in general and termination of programs is usually considered one of the conditions of correctness, it is clear that program verification is also undecidable in general.

However, let us take a deeper look at program verification and identify what can be done. We start with the following definitions.

Definition 1 (Program State). *A program state is a pair consisting of the values assigned to the program variables and the specific location of the program code where such an assignment occurs during an execution of the program.*

We call the meaning of the program the *semantics* of the program. In this paper we are concerned with the following semantics, called the collecting semantics[3].

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

[3] see Cousot lecture notes: http://www.cs.wisc.edu/~cs704-1/LectureNotes/9.AbstractInterpretation.txt

Definition 2. *Define the collecting semantics as a set of all possible program states that may occur for some execution and some input.*

A key idea in program verification is that the collecting semantics can be approximated using a terminating program that takes as input the program and some approximation parameters and gives either an under-approximation or an over-approximation, which we define as follows.

Definition 3 (Over-Approximation). *Let S be the semantics of a program. We say that any P^l where $S \subseteq P^l$ is an* over-approximation*.*

Definition 4 (Under-Approximation). *Let S be the semantics of a program. We say that any P_l where $P_l \subseteq S$ is an* under-approximation*.*

The approximation is often useful to check certain concerns about the program. These concerns are expressed as some conditions called *error states* that need to be avoided by the program to be considered correct.

If the over-approximation does not contain the error states, then the program is considered correct. However, error states contained in an over-approximation may be spurious. Hence they do not prove the program incorrect.

The spurious error states may be avoided by tightening the over-approximation. If repeated tightening fails to eliminate the error states, then we may suspect that the program is incorrect. By using an under-approximation, we can often prove that the program is incorrect, i.e., falsify it. If an *under-approximation* of the semantics contains some error state, then the program is incorrect. Falsification identifies some of the errors in the program, hence it is a useful aid in debugging the program.

Our program verification approach falls into the category of *Abstract Interpretation*. The *abstract interpretation* technique provides the framework for extracting abstract collecting-semantics of a program [2,3,4,5,6,7,8]. The abstraction usually over-approximates the values of the program variables in a convex state space that models all possible program states and makes verification of correctness possible. Many different abstractions and combinations of abstractions evolved over the years and hence no succinct definition of abstract interpretation exists[4]. Rather, abstract interpretation gathers information about programs in order to provide sound answers to questions about their run-time behaviors. These semantics can then be used to design automatic program analyzers [5]. Abstract interpretation is often understood in terms of *abstract-evaluation* using an *abstract interpreter* on an *abstraction* of the program.

Definition 5 (Semantic operator). *Let P be a program written in a language L. Define the semantic operator S as a mapping from L to the semantic domain of L denoted D.*

$$S : L \rightarrow D \tag{1}$$

We usually write the semantics of P as $S[P] \in D$.

Definition 2 gives an example representation for the semantic domain.

[4] see `www.di.ens.fr/~cousot/AI/` for Cousot's overview of Abstract Interpretation

Definition 6 (Abstraction operator). *The abstraction of a program P written in a language L maps the semantics of P in D to an abstract domain $D^\sharp$*

$$\alpha : D \rightarrow D^\sharp \tag{2}$$

The purpose of the abstract domain it to provide a decidable domain of that can be used to evaluate the program.

Definition 7 (Abstract Evaluation). *Given a program P and an abstraction operator α, $\alpha[S[P]]$ is an abstract evaluation if it halts with an over-approximation of P.*

In the constraint database approach we perform abstract evaluation by creating a Datalog query for a constraint database and execute the query using constraint database approximation techniques.

The constraint database approach to program verification is a widely applicable method similar to abstract interpretation. However, while abstract interpretation methods usually rely on widening operators that yield convex approximations, the constraint database approach can yield non-convex approximations. This extra precision still allows an efficient calculation of approximate program semantics, which are crucial to the problem of program verification.

The remainder of the paper is organized as follows: Section 2 gives a review of constraint database approximation techniques. Section 3 describes the constraint database approach to verifying programs. Section 4 gives experimental results for three sample programs and uses them as a comparison to other techniques. Section 5 discusses the running time of the method and gives the running time for verifying each of the sample programs. Finally, Section 6 gives conclusions and future work.

