Symbolic Model Checking for Real-time Systems*

Thomas A. Henzinger†
Computer Science Department, Cornell University
Ithaca, NY 14853, U.S.A.

Xavier Nicollin‡ Joseph Sifakis‡ Sergio Yovine‡
Laboratoire de Génie Informatique, Institut IMAG
B.P. 53X, 38041 Grenoble cedex, France

Abstract. We describe finite-state programs over real-numbered time in a guarded-command language with real-valued clocks. Model checking answers the question which states of a real-time program satisfy a branching-time specification (given in an extension of CTL with clock variables). We develop an algorithm that computes this set of states symbolically as a fixpoint of a functional on state predicates, without constructing the state space.

For this purpose, we introduce a μ-calculus on computation trees over real-numbered time. Unfortunately, many standard program properties, such as response *for all nonZeno execution sequences* (during which time diverges), cannot be characterized by fixpoints: we show that the expressiveness of the timed μ-calculus is incomparable to the expressiveness of timed CTL. Fortunately, this result does not impair the symbolic verification of "implementable" real-time programs — those whose safety constraints are machine-closed with respect to diverging time and whose fairness constraints are restricted to finite upper bounds on clock values. All timed CTL properties of such programs are shown to be computable as finitely approximable fixpoints in a simple decidable theory.

1 Introduction

Model checking is a powerful technique for the automatic verification of finite-state systems. Model checking algorithms determine the states that satisfy a modal formula by a graph-theoretic analysis of the state space (Kripke structure) [CE81, QS81, CES86]. The main practical limitation of model checking algorithms is caused by the size of the state graph, which grows exponentially with the number of parallel components in a system. One approach to confine this "state explosion problem" relies on the *symbolic* (rather than *enumerative*) representation of sets of states and computes the set that satisfies a formula as a fixpoint of a functional on state predicates. While the theoretical possibility of symbolic model checking by computing fixpoints was realized early [EC80, Sif82], the method has become practical only recently through the symbolic representation of state sets by binary decision diagrams (BDDs) [Bry86]. BDDs were first used for verification purposes by [CBM89] and for model checking by [BCD*90]. By now implementations of symbolic model checking techniques have reported spectacular successes, in particular in the area of hardware verification [CBG*91].

The history of model checking has been considerably shorter in the case of real-time systems. First researchers focused on discrete time (the integers), for which the untimed model checking methods can be readily extended [EMSS89, AH91, Eme91]. New complexities arise if we insist that for the compositional modeling of asynchronous systems, time should not be discretized [Alu91]. We consider dense time (the reals). A standard dense-time approach models systems as a transition relation together with a finite set of real-valued clocks that proceed at a uniform rate and constrain the times at which transitions may occur [AD90, Lew90, AH91, NSY91]. Only recently a graph-theoretic model checking algorithm was found for these systems [ACD90]. Since the time component causes the state space to be infinite, the algorithm depends on a clever construction of a finite quotient of the infinite state graph. The size of this "region graph" of equivalence classes of states grows exponentially not only with the number of components in a system but also with the largest time constant and the number of clocks that are used to specify timing constraints.

*A full version of this paper (including all proofs) is available as a technical report from Cornell University and from IMAG in Grenoble.

†This work was performed while visiting IMAG.

‡Partially supported by the Esprit basic research action SPEC.

0-8186-2735-2/92 $3.00 © 1992 IEEE

Thus the need for an alternative, "symbolic" approach to model checking, which avoids the explicit construction of the region graph, is particularly pressing in the real-time case.

The main problem in devising such a method is the definition of a proper next-state relation whose iteration allows us to compute all program properties we wish to consider. Since our time domain is dense, the next-state relation must not force time to advance by more than an infinitesimal amount. Its iteration, however, must force time to advance beyond any bound. This is because upper-bound clock constraints may restrict the time that a system can spend in a particular set of states, say, a loop. While the system may loop any finite number of times before the time bound expires, it cannot do so infinitely often. In other words, every upper-bound constraint hides a fairness condition.

