Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 212–221
http://www.floc-conference.org/ICLP-home.html

TIMED DEFINITE CLAUSE ω-GRAMMARS

NEDA SAEEDLOEI [1] AND GOPAL GUPTA [1]

[1] Department of Computer Science
University of Texas at Dallas, Richardson, TX 75080, USA
E-mail address: {nxs048000,gupta@utdallas.edu}

Abstract. We propose timed context-free grammars (TCFGs) and show how parsers for such grammars can be developed using *definite clause grammars (DCGs)* coupled with *constraints over reals (CLP(R))*. Timed context-free grammars describe timed context-free languages (TCFLs). We next extend timed context-free grammars to timed context-free ω-grammars (ω-TCFGs for brevity) and incorporate *co-inductive logic programming* in DCGs to obtain parsers for them. Timed context-free ω-grammars describe timed context-free languages containing infinite-sized words, and are a generalization of timed ω-regular languages recognized by *timed automata*. We show a practical application of ω-TCFGs to the well-known *generalized railroad crossing problem*.

Introduction

Using timed automata is a popular approach to designing, specifying and verifying real-time systems [Alu90, Alu94]. Timed and hybrid automata provide the foundational basis for cyber-physical systems (CPS) that are currently receiving a lot of attention [Lee08, Gup06]. Timed automata are ω-automata [Tho90] extended with clocks (or stop-watches). Transitions from one state to another are made not only on the alphabet symbols of the language, but also on constraints imposed on clocks (e.g., at least 2 units of time must have elapsed). A timed automaton recognizes a sequence of timed words, where a timed word is made of symbols from the alphabet of the language the automaton accepts (a regular language), paired with a time-stamp indicating the time that symbol was seen. Since finite automata are equivalent to regular languages it seems natural to think of timed automata as being equivalent to timed regular languages. However, regular expressions are unsuitable for many complex (and useful) applications; in many situations one needs context-free languages. For real-time systems this means that timed regular languages may not be powerful enough, and one has to resort to TCFLs.

In this paper we propose timed grammars as a simple and natural method for describing timed languages. Timed grammars describe words that have real-time constraints placed on the times at which the words' symbols appear. Note that previous approaches to dealing with time typically have discretized time, which resulted in frameworks that cannot model problems faithfully. Lack of a framework that models real-time systems and other continuous physical quantities has been perceived as a problem by the research community [Lee08, Gup06].

Key words and phrases: Constraint Logic Programming over reals, Co-induction, Context-Free Grammars, ω-Grammars.

© N. Saeedloei and G. Gupta
Creative Commons Non-Commercial No Derivatives License

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.212

We extend the concept of context-free grammars to timed context-free grammars and timed context-free ω-grammars. Informally, a timed context-free grammar is obtained by associating clock constraints with terminal and non-terminal symbols of the productions of a CFG. The timed language accepted by a timed CFG contains those strings that are accepted by the underlying untimed grammar but which also satisfy the timing constraints imposed by the associated clock constraints. The language accepted by a timed ω-CFG (ω-TCFG) contains timed strings that are infinite in size. Such languages are useful for modeling complex CPS including real-time systems [Sae] that run forever.

In this paper we describe timed grammars, and show how DCGs together with CLP(R) and co-induction can be used to develop an effective and elegant method for parsing them. We also illustrate timed grammars through examples, in particular we show how the *controller* component of the *generalized railroad crossing problem* of Lynch and Heitmeyer [Hei94] can be specified naturally using timed grammars. Introducing ω-TCFGs and their logic programming realization is the main contribution of this paper. Note that pushdown timed automata (PTA) have been introduced in the past; however, to the best of our knowledge, ours is the first attempt to (i) develop the notion of timed grammars, and (ii) develop practical methods for parsing these timed grammars. We assume that the reader is familiar with *constraint logic programming* over reals *(CLP(R))* [Jaf94]. We next present an overview of *co-inductive logic programming (co-LP)*.

1. Co-inductive Logic Programming

The traditional declarative and operational semantics for logic programming (LP) is inadequate for various programming practices such as programming with infinite data structures and *corecursion* [Bar96]. Recently *co-induction* [Bar96] has been introduced into logic programming by Simon et al [Sim06a, Sim06b] to overcome this problem. Co-inductive LP can be used for reasoning about unfounded sets, behavioral properties of (interactive) programs, elegantly proving liveness properties in model checking, type inference in functional programming, representing and verifying properties of Kripke structures and ω-automata, etc. [Gup07, Sim07].