2 Review of Constraint Database Approximation

The *constraint logic programming* languages proposed by Jaffar and Lassez [9], whose work led to CLP(R) [10], by Colmerauer [11] within Prolog III, and by Dincbas et al. [12] within CHIP, were Turing-complete. Kanellakis, Kuper, and Revesz [13] 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 always be evaluated in a finite constraint database representation [14].[5]

Definition 8 (Addition Constraints). *Addition constraints [15] have the form:*

$$\pm\, x \pm y\ \theta\ b \text{ or } \pm\, x\ \theta\ b \tag{3}$$

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

[5] 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.

By allowing addition constraints, it is easy to express a Datalog program that will not terminate using a standard bottom-up evaluation [15]. Consider the following Datalog program:

$$\begin{array}{lll} D(x,y,z) & :\!- & x-y\leq 0,\ \ -x+y\leq 0,\ \ z\leq 0,\ \ -z\leq 0. \\ D(x,y,z) & :\!- & D(x',y,z'),\ \ x-x'\leq 1,\ \ -x+x'\leq -1, \\ & & z-z'\leq 1,\ \ -z+z'\leq -1. \end{array} \tag{4}$$

This expresses that the *Difference* of x and y is z. Further, based on Equation (4) we can also express a *Multiplication* relation as follows:

$$\begin{array}{lll} M(x,y,z) & :\!- & x\leq 0,\ \ -x\leq 0,\ \ y\leq 0,\ \ -y\leq 0,\ \ z\leq 0. \\ M(x,y,z) & :\!- & M(x',y,z'),\ \ D(z,z',y),\ \ x-x'\leq 1,\ \ -x+x'\leq -1 \\ M(x,y,z) & :\!- & M(x,y',z'),\ \ D(z,z',x),\ \ y-y'\leq 1,\ \ -y+y'\leq -1 \end{array} \tag{5}$$

Using Equations (4) and (5) we can express Diophantine equations which by [16] is Turing complete. Hence, in the limit as $l \to -\infty$, this method is Turing complete and can express any program. In this paper we limit ourselves to verifying programs with integer variables.

The D and M recursive programs must be evaluated until no new facts are discovered. This leads to the well known least fixed point definition.

Theorem 1 (Tarski's fixed point Theorem [17]). *Let $(L, \subseteq)$ be any complete lattice. Suppose $F : L \to L$ is monotone increasing. Then the set of all fixed points of f is a complete lattice with respect to $\subseteq$.*

Let F be the set of facts discovered after evaluation of the Datalog rule given in Equation (4). Repeated application of $D^n(F)$ will not reach a point where it has discovered all the facts. If $D^n(F) = D^{n+1}(F)$, D will have reached a least fixed point.

Definition 9 (Least Fixed Point). *The least fixed point of a function f is a fixpoint v such that v is smaller than or equal to every other fixpoint of f.*

However the programs from (4) and (5) will never reach a fixed point. For example applying the recursive rule D gives the following facts where the bound on the right continues to increase to infinity.

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

For Datalog, we always have a least fixed point [18], but when we add addition constraints there must be an approximation method that terminates the evaluation to find an approximation.

For constraint databases, Revesz [19,15] introduced two methods for approximating the least fixpoint evaluation of addition constraints by modifying the standard bottom-up evaluation.

Definition 10 (Lower-Bound Modification). *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 11 (Upper-Bound Modification). *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.*

Example 1 (Lower/Upper-Bound Modification). Consider the difference relation D and suppose that we set an approximation bound at $b = -2$. The lower bound approximation changes in the constraint tuple the value of any bound b to be $\max(b, u)$. This value will not cause the evaluation given in Equation (6) to change until the bound a bound is equal to 3. Perform the evaluation as follows:

$$\begin{aligned} D(x, y, z) \ :\!-\ & x' - y \leq 2, \ \ -x' + y \leq -2, \ \ -z' \leq 2, \ \ -z' \leq -2 \\ & x - x' \leq 1, \ \ -x + x' \leq -1, \ \ z - z' \leq 1, \ \ -z + z' \leq -1. \end{aligned} \tag{6}$$

Simplifying results in bounds that cause a problem.