Both requirements on the next-state relation seem and, in a precise technical sense we will discuss, generally are contradictory. We show that this result has, fortunately, little relevance to the verification of concrete real-time programs, including loops with upper bounds. We define the class of *divergence-safe* systems, for which inevitability is equivalent to time-bounded inevitability: for any event of a divergence-safe system there is a time bound such that if the event need not occur before that bound, then it need not occur ever. The class of divergence-safe systems contains all systems without fairness constraints other than those induced by constant upper bounds, which — it may be argued — is the only "implementable" notion of fairness. In the case of divergence-safe systems, we show that we can settle for a next-state relation that does not force time to advance ever. Such a relation allows us to compute possibility ($\exists\Diamond$) and its dual, invariance ($\forall\Box$), directly. Inevitability ($\forall\Diamond$) is reducible to time-bounded inevitability, which is but an invariance: time may not progress beyond the upper bound of any event without the event occurring.

We have now informally sketched our course of action to compute bounded and unbounded properties of real-time systems as fixpoints of functionals that are based on a next-state relation. Formally, we introduce a timed μ-calculus, Tμ, that is interpreted over timed computation trees. We compare this fixpoint calculus to standard real-time extensions of branching-time logics and find their expressive powers to be incomparable: the basic operator $\forall\Diamond$ of branching-time logics cannot be characterized by fixpoints in Tμ. However, as hinted above, we are able to give a translation from CTL with clock variables (TCTL of [Alu91]) to Tμ that agrees on all divergence-safe models. This translation forms the basis of a symbolic model checking procedure.

We apply this theory to concrete real-time system descriptions. Real-time systems are defined in a guarded-command language with clocks, which is equivalent to *timed safety automata* — timed automata [AD90] without acceptance conditions. Both languages allow the definition of divergence-safe systems only and, thus, all formulas of Tμ and TCTL can be verified symbolically by computing fixpoints (indeed, finitely approximable fixpoints) of appropriate functionals. The practical computation of these functionals rests on our ability to express, for any given program, the next-state relation symbolically. Indeed, the extraction of the next-state relation from a program turns out to be rather nontrivial in the case that the program is not *nonZeno* [AL91] (not machine-closed with respect to the divergence of time); that is, if it may prevent time from diverging. We show how a symbolic fixpoint approach can be used to test if a guarded-command real-time program is nonZeno and, if not, to convert it into an equivalent nonZeno program.

We wish to conclude this introduction by pointing out that there is, of course, nothing "magical" about symbolic methods. Timed model checking is intrinsically difficult (PSPACE-hard) and already the known graph-theoretic algorithm on the exponential region graph is worst-case near-optimal [ACD90]. In practice, however, the "intuitive complexity" [BCD*90] of the state space is typically much smaller than the region graph. Only a symbolic method can exploit this phenomenon, by representing unions of regions symbolically as state predicates. Our model checking algorithm constructs (the equivalent of) the full region graph but in extreme cases; typically it works on a quotient of the region graph that depends on the formula being checked.

The remainder of the paper is organized as follows. In the next section, we define our model of real-time systems. Section 3 presents a guarded-command language for the description of real-time systems. In Section 4, we introduce the timed μ-calculus Tμ and study its complexity and expressiveness compared to the branching-time logic TCTL. Section 5 develops symbolic model checking algorithms for both Tμ and TCTL.

2 A Model for Real-time Systems

In this section we present a formal branching-time semantics for real-time systems. Our semantics integrates elements from several models that have been

proposed in the literature; we are particularly indebted to the clocks of [AD90], the dense trees of [ACD90], the transition-delay dichotomy of [HMP91] and [NSY91], and the relative safety of [Hen91].

2.1 State sequences

We model time as the nonnegative real numbers R^+. The state of a system is determined by the values of a finite set P of boolean variables (*propositions*), representing data and control, and by the values of a finite set C of real-valued variables (*clocks*). The clocks allow the system to make time-dependent decisions.

Definition 2.1 (state) A *state* σ is an interpretation of all propositions and clocks; that is, σ assigns to each proposition $p \in P$ a boolean value $\sigma(p) \in \{\texttt{true}, \texttt{false}\}$ and to each clock $x \in C$ a nonnegative real $\sigma(x) \in \mathsf{R}^+$. We write Σ for the set of all states. ■

For $\delta \in \mathsf{R}^+$, let $\sigma + \delta$ denote the state that agrees with the state $\sigma \in \Sigma$ on all propositions and assigns the value $\sigma(x) + \delta$ to each clock $x \in C$. Given a set $L = \{v_1 := l_1, \ldots v_n := l_n\}$ of variables $v_i \in P \cup C$ and corresponding values $l_i \in \{\texttt{true}, \texttt{false}\} \cup \mathsf{R}^+$, we write $\sigma[L]$ for the state that assigns the value l_i to the variable v_i for all $1 \leq i \leq n$ and agrees with $\sigma \in \Sigma$ on all other propositions and clocks.