Co-induction 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 *co-inductive hypothesis rule* and systematically computes elements of the *gfp* of a program (w.r.t. a query) 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.

In the co-inductive 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* [Llo87], and *maximal models* (computed using *greatest fixed-points)* [Sim06a]. The operational semantics under co-induction is identical to Prolog's operational semantics, except for the following addition [Sim06a]: 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 *co-inductive 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).` succeeds in two resolution steps with the (infinite) answer: `A = [1 | A]` which is a finite representation of the infinite answer: `A = [1, 1, 1, ....]`. Now for a slightly more non-trivial example, consider the program `P1` below:

```
bit(0).                    stream([]).
bit(1).                    stream([H| T]) :- bit(H), stream(T).
```

A call to `?- stream(X).` in a standard LP system will systematically generate all finite bit streams one by one starting from the `[]` stream. Suppose now we remove the base case and obtain the program `P2`:

```
bit(0).
bit(1).                    stream([H| T]) :- bit(H), stream(T).
```

Under co-inductive semantics, posing the query `?- stream(X).` will produce infinite sized streams as answers, e.g., `X = [0, 0, 0 | X]`, `X = [1, 1, 1 | X]`, `X = [0, 1, 0 | X]`, etc.

2. Timed Context-free ω-Grammars

A timed grammar is a grammar extended with real-valued variables.[1] These variables model the clocks in the grammar. All clocks advance at the same rate (identical to the rate at which a wall clock advances). Clock constraints are used to restrict the kind of strings generated by the underlying untimed grammar. Clocks may be reset to zero when a particular symbol of the language is seen. Informally, timed grammars are collections of production rules in which clock expressions (clock constraints and resets) can appear after both the terminal and non-terminal symbols on the right hand side. The timed language accepted by a timed grammar consists of strings that can be derived from rules of the grammar and that satisfy the clock constraints appearing in that grammar. Since CFGs are suitable for describing a large class of complex applications, we consider timed CFGs only (though, in general, one could have timed versions of context-sensitive grammars, as well as timed Turing machines). We use a timed CFG to describe a timed language by generating each string of that language in a manner similar to a CFG. Informally, during the derivation, the right hand side of a rule can be substituted for a non-terminal symbol only if the timing constraints accompanying that non-terminal symbol are satisfied. Similarly, a terminal symbol cannot be generated if its accompanying clock constraint is violated. Timed context-free grammars extend CFGs with:

- A fixed number of *clocks*, which may be reset to zero. Clock names are global to all the production rules, i.e., all occurrences of a clock name c refer to the same clock.
- *Clock resets*, which are written within curly braces and can appear after a terminal or a non-terminal symbol on the right hand side of a production rule. Resetting the clock after a terminal symbol a, denoted $a\{c := 0\}$ where c is the clock, is used to remember the time at which a has been seen; while resetting the clock after a non-terminal symbol B, is used to remember the time at which the *last* terminal symbol in the string that is reduced to B has been seen.

[1] Grammars extended with real-valued variables can be used to model other cyber-physical phenomenon, however, in this paper our focus is on real-time systems.

- *Clock constraints*, which are put in the timed grammar in exactly the same manner as *clock resets*, i.e., within curly braces; however, they are used to indicate the timing constraints between the time stamps of the various symbols that appear in an accepted string. For example $a\{c < 2\}$ indicates that the symbol a must appear within two units of time since the clock c was reset.

Each *terminal* as well as *non-terminal* symbol appearing on the right hand side of a production rule maybe followed by curly braces that enclose a non-empty sequence of *clock resets* and *clock constraints*. Before we formally define timed grammars, let us consider some examples.

Example 2.1. Consider a language in which each sequence of a's is followed by a sequence of an equal number of b's, with each accepted string having at least two a's and two b's. For each pair of equinumerous sequences of a's and b's, the *first* symbol b must appear within 5 units of time from the *first* symbol a and the *final* symbol b must appear within 20 units of time from the *first* symbol a. The grammar annotated with clock expressions is shown below: c is a clock which is reset when the first symbol a is seen.

