In this paper, first, we introduce a temporal logic for specification of real-time systems. Our logic, called RTLTL, is an extension of linear temporal logic (LTL) with clocks and clock operations adopted from timed automata. RTLTL is a natural and simple language for specifying properties of real-time systems. Second, we develop a model checker for RTLTL based on logic programming. We show how the semantics of RTLTL is encoded into a logic program extended with constraints and coinduction. Given this coinductive constraint logic program, for a model checking question we simply ask its corresponding query in a logic programming system.

1 Introduction

Linear temporal logic has been used as a language for specifying properties of reactive systems and their behavior over time [18, 20, 19]. However, LTL (and other temporal logics with no explicit timing information) is interpreted over models that abstract away the actual times at which events occur; therefore, only temporal ordering information of events is retained.

For real-time systems specifications, capabilities for quantitative reasoning about delays and other timing requirements are necessary. Several attempts have been made to extend temporal logics (both linear and branching models) with time, e.g., by introducing bounded temporal operators [9, 15], or quantifier constructs for referencing time [3]. Other approaches include employing first-order temporal logics and assuming that one of the state variables represents time [1, 21]. These formalisms not only use different notations, but make different assumptions when modeling time. However, they are all discrete-time logics.

Model checking [6] is an automatic technique used for verifying hardware and software systems. It is done by constructing a model of the system as a finite-state Kripke structure and then determining whether the model satisfies various properties specified as temporal logic formulae. Model checking has been widely accepted and applied to prove correctness of many real-life systems.

A key issue in the design of a model-checking tool is the choice of the formal language with which properties of interest are specified and verified. It is well known that the language of choice should provide capabilities to specify and verify all $\omega$-regular properties. In addition, for model-checking real-time systems, capabilities for modeling time as well as reasoning over time are required.

Our objective in this paper is the development of (i) a real-time extension of LTL, called RTLTL, and (ii) a model checker for RTLTL based on Logic Programming (LP).

In our approach for extending LTL with real-time, we allow clocks and clock operations (clock resets and checking satisfaction of clock constraints) in the syntax of RTLTL. Clocks can be reset in the states and the relations between times of different states can be specified using clock constraints. In contrast to
other extensions, in our work the notion of time and clocks is adopted directly from the well-understood formalism of timed automata [2].

Logic programming [17, 28] has been successfully used for modeling and verification of real-time systems [29, 11], where executable Horn clauses are used directly to model behavior of systems related to time. Examples include modeling timed automata and hybrid automata as logic programs, where each transition of the automaton is presented as a LP rule extended with constraints [24], as well as translating the semantics of linear hybrid automata to constraint logic programs [5]. Logic programming has been also widely used as an implementation platform for interpreters [12], verification systems and model checkers [7, 31]. The latter typically involves translating the semantics of a given temporal logic, say $\mathcal{T}$, to a logic program, say $P_\mathcal{T}$. In this setting a model checking question can be formulated as a query which can be asked against $P_\mathcal{T}$.

We take the approach mentioned in the preceding paragraph and translate the semantics of RTLTL directly to a logic program. However, for handling global and release operators of the RTLTL which give rise to infinite computations, we resort to coinductive logic programming.

Coinductive logic programming (co-LP) [10, 27] has been proposed as a powerful technique for finitely reasoning about (rational) infinite structures and their properties. Recently, the authors have extended co-LP with constraints. The resulting formalism, coinductive constraint programming (co-CLP) [25], can be viewed as combination of coinductive logic programming and constraint logic programming [14]. Co-CLP is a programming language that is suitable for modeling infinite (rational) behaviors of infinite objects with constraints imposed on them.

We employ the principles of coinductive constraint logic programming [25] to propose a practical model checker for RTLTL. In our approach constraint logic programming is used to model timing constraints realized through clocks and clock operations, and coinduction is used to naturally model infinite computations realized through global and release temporal operators.

Next, we give an overview of coinductive constraint logic programming. More details can be found in [25].

## 2 Coinductive Constraint Logic Programming

Coinductive constraint logic programming [25] is a paradigm that combines constraint logic programming (CLP) [14] and coinductive logic programming [10, 27, 26].

Constraint logic programming is a natural and expressive paradigm which combines two declarative paradigms: logic programming [17] and constraint solving. CLP’s declarative semantics is defined in terms of a least fixed-point, i.e., it is inductive. Therefore, it cannot be directly used for reasoning about infinite objects and their properties; one needs to resort to greatest fixed-point semantics for that purpose.

Coinductive logic programming is an extension of logic programming which provides an elegant technique for finitely reasoning about infinite (rational) structures and their properties [10, 27, 26].