The behavior of a system is represented by an infinite sequence of states. In any state the system may either change its data and reset some of the clocks, or let time pass. Whenever time passes, all clock values increase uniformly with the rate of time. It follows that at any time the value of a clock x is equal to the amount of time that has elapsed since the last time x was reset.

Definition 2.2 (state sequence and divergence) A *state sequence* $\bar{\sigma} = \sigma_0\sigma_1\sigma_2 \ldots$ is an infinite sequence of states $\sigma_i \in \Sigma$ such that for all $i \geq 0$ either

1. (σ_i, σ_{i+1}) is a *transition*: $\sigma_{i+1}(x) = 0$ or $\sigma_{i+1}(x) = \sigma_i(x)$ for all clocks $x \in C$ (in this case we say that σ_{i+1} is a *transition successor* of σ_i); or

2. (σ_i, σ_{i+1}) is a *delay*: $\sigma_{i+1} = \sigma_i + \delta_i$ for some time lapse $\delta_i \in \mathsf{R}^+$ (in this case we say that σ_{i+1} is a *time successor* of σ_i).

If (σ_i, σ_{i+1}) is a transition, we define the corresponding time lapse δ_i to be 0. For all $i \geq 0$, the *time* τ_i of state σ_i is the real number $\sum_{0 \leq j < i} \delta_j$. The state sequence $\bar{\sigma}$ *diverges* if the corresponding time sequence $\tau_0\tau_1\tau_2 \ldots$ diverges; that is, for all $\tau \in \mathsf{R}^+$ there is some $i \geq 0$ such that $\tau_i > \tau$. ■

The modeling of timed behavior by divergent timed state sequences reflects several choices we make [AH91]. First, we require time to *progress* past any real number. Second, we choose time to be *weakly monotonic*: time must not advance with transitions (i.e., when data is modified). The weak monotonicity of time allows the modeling of simultaneous activities by sequential interleaving. Not all researchers find this abstraction convenient. Our preference for weak monotonicity is no precondition for any of the results we will present and may be reversed. Third, we restrict ourselves to the modeling of systems that satisfy the condition of *finite variability*: the number of transitions in any bounded time interval of a divergent state sequence is finite.

The possible behaviors of a system are collected in a set of state sequences.

Definition 2.3 (fusion, suffix, and stutter closure) A set Π of state sequences is *fusion-closed* if for all $\bar{\sigma}, \bar{\sigma}' \in \Pi$ and $i, j \geq 0$, if $\sigma_i = \sigma'_j$ then the state sequence $\bar{\sigma}''$ such that $\sigma''_k = \sigma_k$ for all $0 \leq k < i$ and $\sigma''_k = \sigma'_{k-i+j}$ for all $k \geq i$, is in Π.

Given a state sequence $\bar{\sigma}$ and $i \geq 0$, we write $\bar{\sigma}^i$ for the state sequence that results from $\bar{\sigma}$ by deleting the first i states. A set Π of state sequences is *suffix-closed* if for all $\bar{\sigma} \in \Pi$ and $i \geq 0$, $\bar{\sigma}^i \in \Pi$.

A set Π of state sequences is *stutter-closed* if for all $i \geq 0$ and $\delta \in \mathsf{R}^+$, a state sequence $\bar{\sigma}$ with $\delta_i \geq \delta$ is in Π iff the state sequence $\bar{\sigma}'$ such that $\sigma'_k = \sigma_k$ for all $0 \leq k \leq i$, $\sigma'_{i+1} = \sigma_i + \delta$, and $\sigma'_{k+1} = \sigma_k$ for all $k > i$, is in Π. ■

Definition 2.4 (premodel and real-time system) A *premodel* is a set of state sequences that is fusion-closed, suffix-closed, and stutter-closed. A *real-time system* is a premodel that contains only divergent state sequences. ■

Fusion closure asserts that the future of a system is completely determined by the present state of the system and does not depend on its past. In other words, every state contains the complete information about the future evolution of the system; given a fusion-closed and suffix-closed set Π of state sequences and a state σ, the subset of sequences in Π that start at σ form a tree with root σ. Stuttering closure ensures *time additivity*, and the uniqueness of $\sigma + \delta$ implies *time determinism* [NSY91].

