Verifying Complex Continuous Real-Time Systems with Coinductive CLP(R)

Neda Saeedloei and Gopal Gupta
Department of Computer Science,
University of Texas at Dallas,
Richardson, TX 75080.
Email: gupta@utdallas.edu

***Abstract*—Timed automata has been used as a powerful formalism for specifying, designing, and analyzing real time systems. We consider the generalization of timed automata to Pushdown Timed Automata (PTA). We show how PTAs can be elegantly modeled via logic programming extended with *coinduction* and *constraints over reals*. We use this logic programming realization of a PTA to develop an elegant solution to the *generalized railroad crossing problem* of Lynch and Heitmeyer. Interesting properties of the system can be verified merely by posing appropriate queries to this coinductive constraint logic program.**

I. Introduction

Design, specification, implementation and verification of real-time systems is an important area of research, as real-time systems are ubiquitous. Timed automata is a popular approach to designing, specifying and verifying real-time systems [1], [2]. Timed automata are ω-automata [3] extended with stop watches. Transitions from one state to another are not only made on the alphabet symbols of the language but also on constraints imposed on stop-watches (e.g., at least 2 units of time must have elapsed).

Timed automata are suitable for specifying a large class of real-time systems; however, they suffer from the same limitations that any automaton suffers, in that they can recognize only timed regular languages. This restriction to regular languages renders them unsuitable for many complex, useful applications where the language involved may not be regular. To overcome this problem, timed automata have been extended to pushdown timed automata which recognize timed context-free languages [4]. A PTA recognizes a sequence of timed words, where a timed word is a symbol from the alphabet of the language the automaton accepts, paired with the time-stamp indicating the time that symbol was seen. The sequence of timed words in a string accepted by a PTA must obey the rules of syntax laid down by the underlying untimed PDA, while the time-stamps must obey the timing constraints imposed on the times at which the symbols appear. Note that the concept of a PTA can be extended to the concept of timed context-free grammars or even timed context-sensitive grammars [5]; however, we don't elaborate on them here.

Earlier, Gupta and Pontelli showed how timed automata can be elegantly modeled via constraint logic programming over reals or CLP(R) [6]. Subsequently, Simon et al [7], [8] showed how coinduction can be introduced in logic programming to elegantly model and verify properties of ω-automata. In this paper we extend that work to show how coinduction and CLP(R) can also be used to elegantly model PTAs. We show how a coinductive CLP(R) rendering of a PTA can be used to verify safety and liveness properties of a system. We illustrate the effectiveness of our approach by showing how the *generalized railroad crossing problem* [9] can be elegantly modeled, and how its various safety and utility (liveness) properties can be elegantly verified.

The rest of the paper is organized as follows. We present an overview of timed automata, constraint logic programming over reals, and coinductive logic programming respectively. Next, we consider pushdown timed automata and timed grammars and show how they can be elegantly modeled via coinductive CLP(R). Note that the formulation of PTAs is our own, though they were first introduced in [10], [4]. We illustrate our method of modeling and verifying PTAs to elegantly solve the generalized railroad crossing problem. The railroad crossing problem considers verifying the safety and liveness properties of a gated train crossing with multiple tracks through which multiple trains can travel simultaneously in both directions.

II. Background

We give an overview of timed automata, CLP(R) and coinductive logic programming in the next 3 subsections. More details can be found in [11], [12] and [13] respectively.

A. Real-time systems and Timed Automata

Real-time systems are ubiquitous. Almost every embedded system found in various devices (e.g., cars) has to operate under real-time constraints. Various extensions of finite state automata have been proposed for embedding the notion of *time* and *time constraints* [14], [1], [15], [16] to model real-time systems and verify their properties. Timed automata is one of the most popular formalism. A *timed automaton* is a generalization of an automaton capable of recognizing infinite words (also known as an ω-*automaton* [3]). A ω-automaton over the alphabet Σ is a tuple $M = \langle \Sigma, \Delta, Q, Q_0, F \rangle$, where Q is the (*finite*) set of states, $Q_0 \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the

set of final states, and $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation. Given an infinite string $s_0 s_1 s_2 \ldots$, where $s_i \in \Sigma$, a derivation is defined as a sequence of transitions

$$q_0 \overset{s_0}{\mapsto} q_1 \overset{s_1}{\mapsto} q_2 \ldots$$

such that $q_0 \in Q_0$ and $(q_{i-1}, s_{i-1}, q_i) \in \Delta$.

Different notions of acceptance have been proposed. A *Büchi* automaton accepts a sequence $\bar{s} = s_0 s_1 s_2 \ldots$ iff there exists a state $q \in F$ and an infinite set of indices I such that $(\forall i \in I)(q = q_i)$. A *Müller* automaton defines F to be a subset of 2^Q, and a derivation for $\bar{s}$ lead to the acceptance of the string iff there exists $A \in F$ and an infinite set of indices I_a for each $a \in A$ such that $(\forall q \in A)(\forall i \in I_q)(q_i = q)$.

The purpose of a timed automaton is to recognize *timed words*. A timed word $(\bar{s}, \bar{t})$ associates a time value t_i with each symbol s_i of an infinite word $\bar{s}$. A timed automaton is obtained from a ω-automaton by adding:

- a finite set $\mathcal{C}$ of *clocks*;
- a set $\mathcal{P}$ of propositions over $\mathcal{C}$;
- a labeling function $\tau_C : \Delta \mapsto 2^{\mathcal{C}}$ (*reset* function);
- a labeling function $\tau_P : \Delta \mapsto prop(\mathcal{P})$, where $prop(A)$ is the set of propositional formulae over the set of atomic propositions A.

Thus, in a timed automaton, each transition (p, a, q, C', ϕ) not only consumes a symbol a from the string, but additionally

- resets all the clocks in $C' \in 2^{\mathcal{C}}$ $((\forall c \in C')(c := 0))$;
- verifies that the formula ϕ is satisfied by the current values of the clocks.

The set $\mathcal{P}$ of propositions is typically limited to propositions of the form $x \leq c$ and $c \leq x$, where $x \in \mathcal{C}$.

A derivation in a timed automaton is described as a sequence of transitions between states. A state for a timed automaton is a pair $\langle q, \nu \rangle$, where $q \in Q$ and ν is a clock valuation function ($\nu : \mathcal{C} \mapsto \mathbf{R}^+$, with $\mathbf{R}^+$ the set of non-negative real numbers). The initial state is $\langle q_0, \nu_0 \rangle$, with $q_0 \in Q_0$, and $\nu_0(x) = 0$ for all clocks $x \in \mathcal{C}$.
The transition

$$\langle q_i, \nu_i \rangle \xrightarrow{s_i, t_i} \langle q_{i+1}, \nu_{i+1} \rangle$$

takes place if the following conditions are met:

1) there is a transition (q_i, s_i, q_{i+1}) in Δ;
2) the proposition $\tau_P(q_i, s_i, q_{i+1})$ is satisfied by the clock valuation $\{\nu_i(x) + t_i - t_{i-1} : x \in \mathcal{C}\}$;

where the function ν_{i+1} is defined as follows:

$$\nu_{i+1}(x) = \begin{cases} 0 & \text{if } x \in \tau_C(q_i, s_i, q_{i+1}) \\ \nu_i(x) + t_i - t_{i-1} & \text{otherwise} \end{cases}$$

Thus, each transition can take place only if there is a correct matching on the input symbol and the current clock evaluation satisfies the time constraint on the transition. The effect of the transition is to lead to a new state and to reset the clocks specified by the function τ_C.