Coinductive LP does not handle constraints, while CLP’s declarative and operational semantics is inadequate for various programming techniques which involve infinite computations besides constraint solving. Such techniques have interesting applications for example in model checking and in verifying real-time systems, hybrid systems and cyber-physical systems [16, 13]. Coinductive constraint logic programming [25] is a paradigm that bridges this gap. It allows a special class of formulae (constraints) to be combined with traditional coinductive logic programming. We next present a brief overview of co-CLP.

Given a signature $\Sigma$, a $\Sigma$-structure $\mathcal{D}$ consists of a set $D$ and an assignment of functions and relations
on $D$ to the symbols of $Σ$ which respects the arities of the symbols. It is assumed that $Σ$ contains the binary predicate symbol $=•$, interpreted as identity in $D$, and a special binary relation $\equiv_0$ which is used for unification. $Σ^D$ is an extension of $Σ$ in which there is a constant for every element of $D$. Furthermore, $Σ^D$ is the extension of $Σ^D$ with the set of all functions and predicate symbols in program $P$. The set of predicate symbols defined by $Σ$ is denoted by $Π_Σ$ and the set of predicate symbols defined by program $P$ is denoted by $Π_P$. $Π_Σ$ and $Π_P$ are disjoint.

A term is a tree where every leaf node is labeled with a variable or with some $f/0$ (constant) and every inner node with $n$ children, $n > 0$, is labeled with some $f/n$. We write $f(t_1, \ldots, t_n)$ for the term with root $f$ and direct subtrees $t_1, \ldots, t_n$. A term $t$ is called finite if all paths in the tree $t$ are finite, otherwise it is infinite. A term (tree) is rational if it only contains finitely many different subterms (subtrees). Terms that are built upon variables and function symbols in $Σ^D$ are denoted by $T^D$.

An atom is a tree $p(t_1, \ldots, t_n)$, where $p \in Π_Σ$ and $t_1, \ldots, t_n$ are rational terms. A primitive constraint is an expression of the form $p(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms from $T^D$ and $p$ is a predicate symbol from $Π_Σ$. In other words, primitive constraints only contain functors from $Σ$. This ensures that the constraint solver will only deal with primitive constraints which it can solve. A constraint is a first-order formula built from primitive constraints.

A coinductive constraint logic program is composed of a collection of clauses, where each clause has the form: $a : - c, b_1, \ldots, b_n$, in which $a$, and $b_i, 1 \leq i \leq n$, are user-defined atoms, while $c$ is an arbitrary conjunction of constraints.

Consider the definition of a stream (list) of numbers given as program P1 below:

```
stream([H1, H2 | T]) :-
    number(H1), number(H2), \{H2 - H1 >= 3\}, stream(T).

number(0).
number(s(N)) :- number(N).
```

In standard CLP the query `?- stream(X).` fails, since the model of P1 does not contain any instances of `stream/1`. However, the query `?- stream(X).` under the co-CLP interpretation of P1 should produce all infinite (rational) streams such as $X = \{1, 5 | X\}$, $X = \{2, 6, 3, 7 | X\}$, etc., as answers. The model of P1 does contain instances of `stream/1` (but proofs may be of infinite length).

Co-CLP allows programmers to manipulate infinite (rational) structures in the presence of constraints. As a result, unification must be extended and the “occurs check” removed: unification equations such as $X = \{1 | X\}$ are allowed in co-CLP; in fact, such equations will be used to represent infinite (rational) structures in a finite manner.

From the semantic point of view, predicates and functions in $Σ$ and constraints are interpreted using the predefined interpretation, i.e., the domain of computation $𝒟$. In particular, a constraint $c$ is solvable if $𝒟 \models \exists c$, where $\models$ is the standard entailment relation and $\exists c$ is the existential closure of $c$. A solution $θ$ for $c$ is a mapping from the variables in $c$ to $𝒟$, such that $𝒟 \models cθ$. User-defined functions and predicates will be given the standard interpretation in the Herbrand universe and the Herbrand base $[17]$.

From the procedural point of view, execution of a coinductive constraint logic 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. The operational semantics of co-CLP also relies on the coinductive hypothesis rule and systematically computes elements of the

---

[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.
Model Checking Timed LTL: A Logic Programming-based Approach

greatest fixed point (gfp) of a program via backtracking. The coinductive hypothesis rule states that during execution, if the current resolvent \( R \) contains a call \( G' \) that unifies with an ancestor call \( G \) and the set of accumulated constraints are satisfied, then the call \( G' \) succeeds; the new resolvent is \( R' \theta \) where \( \theta = \text{mgu}(G, G') \) and \( R' \) is obtained by deleting \( G' \) from \( R \). If \(?-(c_1, g_1, \ldots, g_n) \) is a goal, and \( p: -c_2, b_1, \ldots, b_k \) is a clause in the program, then the resolvent of the goal w.r.t. the given clause is

\[ ?-(c_1, c_2, g_1 = u p, b_1, \ldots, b_k, g_2, \ldots, g_n) \]

as long as \( \mathcal{D} \models E(c_1 \land c_2) \) where \( E \) is the current system of equations (substitution) which includes unification of arguments of \( g_1 \) and \( p \). \( E(c_1 \land c_2) \) is the result of applying the substitution \( E \) on \( c_1 \land c_2 \). The constraint solver is used to test the validity of the condition on the constraints.