Definition 2.5 (safety) A premodel Π is *safe* if for any state sequence $\bar{\sigma}$, if all finite prefixes of $\bar{\sigma}$ are prefixes of sequences in Π, then $\bar{\sigma} \in \Pi$. The *safety closure* $\overline{\Pi}$ of a premodel Π is the least safe premodel that contains Π. ■

This notion of safety corresponds to safety in linear time [ADS86]. Because of divergence and stutter closure, no real-time system can be safe. Thus we define the less stringent requirement of divergence safety, which corresponds in the linear-time framework to the notion of *safety relative to the divergence of time* [Hen91].

Definition 2.6 (divergence safety) A real-time system Π is *divergence-safe* if for any divergent state sequence $\bar{\sigma}$, if all finite prefixes of $\bar{\sigma}$ are prefixes of sequences in Π, then $\bar{\sigma} \in \Pi$. The *divergence safety closure* $\widetilde{\Pi}$ of a real-time system Π is the least divergence-safe real-time system that contains Π. ■

Example 2.1 Let Π_1 be the real-time system that changes the value of a proposition p from `false` to `true` before time 10; that is,

$$\Pi_1 = \{\bar{\sigma}^k \mid k \geq 0 \ \wedge\ \bar{\sigma} \text{ diverges} \ \wedge\ \exists i \geq 0.(\tau_i < 10 \wedge \forall j \geq i.\, \sigma_j(p) = \texttt{true} \ \wedge\ \forall j < i.\, \sigma_j(p) = \texttt{false})\}.$$

It is easy to check that Π_1 is divergence-safe. Take a divergent state sequence $\bar{\sigma}$ such that every prefix of it is the prefix of a state sequence in Π_1. Since $\bar{\sigma}$ diverges, it has a finite prefix $\sigma_0\sigma_1 \ldots \sigma_n$ such that $\tau_n \geq 10$, which is the prefix of some sequence in Π_1. Thus, for some $i < n$, $\sigma_i(p) = \texttt{true}$ and $\tau_i < 10$, and for all $j < i$, $\sigma_j(p) = \texttt{false}$. Hence, $\bar{\sigma} \in \Pi_1$.

Now, consider the real-time system Π_2 that changes the value of p *eventually*; that is,

$$\Pi_2 = \{\bar{\sigma}^k \mid k \geq 0 \ \wedge\ \bar{\sigma} \text{ diverges} \ \wedge\ \exists i \geq 0.\ (\forall j \geq i.\, \sigma_j(p) = \texttt{true} \ \wedge\ \forall j < i.\, \sigma_j(p) = \texttt{false})\}.$$

Π_2 is not divergence-safe, since every divergent state sequence $\bar{\sigma}$ with $\sigma_i(p) = \texttt{false}$ for all $i \geq 0$, is such that every prefix of it is the prefix of a sequence in Π_2, but $\bar{\sigma}$ is not in Π_2. ■

2.2 Step sequences

We now introduce a notion of execution for real-time systems. For this purpose, the behavior of a system in continuous time is discretized by considering a run as a succession of discrete steps. A step of a system consists of two phases, namely, a delay followed by a transition. This notion of step is at the base of many models for real-time computation [AH89, NS91, NSY91].

Definition 2.7 (step) Let Π be a premodel.

1. For all states $\sigma, \sigma' \in \Sigma$, we write $\sigma \Rightarrow_\Pi \sigma'$ if σ' is a transition successor of σ in some state sequence of Π.
2. For all states $\sigma \in \Sigma$ and $\delta \in \mathsf{R}^+$, we write $\sigma \rightarrow_\Pi^\delta \sigma + \delta$ if $\sigma + \delta$ is a time successor of σ in some state sequence of Π. Note that $\rightarrow_\Pi^0$ is the identity relation on states and $\rightarrow_\Pi^0 \subseteq \Rightarrow_\Pi$.
3. For all $\delta \in \mathsf{R}^+$, let $\triangleright_\Pi^\delta = \rightarrow_\Pi^\delta \circ \Rightarrow_\Pi$.
4. For all states $\sigma, \sigma' \in \Sigma$, we write $\sigma \triangleright_\Pi \sigma'$ if $\sigma \triangleright_\Pi^\delta \sigma'$ for some $\delta \in \mathsf{R}^+$.

