A Constraint-based Approach for Specification and Verification of Real-time Systems*

Gopal Gupta & Enrico Pontelli
Laboratory for Logic, Databases, and Advanced Programming
Department Computer Science
New Mexico State University
http://www.cs.nmsu.edu/lldap

Abstract

We develop a general constraint logic programming (CLP) based framework for specification and verification of real-time systems. Our framework is based on the notion of timed automata that have traditionally been used for specifying real-time systems. In our framework, a user models the ordering of real-time events as the grammar of a language accepted by a timed automata, the real-time constraints on these events are then captured as denotations of the grammar productions specified by the user. The grammar can be specified as a Definite Clause Grammar (DCG), while the denotations can be specified in constraint logic. The resulting specification can hence be regarded as a constraint logic program (CLP), and is executable. Many interesting properties of the real-time system can be verified by posing appropriate queries to this CLP program. A major advantage of our approach is that it is constructive in nature, i.e., it can be used for computing the conditions under which a property will hold for a given real-time system. Our framework also suggests new types of formalisms that we call Constraint Automata and Timed Push-down Automata.

1 Introduction

There has been tremendous amount of research in specification of real-time system [6, 12, 7]however, the process of designing and verifying the correctness of a real-time system is still not completely automated. Verifying the implementation of a real-time system manually can be error-prone. In this paper we present a framework for specifying and automatically verifying real-time systems. Our approach is based on the Constraint Logic Programming paradigm and the Timed Automata formalism. Timed automata have been devised for specifying real-time systems [2]. A timed automata is an ω-automata extended with the notion of time. ω-automata are automata that accept languages with infinite words. A timed-automata considers execution in a real-time system to be an infinite sequence of events. Transitions in a timed-automata are made not only on alphabet symbols in the language but also based on time-constraints. Timed automata are adequate for specifying real-time systems [2, 3]. Research has also been done on verifying timed automata, though the approaches adopted are ad-hoc, in our opinion.

Our approach, in contrast, provides a uniform framework for both specifying and verifying a real-time system. Our approach is also based on the concept of timed-automata, the main difference of our approach is that we use *constraint logic programming as a vehicle to realize this timed automata*.

An automata essentially accepts a language. In our framework, the specification of a real-time system consists of two parts: a grammar for the language to be accepted by the automata, and, a function for each production in the grammar that captures the constraints imposed on the transition that corresponds to that production. This function is a mapping from a parse tree to the *meaning* of the timed automata. One possible meaning of a timed automata can be given by an ordered sequence of $\langle e, \tau_e \rangle$ pairs where e is an event and τ_e is the time at which the event occurred in the real-time system. In general, any semantics (*trace semantics*, *interleaving semantics*, *fictitious clock semantics*, etc. [6]) can be chosen. Our framework is along the lines of denotational semantics approach [23] for design of programming languages, where syntax and semantics are cleanly separated, and meaning of language constructs are given in terms of their components [22]. So it is up to the person specifying the real-time system to choose the semantics he/she wishes. Also, while the formalism we choose for

*The authors are supported by NSF grants CCR 9625358, HRD 9628450, INT 9515256, and by NATO Grant CRG 921318.

specifying syntax (context free grammar) is more powerful than what is required for a regular languages, its advantage is that it can also be used for modeling *Pushdown Timed Automata* (and also perhaps *Timed Turing Machines*).

If the grammar is specified as a context free grammar then a parser for it is obtained immediately by using a Definite Clause Grammar [25] facility of a logic programming system. Likewise, if the functions corresponding to each production are specified in constraint logic [15], then an executable specification of the timed automata is immediately obtained. The executable specification thus obtained takes a stream of events (events are alphabet of the language), parses it producing a parse tree, then passes the parse tree as input to the function that captures the meaning of the automata. This function produces as output a list of pairs consisting of events and their timings.

A major advantage of our approach is that the resulting program can be used for verifying interesting properties of the real-time system. For example, given a specification of the real-time system for a train crossing, our system can *automatically* verify that when the train is at the crossing the gate cannot be open. The properties that a real-time system should satisfy can be expressed as constraints over the meaning (the list of $\langle e, \tau_e \rangle$ pairs) of the timed-automata, and thus can be naturally expressed in the framework of constraints. Thus, queries for verifying if a certain property is satisfied can be formulated as constraints, and posed to the executable specification (a constraint logic program). If the property is satisfied, an answer will be found for the query, else the query will fail. It should be noted that using our formal approach we were able to spot an error in one of the examples used by Alur and Dill [3]. This demonstrates the power of our approach. (It should also be noted though that later versions of this paper had the error fixed [5]).

The properties to be verified can be thought of as post-conditions while any restrictions on the input event list can be thought of as pre-conditions of the axiomatic semantics frameworks [17]. Thus, our formal framework is consistent with the view that *program assertions are theorems over denotational semantics of the language in which the programs are written.* [22].

Our research also suggests a much more general notion of automata, namely, the *Constraint Automata*. A constraint automata has constraints attached to its edges. It can be shown that timed automata is an instance of the much more general constraint automata. We demonstrate this through an example by capturing properties of both time and distance in our constraint automata. We assume familiarity with logic programming [25].