Regular constraint logic programming execution extended with the coinductive hypothesis rule is termed co-constraint logic programming (or co-CLP)[25]. The coinductive hypothesis rule works for only those infinite proofs that are regular (or rational), i.e., infinite behavior is obtained by a finite number of finite behaviors interleaved an infinite number of times. Even with the restriction to rational proofs, there are many applications of co-CLP. These include model checking, modeling \( \omega \)-automata, etc.

3 RTLTL

The real-time logic RTLTL extends LTL with real-valued variables, called clocks, and clock operations in exact same way as timed automata extends finite state automata with clocks and clock operations.

Given an infinite set \( C \) of clock variables \( (c, d, \ldots) \), the clock constraints \( \delta \) and clock resets \( \gamma \) are defined by

\[
\delta := (c \sim r) \land \delta \mid \epsilon \\
\gamma := (c := 0) \land \gamma \mid \epsilon
\]

where \( r \) is a constant in \( \mathbb{R}_{\geq 0} \), and \( \sim \in \{<, >, \leq, \geq\} \).

Assuming an infinite set \( P \) of propositions \( (p, q, \ldots) \), the formulas \( \phi \) of RTLTL are defined as follows:

\[
\phi := p \mid \delta p \mid \gamma \phi \mid \neg \phi \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2 \mid X \phi \mid \phi_1 U \phi_2 \mid \phi_1 R \phi_2
\]

A clock \( c \) can be reset (set to zero) in a state (we formally define states later in this section) when a particular proposition is true. At any instant, reading of a clock is equal to the time that has elapsed since the last time the clock was reset. Clock constraints indicate timing constraints between states. Empty clock constraints and resets, denoted by \( \epsilon \), can be simply dropped.

Note that our syntax allows a formula of the form \( \phi = \gamma \delta p \); however, we do not allow \( \delta \) and \( \gamma \) to share clocks. The reason for this is to prevent resetting a clock to interfere with checking the satisfiability of a clock constraint.

Additional logical operators: \( \rightarrow \) (implication), and \( \leftrightarrow \) (equivalence), along with formulae \( \text{true} \) and \( \text{false} \) are defined as usual. RTLTL uses temporal operators of LTL: the next operator \( X \), the until operator \( U \) and the release operator \( R \). Two derived quantifiers \( F \) and \( G \), which stand for “eventually, in a finite number of states”, and “globally, at every state”, respectively, can be defined in terms of \( U \) and \( R \): \( F \phi \equiv \text{true} U \phi \) and \( G \phi \equiv \text{false} R \phi \). For instance, the formula

\[ 2 \text{mgu} \text{ is a shorthand for “most general unifier”}. \]
expresses the response property that: whenever \( p \) happens, eventually \( q \) must happen within 2 units of time. Here the clock \( c \) is reset when \( p \) happens, i.e., \( c \) starts counting time from zero; when \( q \) eventually becomes true, the elapsed time on clock \( c \) must be less than 2.

3.1 Semantics of RTLTL

We have chosen a continuous strictly-monotonic real-time semantics for our logic. The semantics of RTLTL is given in terms of infinite runs of timed Kripke structures [30]. Timed Kripke structures are timed automata [2], in which states are augmented with propositions.

Definition 1 A timed Kripke structure is a tuple \( M = \langle S, S_0, L, \Gamma, E \rangle \), where

- \( S \) is the (finite) set of states;
- \( S_0 \subseteq S \) is the set of initial states;
- \( L : S \to 2^P \) is a function which assigns to each state the set of atomic propositions true in that state;
- \( \Gamma \) is a finite set of real-valued variables, called clocks;
- \( E \subseteq S \times S \times 2^\Gamma \times \phi(\Gamma) \) gives the set of transitions. An edge \( \langle s, s', \gamma, \delta \rangle \) represents a transition from state \( s \) to state \( s' \). The set \( \gamma \subseteq \Gamma \) gives the clocks to be reset with this transition. The set of clock constraints \( \delta \) over \( \Gamma \) is specified by \( \phi(\Gamma) \).

With abuse of notations, we have used \( \gamma \) both as a set of clocks to be reset (subset of \( \Gamma \)) and also as a clock reset operation in the syntax of the logic. An example timed Kripke structure is shown in Fig. 1.