1. $S \to R\ S$
2. $R \to a\ \{c := 0\}\ T\ b\ \{c < 20\}$
3. $T \to a\ T\ b$
4. $T \to a\ b\ \{c < 5\}$

Note that, for example, in the first production ($S \to R\ S$), the sets of clock constraints associated with R and S are empty, and are therefore omitted. Note also that in the above grammar, the first rule is co-inductive (i.e., a recursive rule with no base case). Thus, this grammar is an ω-grammar.

Example 2.2. The following grammar describes a language in which sequences of a's are followed by a final symbol b, which must appear within 5 units of time from the *last* symbol a.

$S \to a\ \{c := 0\}\ S$
$S \to b\ \{c < 5\}$

Example 2.3. With a slight change in the timed grammar in the previous example we can capture a language in which sequences of a's are followed by a final symbol b that appears within 5 units of time from the *first* symbol a. The timed grammar for this timed language is as follows.

$S \to a\ \{c := 0\}\ R$
$R \to a\ R$
$R \to b\ \{c < 5\}$

Note the difference in how the clocks are reset in the last two examples. The clock c in Example 2.2 is reset on every occurrence of the symbol a; while in Example 2.3 the clock c is reset only on the first occurrence of symbol a.

Definition 2.4. A *timed context-free grammar* is a 6-tuple $G = \langle V, T, C, E, R, S \rangle$, where

- V is a finite set of non-terminal symbols;
- T is a finite set of terminal symbols, disjoint from V, which is the alphabet of the language defined by the grammar;
- C is a finite set of clock identifiers;

- E is a set of clock expressions over C (clock constraints and clock resets);
- R is a finite set of productions of the form $A \to (a\delta)^*$, where $A \in V$, $a \in (V \cup T)$, and δ denotes a (possibly empty) collection of clock expressions from E (* denotes Kleene closure);
- $S \in V$ is a special symbol called the *start* symbol.

The set E of clock expressions is limited to expressions of the form $\{c := 0\}$ and $\{c \sim x\}$, where c is a free variable, x is a constant, and $\sim \in \{=, <, \leq, >, \geq\}$.

Definition 2.5. A *timed context-free language (TCFL)* is a set of timed strings. A timed string is a sequence of pairs of the form $(a,\ t_a)$ where a is a symbol from the alphabet, and t_a is the time at which the symbol a was seen (by time we mean wall clock or physical time). If $(a,\ t_a)$ is immediately followed by $(b,\ t_b)$ in a timed string, then $t_b > t_a$, i.e., two or more symbols cannot appear at the same instant.

We now formally define the language generated by a timed context-free grammar $G = \langle V, T, C, E, R, S \rangle$. Note that because time flows linearly we only consider left to right *derivations*. Note also that because clocks may reset during a derivation, the reset times have to be recorded as part of the state. We could carry this state locally with each step of the derivation; however, to keep the exposition below simple we maintain this state as a global entity, which is accessed during each step of the derivation. For each clock c, $reset(c)$ denotes the wall clock time at which clock c was last reset. The wall clock is treated as a global variable whose value can be read at any time. We consider two cases, one where we reduce using a production that has a terminal symbol occurring in the leftmost position on its RHS, and the other where this production has a non-terminal symbol as the leftmost symbol in the RHS.

Case I: If $A \to a\sigma B$ is a production of R, where $a \in T$, σ is set of clock expressions (possibly empty) from E, and B is in $(V \cup T \cup E)^*$, then $A\gamma F$ *yields* $(a, t_a)B\gamma F$, written $A\gamma F \underset{G}{\Rightarrow} (a, t_a)B\gamma F$, where F is in $(V \cup T \cup E)^*$, γ is set of clock expressions (possibly empty) from E associated with non-terminal symbol A, and for each clock expression $k \in \sigma$:

- if $k = \{c := 0\}$, then $reset(c) = t_a$, i.e., we record that the clock c was reset at wall clock time t_a.
- if $k = \{c \sim x\}$, then $t_a - reset(c) \sim x$ must hold. If this clock constraint does not hold, then the derivation fails.

Case II: If $A \to D\sigma B$ is a production of R, where $D \in V, \sigma$ is set of clock expressions (possibly empty) from E, and B is in $(V \cup T \cup E)^*$, then $A\gamma F \underset{G}{\Rightarrow} D\sigma B\gamma F$, where F is in $(V \cup T \cup E)^*$, and γ is set of clock expressions (possibly empty) from E associated with non-terminal symbol A.