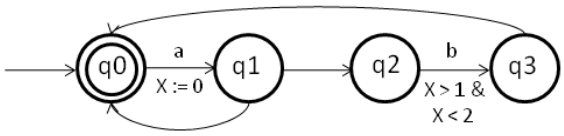

Figure 1. A Sample Timed Automaton

Figure 1 shows a simple timed automaton. It describes a system in which signals are recognized. Each signal a can (but need not) be followed by a b signal, with the constraint that the b signal must arrive at least one time unit after a and at most two time units after a. This is a Büchi automaton, the final state is $F = \{q_0\}$.

In the rest of this work we will limit our considerations to *deterministic timed automata* with Müller termination. A deterministic automaton is one that satisfies the following properties:

1) the set Q_0 is a singleton set, i.e., there is a single initial state;
2) for each $a \in \Sigma$, the number of a transitions originating from each state q_i is ≤ 1.

These conditions imply that for each word there is at most one possible derivation. In the specific case of timed automata, the second condition is modified as follow: for each $a \in \Sigma$ and $q \in Q$, if there are two transitions of the form$(q, a, ?_1)$ and $(q, a, ?_2)$ in Δ, then $\tau_P(q, a, ?_1) \wedge \tau_P(q, a, ?_2)$ is unsatisfiable (i.e., the two propositions are mutually exclusive), where $?_i$ stands for any arbitrary state.

B. Constraint Logic Programming over Reals

Constraint Logic Programming (CLP) is a many-sorted version of logic programming, in which different sorts are associated to different interpretation domains, and corresponding formulae are manipulated using predefined constraint solvers. The intuitive idea is to introduce special classes of formulae (constraints) which are not handled using traditional resolution, but are interpreted under a predefined specific interpretation and handled by external constraint solvers. For a more precise and complete presentation of CLP the reader is referred to the literature [17], [18].

The language at hand is built on two collections of symbols, Σ, containing all the function symbols, and Π, containing the predicate symbols. Furthermore, predicate symbols are divided into two separate classes, $\Pi = \Pi_c \cup \Pi_p$; Π_p contains the user-defined predicates, while Π_c contains the constraint predicates. Constraint predicates will be interpreted with respect to a predefined interpretation structure, while user-defined predicates will be subject to the user definitions. Π_c is always assumed to contain the equality symbol $=$.

Terms are objects created using the symbols from Σ and $\mathcal{V}$, where $\mathcal{V}$ is a collection of variables. A term is either a

simple variable or the application $f(t_1, \ldots, t_n)$ of a n-ary symbol $f \in \Sigma$ to n terms $t_1, \ldots, t_n$ ($n \geq 0$).

An atom is the application $p(t_1, \ldots, t_n)$ of a predicate symbol p to n terms $t_1, \ldots, t_n$. If $p \in \Pi_C$, then the atom is said to be a *constraint*. The program is composed by a collection of clauses, where each clause has the form:

$$head \ :- \ c \mid b_1, \ldots, b_k$$

$head$, b_i are user defined atoms while c is an arbitrary conjunction of constraints.

From the semantic point of view, constraints are interpreted using a predefined interpretation (i.e., a domain $\mathcal{D}$ together with an interpretation function I_D). In particular, a constraint c is solvable if $\mathcal{D} \models \exists(c)$. A solution θ for c is a mapping from the variables in c to $\mathcal{D}$, such that $\mathcal{D} \models c\theta$.

From the procedural point of view, execution of a constraint program requires the use of constraint solvers capable of deciding the solvability of each possible constraint formula[1]. Resolution is extended in order to embed calls to the constraint solvers. If $?{-}c_1 \mid g_1, \ldots, g_n$ is a goal, and p: $-c_2 \mid b_1, \ldots, b_k$ is a clause in the program, then the resolvent of the goal w.r.t. the given clause is

$$?{-}(c_1, c_2, g_1 = p) \mid b_1, \ldots, b_k, g_2, \ldots, g_n$$

as long as $\mathcal{D} \models (c_1 \wedge c_2 \wedge (g_1 = p))$. The constraint solver is used to test the validity of the condition on the constraints.

Frequently, constraint solvers are capable not only of checking solvability, but also of simplifying the constraints (eventually computing explicit solutions whenever possible); in this case, in the resolvent the constraint $c_1, c_2, (g_1 = p)$ is replaced with its simplified form.

An example of a CLP system is *CLP($\mathcal{R}$)*, where the constraint domain is the domain of real numbers, Σ contains real numbers and arithmetic operations ($+$, $*$, etc.), and Π_C contains the equality $=$ and the disequation predicates ($\leq$, $\geq$, etc.).

C. Coinductive Logic Programming

One difficulty in modeling timed automata and PTAs is that the underlying automaton is an ω-automaton which accepts infinite strings. Standard logic programming (which computes least fixed-points) is not equipped to model automata that accept infinite strings (which belong to the greatest fixed-points). Recently *coinduction* [19] has been introduced into logic programming by Simon et al [7], [8] to overcome this problem. Coinductive LP can also be used for reasoning about unfounded sets, behavioral properties of (interactive) programs, elegantly proving liveness properties in model checking, type inference in functional programming, etc. [13].

Coinduction is the dual of induction and corresponds to the greatest fixed-point (*gfp*) semantics. Simon et al's work gives an operational semantics—similar to SLD resolution—for computing the greatest fixed-point of a logic program. This operational semantics (called co-SLD resolution) relies on the *coinductive hypothesis rule* and systematically computes elements of the *gfp* of a program via backtracking. It is briefly described below. The semantics is limited only to *regular proofs*, i.e., those cases where the infinite behavior is obtained by infinite repetition of a finite number of finite behaviors (and, thus, is powerful enough for modeling ω-automata).

In the coinductive LP (co-LP) paradigm the declarative semantics of the predicate is given in terms of *infinitary Herbrand (or co-Herbrand) universe, infinitary Herbrand (or co-Herbrand) base [20], and maximal models (computed using greatest fixed-points)* [7]. The operational semantics under coinduction is identical to Prolog's operational semantics except for the following addition [7]: a predicate call `p(`$\bar{t}$`)` succeeds if it unifies with one of its ancestor calls. Thus, every time a call is made, it has to be remembered. This set of ancestor calls constitutes the *coinductive hypothesis set*. Under co-LP, infinite *rational* answers can be computed, and infinite rational terms are allowed as arguments of predicates. Infinite terms are represented as solutions to unification equations and the occurs check is omitted during the unification process: for example, `X = [1 | X]` represents the binding of `X` to an infinite list of `1`'s. Thus, in co-SLD resolution, given a single clause (note the absence of a base case)

```
p([ 1 | X ]) :- p(X).
```

the query `?- p(A)` will succeed in 2 resolution steps with the (infinite) answer:

```
A = [1 | A]
```

which is a finite representation of the infinite answer: `A = [1, 1, 1, ....]`.