Figure 1: A Timed Kripke Structure

\[ G((c := 0)p \rightarrow F(c < 2)q) \]

expresses the response property that: whenever \( p \) happens, eventually \( q \) must happen within 2 units of time. Here the clock \( c \) is reset when \( p \) happens, i.e., \( c \) starts counting time from zero; when \( q \) eventually becomes true, the elapsed time on clock \( c \) must be less than 2.

Definition 2 A state sequence \( s = s_0 s_1 s_2 \ldots \) is an infinite sequence of states. A time sequence \( \tau = \tau_0 \tau_1 \tau_2 \ldots \) is an infinite sequence of times \( \tau_i \in \mathbb{R}_{\geq 0}, i \geq 0 \), that satisfies:

1. Monotonicity: \( \tau_i \leq \tau_{i+1} \) for all \( i \geq 0 \), and
2. Progress: For all \( t \in \mathbb{R}_{\geq 0} \), there is some \( i \geq 0 \) such that \( \tau_i > t \).

A timed state sequence is a pair consisting of a state sequence \( s \) and a time sequence \( \tau \). The timed state sequence \( \rho^i \) can be obtained from \( \rho \), by deleting the first \( i \) elements.
RTLTL formulas assert about timed state sequences. For instance, the formula \( Xp \) asserts that the second state in a timed state sequence satisfies the proposition \( p \). The formula \( pUq \) asserts that there is a state satisfying the proposition \( q \) in a timed state sequence, and all states before this \( q \)-state satisfy the proposition \( p \). Similarly, the formula \( pRq \) asserts that if there is a state satisfying the proposition \( p \) in a timed state sequence, then, this \( p \)-state and all states before it satisfy the proposition \( q \).

**Definition 3** A clock interpretation \( \mathcal{I} \) for a set \( \Gamma \) of clocks is a mapping from \( \Gamma \) to \( \mathbb{R}_{\geq 0} \). It assigns a real value to each clock in \( \Gamma \).

A clock interpretation \( \mathcal{I} \) satisfies a clock constraint \( \delta \), denoted by \( \mathcal{I} \models \delta \), iff the expression obtained by applying \( \mathcal{I} \) to \( \delta \), \( \mathcal{I}(\delta) \), evaluates to true. For \( t \in \mathbb{R}_{\geq 0} \), \( \mathcal{I} + t \) denotes the clock interpretation which maps every clock \( c \) to the value \( \mathcal{I}(c) + t \). For \( \gamma \subseteq \Gamma \), \( [\gamma \rightarrow t] \mathcal{I} \) denotes the clock interpretation for \( \Gamma \) which assigns \( t \) to each \( c \in \gamma \), and agrees with \( \mathcal{I} \) over the rest of the clocks. We assume all clocks advance at the same rate.

A run \( r \), denoted by \((s, \mathcal{I})\), of a timed Kripke structure over a time sequence \( \tau \) is an infinite sequence of the form

\[
\cdots \% s_0, \mathcal{I}_0 \xrightarrow{\tau_1} s_1, \mathcal{I}_1 \xrightarrow{\tau_2} s_2, \mathcal{I}_2 \xrightarrow{\tau_3} \cdots
\]

with \( s_i \in S \) and \( \mathcal{I}_i \in [\Gamma \to \mathbb{R}_{\geq 0}] \), for all \( i \geq 0 \), satisfying two requirements:

- \( s_0 \in S_0 \), and \( \mathcal{I}_0(x) = 0 \) for all clocks \( x \in \Gamma \).
- For all \( i \geq 1 \), there is an edge in \( E \) of the form \( \langle s_{i-1}, s_i, \gamma, \delta \rangle \) such that \( (\mathcal{I}_{i-1} + \tau_i - \tau_{i-1}) \) satisfies \( \delta \) and \( \mathcal{I}_i \) equals \( [\gamma \rightarrow 0](\mathcal{I}_{i-1} + \tau_i - \tau_{i-1}) \).

We next define the semantics of RTLTL with respect to a timed Kripke structure. We only consider non-Zeno behaviors, that is, only a finite number of transitions can happen within a finite amount of time. We also assume that the set of clocks in the Kripke structure is disjoint from those of RTLTL formulae.