$$D(x, y, z) \ :\!-\ x - y \leq 3, \ \ -x + y \leq -3, \ \ z \leq 3, \ \ -z \leq -3 \tag{7}$$

Applying the lower bound rule, change the any bound greater than 3 to be 2. This results in the following:

$$D(x, y, z) : - x - y \leq 2, -x + y \leq -3, z \leq 2, -z \leq -3 \tag{8}$$

The last step in the requires checking the satisfiability of the modified clause. Combining the first two and last two constraints results in:

$$D(x, y, z) \ :\!-\ 0 \leq -1 \tag{9}$$

Since Equation (9) is not satisfiable, the evaluation does not add a clause to the relation, and the evaluation of the recursive clause halts.

Applying the upper bound rule, to Equation (7), requires that we delete the second and fourth constraints which gives:

$$D(x, y, z) \ :\!-\ x - y \leq 3, \ \ z \leq 3 \tag{10}$$

This fact is added to the relation and we use it in the recursive call:

$$\begin{aligned} D(x, y, z) \ :\!-\ & x' - y \leq 3, \ \ z' \leq 3 \\ & x - x' \leq 1, -x + x' \leq -1, \ \ z - z' \leq 1, \ \ -z + z' \leq -1. \end{aligned}$$

The result of this is

$$D(x, y, z) \ :\!-\ x - y \leq 4, \ \ z \leq 4 \tag{11}$$

This constraint is not added to the relation because it is subsumed by (10). Trying all the other facts in the recursive clause also results in a subsumed fact and hence the evaluation halts.

These modifications lead to the following approximation theorem.

Theorem 2 (Revesz [15]). *For any Datalog program P and constant $l < 0$ the following is true:*

$$P_l \subseteq \mathit{lfp}(P) \subseteq P^l \tag{12}$$

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

Hence l can be considered a parameter that controls how tight the over/under-approximation will be.

We implemented these two constraint modifications in CDB-PV to allow the over-approximation and under-approximation of the semantics of Datalog with addition constraints for program verification.

3 The Constraint Database Approach to Verification

CDB-PV is built on the MLPQ [20] constraint database system which provides a high degree of precision by allowing non-convex and disjoint regions to represent collecting semantics. We control the level of approximation by a single parameter l which corresponds to the parameters used in Definitions 10 and 11. Figure 1 provides an overview of the constraint database approach to the verification of programs [21].

The first two steps form the framework that translates a program into Datalog. The next step calculates an over-approximation given the bounding parameter l. The results from the over-approximation (or under-approximation) often contain a large set of data due to the disjoint representation of variable values. The constraint database approach simplifies interpretation of results by providing native facilities to query the results for error states using Datalog or SQL. Not finding the error state in the over-approximation verifies program correctness. If we suspect that the error state is present, we perform the under-approximation. Finding the error state in the under-approximation falsifies the program.

In theory the constraint database approach can reach arbitrary precision by calculating the under-approximation and over-approximation repeatedly as $l \to -\infty$. Hence this method approaches a precise evaluation. While this may not be possible for every constraint in an invariant, it may be reasonable to lower l to a point where the constraint in the over-approximation and under-approximation that identifies an error state converges. This method provides a way to find constraints in invariants that converge in parallel with constraints that do not even though a precise evaluation is not possible in general.

The framework for translating a program to Datalog adapts the standard pre-condition transition system used in static analysis and compiler optimization techniques.

Definition 12 (Transition System). *A transition system is a tuple $(S, \wedge, \to)$ where S is a set of states, $\wedge$ is a set of labels and $\to \subseteq S \times \wedge \times S$ is a ternary relation of labeled transitions. If $p, q \in S$ and $\beta \in \wedge$, then $(p, \beta, q) \in \to$ is written as:*