An important application of coinductive LP is in directly representing and verifying properties of Kripke structures and ω-automata (automata that accept infinite strings). Just as automata that accept finite strings can be directly programmed using standard LP, automata that accept infinite strings can be directly represented using coinductive LP (one merely has to drop the base case). Consider the automata (over finite strings) shown in Figure 2 which is represented by the logic program below.

```
automaton([X|T], St):-
          trans(St, X, NewSt),
          automaton(T, NewSt).
automaton([], St) :- final(St).
trans(s0, a, s1).        trans(s1, b, s2).
trans(s2, c, s3).        trans(s3, d, s0).
```

[1]It is common for the programmer to identify only special types of constraint formulae, the *admissible* constraints; these are the only constraints which are admitted during the execution of a program. This is because it is not possible to devise a constraint solver that will solve any arbitrary set of constraints.

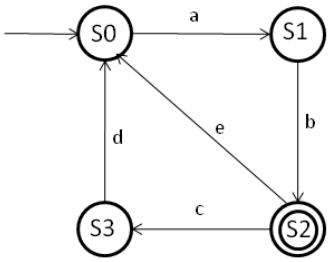

Figure 2. An Automaton

```
trans(s2, e, s0).          final(s2).
```

A call to `?- automaton(X, s0)` in a standard LP system will generate all finite strings accepted by this automaton. Now suppose we want to turn this automaton into an ω-automaton, i.e., it accepts infinite strings (an infinite string is accepted if states designated as final state are traversed an infinite number of times), then the (coinductive) logic program that simulates this automaton can be obtained by simply dropping the base case (for the moment, we'll ignore the requirement that final states occur infinitely often)

```
automaton([X|T], St) :-
            trans(St, X, NewSt),
            automaton(T, NewSt).
```

Under coinductive semantics, posing the query `| ?- automaton(X, s0).` will yield the cyclical solutions:

```
X = [a, b, c, d | X];
X = [a, b, e | X];
```

This feature of coinductive LP can be leveraged to modeling and verifying properties of (timed) ω-automata directly and elegantly.

III. Pushdown Timed Automata

A Pushdown Timed Automaton extends a timed automaton with a stack in exactly the same manner that a pushdown automaton extends a finite automaton. Thus, a PTA is obtained from a timed automaton by adding:

- ϵ (empty string) to the input alphabet Σ.
- a stack alphabet $\Gamma_\epsilon = \Gamma \cup \epsilon$
- a stack represented by Γ_ϵ^*.

Acceptance conditions for an infinite string for a PTA are similar to those for timed automata except that, additionally, the stack must be empty. The transition relation is extended to include the state of the stack (to represent that a stack symbol may be pushed or popped during a transition). Thus, the transition function becomes:

$$\langle q_i, \nu_i, a\gamma \rangle \xrightarrow{s_i, t_i} \langle q_{i+1}, \nu_{i+1}, b\gamma \rangle$$

where $a, b \in \Gamma_\epsilon$ and $\gamma \in \Gamma_\epsilon^*$. Note that s_i may be an empty word (ϵ).

Pushdown timed automata have been introduced earlier [4], [21], [10]. Our aim in this paper is to show how PTAs can be modeled and their properties verified with coinductive constraint logic programming over reals with the same ease as that for timed automata [6].

In many cases real-time systems that are naturally modeled as PTAs can be modeled as timed automata by imposing restrictions (such as limiting the size of the string, i.e., limiting the number of allowable events), but, our experience indicates that such a timed automaton will have an enormous number of states, and thus would be unwieldy and time consuming to specify. Proving its safety and liveness properties will also be quite cumbersome simply due to the large size of the automaton.

From PTA one can also develop the notion of timed grammars [5]; however, in this paper we restrict ourselves to PTAs.

As an example of a PTA, consider a language in which sequences of a's are followed by sequences of an equal number of b's (each such string has at least two a's and at least two b's). For each pair of equinumerous sequences of a's and b's, the first b symbol must appear within 5 units of time from the first a symbol and the final b symbol must appear within 20 units of time from the first a symbol. The grammar annotated with clock constraints is shown below. Note that c is a clock; clock expressions are written within braces.

$S \rightarrow R \;\; S$
$R \rightarrow a \;\; \{c := 0\} \;\; T \;\; b \;\; \{c < 20\}$
$T \rightarrow a \;\; T \;\; b$
$T \rightarrow a \;\; b \;\; \{c < 5\}$

Note also that the first rule is coinductive (i.e., a recursive rule with no base case) and accepts infinite strings. Thus, the above grammar is an ω-grammar. The PTA realizing this timed grammar is shown in Figure 3. Note that in Figure 3, S0 is the final state. Actions `push(1)` and `pop()`, respectively push 1 onto the stack and pop the stack (the automaton in Figure 3 will accept empty string also; we allow this for simplicity of presentation). The requirement that the stack be empty ensures that only strings with equal numbers of a's and b's are accepted. Note that the global time (or wall clock time) keeps advancing at the normal uniform rate, as the automaton makes transitions.

IV. Modeling PTAs with coinductive CLP(R)

Gupta and Pontelli [6] showed how constraints over reals can be used to model continuous real-time systems. They showed that continuous time and associated clock constraints can be elegantly specified as a CLP(R) program, which, in turn can be used to verify interesting properties of the system, e.g., safety and liveness properties. In their technique, each transition of the automaton is modeled as a logic programming fact. It is extended with extra arguments to model time: one argument is added to model the global

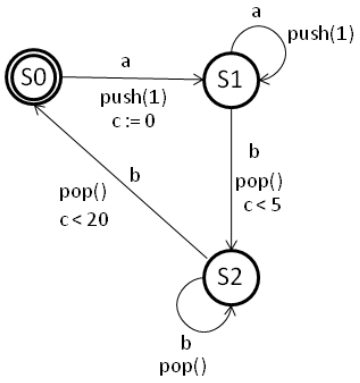

Figure 3. A Pushdown Timed Automaton

(wall clock) time, a pair of arguments are added for each stop watch used in the automaton. The first argument of this pair is used to remember the last (wall clock) time the stopwatch was reset, while the second one is used to pass on the stop watch value to the next transition.

We described earlier how ω-automata can be modeled as coinductive logic programs. Thus, coinductive logic programming extended with constraints provides a natural and practical formalism for representing real-time systems modeled as timed automata [13], [22]. Introduction of coinduction significantly simplifies Gupta and Pontelli's realization of ω-automata [13]: the *driver* program used to compose the various automata involved in the system is considerably simplified.