**Definition 4** (Semantics of RTLTL). Given a run \( r = (s, \mathcal{I}) \), of a timed Kripke structure over a time sequence \( \tau \), the pair \((\rho, \mathcal{I})\) satisfies the RTLTL formula \( \phi \) iff \( \rho^0 \models_{\mathcal{I}_0} \phi \), where \( \mathcal{I}_0 : \Gamma \to \{\tau_0\} \) and \( \models_{\mathcal{I}} \) is inductively defined as follows:

\[
\begin{align*}
\rho \models_{\mathcal{I}} p & \iff p \in L(s_0); \\
\rho \models_{\mathcal{I}} \neg \phi & \iff \rho \models_{\mathcal{I}} \phi; \\
\rho \models_{\mathcal{I}} \phi_1 \wedge \phi_2 & \iff \rho \models_{\mathcal{I}} \phi_1 \text{ and } \rho \models_{\mathcal{I}} \phi_2; \\
\rho \models_{\mathcal{I}} \phi_1 \vee \phi_2 & \iff \rho \models_{\mathcal{I}} \phi_1 \text{ or } \rho \models_{\mathcal{I}} \phi_2; \\
\rho \models_{\mathcal{I}} X \phi & \iff \rho^1 \models_{\mathcal{I} + \tau_1 - \tau_0} \phi; \\
\rho \models_{\mathcal{I}} \phi_1 U \phi_2 & \iff \rho^i \models_{\mathcal{I} + \tau_1 - \tau_j} \phi_2 \text{ for some } i \geq 0, \text{ and } \rho^j \models_{\mathcal{I} + \tau_j - \tau_0} \phi_1 \text{ for all } 0 \leq j < i; \\
\rho \models_{\mathcal{I}} \phi_1 R \phi_2 & \iff \text{ for all } j \geq 0, \text{ if for every } i < j \text{ } \rho^j \models_{\mathcal{I} + \tau_j - \tau_0} \phi_1 \text{ then } \rho^j \models_{\mathcal{I} + \tau_j - \tau_0} \phi_2.
\end{align*}
\]

The formulas of RTLTL are interpreted over runs of timed Kripke structures. In other words, the truth value of a RTLTL formula is determined by a timed state sequence and a clock interpretation. A timed state sequence \( \rho \) is a model of the RTLTL formula \( \phi \), denoted by \( \rho \models \phi \), if \((\rho, \mathcal{I})\) satisfies \( \phi \) for any interpretation \( \mathcal{I} \).
Given a timed Kripke structure model of a real-time system $M$, and a property $\phi$ expressed as a RTLTL formula, we would like to know if $M \models \phi$. Intuitively, a model of a RTLTL formula $\phi$ with respect to $M$ is a set of states of $M$ that satisfy the clock constraints of $\phi$. For instance, consider evaluating whether or not the state $s_2$ in Fig. 1 is in the model of $G(F(c > 2)s)$. Proposition $s$ does not hold at $s_2$, but it holds in one of its descendent states; moreover, when $s$ holds in that state, more than 3 time units has elapsed since state $s_0$. Thus, the constraint $c > 2$ accompanying proposition $s$ in the RTLTL formula is indeed satisfied. Therefore, $G(F(c > 2)s)$ holds at state $s_2$. As another example, consider evaluating whether $s_0 \models F(c > 1)p \lor F(c > 1)t$. It is easy to see that $s_0$ has two successor states in which either $p$ or $t$ is true. Moreover, the clock constraint $c > 1$ in the formula is satisfied in both successors; hence, $s_0 \models F(c > 1)p \lor F(c > 1)t$.

### 3.2 RTLTL as a Specification Language

Using clocks and operations on clocks we can relate the times of different states. A typical real-time requirement for a reactive system is a bounded-response property. For instance, “a multi-valued switch must be turned from position $p$ to position $q$ within 10 time units”, can be specified by the formula:

$$(c := 0)p \rightarrow p \lor (c < 10)q.$$  

The meaning of this formula over timed state sequences depends on the clock interpretation. For instance, the timed state sequence

$$\{(p), 5\} \rightarrow \{(p), 7\} \rightarrow \{(p), 8\} \rightarrow \{(p), 10\} \rightarrow \{(q), 14\} \rightarrow \{\}, 15$$

satisfies the formula.

The non-local property that “every stimulus $p$ is followed by a response $q$ and, then, by another response $r$ within 3 units since $p$" is also a bounded-response property which can be specified by the formula:

$$G((c := 0)p \rightarrow (q \land F(c < 3)r)).$$

While bounded-response properties assert that “something good” will happen within a specified amount of time; bounded-invariance properties assert that “nothing bad” will happen within a specified amount of time. For instance, “after every stimulus $p$, $q$ will not happen within 3 units of time” is an example of a bounded-invariance property which can be specified by the RTLTL formula:

$$G((c := 0)p \rightarrow (c < 3)\neg q).$$

### 4 Model Checking RTLTL in Logic Programming

Given a system modeled as a Kripke structure, and a property expressed as a temporal logic formula, verifying that the model satisfies the property involves a systematic search in the state space of the Kripke structure for a counter-example that falsifies the given formula. The vast majority of interesting properties that are to be verified can be classified into safety properties and liveness properties.