$$p \xrightarrow{\beta} q$$

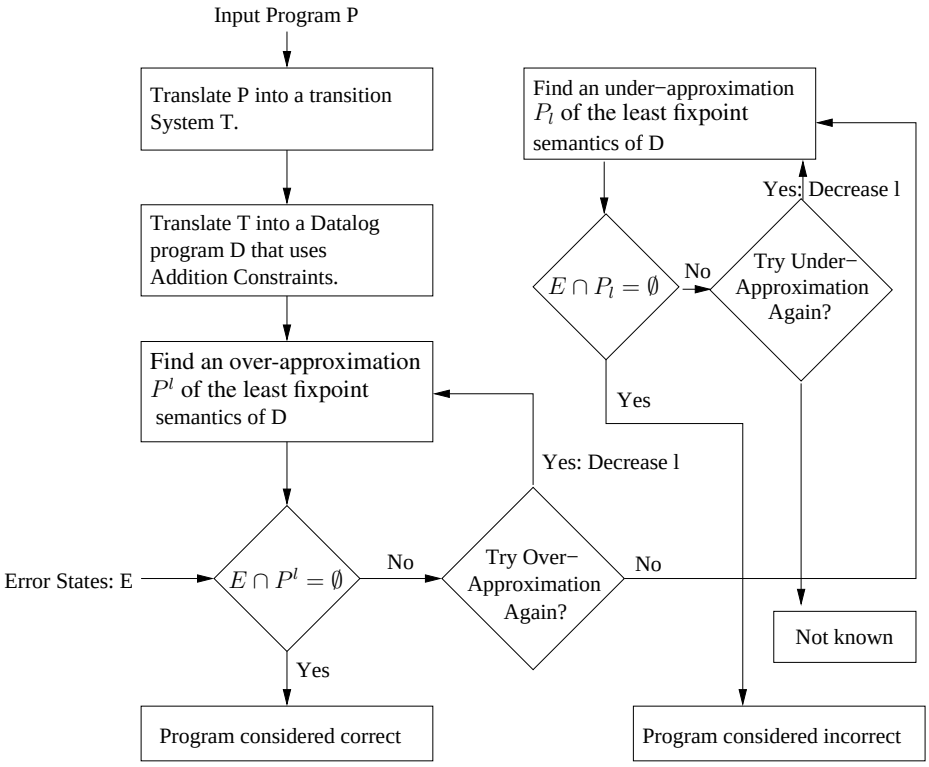

Fig. 1. Constraint Database Approach.

where β is a set of conditions and operations to the source state variables that must be made to enter the target state.

The abstract domain we use consists of addition constraints. The framework translates a program into Datalog in the first two steps shown in Figure 1.

Given a program P with n lines of code and m variables, step (1) gives the transition system where a program statement (or state) $p_i \in S$ denotes the program statement on line i about to be executed. A transition from some state p_j to p_i denoted $p_j \xrightarrow{\beta} p_i$ represents the rule to enter state p_i where β contains the "execution" of the program statement on line j. The values changed by β affect the values available for execution in p_i. *How the new values affect the values of variables in state p_i will determine the type of approximation*. In Datalog these values will be added to the set of existing values. In abstract interpretation the new values cause widening of the existing invariants.

For example, Miné [22] defines an abstract interpretation widening technique as follows:

Definition 13 (Widening Operator of [22]). *Let M and N be two ABMs. Then the* widening *of M by N, written as $M \nabla N$ is defined as:*

$$[M \vee N][i,j] = \left\{ \begin{array}{ll} M[i,j] & \text{if } M[i,j] \leq N[i,j] \\ -\infty & \text{if } N[i,j] \leq M[i,j] \end{array} \right\}$$

Example 2. The simple program shown in Table 1 is translated into Datalog where transition system is given in Figure 2. The constraint relations `L2` - `L7` shown at the right represent the variable values prior to the execution of the corresponding lines at the left. `L1` remains undefined because no variables have been assigned prior to the execution of Line 1.

1. a ← 0 2. a ← a + 1 3. **if** a > 2 **then goto** 6 4. **if** a = 2 **then goto** 7 5. **goto** 2 6. ... 7. ...	→	begin%RECURSIVE% L2(a) :- a=0. L2(a) :- L5(a). L3(a) :- L2(a1), a-a1=1. L4(a) :- L3(a), a≤2. L5(a) :- L4(a), a<2. L5(a) :- L4(a), a>2. L6(a) :- L3(a), a>2. L7(a) :- L4(a), a=2. end%RECURSIVE%

Table 1. Simple Goto Program.