PTAs and timed grammars accept infinite strings. As is well known, automata and grammars can be elegantly modeled via logic programming [23]. Specifically, the definite clause grammar facility of Prolog allows one to obtain a parser for context-free grammars or even context-sensitive grammars with a minimal amount of work. By extending logic programming with coinduction, one can develop language processors that recognize infinite strings. Definite clause grammars (DCGs) extended with coinduction can act as recognizers for ω-PDAs and ω-grammars. Further, incorporation of coinduction and constraint logic programming over reals into the definite clause grammar allows modeling of time aspects of the system. Once a timed system is modeled as a coinductive constraint logic program, it can be used to (i) verify if a particular timed-string will be accepted or not; and, (ii) systematically generate all possible timed strings that can be accepted. The coinductive CLP realization of the system can also be used to verify system properties by posing appropriate queries.

Consider the PTA shown in Figure 3 in section III. The logic programming rendering of this PTA is shown below. To keep matters simple, this logic program models the PTA as a collection of transition rules (one rule per transition in the PTA), where each rule is extended with stack actions as well as clock constraints. The first 3 arguments of the `trans/8` are self-explanatory. The fourth argument represents the global (wall clock) time. The pair of arguments, `Tr` and `To`, represent the stopwatch `c` of the timed automaton, while the last two arguments represent the stack actions.

The coinductive `driver/6` rule realizes the automata, calling the `trans/8` rule repeatedly. The CLP(R) constraints are enclosed within curly braces, as is the convention in most Prolog systems. The constraint `Ta > T` advances the time on the wall clock after every transition, and that the predicate `driver/6`'s coinductive success will depend only on the first four arguments, i.e., the wall-clock time will be ignored to check if the `driver/6` predicate is cyclical. The driver generates the timed trace of events as output.

```
trans(s0,a,s1,T,Tr,To,_,[1])  :- {To=T}.
trans(s1,a,s1,T,Tr,To,C,[1|C]):-{To=Tr}.
trans(s1,b,s2,T,Tr,To,[1|C],C):-
        {T - Tr < 5,  To = Tr}.
trans(s2,b,s2,T,Tr,To,[1|C],C):-{To=Tr}.
trans(s2,b,s0,T,Tr,To,[1|C],C):-
        {T - Tr < 20, To = Tr}.

:-coinductive(driver/6).
driver([X| R],Si,C1,Tr,T,[(X, T)| S]) :-
           trans(Si,X,Sj,T,Tr,To,C1,C2),
           {Ta > T},
           driver(R,Sj,C2,To,Ta,S).
```

Given this program one can pose queries to it to check if a timed string satisfies the timing constraint. Alternatively, one can generate possible (cyclical) legal timed strings. Finally, one can verify properties of this timed language (e.g., checking the trivial property that all the a's are generated within 5 units of time, in any timed string that is accepted).

Next we show how our coinductive CLP(R) realization of PTAs can be used to elegantly solve the generalized railroad crossing problem.

V. The Generalized Railroad crossing

The Generalized Railroad Crossing(GRC) problem has been proposed [9] as a benchmark problem in order to compare the formal methods that have been invented for specifying, designing, and analyzing real-time systems. It also provides a better way to understand the use of these methods in developing practical real-time systems. Informally, the GRC problem consists of several tracks and an unspecified number of trains traveling in both directions. There is a gate at the railroad crossing that should be operated in a way that guarantees the *safety* and *utility* properties. The safety property stipulates that the gate must be down while one or more trains are in the crossing. The utility property states that the gate must be up when there is no train in the crossing. The formal statement of the GRC problem, taken directly from [9], is as follows.

The system to be developed operates a gate at a railroad crossing. The railroad crossing I lies in a region of interest R, i.e., $I \subseteq R$. A set of trains travel through R on multiple tracks in both directions. A sensor system determines when each train enters and exits region R. To describe the system formally, we define a gate function $g(t) \in [0, 90]$, where $g(t) = 0$ means the gate is down and $g(t) = 90$ means the gate is up. We also define a set $\{\lambda_i\}$ of *occupancy intervals*, where each occupancy interval is a time interval during which one or more trains are in *I*. The *i*th occupancy interval is presented as $\lambda_i = [\tau_i, \nu_i]$, where τ_i is the time of the *i*th entry of a train into the crossing when no other train is in the crossing and ν_i is the first time since τ_i that no train is in the crossing (i.e., the train that entered at τ_i has exited as have any trains that entered the crossing after τ_i). Given two constants ξ_1 and ξ_2, $\xi_1 > 0$, $\xi_2 > 0$, the problem is to develop a system to operate the crossing gate that satisfies the following two properties:
Safety Property: $t \in \cup_i \lambda_i \Rightarrow g(t) = 0$
Utility Property: $t \notin \cup_i [\tau_i - \xi_1, \nu_i + \xi_2] \Rightarrow g(t) = 90$

Note that in the problem description above, I and R are used respectively to denote the railroad crossing, and the region from where a train passes a sensor until it exits the crossing. Some positive real-valued constants are also defined by the GRC as follows:

- ϵ_1, a lower bound on the time from when a train enters R until it reaches I.
- ϵ_2, an upper bound on the time from when a train enters R until it reaches I.
- γ_{down}, an upper bound on the time to lower the gate completely.
- γ_{up}, an upper bound on the time to raise the gate completely.
- ξ_1, an upper bound on the time from the start of lowering the gate until some train is in I.
- ξ_2, an upper bound on the time from when the last train leaves I until the gate is up (unless the raising is interrupted by another train getting "close" to I).
- β, an arbitrarily small constant used to take care of some technical race conditions.
- δ, the minimum useful time for the gate to be up. (For example, this might represent the minimum time for a car to pass through the crossing safely.)

Some restrictions are placed on the values of the various constants:

1) $\epsilon_1 \leq \epsilon_2$.
2) $\epsilon_1 > \gamma_{down}$. (The time from when a train arrives until it reaches the crossing is sufficiently large to allow the gate to be lowered.)
3) $\xi_1 \geq \gamma_{down} + \beta + \epsilon_2 - \epsilon_1$. (The time allowed between the start of lowering the gate and some train reaching I is sufficient to allow the gate to be lowered in time for the fastest train, and then to accommodate the slowest train.)
4) $\xi_2 \geq \gamma_{up}$. (The time allowed for raising the gate is sufficient.)

We show that the GRC problem can be elegantly modeled as a PTA. The PTA has been implemented as a coinductive CLP(R) program. We have verified the safety and utility properties, as well as other interesting properties of the system, by posing appropriate queries to the program.

VI. Solving the GRC with coinductive CLP(R)

For simplicity of presentation, we present our problem in two steps. We first restrict the number of tracks to one, but allow arbitrary number of trains to travel on it, one after the other (it is theoretically possible for the gate to never go up once it goes down, if an infinite number of trains arrive one after the other within close enough interval). Next, we address the full GRC.