It is well known that safety properties can be verified by reachability analysis, i.e., if a counterexample to the postulated property exists, it can be finitely determined by enumerating all the reachable states of the Kripke structure. Verification of safety properties using logic programming, amounts to computing elements of the least fixed points (LFP) of a program, and is elegantly handled by standard
LP systems extended with tabling \[22\]. Verification of liveness properties is less straightforward, because counterexamples take the form of infinite traces, which are semantically equivalent to elements of the greatest fixed points (GFP) of a logic program: coinductive logic programming is more suitable for directly computing such counterexamples without the expensive transformations required by some other approaches suggested in the literature \[22\].

Intuitively, a state is live if it can be reached via an infinite loop (cycle). Liveness counterexamples can be found by (coinductively) enumerating all possible states that can be reached via infinite loops and then determining if any of these states constitute valid counterexamples.

In the following we will describe a model checking procedure which is the direct translation of semantics of RTLTL into a logic program. In linear temporal logic one checks if a temporal logic formula is true along a path. For instance, in order for $F\phi$ hold at state $s$, either $\phi$ should hold in state $s$ or, there must be a transition from $s$ to state $t$ of the timed Kripke structure such that $F\phi$ holds at $t$. This can be logically described as $F\phi \equiv \phi \lor (XF\phi)$. Similar equivalence relations can be obtained and will be used to evaluate whether a state $s$ of the timed Kripke structure is in the model of a RTLTL formula $\phi$.

$$G\phi \equiv \phi \land (XG\phi)$$
$$\phi_1 U \phi_2 \equiv \phi_2 \lor (\phi_1 \land (\phi_1 U \phi_2))$$
$$\phi_1 R \phi_2 \equiv \phi_2 \land (\phi_1 \lor (\phi_1 R \phi_2))$$

To verify that a state $s$ satisfies a formula $\phi$, we negate the formula and check that the state does not satisfy the negated formula. We make sure that the negated formulae are normalized: only atomic propositions can be negated. Among the temporal operators, the following equivalence relations hold:

$$\neg Gp \equiv F \neg p$$
$$\neg pUq \equiv \neg pR \neg q$$
$$\neg Fp \equiv G \neg p$$
$$\neg pRq \equiv \neg pU \neg q$$
$$\neg p \land q \equiv \neg p \lor \neg q$$
$$\neg p \lor q \equiv \neg p \land \neg q$$
$$\neg Xp \equiv X \neg p$$
$$\neg \neg p \equiv p$$

In our encoding, we use the predicate `negate/2` to convert the negated formulae to their negation normal forms.

```
normalize(~ (P v Q), NP ^ NQ) :- normalize(~P, NP), normalize(~Q, NQ).
normalize(~ (P ^ Q), NP v NQ) :- normalize(~P, NP), normalize(~Q, NQ).
normalize(~ (P u Q), NP r NQ) :- normalize(~P, NP), normalize(~Q, NQ).
normalize(~ (P r Q), NP u NQ) :- normalize(~P, NP), normalize(~Q, NQ).
normalize(~ x P, x NP) :- normalize(~P, NP).
normalize(~ f P, g NP) :- normalize(~P, NP).
normalize(~ g P, f NP) :- normalize(~P, NP).
normalize(~ P, ~ NP) :- normalize(P, NP).
normalize(~ ~ P, NP) :- normalize(P, NP).
normalize(~ ~ P, P) :- proposition(P).
normalize(P, P) :- proposition(P).
```

Next, we describe our encoding of timed Kripke structures using Horn clauses. Given $S$ the set of states and $P$ the set of propositions of a timed Kripke structure, we define the following predicates:

- `proposition/1`, which specifies all the propositions of the timed Kripke structure,
- `trans/5` $\subset S \times S \times \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0}$, which represents the transitions of the timed Kripke structure, taking clock operations into consideration, and
- `holds/2` $\subset S \times P$, which specifies the set of propositions that are true in a state.
For instance, the timed Kripke structure of Fig. 1 is modeled by the following logic programming rules and facts:

\[
\begin{align*}
\text{proposition}(p). & \quad \text{proposition}(s). \\
\text{proposition}(q). & \quad \text{proposition}(t). \\
\text{holds}(s0, q). & \quad \text{holds}(s2, q). \\
\text{holds}(s1, t). & \quad \text{holds}(s3, p). \\
\text{holds}(s2, p). & \quad \text{holds}(s4, s). \\
\end{align*}
\]

\[
\begin{align*}
\text{trans}(s0, s1, W, Ci, Co) :& \quad \{W - Ci > 1, W - Ci < 2, Co = Ci\}. \\
\text{trans}(s0, s2, W, Ci, Co) :& \quad \{W - Ci > 1, Co = Ci\}. \\
\text{trans}(s2, s3, W, Ci, Co) :& \quad \{Co = W\}. \\
\text{trans}(s3, s4, W, Ci, Co) :& \quad \{W - Ci > 3, Co = Ci\}. \\
\text{trans}(s4, s2, W, Ci, Co) :& \quad \{Co = Ci\}. \\
\end{align*}
\]