4 Experiments and Results

We tested the constraint database approach on three different examples. The first example compares our constraint database approach the widening technique of Miné using the simple code example from Example 2. The second experiment gives a tight bound for the automaton from [6]. The final example demonstrates the verification of a search algorithm involving Euclidean distance calculations. We give an examination of running time for the method and each individual sample program in Section 5.

The left side of Table 2 shows invariants found by two abstract evaluation passes using the Miné widening technique given in Example 2. In the second entry, an invariant of $a \geq 1$ entering line (3) indicates that line (6) is executed.

We recursively evaluate the Datalog program in MLPQ with $l = -2$ and either over-approximation or under-approximation to obtain the result on the right in Table 2. The resulting invariants for line 3 never indicate that $a > 2$ and hence we find that the

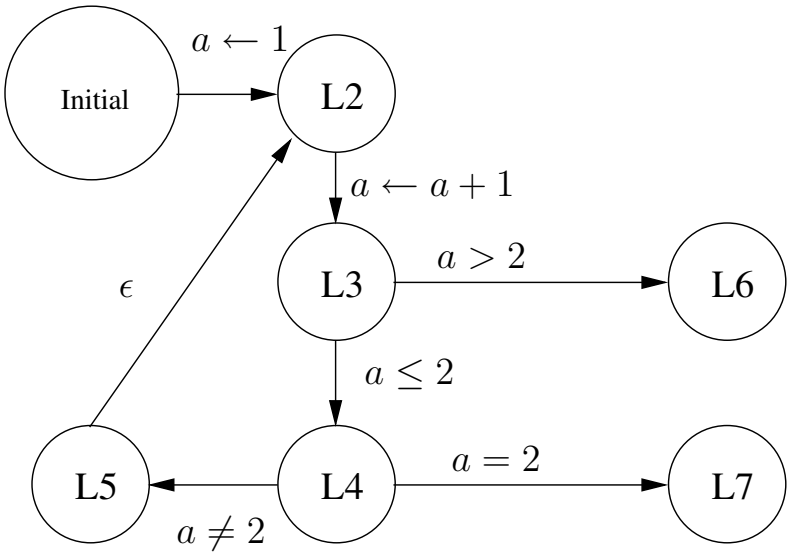

Fig. 2. Transition System for the Simple Program.

Miné Widening Results		
Line	1st Entry	2nd Entry
2	$0 \leq a \leq 0$	$0 \leq a$
3	$1 \leq a \leq 1$	$1 \leq a$ if condition $a > 2$ true goto 6
4	$1 \leq a \leq 1$	
5	$1 \leq a \leq 1$	

MLPQ output
L2(a) :— a=0.
L2(a) :— a=1.
L3(a) :— a=1.
L3(a) :— a=2.
L4(a) :— a=1.
L4(a) :— a=2.
L5(a) :— a=1.
L7(a) :— a=2.

Table 2. Invariants Obtained by Miné Widening.

`L6(a)` relation is missing. Suppose that line (6) identifies an error, then our program technique identifies the unentered error state correctly where the abstract interpretation method of Miné does not.

Consider the subway train speed regulation system in Figure 3 described by [6]. Each train detects "beacons" that are marks placed along the track and receives a "second" signal from a central clock.

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 decelerating by applying its brake. The goal of the speed regulation system is to keep $\mid b - s \mid \leq 20$ while the train is running.

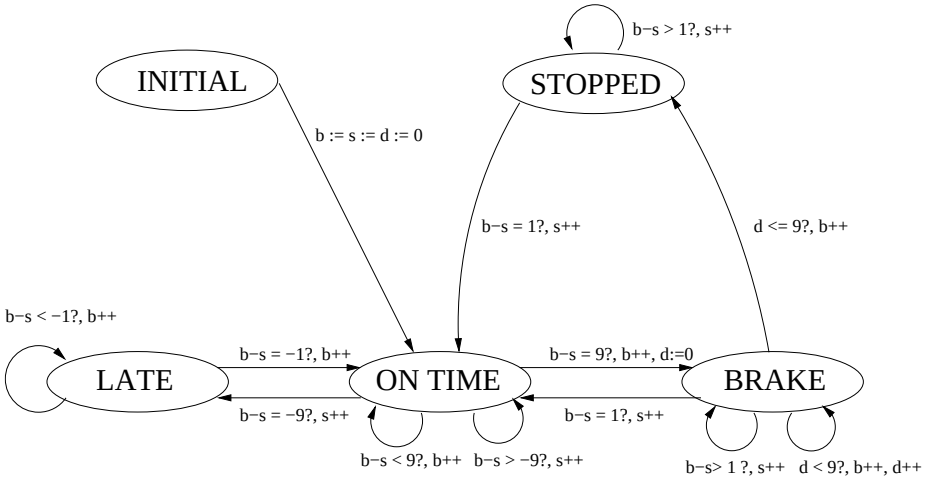

Fig. 3. Subway Automaton.