Our work incorporates a number of novel ideas. We propose a constraint-based framework in which timed-automata based specifications of real-time systems are automatically executable. Our constraint-based approach gives a computational interpretation for timed automata. Thus, many interesting properties can be automatically verified quite easily. Our research also suggests a generalization of timed automata to constraint automata. Our approach also provides a computational interpretation for more powerful abstractions for real-time systems such as *timed pushdown automata* and *timed Turing machines* that can also be modeled as constraint logic programs.

2 Timed Automata

Various extensions of finite state automata have been proposed for embedding the notion of *time* and *time constraints*, for example [10, 2, 21, 16]. A *timed automata* is a generalization of an automata capable of recognizing infinite words (also known as ω-*automata* [24]). A ω-automata 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 $\Delta \subseteq Q \times \Sigma \mapsto Q$ is the transition function. 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. *Büchi* automata 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 $(\forall i \in I)(q = q_i)$. *Müller* automata defines F to be a subset of 2^Q, and a derivation for $\bar{s}$ leads to the acceptance of the string iff there exists $A \in F$ and infinite sets 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 automata is to recognize *timed words*. A timed word $(\bar{s}, \bar{t})$ associates to each symbol s_i of an infinite word $\bar{s}$ a time value t_i. A timed automata is obtained from a ω-automata by adding: (i) a finite set $\mathcal{C}$ of *clocks*; (ii) a set $\mathcal{P}$ of propositions over $\mathcal{C}$; (iii) a labeling function $\tau_C : \Delta \mapsto 2^{\mathcal{C}}$ (*reset* function); (iv) a labeling function $\tau_P : \Delta \mapsto \mathit{prop}(\mathcal{P})$, where $\mathit{prop}(A)$ is the set of propositional formulae over the set of atomic propositions A. Thus, in a timed automata, each transition (p, a, q, C, γ) not only consumes a symbol a from the string, but additionally

- resets all the clock in C $((\forall c \in C)(c := 0))$;
- verifies that the formula γ 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 automata is described as a sequence of transitions between states. A state for a timed automata 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 function ν_{i+1} is defined as follows:

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

3. the proposition $\tau_P(q_i, s_i, q_{i+1})$ is satisfied by the clock valuation $\{\nu_i(x) + t_i - t_{i-1} : x \in \mathcal{C}\}$.

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 shows a simple timed automata. It describes a system in which signals are recognized. Each signal a can be either by itself or followed by a b signal, with the constraint that the b signal must arrive at least one time unit after a and at worst two time units after a. As a Büchi automata, the final state is $F = \{q_0\}$.

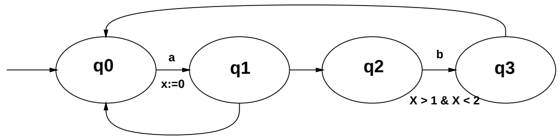

Figure 1. Timed Automata

In the rest of this work we will limit our considerations to *deterministic timed automata* with Müller termination. A deterministic automata 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 follows: 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.

Furthermore, it has been shown that the problem of verifying whether the language of a deterministic timed automata is included in that of another timed automata is decidable [3].

Example 2.1 *Let us consider the following problem. An automatic controller is designed to handle a gate at a railroad crossing. The system is composed of three entities, a gate-controller, the gate itself, and the train.*
The train is modeled by the automata shown in figure 2(i). It recognizes four different events: (i) `approach` *denoting approach of the train to the gate; (ii)* `in` *denoting that the train is at the gate; (iii)* `out` *denoting that the train has just left the gate; and, (iv)* `exit` *denoting that the train has left the gate area.*
The controller is modeled by the automata in figure 2(ii); it recognizes the approach *and* exit *events described above, together with the two events: (i)* `lower` *denoting starting of the lowering of the gate; and, (ii)* `raise` *denoting starting of the raising of the gate.*
Finally, the gate is modeled by the automata in figure 2(iii). Timing constraints force the train to employ at most 5 units of time to cross the gate area, with a speed which should allow 2 units of time to the gate to lower before the arrival of the train. Analogously, the controller should be able to react in less than one unit of time to the approach of the train, and the gate should employ no more than a unit of time to completely lower or raise the gate.

3 Constraint Logic Programming

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. A more precise and complete presentation of CLP can be found elsewhere [19, 18].

A CLP language is built upon two collections of symbols, Σ, containing all the function symbols, and Π, containing the predicate symbols. Furthermore, predicate symbols are classified in 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 object 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

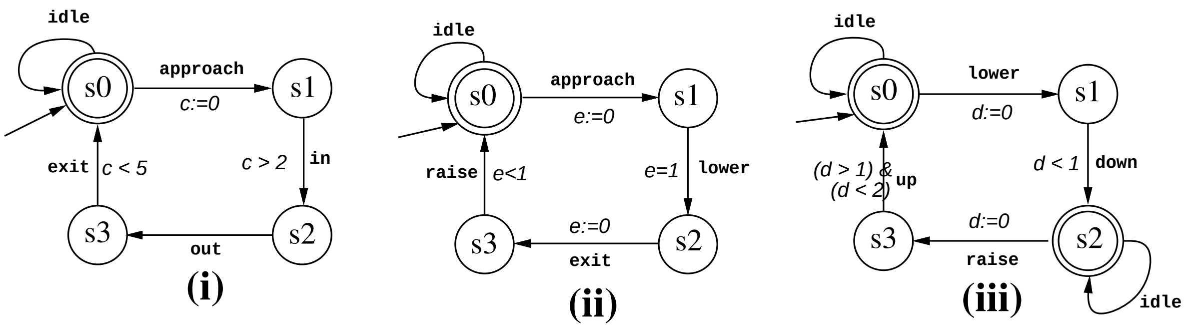

Figure 2. Train, Controller, and Gate automata

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 solving the constraints, but also of simplifying them (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.).