In the rules for \texttt{trans/5}, \texttt{Ci} and \texttt{Co} model the clock \(c\) of the timed Kripke structure: \texttt{Ci} remembers the last (wall clock) time \(c\) was reset; while, \texttt{Co} is used to pass on the clock value to the next transition. The third parameter, \(W\), is the current (wall clock) time.

Note that the set of constraints used for modeling clocks are directly handled in CLP(R). The CLP(R) constraints are enclosed within the curly braces, as it is the convention in most Prolog systems.

Having (i) expressed the RTLTL formulae as equivalence relations in terms of next temporal operator, and (ii) represented timed Kripke structures as set of CLP(R) rules, we present our co-CLP encoding of RTLTL model checking in Fig. 2. To keep the exposition simple we assume that RTLTL encoding of model checking involves a single clock.

In order to verify that a state \(s\) of a timed Kripke structure models a property \(\phi\), first we negate the property and we evaluate the query \(\text{verify}(s, \neg \phi, \text{path})\), where \(\neg \phi\) is in negation normal form. If the call to \texttt{verify/3} fails, then there is no path on which the negated formula holds, implying that the formula \(\phi\) holds in state \(s\). In contrast, if \texttt{verify/3} returns a path as an answer, then that specific path is a counter-example for which the original formula \(\phi\) does not hold.

Note that the temporal operators \(X\), \(F\), \(G\), \(U\) and \(R\) are represented with corresponding lower case letters, while \(\wedge\), \(\vee\), and \(\sim\) represent \(\land\), \(\lor\), and negation, respectively.

Temporal operators whose meaning is given in terms of LFPs, e.g., \(\text{next}, \text{future}\), and \(\text{until}\), are realized via inductive logic programming, while those whose meaning is given in terms of GFPs, e.g., \(\text{global}\) and \(\text{release}\), are realized using coinductive logic programming.

For instance, consider the problem of determining whether \(G\phi\) holds at state \(s\). Assuming \(s'\) is a successor of \(s\), obviously, \(\phi\) has to hold at \(s\) and using the identity: \(G\phi \equiv \phi \land (XG\phi)\) \(G\phi\) must hold in \(s'\). In other words, \(s \models G\phi\) only if \(s' \models G\phi\). Using similar reasoning, assuming \(s''\) is a successor of \(s'\), then, \(s'' \models G\phi\) must be true and so on. Hence, we seek the greatest fixed points when encoding the \(\text{global}\) temporal operator.

Note that \texttt{coverify/6} is declared as coinductive only on the first three arguments; i.e., time will be ignored to check if \texttt{coverify/6} is cyclical.

## 5 Related Work

Several researchers have considered various timed extensions of LTL. Koymans [15] considers PTL and adds an infinite supply of real-time modalities such as \(\diamond_{\leq \delta}\) (“eventually within \(\delta\) time units”) and \(\boxdot_{\leq \delta}\)
verify(S, F, [S], W, Ci, Di) :- proposition(F), holds(S, F).
verify(S, ~ F, [S], W, Ci, Di) :- proposition(F), \+holds(S, F).
verify(S, const(D) F, [S], W, Ci, Di) :- proposition(F), holds(S, F), const(D, W, Di).
verify(S, ~ const(D) F, [S], W, Ci, Di) :- proposition(F), \(+holds(S, F); \+const(D, W, Di)).
verify(S, reset(D) F, Path, W, Ci, Di) :- \{Do = W\}, verify(S, F, Path, W, Ci, Do).
verify(S, F1 ^ F2, Path, W, Ci, Di) :- verify(S, F1, Path1, W, Ci, Di), verify(S, F2, Path2, W, Ci, Di),
\( append(Path1, _, Path2) -> Path = Path2 ; append(Path2, _, Path1) -> Path = Path1 \).
verify(S, F1 v F2, Path, W, Ci, Di) :- verify(S, F1, Path, W, Ci, Di); verify(S, F2, Path, W, Ci, Di).
verify(S, x F, [S|P], W, Ci, Di) :- trans(S, S2, W, Ci, Co), \{W2 > W\}, \{Do = Di\}, verify(S2, F, P, W2, Co, Do).
verify(S, f F, Path, W, Ci, Di) :- verify(S, F, Path, W, Ci, Di); verify(S, x f F, Path, W, Ci, Di).
verify(S, F1 u F2, Path, W, Ci, Di) :- verify(S, F2, Path, W, Ci, Di); verify(S, F1 ^ x(F1 u F2), Path, W, Ci, Di).
verify(S, g F, Path, W, Ci, Di) :- coinductive coverify(S, g F, Path, W, Ci, Co).
verify(S, F1 r F2, Path, W, Ci, Co) :- verify(S, F1 ^ F2, Path, W, Ci, Co).
verify(S, F1 r F2, Path, W, Ci, Co) :- verify(S, F2 ^ x(F1 ^ F2), Path, W, Ci, Co).