If two sets of clock expressions, γ and σ, appear next to each other during a derivation, they are replaced by $\gamma \cup \sigma$.

Definition 2.6. Suppose that $\alpha_1, \alpha_2, \ldots, \alpha_m$ are strings in $(V \cup T \cup E \cup (T \times \mathbf{R}^+))^*$, $m \geq 1$, and

$$\alpha_1 \underset{G}{\Rightarrow} \alpha_2,\ \alpha_2 \underset{G}{\Rightarrow} \alpha_3,\ \ldots,\ \alpha_{m-1} \underset{G}{\Rightarrow} \alpha_m.$$

Then we say that $\alpha_1 \overset{*}{\underset{G}{\Rightarrow}} \alpha_m$, or α_1 *derives* α_m in grammar G. Alternatively, $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ if β follows from α by application of zero or more productions of R. Note that $\alpha \overset{*}{\underset{G}{\Rightarrow}} \alpha$ for each string α. If it is clear which grammar G is involved, we use $\Rightarrow$ for $\underset{G}{\Rightarrow}$ and $\overset{*}{\Rightarrow}$ for $\overset{*}{\underset{G}{\Rightarrow}}$.

Definition 2.7. The language generated by G [denoted L(G)] is

$\{w \mid w = (w_1, t_1), (w_2, t_2), \ldots, (w_n, t_n)\ where\ w_i \in T, t_i \in \mathbf{R}^+, t_1 < t_2 \cdots < t_n, and\ S \overset{*}{\underset{G}{\Rightarrow}} w\}$.

That is, a timed string is in $L(G)$ if:

1. The timed string (timed word) consists of a sequence of pairs; the first element of each pair is a terminal symbol and the second is a real number.
2. The timed string can be derived from S and satisfies the time constraints imposed by the grammar.

We call L a *timed context-free language (TCFL)* for some timed CFG G, if $L = L(G)$.

As mentioned before, timed context-free ω-grammars (ω-TCFG) are CFGs with co-recursive grammar rules (i.e., recursive rules with no base cases). ω-TCFGs generate TCFLs with infinite sized words.

Adding clocks to a context-free grammar results in a grammar which is not context-free any more, rather it is context-sensitive. Intuitively, this is easy to see, as whether a non-terminal symbol can be reduced depends not only on if a matching production exists but also that the clock constraints associated with the non-terminal are consistent with the past clock resets. In other words a pushdown timed automaton would not be able to recognize a TCFG, because an extra component is needed to save the information about various clocks which are used in the TCFG. Thus, a TCFL cannot be recognized by a push down automaton. We conjecture that TCFGs are equivalent to *linear bounded automata* (LBA). A linear bounded automaton is a nondeterministic Turing machine which, instead of having potentially infinite tape for storage, is restricted to the portion of the tape containing the input x plus the two tape squares holding the end-markers.

Example 2.8. Consider the timed ω-grammar of Example 2.1. Consider a timed word

$$(a, 2), (a, 4), (b, 5), (b, 10), \ldots$$

Below we give the initial segment of the derivation of this timed word. The global state will record the time at which clock c is reset; c will be set to the value 2 when the first symbol a is seen.

$$\begin{aligned}
S &\Rightarrow R\ S \\
&\Rightarrow a\ \{c := 0\}\ T\ b\ \{c < 20\}\ S \\
&\Rightarrow (a, 2)\ T\ b\ \{c < 20\}\ S \\
&\Rightarrow (a, 2)\ a\ b\ \{c < 5\}\ b\ \{c < 20\}\ S \\
&\Rightarrow (a, 2)\ (a, 4)\ b\ \{c < 5\}\ b\ \{c < 20\}\ S \\
&\Rightarrow (a, 2)\ (a, 4)\ (b, 5)\ b\ \{c < 20\}\ S \\
&\Rightarrow (a, 2)\ (a, 4)\ (b, 5)\ (b, 10)\ S \ldots
\end{aligned}$$