A. 1-Track Generalized Railroad Crossing Problem

The 1-Track Generalized Railroad Crossing Problem consists of one track and an unspecified number of trains traveling through the track in one direction. Our task is to develop (specify and prove correct) the system to control the gate at the crossing. The system consists of three components, an automaton to control the gate (gate automaton), an automaton that corresponds to the track and models the behavior of trains traveling through the track (track automaton), and an automaton that acts as an overall controller (controller automaton).

The gate automaton is modeled as a timed automaton with four states which takes actions based on four different events: *(i)* `lower` indicates starting of lowering the gate; *(ii)* `down` indicates the gate being down; *(iii)* `raise` indicates starting of raising the gate; and, *(iv)* `up` indicates the gate being up.

The track is modeled as a timed automaton with five states which takes actions based on three events: *(i)* `approach` indicates a train approaching the crossing area; *(ii)* `in` indicates the train being in the crossing area; and, *(iii)* `exit` indicates that the train has left the crossing area. The track automaton assumes that there cannot be two trains at the same time in the crossing area. In other words trains travel in a safe distance from each other. If there is a new `approach` signal received while there is a train already in the crossing, the previous train should leave the crossing area before the new train can enter the crossing area. The range of sensors is such that the *approach* signal of only at most one approaching train is registered (thus, there could be one train in the crossing and another one approaching

that the system reacts to). There could be a situation in which multiple trains travel one after the other (in a safe distance from each other). In this situation, the system will work properly in the sense that the first train will enter the crossing area, the second train will enter the crossing area after exiting the first train, the third train will take the place of the second train and so on. Therefore the gate remains down until the last train exits the crossing area. Note that the gate crossing system is not responsible for ensuring safe distance between trains, its task is to ensure the safety and utility of the crossing.

The controller automaton is modeled as a PTA with four states. The controller automaton must keep track of trains currently in the system (i.e., those trains whose approach signal has been received): it has to ensure that the number of `approach` events is identical to those of `exit` events. Timed automata are not appropriate for specifying the controller automaton, for two reasons: *(i)* we don't know the number of `approach` events in advance, so we cannot design one general timed automaton for the controller that works for an arbitrary number of tracks and trains. In other words, we would have to have different timed automata for different numbers of tracks. *(ii)* The controller automaton would become too complicated as the number of tracks increases. More tracks means more states and transitions, therefore a more complicated automaton. Then use of a stack in a PTA eliminates the need for new extra states and transitions as the number of `approach` signals and tracks increase.

The controller automaton must respond to four events: `approach`, `lower`, `exit`, and `raise` described above. On receiving an `approach` signal at state s0 the controller clock will be reset. This will ensure lowering of the gate before the train gets into the crossing area. The controller clock will not get reset if the `approach` signal is received while in other states. The stack in PTA is used to keep track of the number of trains in the system. On receiving an `approach` signal the controller pushes the symbol "1" onto the stack and on receiving the `exit` signal, it pops a "1" from the stack. When the stack is empty, the controller sends the `raise` signal to the gate as the last train has left the system and it is safe to raise the gate. To implement this, we have used a counter which is increased by 1 on receiving an `approach` signal and decreased by 1 on receiving an `exit` signal. A transition is activated by a pair (event, state of counter) and triggers an action on the counter. Testable states of the counter are "= 0" and "$\neq$ 0", and counter actions are `increment` and `decrement`. The automaton may ignore the state of the counter and act on the input signal. For example on receiving an `approach` signal at any state, the automaton increments the counter regardless of its current state. The automaton can act based on both the input signal, and the counter state. On receiving the `exit` signal in state s2, the automaton checks the state of the counter and it might stay in the same state or move to state s3. If the counter is equal to zero, the controller will go to state s3 and reset its clock. This will ensure that a `raise` signal will be sent to the gate automaton within $\xi_2 - \gamma_{up}$ units of time after the controller clock is reset. If the counter is not equal to zero, the controller remains in state s2 and its clock will not get reset. Figure 4 shows the timed automata for track *(i)*, gate *(iii)*, and the PTA for controller *(ii)*. Note that in the controller automaton in Figure 4, `s` is the stack (modeled via a counter in the CLP(R) code, as mentioned above).

For modeling the 1-track GRC problem we set ϵ_1 = 2, ϵ_2 = 3, γ_{down} = 1, $\gamma_{up} = 2$, ξ_1 = 2, $\xi_2 = 3$. Note that these values are taken directly from [2]. A real-time system designer can choose other values for these parameters. The GRC does not put any restrictions on how long a train can take to pass the gate crossing (theoretically speaking, a train can even stop at the gate and stay their indefinitely). To disallow such behaviors, we put an upper bound on the maximum time a train should take to exit the crossing. We introduce a constant σ, which is the maximum time in which the `exit` signal should appear since the `approach` signal was seen. For GRC, $\sigma = \infty$. Following [2] we set $\sigma = 5$.

The behavior of the timed automaton for the track can be specified by the following CLP(R) rules.

```
track(s0, approach, s1, GT, TI, TO, L) :-
    {TO = GT}.
track(s1,  in,  s2,  GT,  TI,  TO,  L) :-
    {GT - TI > 2, GT - TI < 3, TO = TI}.
track(s2, approach, s3, GT, TI, TO, L) :-
    {TO = GT}.
track(s4,  in,  s2,  GT,  TI,  TO,  L) :-
    {GT - TI > 2, GT - TI < 3, TO = TI}.
track(s3,  exit,  s4, GT, TI,  TO,  L) :-
    first(L, First),
    {GT - First < 5, TO = TI}.
track(s2,  exit,  s0, GT, TI,  TO,  L) :-
    first(L, First),
    {GT - First < 5, TO = TI}.
track(X,  lower,  X,  GT,  TI,  TI,  L).
track(X,  down,   X,  GT,  TI,  TI,  L).
track(X,  raise,  X,  GT,  TI,  TI,  L).
track(X,  up,     X,  GT,  TI,  TI,  L).
```

The first argument of the `track` predicate is the current state of the track. The second argument is one of the events triggering an action explained above. The third argument is the new state that results. `GT` represents the global wall clock. TI is the last time the track clock was reset. TO is the new reset time which depends on whether the track clock gets reset in this transition or not (it is set to either GT or TI). The last argument, `L`, keeps track of all the reset times: since there might be more than one train in the system at any given time, every time a train gets into or exits the crossing