[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.

4 A CLP Framework for Real-time Systems

In this section we present our constraint logic programming-based framework for specifying real-time systems. Our framework captures the syntax and semantics of real-time automata as a constraint logic program. The resulting constraint logic program can be directly executed. Additionally, it can be used for verifying interesting properties (e.g., safety, liveness, etc. [3]).

Our approach is inspired by the *denotational* method of specifying programming language semantics [22]. In the denotational approach a language is specified by giving both its syntax and semantics formally. The syntax is specified as a context free grammar, while the semantics is given via *valuation functions*[22]. A valuation function is given for each production in the grammar. Each valuation function takes as arguments (i) the parse tree produced by that production, and (ii) the global *state* of the computation (the *memory store* for programming languages), and returns a new store as its value. The valuation functions thus describe the meaning of each grammar production, in terms of their effect on the store.

A timed automata is a recognizer for a certain language. It also describes the behavior of the real-time system for a given string in that language. Thus, a timed automata describes the syntax and semantics of a language. Following, the denotational approach, in our framework the syntax of the language is described as a context free grammar. The behavior of the timed automata is captured as *valuation functions* (the global state now is *time*). Since our approach is based on Constraint Logic Programming, the grammar is expressed as a Definite Clause Grammar [25], and the valuation functions are expressed as Horn clauses with constraints. The valuation functions are a mapping from the stream of events in the real-time system and global time to the list of $\langle e, \tau_e \rangle$ pairs. Both the syntax and semantic specification are immediately executable.

Example: The DCG and the semantic functions for the timed automata for the train (Figure 2) are shown below.

The parse tree is constructed during the parsing operation.

```
SYNTAX:
s0(s0(approach,S1)) --> approach, s1(S1).
s0(s0(idle,S0) --> idle, s0(S0).
s0(s0(idle)) --> idle.
s1(s1(in,S2)) --> in, s2(S2).
s2(s2(out,S3)) --> out, s3(S3).
s3(s3(exit,S0)) --> exit,  s0(S0).
s3(s3(exit)) --> exit.

SEMANTICS:
pS0(s0(a,S1), T1, T2, Out) :-  T3 > T1,
      Out = [(a, T3) | Out1],
      pS1(S1, T3, T3, Out1).
pS0(s0(idle,S0), T1, T2, Out) :-  T3 > T1,
      Out = [(idle, T3) | Out1],
      pS0(S0, T3, T2, Out1).
pS0(s0(idle), T1, T2, Out) :-  T3 > T1,
      Out = [(idle, T3)|Out1], Out1 = [].
pS1(s1(in,S2), T1, T2, Out) :- T3 > T1,
      T3 - T2 > 2, Out = [(in,T3)|Out1],
      pS2(S2, T3, T2, Out1).
pS2(s2(out,S3), T1, T2, Out) :- T3 > T1,
      Out = [(out, T3) | Out1],
      pS3(S3, T3, T2, Out1).
pS3(s3(exit,S0), T1, T2, Out) :- T3 > T1,
      T3 - T2 > 5, Out = [(exit,T3)|Out1],
       pS0(S0, T3, T2, Out1).
pS3(s3(exit,S0), T1, T2, Out) :- T3 > T1,
      T3 - T2 < 5,
      Out = [(exit,T3)|Out1], Out1 = [].
```

As mentioned earlier, the meaning of the automata is given as a list of pairs (e, τ_e) where e is an event, and τ_e is the time it occurred. The meaning is output in the last argument of the various predicates. The first three arguments of the above semantic predicates are, respectively, the parse tree produced during the parsing phase, the time the previous event took place, and the last time the clock was reset (the last two arguments describe the global state). Because CLP languages are single assignment languages, two arguments need to be passed as part of the input state for every local clock used by the timed-automata. The timing relations between various events are expressed as constraints over time.

The first rule, for instance, states that when the approach event occurs (approach signal is raised), state S1 is made the current state and the clock is reset to zero (thus, T3 is passed as both the 2nd and the 3rd argument in the call to `pS1`)[2]. The remaining rules have similar interpretations. Note that the above constitutes an executable specification of the timed automata for the train in Figure 2.

The above scheme requires that the syntax be specified as a context free grammar, when in fact regular expressions would be powerful enough. The reason we chose CFGs is because we want to stay close to the denotational semantics approach. Also, one could generalize timed automata to timed pushdown automata, in which case our framework will still work (one can conceive of real-time systems that cannot be specified as a timed automata, for example, a real-time system in which we have to make sure that the number of occurrences of an arbitrary event A is identical to those of event B).