3. Modeling ω-TCFGs with Co-inductive CLP(R)

To model and reason about ω-TCFGs we should be able to handle the fact that: (i) the underlying language is context-free, (ii) accepted strings are infinite, and (iii) clock constraints are posed over continuously flowing time. All three aspects can be elegantly handled within LP. It is well known that the definite clause grammar (DCG) facility of Prolog allows one to obtain parsers for context-free grammars (and even for context-sensitive grammars) with minimal effort. By extending LP with co-induction, one can develop language processors that recognize infinite strings. DCGs extended with co-induction can act as recognizers for ω-grammars. Further, incorporation of co-induction and CLP(R) into the DCG allows modeling of time aspects of ω-TCFGs. Once an ω-TCFG is modeled as a co-inductive CLP(R) program, it can be used to (i) check whether a particular timed string will be accepted or not; and, (ii) systematically generate all possible timed strings that can be accepted (note that a CLP(R) system will represent time-stamps of terminal symbols as variables in the output, with constraints imposed on them). The LP realization of the system based on co-induction and CLP(R) can also be used to verify system properties by posing appropriate queries.

We have developed a system that takes an ω-TCFG and converts it into a DCG augmented with co-induction and CLP(R). The resulting co-inductive constraint logic program acts as a parser for the ω-TCFL recognized by the ω-TCFG. The general method of this system is outlined next (the system is shown in `http://www.utdallas.edu/~nxs048000/tGrammar.yap`). The method takes an ω-TCFG as input and generates a parser as a collection of DCG rules (one rule per production in the ω-TCFG), where each rule is extended with clock expressions. For simplicity of presentation the method is explained for a timed grammar with one clock, but it can handle any number of clocks in a similar manner. We assume c is the clock; Ti, and To are used to remember the last wall clock time this clock was reset, and to pass on this clock's value to the next step in the derivation respectively. T, and Tn are used to represent the wall clock time, and the new wall clock time after each step in the derivation respectively. The method replaces every component of the timed grammar with its corresponding timed component as follows.

- A *non-terminal* symbol s is replaced with predicate $s(T, Ti, Tn, To)$;
- A *terminal* symbol a is replaced with $[(a, t_a)]$, where t_a is the wall clock time at which the symbol a appeared;
- A *clock constraint* of the form $\{c \sim x\}$, where $\sim \in \{=, <, \leq, >, \geq\}$ is replaced with $\{\{T - Ti \sim x\}\}$;
- A *clock reset* of the form $\{c := 0\}$ is replaced with $\{\{Ti = T\}\}$ or $\{\{To = T\}\}$ (depending on where it appears in the production).

Note that everything within curly braces in DCG's is treated as a standard Prolog code, i.e., curly braces would be simply dropped and the code within them would be executed. Constraint solving is done using the CLP(R) system (after dropping a pair of braces; since it is the convention in most CLP(R) systems to put constraints inside a pair of curly braces).

For each production in the timed grammar, the method replaces each component (starting with the left hand side) with its corresponding timed component as explained above, advances the clock if necessary, passes on the wall clock time and last reset time to the next component, and repeats this step until the end of production rule is reached. If T represents the wall clock time, then time is advanced by posting the constraint $T' > T$ where T' represents the current wall clock time. Advancing the clock is necessary since the

underlying assumption is that multiple symbols of the timed string are not seen at the same instant. if (a, t_1) is immediately followed by (b, t_2) in a timed string, then $t_2 > t_1$. Thus the wall clock should be advanced after each symbol in the timed string.

To illustrate the process, we describe the logic programming rendering of the timed ω-grammar shown in Example 2.1 in section 2.

```
:- coinductive(s/6).
s(T, Ti, Tn, To) --> r(T, Ti, T1, To1), {{T2 > T1}}, s(T2, To1, Tn, To).
r(T, Ti, Tn, To) --> [(a, T)],    {{Ti = T, T1 > T}}, t(T1, Ti, T2, To),
                     {{Tn > T2}}, [(b, Tn)],  {{Tn - To < 20}}.
t(T, Ti, Tn, To) --> [(a, T)],    {{T1 > T}}, t(T1, Ti, T2, To), {{Tn > T2}},
                     [(b, Tn)].
t(T, Ti, Tn, To) --> [(a, T)], {{Tn > T}}, [(b, Tn)], {{Tn - Ti < 5, To = Ti}}.
```