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 counter automaton enforces the conditions described above using guard constraints followed by question marks, and $x++$ and $x--$ as abbreviations for the assignments $x := x + 1$ and $x := x - 1$, respectively.

The subway counter automaton from Figure 3 can be translated into the Datalog program shown in Table 3. It expresses the semantics (combinations of states and state variable values) of the automaton using difference constraints.

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 under-approximation using the MLPQ constraint database system.

MLPQ Under Approximation

BRAKE	LATE	ONTIME	STOPPED
$1 \leq b - s \leq 19$	$-10 \leq b - s \leq -1$	$-9 \leq b - s \leq 9$	$1 \leq b - s \leq 20$
$10 \leq b \leq 19$	$10 \leq s \leq 19$	$0 \leq b \leq 9$	$11 \leq b \leq 20$
$0 \leq s \leq 18$	$0 \leq d \leq 9$	$0 \leq s \leq 18$	$0 \leq s \leq 9$
$0 \leq d \leq 9$		$0 \leq d \leq 9$	$0 \leq d \leq 9$

```
//Subway Automaton
begin%RECURSIVE%

ONTIME(b,s,d) :- b=0, s=0, d=0.
ONTIME(b,s,d) :- STOPPED(b,s1,d), b-s1=1, s-s1=1.
ONTIME(b,s,d) :- ONTIME(b1,s,d), b1-s<9, b-b1=1.
ONTIME(b,s,d) :- ONTIME(b,s1,d), b-s1>-9, s-s1=1.
ONTIME(b,s,d) :- ONBRAKE(b,s1,d), b-s1=1, s-s1=1.
ONTIME(b,s,d) :- LATE(b1,s,d), b1-s=-1, b-b1=1.
ONBRAKE(b,s,d) :- ONTIME(b1,s,d1), b1-s=9, b-b1=1, d=0.
ONBRAKE(b,s,d) :- ONBRAKE(b1,s,d1), d1<9, b-b1=1, d-d1=1.
ONBRAKE(b,s,d) :- ONBRAKE(b,s1,d), b-s1>1, s-s1=1.
STOPPED(b,s,d) :- ONBRAKE(b1,s,d), d≤9, b-b1=1.
STOPPED(b,s,d) :- STOPPED(b,s1,d), b-s1>1, s-s1=1.
LATE(b,s,d) :- ONTIME(b,s1,d), b-s1=-9, s-s1=1.
LATE(b,s,d) :- LATE(b1,s,d), b1-s<-1, b-b1=1.

end%RECURSIVE%
```

Table 3. Subway Datalog Program.

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

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

The Verimag laboratory has software for testing program correctness using abstract interpretation. [6] gave the following over approximation derived using Verimag's software for the subway automaton.

Verimag Over Approximation

BRAKE	LATE	ONTIME	STOPPED
$1 \leq b - s \leq d + 10$	$-10 \leq b - s \leq -1$	$-9 \leq b - s \leq 9$	$1 \leq b - s \leq 19$
$d + 10 \leq b$	$s \geq 10$	$b \geq 0$	$19 \leq 9s + b$
$0 \leq d \leq 9$		$s \geq 0$	$b \geq 10$

We showed in [23] that the Verimag system produced incorrect results. However a over-approximation still needs to be found to verify the automaton. We made several runs with different l values ranging from -10 to -30 for both over and under approximations. By increasing the l value to -20 alone and performing the evaluation with both approximations, we derive a tight bound $-10 \leq b - s \leq 20$ across the four constraint relations.