One can, however, take advantage of the fact that in the case of timed automata, the language being recognized is regular. Thus, the rules in the transition table can be directly encoded as clauses in constraint logic (in other words, syntax checking and semantic interpretation are fused together). In fact, that is exactly the approach we adopt from this point onwards. Accordingly, the behavior of the timed-automata for the train can be captured by the following constraint rules:

```
train(s0,approach,s1,T1,T2,T3) :- T3 = T1.
train(s1,in,s2,T1,T2,T2) :- T1 - T2 > 2.
train(s2,out,s3,T1,T2,T2).
train(s3,exit,s0,T1,T2,T2) :- T1 - T2 < 5.
train(X,lower,X,T1,T2,T2).
train(X,down,X,T1,T2,T2).
train(X,raise,X,T1,T2,T2).
train(X,up,X,T1,T2,T2).
```

The first argument of the heads of the clauses is the current state, the second argument is the event which triggers the transition, while the third argument is the new state the transition results in. The last three arguments are clock-times. T1 is the current time (wall clock time), while T2 the last time the clock was reset. T3 is an output argument which will be set to T1 if the clock is reset in this transition, and to T2 if the clock is not reset. If there were more than 1 clocks, then two arguments will be introduced for each clock. In the example above, T2 and T3 are used to represent the single clock used by the timed automata for the train.

The rules can be understood as follows: when the approach event occurs in state s0, a transition is made to state s1, and the clock is reset. The reseting of the clock is denoted by setting the output argument T3 to T1 (the current time). In state s1, when event `in` takes place, a transition can be made only if at least 2 units of time have elapsed since the resetting of the clock. The last time (on the global clock) the local clock was reset is contained in argument T2, thus, T1 - T2 must exceed 2. Since the clock is not reset in this transition, the value of T2 is copied into T3. The other rules can be similarly understood. The last 4 rules essentially are no-ops, in that the train automata doesn't change state on seeing `lower`, `down`, `up`, or `raise` event. It should be noted that the relationship between times of

[2]Note that in logic programming, variable names begin with an upper case letter, while constants begin with a lower case letter.

occurrence of different events is being expressed as constraints.

Also, in general, if a timed automata has n local clocks, then there will be one rule for each transition from every state. This rule will be of the form:

p(S_i, W, S_x, T_g, T_{c_1}, $T1_{c_1}$, ..., T_{c_n}, $T1_{c_n}$) :- $\mathcal{C}$.

where S_i is the current state, S_n the state reached on seeing the symbol W, T_{c_i} is the last time the ith clock was reset, $T1_{c_i}$ is the output value corresponding to the ith clock (thus, it is set to T_{c_i} if the ith clock is not set in this transition, and to T_g if it is). The constraints C represent the timing constraints between the two states S_i and S_x.

There may be multiple automata involved in the real-time system, and all of them can be encoded following the above technique. Once the transition rules have been coded, a driver program needs to be supplied. We assume that the events stream is available as a list. The driver takes the following as input: (i) the list of events (possibly infinite), (i) current global time, (ii) current time of all the local clocks, (iii) current state in each of the automata. The driver produces an output consisting of the meaning of the real-time system. This meaning is represented as a list of pair, where a pair is produced for each event and consists of the event name and the time at which that event took place. For each event in the input event stream, the driver calls the transition rule for each automata computing the new values of local clocks and the new states. It also adds the $\langle e, \tau_e \rangle$ pair to its output argument. Note that the driver can be generated automatically. Because the driver composes all the automata involved in the specification, and calls each one of them for every input event, the automata in the program should be defined so that they have a transition for every possible event (even though this transition is a no-op). For example, the gate automata's CLP definition below has to define such no-op transitions for the events `approach, in, out`, and `exit`.

Once the driver has been generated, we have an executable specification of the real-time system. The transition rules for the `gate` and `controller`, along with the driver are given below:

```
gate(s0,lower,s1,T1,T2,T1).
gate(s1,down,s2,T1,T2,T2) :-
                        T1 - T2 < 1.
gate(s2,raise,s3,T1,T2,T1).
gate(s3,up,s0,T1,T2,T2) :-
                        T1 - T2 > 1,
                        T1 - T2 < 2.
gate(X,approach,X,T1,T2,T2).
gate(X,in,X,T1,T2,T2).
gate(X,out,X,T1,T2,T2).
gate(X,exit,X,T1,T2,T2).

controller(s0,approach,s1,T1,T2,T1).
controller(s1,lower,s2,T1,T2,T2) :-
                        T1 - T2 = 1.
controller(s2,exit,s3,T1,T2,T1).
controller(s3,raise,s0,T1,T2,T2) :-
                        T1-T2 < 1.
controller(X,in,X,T1,T2,T2).
controller(X,out,X,T1,T2,T2).
controller(X,up,X,T1,T2,T2).
controller(X,down,X,T1,T2,T2).

driver([],S0,S1,S2,T,T0,T1,T2,[]).
driver([X|S],S0,S1,S2,T,T0,T1,T2,[(X,T)|R]):-

   train(S0, X, S00, T, T0, T00),
   gate(S1, X, S10, T, T1, T10) ,
   contr(S2, X, S20, T, T2, T20) , TA > T,
   driver(S,S00,S10,S20,TA,T00,T10,T20,R).
```

Note that if the event stream is non-terminating, then the driver will execute forever. The driver terminates when the event stream terminates.