\( const(less(d, A), W, C) := \{W - C < A\}\).
\( const(lesseq(d, A), W, C) := \{W - C <= A\}\).
\( const(greater(d, A), W, C) := \{W - C > A\}\).
\( const(greatereq(d, A), W, C) := \{W - C >= A\}\).
\( const(eq(d, A), W, C) := \{W - C = A\}\).

Figure 2: Model checker for RTLTL
N. Saeedloei, G. Gupta

(“next state within $\delta$ time units) into the logic. These bounded temporal operators are easily defined in RTLTL. For instance, $\Diamond_{\leq \delta} \phi$ can be expressed by the RTLTL formula:

$$(c := 0) \rightarrow F(c < \delta) \phi.$$ 

Using bounded temporal operators, one can only relate the times of adjacent temporal contexts; using RTLTL however, we can specify timing constraints between arbitrary states.

Alur et al [3] propose a real-time logic, TPTL, as an extension of PTL with freeze quantifiers. A variable $x$ can be bound by a freeze quantifier “$x$.”, which freezes $x$ to the time of the local temporal context. The variable $x$ will be frozen immediately upon introduction and can be used in future states. We can define variables bounded by freeze quantifiers naturally in RTLTL. For instance, the equivalence of the TPTL formula $x.\phi$ in RTLTL is $(c := 0)\phi$.

The underlying assumption in TPTL is that every variable is bound to the time of a particular state. Thus, in the TPTL formula $\Diamond y.\phi$, the variable $y$ is bound to the time of the state at which $\phi$ is “eventually” true. In RTLTL, we use clocks to keep track of time in various states and we assert timing constraints among various states using clock constraints. For instance the TPTL formula

$$\Box x. (p \rightarrow \Diamond(q \land \Diamond z. (r \land z \leq x + 5))).$$

is specified in RTLTL by

$$\Box ((c := 0)p \rightarrow F(q \land F(c < 5)r)).$$

Metric interval temporal logic (MITL) extends propositional linear-time logic with intervals. It uses non-negative reals as time domain and has strictly-monotonic real-time semantics. MITL uses the bounded-operator syntax; however, it disallows the temporal operators to be bounded by singular intervals. This logic comes close to our real-time extension of LTL; however, our logic has simpler syntax. Moreover, our notion of time and clocks is adopted directly from timed automata. moreover, the semantics of our logic is defined in terms of semantics of timed automata.

There have been several attempts to connect logic programming and constraints with verification and model checking. Pontelli and Gupta [11] use CLP for modeling and verification of real-time systems. To verify a property of interest, they formulate the negated property as a query and try to prove false. While safety properties can be formulated as reachability problems in their framework, liveness properties cannot be handled, as only least fixed points computations are possible.

Gupta et. al propose an elegant LTL model checker (but not timed LTL) based on coinductive logic programming [12]. Our current work is the extension of this model checker with time.

Delzanno and Podelski [8] also use constraint logic programming for encoding discrete, infinite-state systems. They consider computation tree logic (CTL) properties and verify them by computing the least and greatest fixed points of the logical consequence operator of CLP programs.

Urbina [29] models hybrid systems as constraint logic programs. However, liveness properties of timed or hybrid automata cannot be directly expressed using fixed point computations in his work.

6 Conclusion

This paper has presented (i) a real-time extension of linear temporal logic with real time, called RTLTL, and (ii) a model checker for RTLTL based on coinductive constraint logic programming.

Our effort was driven by our desire to keep the logic simple, without introducing new constructs and operators. Our work makes use of many familiar concepts such as clocks, clock operations and clock
interpretations, from timed automata [4]. Therefore, RTLTL can be considered as combination of two well-known formalisms, LTL and timed automata, to provide capabilities for real-time specifications requirements. Our logic has a dense-time model semantics which is defined in terms of infinite time state sequences of timed automata.

Our logic programming-based model checker is a direct translation of the semantics of RTLTL into a logic program, where clocks and clock operations are handled by constraint logic programming. While safety properties are verified by reachability analysis and computationally realized using logic programming; coinduction is used for verification of liveness properties.

References