Suppose a yacht is traveling through the ocean between two ports. The yacht does not have enough supplies to make the trip, hence it must resupply at several possible locations. The program shown in Table 4 determines if a point $(22, 19)$ can be reached from a starting position of $(0, 0)$. It includes the `Depot` relation containing possible resupply locations. The `Leg` and `Reach` rules calculate Euclidian distance using the `D` (difference) and `M` (multiplication) relations. The approximation value l limits the evaluation of `D` and `M` which may be pre-computed to save time. The reach relation determines if the destination can be reached. This example can be extended by adding a relation for multiple destinations. In that case knowing error destinations would allow us to query the reach relation for incorrect values.

The results of running this program in MLPQ verify that the `reach` relation contains the values $x = 22$ and $y = 19$. This verifies the Resupply Depot program as correct.

```
begin%SupplyDepotOptimized%

Depot(id,x,y) :- id=1, x=0, y=19.
Depot(id,x,y) :- id=2, x=6, y=8.
Depot(id,x,y) :- id=3, x=15, y=12.
Depot(id,x,y) :- id=4, x=25, y=5.
D(x,y,z)      :- x-y=0, z=0.
D(x,y,z)      :- D(x1,y,z1), x-x1=1, z-z1=1.
D(x,y,z)      :- D(x1,y,z1), x-x1=-1, z0z1=-1
M(x,y,z)      :- x=0, y=0, z=0.
M(x,y,z)      :- M(x1,y,z1), D(z,z1,y), x-x1≥1, x1-x≥-1.
M(x,y,z)      :- M(x,y1,z1), D(z,z1,x), y-y1≥1, y1-y≥-1.
Leg(x,y)      :- x=0, y=0.
Leg(x,y)      :- Leg(x1,y1), Depot(id,x,y), AD(x,x1,dx),
                 AD(y,y1,dy), M(dx,dx,dx2), M(dy,dy,dy2),
                 dx2+dy2≤100 ,dx≤10, dy≤10.
Reach(x,y)    :- x=22, y=19, Leg(x1,y1), AD(x,x1,dx),
                 AD(y,y1,dy), M(dx,dx,dx2), M(dy,dy,dy2),
                 dx2+dy2≤100, dx≤10, dy≤10.

end%SupplyDepotOptimized%
```

Table 4. Yacht Resupply Example.

5 Running Time of Methods and Sample Programs

Calculating the *lfp* of a Datalog program is exponential in the worst case. However, the running time depends on the amount of recursion in the Datalog program. We also note that a program need only be verified once and longer running times may be tolerated for program verification. The running times we report below indicate the time for the Datalog query to complete. The CDB-PV system uses 4,368kb of memory with no program loaded. The memory usage reported indicates the memory usage of the CDB-PV system when running the particular example and includes the 4,368kb. All runs were performed on an AMD Athlon 2000 with 1 GB or RAM except the Subway program which was run on an AMD X2 64bit computer with 1 GB of RAM.

The simple program from Table 2 can be evaluated precisely or with the under/over-approximation in the same running time. As this program has no recursion, the running time is less than our ability to measure (e.g. < 0.001 seconds) and the memory used is 5,556kb.

The yacht example from Table 4 depends on the D and M relations given in Equations (4) and (5). The under-approximation and over-approximation converge with $l = -15$. We executed the query with $l = -10, ... - 15$. In this more complicated example we still have good running times and memory usage as shown in Table 6.

Bound	**Under-Approximation**		**Over-Approximation**	
	Time	Memory	Time	Memory
-10	1.969	5680	33.593	7020
-11	2.297	5716	37.886	7176
-12	2.922	5712	20.395	6716
-13	3.375	5724	25.014	7012
-14	3.966	5752	31.869	7336
-15	5.106	5764	35.538	7704

Table 5. Yacht Program Running Times (seconds) and Memory Usage (KB).

These results show a dip on the over-approximation at $l = -12$. We believe that runs with larger l values require more calculations to find the upper bound of the `Reach` relation. With $l < -12$, the calculation cost of D and M dominate the time and memory.

The subway Datalog program from Table 3 has more recursion and complicated calculations. We expect it to take the longest time. The running times and memory usage for various runs are given in Table 6.