5 Verification of Real-time Systems

Verification of real-time systems is quite important, as real-time systems are used in many critical situations (e.g. nuclear reactors, auto-pilots, etc.). We want to formally prove that a specification of a real-time system is correct. Normally, proof of correctness involves showing that an implementation is faithful to its specification. In case of real-time systems we are interested in proving the correctness of the specification. Thus, verification means checking to see that the specification satisfies certain properties. For example, we may want to verify that the specification for a real-time system for controlling a railroad crossing satisfies the following property: when the train is at the gate, the gate is always closed. These properties are specified by the designer, and ensure correctness of the real-time system specification. These properties can be thought of as assertions that a real-time system must satisfy, for it to be certified as working correctly.

In our framework, these assertions are stated as predicates over the *meaning* of the real-time system. Recall that the *meaning* of the real-time system is computed by our specification as a list of ordered pairs. If the predicate is true over the meaning of the real-time system, then the property is verified. We can also show that a property holds by taking its negation, and then showing that no matter what the input event stream is, the negated property is never true. We adopt the latter approach.

A problem that we face in writing assertions over the meaning of the real-time system is that the meaning can be an infinite list, since the input may be an infinite stream. This finite aspect has to be captured in the driver by building restrictions in the driver's code. Once the finite patterns in the input stream have been recognized, it becomes straigtforward to verify a property, as is shown later.

Alternatively, following the axiomatic semantics approach, we can put restrictions on the event list (pre-conditions), and then verify properties (post-conditions) subject to these pre-conditions. Thus, the pre-conditions can ensure that inputs event-lists are finite. In other words, the prevoiusly mentioned finite patterns in the infinite event list can be captured by specifying pre-conditions. Thus, our queries to the program will appear as follows:

```
pre_condition(X),
driver(X, ...),
post_condition(X)}
```

where post_condition is the verification condition, while pre_condition is the condition imposed on the input to ensure finiteness. The pre_condition should be such that it generates all sentences that satisfy the restitions. For example, if we want to check that the event in (train is in) never occurs before event down (gate is down), then the pre-condition is that the in or the down events must occur between two approach events. This pre-condition can be expressed so that it can act as a generator all possible strings that start with an approach and end in approach:

```
pre_condition(L) :-  gen(N, L).
gen(N, []).
gen(N, [X|T]) :- event(X),
    (X = approach ->
        (N = 0 -> gen(1, T);
                  gen(_, [])); gen(N, T)).
event(approach). event(exit). event(raise).
event(lower). event(up). event(down).
```

In this formulation pre_condition will also generate event lists that are not legal according to the automata definition (these will be rejected by the driver call), introducting inefficiency. We can make this program efficient by having the *driver* itself generate all possible legal strings that the automata accepts, i.e., we can *merge* the pre_condition into the driver. Thus, the driver can be rewritten as:

```
driver(_,[],_,_,_,_,_,_,_,[]) :-
driver(N,[X|S],S0,S1,S2,T,T0,T1,T2,
                              [(X,T)|R]) :-
     train(S0,X,S00,T,T0,T00),
     gate(S1,X,S10,T,T1,T10) ,
     contr(S2,X,S20,T,T2,T20) ,
     TA > T,
        (X = approach ->
          (N = 0 -> M = 1; Rest = []);
                    M = N),
     driver(M,S,S00,S10,S20,TA,
                         T00,T10,T20,R).
```

The driver thus acts as a generator, generating all possible strings that begin with *approach* and end in approach and that will be accepted by the automata. Now a property can be verified by calling the driver with uninstantiated input, and checking that the negated property does not hold for every possible meaning of the automata. Suppose, we want to verify that when the train is at the crossing, the gate must be down. This boils down to the following fact: in every possible meaning of a single run of the real-time system, the event down must occur before the event in. The negated property will be that the event in occurs before down. Thus, the query:

```
?- driver(0,X,s0,s0,s0,0,0,0,0,R),
          append(A, [(down,_)|_], R),
          append(_, [(in,_)|_], A).
```

will fail when run on a constraint logic programming system (we used the CLP(R) [19] system). The append program is the standard logic program for appending two lists.

Likewise, if we want to verify that the gate will be down at least 4 units of time. That is, an up cannot follow down within 4 time units, then we pose the following query:

```
?- driver(0,s0,s0,s0,0,0,0,0,X,R),
         append(A, [f(up,T2) |_], R),
         append(_, [f(down,T1)|_], A),
         T2 - T1 < 4.
```

and the query will fail.

It is interesting to note that using our system we were able to spot an error in the paper of Alur and Dill [3]. In the paper, the train automata of Figure 2 had the exit constraint erroneously labeled as c > 5. With this constraint changed, their real-time liveness property does not hold. The real-time liveness property states that the gate is never closed at a stretch for more than 10 units of time. In our system, this will be expressed as the query:

```
?- driver(0,s0,s0,s0,0,0,0,0,X,R),
         append(A, [f(up,T2) |_], R),
         append(_, [f(down,T1)|_], A),
         T2 - T1 > 10.
```

This indeed produces a solution (i.e., the gate can be closed for more than 10 units of time), if the exit constraint in the program is changed from T1 - T2 < 5 to T1 - T2 > 5. However, we did find that the error had been fixed in their subsequent paper [5].