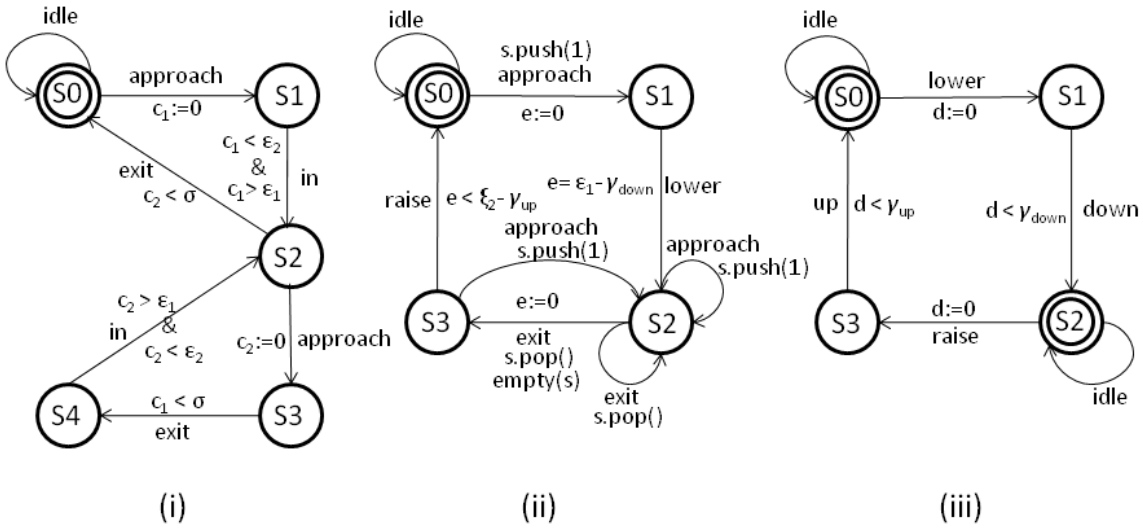

Figure 4. track, controller, and gate automaton

area, it should use its own approach time rather than the last approach time. The list `L` keeps track of all the stop-watches that are active due to multiple trains signaling an `approach`.

Since trains enter into and exit the crossing area in the order they arrive in the system,[2] each train on `exit` event uses the first element of list `L` as its own approach time. After each `exit` the first element of list `L` is removed from the list.

```
gate(s0, lower, s1, GT, TI, TO) :-
          {TO = GT}.
gate(s1, down , s2, GT, TI, TO) :-
          {TO = TI, GT - TI < 1}.
gate(s2, raise, s3, GT, TI, TO) :-
          {TO = GT}.
gate(s3, up,    s0, GT, TI, TO) :-
          {TO = TI, GT - TI < 2}.
gate(X, approach, X, GT, TI, TI).
gate(X, in,       X, GT, TI, TI).
gate(X, exit,     X, GT, TI, TI).
```

The `gate` predicate has the same arguments as the track predicate except for the last argument, `L`.

```
contr(s0,approach,C1,C2,s1,GT,TI,TO):-
     C2 is C1 + 1, {TO = GT}.
contr(s1,lower,   C1,C1,s2,GT,TI,TO):-
     {GT - TI = 1, TO = TI}.
contr(s2,exit,    C1,C2,s3,GT,TI,TO):-
     C1 = 1, C2 is C1 - 1, {TO = GT}.
contr(s2,approach,C1,C2,s2,GT,TI,TO):-
     C2 is C1 + 1, {TO = TI}.
contr(s2,exit,    C1,C2,s2,GT,TI,TO):-
     C1 > 1, C2 is C1 - 1, {TO = TI}.
contr(s3,approach,C1,C2,s2,GT,TI,TO):-
     C2 is C1 + 1, {TO = TI}.
contr(s3,raise,  C1,C1,s0,GT,TI,TO) :-
     {GT - TI < 1, TO = TI}.
contr(s2,in,C1,C1,s2,GT,TI,TI).
contr(s0,up,C1,C1,s0,GT,TI,TI).
contr(s2,down,C1,C1,s2,GT,TI,TI).
```

The `controller` predicate has the same arguments as the `gate` predicate with two extra arguments, C1 and C2. C1 is the current counter in the system (the number of active approach signals), and C2 is the new counter after the transition. If the event is `approach` the counter will be incremented by one, if the event is `exit` it will be decremented by one, otherwise it will not change.

The main `driver` predicate is shown below.

```
driver(C1,ST,SC,SG,GT,CT,CC,CG,[X|Rest],
                        Resets,[(X,GT)|R])
    :-
      track(ST,X,STO,GT,CT,CTO,Resets),
      contr(C1,SC,X,C2,SCO,GT,CC,CCO),
      gate(SG,X,SGO,GT,CG,CGO),
      {TA > GT},
      (X = approach ->
        add-to-list(GT,Resets,NewResets);
      (X = exit ->
        delete-first(Resets,NewResets);
      NewResets = Resets)),
      driver(C2,STO,SCO,SGO,TA,CTO,CCO,CGO,
```

[2]For the GRC (multiple tracks), this assumption may not hold: a long slow train in one track that approaches first, may be overtaken by a fast short train that approaches later but exits first. However, this situation can be handled easily in our framework and indeed we handle it in our treatment of the full GRC (see the code for the `driver/12` predicate in Section VI-B).

```
                                    Rest,NewResets,R).
```

The `driver` takes as input (i) the system counter, starting at zero; (ii) the current states of track, controller, and gate; (iii) The current global time; (iv) The last reset time for each of the automaton; (v) the list of events; and, (vi) the list of `approach` times, starting with empty set, []. The `driver` generates output as a list of pairs which is the last argument of `driver`. The first element of the pair is the event name and the second element is the time at which the event happened. Note that `driver/11` is specified coinductively, as it executes forever. The `add-to-list/3` and the `delete-first/2` predicates manage the stop-watches that have to maintained for each track.

Given the logic programming definitions of controller, track, and gate automata and the driver routine that composes the three automata, one can check if a given sequence of timed events is legal or not, i.e., use the logic program as a simulator. One can also generate a sample sequence of timed events accepted by the system (here we use the generational power of logic programming). Finally, properties of interests are verified as follows: given a property `Q` to be verified, specify its negation as a logic program. Let's call this predicate `notQ`. If the property `Q` holds, the query `notQ` will fail w.r.t. the logic program that models the system. If the query `notQ` succeeds, the answer provides a counter example to why the property `Q` does not hold.

We prove the safety property of the system in two phases. In the first phase we show that system is safe after the first `approach` signal. We define the `firstinbeforedown/1` predicate in which we have negated the safety property by looking for any solution that contains `in` before `down` after the first `approach`. In other words we look for any possibility that the first train is in the crossing area before the gate is down. In this predicate we assume that the first `approach` occurs at time T = 0.0. The call to `firstinbeforedown` will fail which indicates that the system is safe after the first train approaches.

```
firstinbeforedown(X) :-
       driver(0,s0,s0,s0,0,0,0,0,X,[],R),
       append(B, [(in, _)|_], R),
       append([(approach, 0)], A, B),
       \+member(down, A).
```

In second phase we show that the safety property holds for the subsequent approaches of the trains. We define the `inbeforedown/1` predicate in which we have negated the safety property again. In this predicate we check for any possibility of the gate being `up` followed by `in` without `down` in between. The call for this query also fails.

```
inbeforedown(X) :-
       driver(0,s0,s0,s0,0,0,0,0,X,[],R),
       append(C, [(in, _)|_], R),
       append(A, B, C),
       append(_, [(up, _)], A),
       \+member((down, _), B).
```