Note that the predicates s, r, and t are defined as DCG rules; therefore, two arguments (for the difference lists) will be added to each of them by a Prolog compiler. The first explicit argument of these predicates is the wall clock time; Tn is the new wall clock time after each predicate call. The pair of arguments, Ti and To, represent the clock c of the timed grammar. In fact, a pair of arguments have to be added for each clock that is used in the grammar (in the current example there is only one clock). The first argument of this pair is used to remember the last wall clock time this clock was reset, while the second one is used to pass on this clock's value to the next predicate call. Given this program one can pose queries to it to check if a timed string satisfies the timing constraints imposed in the timed grammar. Alternatively, one can generate possible legal timed strings. Finally, one can verify properties of this timed language (e.g., checking the simple property that all the a's are generated within 5 units of time, in any timed string that is accepted).

The co-inductive termination of predicate $s/6$ will depend only on the two arguments (for the difference lists) that are added by the Prolog compiler, i.e., the wall-clock time and other arguments will be ignored when checking if the $s/6$ predicate is cyclical. In the Co-LP system we have used for our work [2], one can declare the arguments w.r.t. which a predicate should behave co-inductively. Only those arguments will be employed by the system for determining co-inductive termination for that predicate. For truly co-inductive termination, the constraints induced in a given cycle in the grammar should also be taken into account: one must ensure that if a cycle `P` is part of an accepting string, then the constraints generated in one traversal of cycle `P` must be entailed by those generated in the next traversal of `P`. This is indeed the case for practical timed systems, where all clocks involved are reset in every accepting cycle. Due to this resetting, the same constraints are repeated in every such cycle and therefore co-inductive termination is justified w.r.t. constraints also.

Next we illustrate applications of our co-inductive CLP(R) realization of ω-TCFGs by using it for the *controller* component of the GRC problem with two tracks.

4. Timed ω-Grammar for GRC

Informally, the GRC problem [Hei94] describes a railroad crossing system with several tracks and an unspecified number of trains traveling through the tracks 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

[2] The interpreter for Co-LP that we have used is based on YAP, and can be found in `http://www.utdallas.edu/~nxs048000/co-lp.yap`

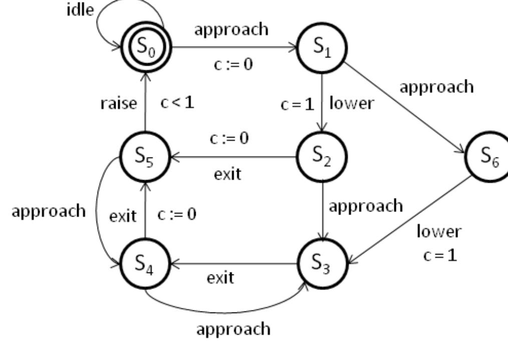

Figure 1: The controller for the GRC with two tracks

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 system is composed of three subsystems: a gate, a set of tracks and an overall controller. We show a timed context-free ω-grammar for the *controller* component of the GRC system with two tracks. The behavior of the controller can be expressed graphically via a timed push down automaton (Figure 1), and described using the ω-TCFG shown below:

$C \rightarrow approach\ \{c := 0\}\ L\ exit\ \{c := 0\}\ raise\ \{c < 1\}\ C$
$C \rightarrow approach\ \{c := 0\}\ L\ N\ exit\ \{c := 0\}\ raise\ \{c < 1\}\ C$
$L \rightarrow lower\ \{c < 1\}$
$L \rightarrow approach\ lower\ \{c < 1\}\ exit$
$N \rightarrow approach\ exit$
$N \rightarrow approach\ exit\ N$
$N \rightarrow exit\ approach$
$N \rightarrow exit\ approach\ N$