Using our system one can also find out the minimum and the maximum amount of time the gate will be closed by posing the following query:

```
?- driver(0,s0,s0,s0,0,0,0,0,X,R),
         append(A, [f(up,T2) |_], R),
         append(_, [f(down,T1)|_], A),
         N < T2 - T1 < M, M > 0, N > 0.
```

We obtain the answer `M < 7, N > 1`. This tells us that the minimum time the gate will be down is 1 units, and the maximum time it will be down is 7 units. Other properties of the real-time system can similarly be tested.

The ability to compute values of unknowns is a distinct advantage of a logic programming based approach and is absent from all other approaches considered to-date.

6 Automatic Generation of the Driver

In the example above we have relied on the programmer to indicate the general boundaries of the input fragment to consider for verification. On one hand, one can argue that this is not more complex than developing a complete timed automata to express the property to be verified [3], as in both cases a thorough understanding of the problem and its properties is required. On the other hand, the restriction typically posed on the nature of the time constraints in the timed-automata frameworks [3, 10] are such that the existence of an ω-automata which accepts the same language as the timed automata is guaranteed. This is possible because the timing constraints imposed by transitions of a timed-automata can be captured as time-intervals which can then be encoded in the states of a ω-automata [5]. The restrictions imposed on the types of propositions allowed essentially ensure that such a translation is possible. In any timed automata the infinite sentences consist of a finite number of sentences of finite size occurring unbounded number of times. With the restrictions imposed on the propositions, we conjecture that it is possible to detect, a priori, these finite repeating patterns. This follows from the fact that the pre-condition we consider in the previous examples (that the event of interest occurs between two `approach` events) to ensure finiteness, can be inferred from the automata itself. Thus, this suggests the possibility of trying to capture the repetitive patterns in the automata *automatically*, allowing an appropriate driver for capturing a given property to be verified to be generated automatically as well. Work is in progress in this direction.

7 Constraint Automata

In our discussions so far we have only considered inequality constraints over time. We do not have to restrict ourselves to inequality constraints. We can consider arbitrary constraints, as long as the set of constraints accumulated is solvable. Also, we do not have to restrict ourselves to constraints over time, we can also include constraints over other physical quantities (e.g., distance in the case of train; see later example). This suggests that one can formulate a more general form of automata. We call this automata *Constraint Automata*. Timed automata can be regarded as an instance of constraint automata. Of course, we then need to understand the power of constrained automata, and figure out what type of properties are decidable for them.

Informally, a constraint automata is defined as an automata that recognizes a language consisting of a series of n-tuples. The edges of the automata are labeled with constraints involving elements of the tuples. Depending on the restrictions placed on the nature of these constraints, one can obtain constraint automata of varying degrees of power. A timed automata can be trivally seen to be an instance of a constraint automata in which the tuple consists of the event symbol and the remainder of the elments are local clocks.

It is well known that many interesting properties of timed automata are decidable if the constraints on clock are restricted to equality or inequality constraints over variables that range over time (i.e., only conditions of the form $x\ relop\ c$, are allowed where $relop \in \{<, >, <=, >=, =\}$ and c is a numerical constant; thus arithmetic operations are not allowed). If we place similar restrictions on constraints in a constraint automata, these properties will still be decidable. In fact, we believe that these properties will still be decidable if we allowed monotonic functions to appear in equality and inequality constraints (i.e., constraints are of the form $f(x)\ relop\ c, relop \in \{<, >, <=, >=, =, \}, c$ is a constant and f is a function monotonic on all its arguments).

Constraint automata thus allow us to reason about not just time but other physical quantities as well, such as distance. In other frameworks, e.g. that of Alur, Henzinger, and Ho [4], a switch has to be made from purely automata based theory (as in [3]) to more complicated types of temporal logics such as *integrator computation tree logic (ICTL)*.

We illustrate the constraint automata for a problem taken from [4]. Consider the railroad gate controller except that now the apart from time, the automata also include information about distance—see Figure 3. The CLP encoding of this automata is given below.

```
train(s0,approach,s1,X,Xn,T,Ti,To) :-
          To = T, Xn = -1000.
train(s1,in,s2,X,Xn,T,Ti,Ti) :-
          X < 0, Xn = 0,
          (T-Ti)*S = 1000,
          S = 50.
train(s2,exit,s0,X,Xn,T,Ti,Ti) :-
          X < 100, Xn = 100,
          (T-Ti)*S = 1100,
          S = 50.
train(X,lower,X,A,A,T,Ti,Ti).
train(X,raise,X,A,A,T,Ti,Ti).
train(X,up,X,A,A,T,Ti,Ti).
train(X,down,X,A,A,T,Ti,Ti).

gate(s0,raise,s0,Y,Y,T,Ti,Ti).
gate(s1,down,s2,Y,0,T,Ti,Ti) :-  Y > 0,
          (T - Ti)*0.5 = 2.5.
```