Similarly we can check the utility property using the `utility/1` predicate defined below. In this predicate we ensure that the gate is up when there is no train in the crossing area or approaching the crossing. In other words it looks for feasibility of situations in which the gate is down without any train being in the crossing area. So, if a call to the `utility` predicate fails we know that the utility property is satisfied.

```
utility(X) :-
       driver(0,s0,s0,s0,0,0,0,0,X,[],R),
       append(A, B, R),
       member([(down, _)], A),
       \+member((in, _), B).
```

Other interesting properties of the system can also be verified using appropriate queries. For example one can compute the minimum time distance between two consecutive trains (i.e., two consecutive `approach` signals) through a call to the `distance/2` predicate below. The solution produced by this query is $N > 2$, and $M < 5$, which indicates the range of this minimum for the system to operate safely.

```
distance(M, N) :-
       driver(0,s0,s0,s0,0,0,0,0,X,[],R),
       append(A, [(approach, T2)|_], R),
       append([(approach, T1)], B, A),
       \+member((approach, _), B) ,
       {T2 - T1 >= M, T2 -T1 =< N}.
```

B. Solution for the GRC Problem

We next tackle the fully general case of the GRC. Note that most of the issues that are raised by GRC problem have to be handled in the 1-Track GRC problem, too, so solving the GRC problem is straightforward. We use the solution for the 1-track GRC problem and generalize it to handle any number of tracks and an unspecified number of trains traveling through the tracks in both directions. We assume that at most one train can be in the crossing area in each track at any given time. In other words if the number of tracks is equal to some number `n` then there can be at most `n` trains in the crossing area at any given time.

In modeling the GRC problem we make use of the same PTA for the controller that we used for 1-track GRC problem with a slight difference: it has to be cognizant of all the approach signals coming from trains on different tracks. Approach signals from trains on different tracks can arrive arbitrarily close to each other. On receiving an `approach` signal in state s1 the extended controller automaton stays in state s1 and increments the counter. This addition allows the automaton to handle multiple `approach` signals in different tracks arriving very close to each other or additional approach signals arriving after the first `approach` has been

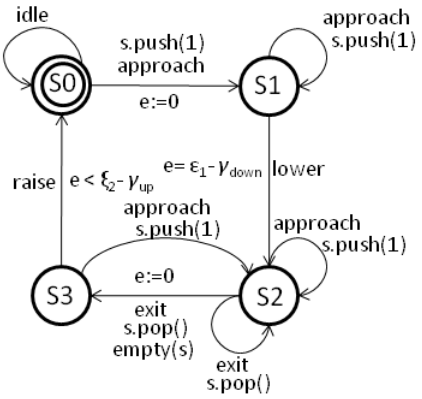

Figure 5. The Controller Automaton

received but before enough time has elapsed for the gate to go down. Note that the controller clock doesn't get reset on this transition. The reason is that the `lower` signal is sent to the gate automaton at $\epsilon_1 - \gamma_{down}$ units of time after the first `approach`. Figure 5 shows the controller automaton for the GRC problem.

To model the full GRC, the number of tracks has to be given as an input to the system so that any number of tracks can be handled. A train can arrive on any track at any instant of time. On receiving a new `approach` signal from a train on a given track, the system will respond to it assuming that there is no other train in that track or that any trains on that track will exit the crossing area before the new train would arrive there.

Thus, the controller predicate remains the same as in the 1-Track GRC problem except for one additional rule needed to handle multiple `approaches` in different tracks at the same time as follows.

```
contr(C1,s1,approach,C2,s1,GT,TI,TO):-
            C2 is C1 + 1, {TO = TI}.
```

We make use of the same timed automaton for the gate that we used for 1-track GRC problem. Likewise, the `gate` predicate remains unchanged. The timed automaton for the track is the same as in the 1-track GRC problem, except that each track has its own track automaton that models the trains running on that track. Tracks are specified by track numbers which is implemented by adding the track number to the track predicate that we used for 1-Track GRC problem. All these track automata are initialized at state s0 and they work in parallel. When an event takes place in a specified track, only that track responds to this event and all automata for other tracks don't respond to this event and remain in their current states.

The `driver/12` predicate for GRC problem is shown below.

```
driver(C1,NoTracks,SC,SG,GT,CC,CG,[X|Rest],
  Tracks,TrkResets,Trains,[(X,Track,GT)|R]):-
  contr(C1, SC, X, C2, SCO, GT, CC, CCO),
  gate(SG, X, SGO, GT, CG, CGO),
  {TA > GT},
  ((X = approach, N is NoTracks + 1,
  random(1, N, Track);
  (X=in; X=exit), member(Track, Trains))->
    nthElement(Track, Tracks, Trk),
    arg(1, Trk, CT), arg(2, Trk, ST),
    nthElement(Track, TrkResets, TrkReset),
    train(Track, ST, X, STO, GT, CT, CTO, TrkReset),
    update(Track, Tracks, (CTO,STO), NewTracks),
    (X = approach ->
       add-to-list(GT, TrkReset, NewTrk),
       update(Track,TrkResets,NewTrk,NewTrkResets),
       add-to-list(Track, Trains, NewTrains);
    (X = exit ->
       delete-first(TrkReset, NewTrk),
       update(Track,TrkResets,NewTrk,NewTrkResets),
       delete-first(Track, Trains, NewTrains);
     NewTrkResets = TrkResets,NewTrains = Trains));
  NewTrkResets = TrkResets,
  NewTrains = Trains,
  NewTracks = Tracks),
  driver(C2,NoTracks,SCO,SGO,TA,CCO,CGO,
  Rest,NewTracks,NewTrkResets,NewTrains,R).
```

In this predicate, `NoTracks` is the number of tracks in the system (given as an input to the program). `Tracks` is a list of $\langle time, state \rangle$ pairs (one pair for each track), specifying the status of each track; `time` indicates the last time that clock was reset in a particular track and `state` indicates the current state of the track automaton for that track. Initially `Tracks` is a list of $\langle s0, 0 \rangle$ pairs, indicating that all tracks are initially in state s0 at time 0. The `update` predicate takes the track number and updates the status of that track in `Tracks` list, producing `NewTracks` list. `TrkResets` is a list of lists. The number of lists in `TrkResets` is equal to the number of tracks in the system. Each list in `TrkResets` contains the `approach` times on each track. Initially `TrkRestes` is a list of empty lists. On receiving `approach` and `exit` signals on any track, the corresponding list for that track in the `TrkResets` is updated. With these changes, our framework can handle situations where `approaches` and `exits` of various trains on various tracks may be inter-mixed: for example, a long slow train in one track that approaches first, may be overtaken by a fast short train in another track that approaches later but exits first. Being able to handle such situations is the reason that we store `approach` times of each track separately in `TrkResets`. The predicates, `add-to-list/3`, `update/4`, `delete-first/2`, and `delete-first/3`, thus, all pertain to management of various track stop-watches (clocks).