The logic programming rendering of this ω-TCFG is presented below.

```
c(T, Ti, Tn, To) --> [(approach, T)], {{Ti = T, T1 > T}},    l(T1, Ti, T2, To1),
                     {{T3 > T2}}, [(exit, T3)], {{To2 = T3, T4 > T3}},
                     [(raise, T4)], {{T4 - To2 <1, T5 > T4}}, c(T5, T5, Tn, To).
c(T, Ti, Tn, To) --> [(approach, T)], {{Ti = T, T1 > T}},    l(T1, Ti, T2, To1),
                     {{T3 > T2}}, n(T3, To1, T4, To2), {{T5 > T4}},
                     [(exit, T5)], {{To3 = T5, T6 > T5}}, [(raise, T6)],
                     {{T6 - To3 < 1, T7 > T6}}, c(T7, T7, Tn, To).
l(T, Ti, Tn, To) --> [(lower, T)],    {{T - Ti < 1, To = Ti, Tn = T}}.
l(T, Ti, Tn, To) --> [(approach, T)], {{T1 > T}}, [(lower, T1)],
                     {{T1 - Ti < 1, Tn > T1}},    [(exit, Tn)], {{To = Ti}}.
n(T, Ti, Tn, To) --> [(approach, T)], {{Tn > T}}, [(exit, Tn)], {{To = Ti}}.
n(T, Ti, Tn, To) --> [(approach, T)], {{T1 > T}}, [(exit, T1)], {{T2 > T1}},
                     n(T2, Ti, Tn, To).
n(T, Ti, Tn, To) --> [(exit, T)], {{Tn > T}}, [(approach, Tn)], {{To = Ti}}.
n(T, Ti, Tn, To) --> [(exit, T)], {{T1 > T}}, [(approach, T1)], {{T2 > T1}},
                     n(T2, Ti, Tn, To).
```

The ω-TCFGs for *gate* and *track* components of the GRC problem along with their corresponding logic programs can be generated in a similar manner.

5. Conclusions and Future Work

In this paper we have extended context-free grammars with clocks and clock expressions. The resulting grammars, called timed CFGs, are means of describing complex languages consisting of timed words, where a timed word is a symbol from the alphabet of the language the grammar generates, paired with the time that symbol was seen. As a result, timed CFGs are suitable for specifying systems whose behavior can be described by recursive structures with time constraints. We have developed a system for generating parsers for timed context-free ω-grammars using the DCG facility of logic programming coupled with CLP(R) and co-induction. We have applied our method of generating parsers for ω-TCFGs to the GRC problem with two tracks, and presented a simple timed context-free ω-grammar for the *controller* component of this problem.

To conclude, a combination of constraint over reals, co-induction, and the language processing capabilities of logic programming provides an elegant and expressive formalism for describing real-time, hybrid, and cyber-physical systems. In fact, our framework is a general framework that can be used for handling not only time but other continuous physical quantities as well.

Acknowledgment

The authors would like to thank D. T. Huynh, Kevin Hamlen, and Feliks Kluźniak for discussions.

References

[Alu90] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In *ICALP*, *Lecture Notes in Computer Science*, vol. 443, pp. 322–335. Springer, 1990.

[Alu94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[Bar96] Jon Barwise and Lawrence Moss. *Vicious circles: on the mathematics of non-wellfounded phenomena*. Center for the Study of Language and Information, Stanford, CA, USA, 1996.

[Gup06] Rajesh Gupta. Programming models and methods for spatiotemporal actions and reasoning in cyber-physical systems. In *NSF Workshop on CPS*. 2006.

[Gup07] Gopal Gupta, Ajay Bansal, Richard Min, Luke Simon, and Ajay Mallya. Coinductive logic programming and its applications. In *ICLP*, pp. 27–44. Springer, 2007.

[Hei94] Constance L. Heitmeyer and Nancy A. Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *IEEE RTSS*, pp. 120–131. 1994.

[Jaf94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.

[Lee08] Edward A. Lee. Cyber-physical systems: Design challenges. In *ISORC*. 2008.

[Llo87] J. W. Lloyd. *Foundations of logic programming / J.W. Lloyd*. Springer-Verlag, Berlin, New York, 2nd edn., 1987.

[Sae] Neda Saeedloei and Gopal Gupta. Logic programming foundations of cyber-physical systems. In Preparation.

[Sim06a] L. Simon. *Coinductive Logic Programming*. Ph.D. thesis, University of Texas at Dallas, Richardson, Texas, 2006.

[Sim06b] Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta. Coinductive logic programming. In *ICLP*, pp. 330–345. 2006.

[Sim07] Luke Simon, Ajay Bansal, Ajay Mallya, and Gopal Gupta. Co-logic programming: Extending logic programming with coinduction. In *ICALP*, pp. 472–483. 2007.

[Tho90] Wolfgang Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pp. 133–192. MIT Press, 1990.