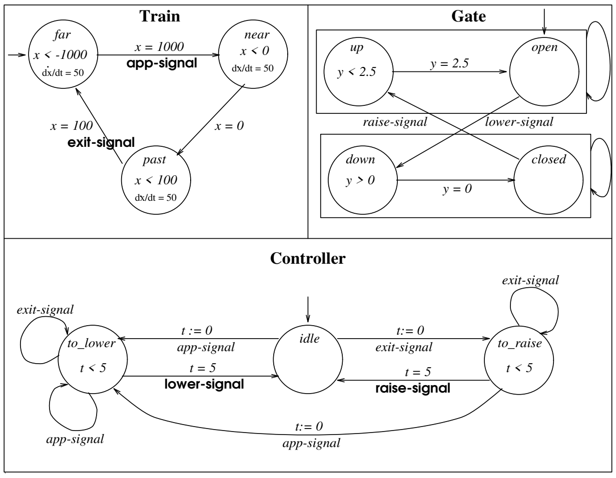

Figure 3. Another Railroad Example

```
gate(s1,lower,s1,Y,Y,T,Ti,Ti).
gate(s2,raise,s3,Y,Yn,T,Ti,To) :-
         Y=0, Yn > 0, To = T.
gate(s2,lower,s2,Y,Y,T,Ti,Ti).
gate(s3,up,s0,Y,2.5,T,Ti,Ti) :-  Y < 2.5,
         (T - Ti)*0.5 = 2.5.
gate(s3,raise,s3,Y,Y,T,Ti,Ti).
gate(X,approach,X,A,A,T,Ti,Ti).
gate(X,in,X,A,A,T,Ti,Ti).
gate(X,exit,X,A,A,T,Ti,Ti).

contr(s0,approach,s1,T,Ti,T).
contr(s1,lower,s0,T,Ti,Ti) :- T - Ti = 5.
contr(s1,approach,s1,T,Ti,Ti).
contr(s1,exit,s1,T,Ti,Ti).
contr(s0,exit,s2,T,Ti,T).
contr(s2,exit,s2,T,Ti,Ti).
contr(s2,raise,s0,T,Ti,Ti) :- T - Ti  = 5.
contr(s2,approach,s1,T,Ti,T).
contr(X,Y,X,T,Ti,Ti) :-
(Y=up; Y=in; Y=down).
```

This automata can be used for verifying many interesting properties. Suppose we want to verify that the minimum distance of 2000 meters should be kept between two trains so that an approach signal is not received while the gate is being raised (assuming trains are traveling at a maximum speed of 50 meters/sec.). The driver used for generating the finite patterns of the input stream for verifying this property is shown below.

```
driver(N,S0,S1,S2,A,B,T,T0,T1,T2,[X|Rest],
                  [f(X,T,A1,B1)|R]) :-
     train(S0,X,S00,A,A1,T,T0,T00),
     gate(S1,X,S10,B,B1,T,T1,T10) ,
     contr(S2,X,S20,T,T2,T20) ,
     TA > T,
     (X = approach ->
               (N = 0 -> M = 1; Rest = []);
                     M = N),
     driver(M,S00,S10,S20,A1,B1,TA,T00,
                          T10,T20,Rest,R).
```

Once the driver has been set up, we can pose the following query to check that indeed 2000 meters is a safe distance between two trains.

```
?- driver(0,s0,s0,s0,-
2000,2.5,0,0,0,0,X,R),
      append(A,[f(up,T1,A1,B1)|_],R),
      append(_,[f(approach,T2,A2,B2)|_],A),
      2000 > 50 * (T1-T2).
```

We can also find the minimum safe distance by posing the following query:

```
?- driver(0,s0,s0,s0,-
2000,2.5,0,0,0,0,X,R),
       append(A, [f(up,T1,A1,B1)|_], R),
       append(_,[f(approach,T2,A2,B2)|_],A),
       D = 50 * (T1-T2).
```

which computes the value of `D` to be 1600 meters. So the minimum distance between two trains should be 1600 meters.

8 More General Timed Languages

An obvious extension of timed automata is a timed push-down automata. Timed automata can only recognize timed regular languages. There are several situations where a regular expression is insufficient to express the property to be verified. For example, in the timed-automata of Figure 3, if we wanted to verify that in any arbitrary run, the number of `approach` events is equal to the number of `exit` events, then this cannot be expressed using timed-automata. However, if we augment the timed automata with a push down stack, then this property can be expressed. Our denotational formalism will not have any problem in expressing such properties, as it is based on context free grammars. One can further generalize a timed push down automata to a timed Turing machine. Timed pushdown automata and timed Turing machines are objects of our current investigations. An open question is the relationship between the hierarchy of timed languages recognized by these machines and the various kinds of temporal logics of varying power that have been used for specifying and verifying real-time systems.

9 Conclusions & Comparison with Other Work

There is a tremendous amount of work that has been done in verification of real-time systems [6]. Real-time systems have been studied with several types of formal semantics assigned to them. A host of Real-time Logics have also been proposed. There are *linear time temporal logics* and *branching time temporal logics*, temporal logics with *freeze quantification*, with *bounded temporal operators*, and with *explicit clock variables*. Some example temporal logics are MTL (metric temporal logic [6]), TPTL (timed temporal logic [6]), RTTL (real-time temporal logic [21], XCTL (Explicit Clock temporal logic [11]), RTCTL (real-time computation tree logic), etc. There are also automata based real-time formalisms such as *timed automata* [3, 20] and *timed transition systems*.