`Trains`, is a list of integers from the domain D, where D = 1 .. `NoTracks`. The `Trains` list (initialized to empty list, []) consists of all the trains currently in the system, which are specified by their track number. For example if `Trains` gets bound to [2, 1, 3, 2], it means that there are currently four trains in the system: one train in tracks

1, and 3 each, and two trains in track 2. On receiving an `approach` signal on a track, the track number is added to the `Trains` list. On receiving an `exit` signal on a track, the first occurrence of that track number is removed from the `Trains` list by `delete-first` predicate. The last argument of `driver` is the output of the program which is a list of $< X, Track, T >$ triples where `X` is an event, `Track` is the track number in which the event happened, and `T` is the time that event occurred. The rest of the arguments of `driver` are as before.

The `safety` and `utility` properties along with other properties of the system can be verified by posing appropriate queries similar to those of the 1-Track GRC problem. In fact, these queries are identical to those of the 1-Track problem except that track numbers are added to queries. Therefore, these queries are not reproduced here.

VII. Conclusions and Related Work

Real-time systems have been studied with several types of real-time logics. Automata based real-time formalisms such as timed automata [2], [9] and timed transition systems have been also proposed to model and analyze a wide range of real-time systems. Jaffar [22] builds on the work of Gupta and Pontelli and translates timed automata to a CLP program and uses it for proving assertions (properties of the system can be expressed in an expressive assertion language). All these papers do not consider PTAs, they limit themselves to timed automata.

Discrete pushdown timed automata were originally introduced by Zhe Dang, et al. [4]. Pushdown timed automata with dense clocks were also considered by Dang [10] and used to give a decidable characterization of the binary reachability of a PTA. However, the treatment in their work is largely theoretical, there is not much focus on how to elegantly and efficiently realize PTAs. In contrast, we are more interested in elegantly modeling and analyzing PTAs applied to complex applications such as the GRC.

Few solutions have been proposed for GRC problem. The most notable is that of Puchol [24], which is based on the ESTEREL programming language. In Puchol's work, time is discretized and thus is not faithful to the original problem. In contrast, our solution treats time as continuous. In Puchol's approach, the number of trains that the solution comprises must be chosen at compile time. Our approach, in contrast, takes the number of tracks as an input and can work on an unspecified number of trains. Verifying safety properties of the system in Puchol's approach is extremely complex: this complexity is such that one cannot be sure if the verification process itself is trustworthy. In our approach, in contrast, safety properties as well as other properties can be verified elegantly by posing simple queries.

UPPAAL has been also proposed as a toolbox for verification of real-time systems [25]. In fact, a model for a Train-Gate example is distributed with UPPAAL. UPPAAL is based on a timed automata formalism and cannot handle PTAs. Significant rewriting of the system would be required to support them. In contrast, PTAs are realized much more simply in our logic programming based approach.

Thus, a combination of constraint over reals, coinduction, and the language processing capabilities of logic programming provides an elegant, expressive, and easy-to-use formalism for modeling and analyzing complex real-time systems.

Acknowledgment

The authors would like to thank Feliks Kluzniak for proofreading the paper and for discussions.

References

[1] R. Alur and D. L. Dill, "Automata for modeling real-time systems," in *Proceedings of the seventeenth international colloquium on Automata, languages and programming*. New York, NY, USA: Springer-Verlag New York, Inc., 1990, pp. 322–335.

[2] ——, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.

[3] W. Thomas, "Automata on infinite objects," in *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, 1990, pp. 133–192.

[4] Z. Dang, O. H. Ibarra, T. Bultan, R. A. Kemmerer, and J. Su, "Binary reachability analysis of discrete pushdown timed automata," in *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 2000, pp. 69–84.

[5] N. Saeedloei and G. Gupta, "Timed grammars," forthcoming.

[6] G. Gupta and E. Pontelli, "A constraint-based approach for specification and verification of real-time systems," in *IEEE Real-Time Systems Symposium*, 1997, pp. 230–239.

[7] L. Simon, "Coinductive logic programming," Ph.D. dissertation, University of Texas at Dallas, Richardson, Texas, 2006.

[8] L. Simon, A. Mallya, A. Bansal, and G. Gupta, "Coinductive logic programming," in *ICLP*, 2006, pp. 330–345.

[9] C. L. Heitmeyer and N. A. Lynch, "The generalized railroad crossing: A case study in formal verification of real-time systems," in *IEEE Real-Time Systems Symposium*, 1994, pp. 120–131.

[10] Z. Dang, "Binary reachability analysis of pushdown timed automata with dense clocks," in *In CAV01, volume 2102 of LNCS*. Springer, pp. 506–517.

[11] R. Alur and T. A. Henzinger, "Logics and models of real time: A survey," 1992.

[12] J. Jaffar and M. J. Maher, "Constraint logic programming: A survey," *J. Log. Program.*, vol. 19/20, pp. 503–581, 1994.

[13] G. Gupta, A. Bansal, R. Min, L. Simon, and A. Mallya, "Coinductive logic programming and its applications," in *ICLP*, 2007, pp. 27–44.

[14] D. L. Dill, "Timing assumptions and verification of finite-state concurrent systems," in *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*. London, UK: Springer-Verlag, 1990, pp. 197–212.

[15] J. Ostroff, *Temporal Logic of Real-time Systems*. John Wiley and Sons Inc, 1990.

[16] T. A. Henzinger, Z. Manna, and A. Pnueli, "Temporal proof methodologies for real-time systems," Stanford, CA, USA, Tech. Rep., 1991.

[17] J. Jaffar and J.-L. Lassez, "Constraint logic programming," in *POPL*, 1987, pp. 111–119.

[18] J. Jaffar and M. J. Maher, "Constraint logic programming: A survey," *J. Log. Program.*, vol. 19/20, pp. 503–581, 1994.

[19] J. Barwise and L. Moss, *Vicious circles: on the mathematics of non-wellfounded phenomena*. Stanford, CA, USA: Center for the Study of Language and Information, 1996.

[20] J. W. Lloyd, *Foundations of logic programming / J.W. Lloyd*, 2nd ed. Springer-Verlag, Berlin ; New York :, 1987.

[21] Z. Dang, T. Bultan, O. H. Ibarra, and R. A. Kemmerer, "Past pushdown timed automata and safety verification," *Theor. Comput. Sci.*, vol. 313, no. 1, pp. 57–71, 2004.

[22] J. Jaffar, A. E. Santosa, and R. Voicu, "A CLP proof method for timed automata," in *RTSS*, 2004, pp. 175–186.

[23] L. Sterling and E. Shapiro, *The art of Prolog (2nd ed.): advanced programming techniques*. Cambridge, MA, USA: MIT Press, 1994.

[24] C. Puchol, "A solution to the generalized railroad crossing problem in Esterel," Austin, TX, USA, Tech. Rep., 1995.

[25] G. Behrmann, D. A, and P. K. Larsen, "A tutorial on Uppaal," in *LNCS, Formal Methods for the Design of Real-Time Systems (revised lectures)*, M. Bernardo and F. Corradini, Eds., vol. 3185. Springer Verlag, 2004, pp. 200–237. [Online]. Available: http://doc.utwente.nl/51010/