6 Conclusion and Future Work

We implemented an arbitrarily precise program verification and falsification method using constraint databases and approximation. The experiments showed that the constraint database method can more accurately approximate collecting semantics than

Bound	Under-Approximation		Over-Approximation	
	Time	Memory	Time	Memory
-18	0:07:01	25,900	1:14:35	237,744
-19	0:09:32	31,028	1:15:50	231,596
-20	0:12:51	32,364	1:33:16	277,648
-21	0:16:35	43,308	1:43:28	286,264
-22	0:20:48	50,164	1:55:09	312,396

Table 6. Subway Automaton Running Times (hh:mm:ss) and Memory Usage (KB).

other methods using widening techniques. Using over- and under-approximation we showed that our previous under-approximation constraint for the subway automaton is tight. In the ship resupply problem we showed our method is powerful enough to explore more complex mathematical expressions. While simple programs can be verified quickly, more complex programs may take longer as seen in the subway automaton verification. However, this method does provide precision beyond other techniques. Future work includes improving running time and memory efficiency while maintaining high precision and accuracy.

References

1. Kilroy, C.: Investigation: Air france 296. `http://www.airdisaster.com/investigations/af296/af296.shtml` (1997)
2. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. Proceedings of the Second International Symposium on Programming (1976) 106–130
3. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages. (1977) 238–252
4. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (1978) 84–96
5. Cousot, P., Cousot, R.: Abstract interpretation frameworks. J. Log. Comput. **2**(4) (1992) 511–547
6. Halbwachs, N.: Delay analysis in synchronous programs. In: CAV '93: Proceedings of the 5th International Conference on Computer Aided Verification, London, UK, Springer-Verlag (1993) 333–346
7. Kerbrat, A.: Reachable state space analysis of lotos specifications. In: Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques VII, London, UK, UK, Chapman & Hall, Ltd. (1995) 181–196
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, Springer, Berlin (January 17–19 2005) 1–24
9. Jaffar, J., Lassez, J.L.: Constraint logic programming. In: POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, New York, NY, USA, ACM Press (1987) 111–119

10. Jaffar, J., Michaylov, S., Stuckey, P.J., Yap, R.H.C.: The CLP(R) language and system. ACM Trans. Program. Lang. Syst. **14**(3) (1992) 339–395
11. Colmerauer, A.: Note sur prolog iii. In: SPLT'86, Séminaire Programmation en Logique. (1986) 159–174
12. Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T., Berthier, F.: The Constraint Logic Programming Language CHIP. Proceedings of the International Conference on Fifth Generation Computer Systems **2** (1988) 693–702
13. Kanellakis, P., Kuper, G., Revesz, P.: Constraint Query Languages. Journal of Computer and System Science **51**(1) (1995) 26–52
14. Revesz, P.: A Closed-Form Evaluation for Datalog Queries with Integer (Gap)-Order Constraints. Theoretical Computer Science **116**(1&2) (1993) 117–149
15. Revesz, P.: Introduction to Constraint Databases. Springer-Verlag (2002)
16. Matiyasevich, Y.V.: Hilbert's Tenth Problem. MIT Press (1993)
17. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math **5**(2) (1955) 285–309
18. Ullman, J.: Principles of database and knowledge-base systems. Computer Science Press Rockville, Md (1988)
19. Revesz, P.Z.: Reformulation and approximation in model checking. In: SARA '02: Proceedings of the 4th International Symposium on Abstraction, Reformulation, and Approximation, London, UK, Springer-Verlag (2000) 202–218
20. Revesz, P., Chen, R., Kanjamala, P., Li, Y., Liu, Y., Wang, Y.: The MLPQ/GIS constraint database system. In: ACM SIGMOD International Conference on Management of Data. (2000)
21. Revesz, P.: The constraint database approach to software verification. In: 8th Int. Conf. on Verification, Model Checking, and Abstract Interpretation, LNCS 4349 (2007) 329–345
22. Miné, A.: The octagon abstract domain. In: In Proceedings Analysis, Slicing and Transformation, IEEE Press (2001) 310–319
23. Anderson, S., Revesz, P.: Verifying the incorrectness of programs and automata. In: Proceedings of the 6th International Conference on Symposium on Abstraction, Reformulation and Approximation. Volume 3607., Springer Verlag (2005) 1–13