Few proposals have been made for modeling real time systems using a logic programming or a constraint logic programming framework. The most notable is that of Brzoska [7, 8], which provides a translation of simple temporal formulas (following the structure of Templog [1]) into a simple CLP scheme over an algebra based on the set of integer numbers with $-$ and $+$. BNR-Prolog [9], a constraint logic language on interval arithmetics, has also been used to capture temporal properties. Some preliminary work on constraint temporal logic has also been done by Früwirth [14]. Constraint automata have been suggested by Fribourg [13], though not in the context of real-time systems (Fribourg admits that he cannot handle real-time, a major weakness). In addition, Fribourg's approach requires checking automatically if one constraint logic program is subsumed by another—a very difficult problem. Our approach, in contrast, can handle real-time, and is also simpler in the sense that verification involves checking if a query will succeed or fail w.r.t a program.

Our work can be seen as providing a computational framework for timed automata based real-time system. We believe that a constraint logic based approach like ours can also be used for providing a computational framework for other formalisms such as various kinds of temporal logics. Also, various kinds of abstract semantics can also be easily realized in a constraint based framework.

The fact that our constraint based framework has a computational interpretation (SLD resolution augmented with constraint solving) adds some extra expressive power. Thus, it is possible in our framework to leave values unspecified and expect that the value will be computed based on the constraints imposed by the real-time system. This is a major strength of our approach. Thus, in our framework we can calculate, as in the example in the previous section, that the minimum safe distance between two trains is 1600 meters. In other approaches this is not possible, as we need to guess the value of a safe distance, and then verify that indeed that is the safe distance. Thus, our framework is more expressive than other timed automata based approaches. Our framework also has weaknesses, the most major one being that we need to identify the finite number of finite repeating patterns in the infinite strings accepted by the timed automata. At the moment this is captured by the programmer by writing the `driver` routine, but we would like to automate this process (note though that the driver that consumes infinite input can be automatically generated). Work is in progress in that direction.

References

[1] M. Abadi and Z. Manna. Temporal Logic Programming. *Journal of Sympbolic Computation*, 8, 1989.

[2] R. Alur and D. Dill. Automata for Modeling Real-time Systems. In *Automata, Languages, and Programming*. Springer Verlag, 1990.

[3] R. Alur and D. Dill. The Theory of Timed Automata. In *Real Time: Theory in Practice*. Springer Verlag, 1992.

[4] R. Alur, T.A. Henzinger. Automatic Symbolic Verification of Embedded Systems. In *Proc. Symposium on Real Time Systems*. IEEE Computer Society, 1993.

[5] R. Alur and D. Dill. The Theory of Timed Automata. *Theoretical Computer Science*, 126, 1994.

[6] R. Alur, T. Henzinger. Logics and Models of Real-time: a Survey. In *Real Time: Theory in Practice*. Springer, '92.

[7] C. Brzoska. Temporal Logic Programming and its Relation to Constraint Logic Programming. In *Proc. International Logic Programming Symposium*. MIT Press, 1991.

[8] C. Brzoska. Temporal Logic Programming with Metric and Past Operators. In *Executable Modal and Temporal Logics*. Springer Verlag, 1995.

[9] J. Cleary and V. Kaushik. Updates in Temporal Logic Programming Languages. Tech. Rep., Univ. of Calgary, 1991.

[10] D. Dill. Timing Assumptions and Verifications of Finite-state Systems. In *Automatic Verification Methods for Finite-state Systems*. Springer Verlag, 1989.

[11] A. P. E. Harel, O. Lichtenstein. Explicit-clock Temporal Logic. In *Proc. IEEE Symp. on Logics in Comp. Sci.*, 1990.

[12] M. Fisher and R. Owens. *Executable Modal and Temporal Logics*. Springer Verlag, 1995.

[13] L. Fribourg and M. V. Peixoto. Concurrent Constraint Automata. Tech. Rep., LIENS 93-10, 1993. 1995.

[14] T. Fruwirth. Temporal Logic & Annotated Logic Programming. In *Executable Modal & Temporal Logics*. Springer, '95.

[15] G. Gupta. Logical denotations: an executable denotational framework. Internal Report, NMSU, 1997.

[16] T.A. Henzinger, Z. Manna, A. Pnueli. Temporal Proof Methodologies for Real-time Systems. In *Proc. Symp. on Principles of Programming Languages*. ACM Press, 1991.

[17] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *CACM*, Vol. 12, pp. 576-580.

[18] J. Jaffar and M. Maher. Constraint Logic Programming. *Journal of Logic Programming*, 19/20, 1994.

[19] J. L. Lassez and J. Jaffar. Constraint logic programming. In *Proc. 14th ACM POPL*, 1987.

[20] N. Lynch, C. Heitmeyer. The Generalized Railroad Crossing: A Case Study. In *Proc., IEEE RTSS*, 1994.

[21] J. Ostroff. *Temporal Logic of Real-time Systems*. Research Studies Press, 1990.

[22] D. Schmidt. *Denotational Semantics: a Methodology for Language Development*. W.C. Brown Publishers, 1986.

[23] D. Scott and C. Strachey. *Towards a Mathematical Semantics for Computer Languages*. Tech. Monograph, Univ. of Oxford, 1971.

[24] W. Thomas. Automata on Infinite Objects. In *Handbook of Theoretical Computer Science*. Elsevier Science, 1990.

[25] L. Sterling & S. Shapiro. The Art of Prolog. MIT Press, '94.