If $\sigma \triangleright_\Pi \sigma'$, we say that σ' is a (*step*) *successor* of σ in Π. ■

In other words, $\sigma \triangleright_\Pi^\delta \sigma'$ means that in state σ, the system modeled by Π may first let time advance by $\delta \in \mathsf{R}^+$ and then perform a transition leading to state σ'. Since every premodel is closed under stuttering, both transition successors and time successors are step successors.

We will define real-time systems Π by giving the successor relation $\triangleright_\Pi \subseteq \Sigma^2$. In this section, we show that these definitions are proper for divergence-safe real-time systems: we prove that a successor relation $\triangleright_\Pi$ uniquely defines a divergence-safe real-time system Π. First, observe that every successor relation $\triangleright_\Pi$ defines a set Π_{step} of step sequences (and a set Π_{step}^{div} of divergent step sequences) that are generated iterating $\triangleright_\Pi$.

Definition 2.8 (step sequence) A *step sequence* of a premodel Π is an infinite sequence $\hat{\sigma} = \sigma_0\sigma_1\sigma_2 \ldots$ of states $\sigma_i \in \Sigma$ such that $\sigma_i \triangleright_\Pi \sigma_{i+1}$ for all $i \geq 0$. The time of a state and the divergence of a step sequence is defined as for state sequences. We write Π_{step} for the set of step sequences of Π and Π_{step}^{div} for the set of divergent step sequences of Π. ■

While $\Pi \subseteq \Pi_{step}$ for every premodel Π, not every step sequence of Π is a state sequence. But we can turn every step sequence into a state sequence by adding stutter states. The step sequences that correspond to state sequences in Π are called the execution sequences of Π:

Definition 2.9 (execution sequence) Let Π be a premodel. For a step sequence $\hat{\sigma} = \sigma_0\sigma_1\sigma_2 \ldots$ of Π, let $ex(\hat{\sigma})$ denote the state sequence $\bar{\sigma}'$ such that for all $i \geq 0$, $\sigma'_{2i} = \sigma_i$ and $\sigma'_{2i+1} = \sigma_i + \delta_i$. An *execution sequence* of Π is a step sequence $\hat{\sigma}$ of Π such that $ex(\hat{\sigma}) \in \Pi$. We write Π_{ex} for the set of execution sequences of Π. ■

In general, it is not the case that $ex(\hat{\sigma}) \in \Pi$ for all step sequences $\hat{\sigma}$ of a real-time system Π; that is, not every step sequence is an execution sequence. Consider, for instance, the real-time system Π_2 of Example 2.1. The divergent step sequence $\hat{\sigma}$ with $\sigma_i(p) =$

false and $\tau_i = i$ for all $i \geq 0$, is in $\Pi_2{}^{div}_{step}$, but $ex(\hat{\sigma})$ is not in Π_2. However, $ex(\hat{\sigma})$ is in the divergence safety closure of Π_2. Indeed, for every *divergence-safe* real-time systems Π, the step sequences generated by the successor relation $\triangleright_\Pi$ are precisely the step sequences that correspond to state sequences in Π.

Proposition 2.1 *For every premodel* Π, $\Pi_{step} = \overline{\Pi}_{ex}$. *For every real-time system* Π, $\Pi^{div}_{step} = \widetilde{\Pi}_{ex}$.

Corollary 2.1 *For all premodels* Π_1 *and* Π_2, $\overline{\Pi}_1 = \overline{\Pi}_2$ *iff* $\triangleright_{\Pi_1} = \triangleright_{\Pi_2}$. *For all real-time systems* Π_1 *and* Π_2, $\widetilde{\Pi}_1 = \widetilde{\Pi}_2$ *iff* $\triangleright_{\Pi_1} = \triangleright_{\Pi_2}$.

Thus we can define divergence-safe real-time systems by their successor relation, just as we can define safe untimed systems by a next-state relation.

3 Guarded-command Real-time Programs

In this section we present a programming language for defining real-time systems. A real-time program is given as set of guarded commands. The guards check the values of propositions and clocks and can be used to enforce timing constraints. The commands assign new values to propositions and clocks.

Definition 3.1 (state predicate) The set of *state predicates* is inductively defined by the following grammar:

$$\phi \quad ::= \quad p \mid x + c \leq d \mid c \leq x + d \mid x + c \leq y + d \mid \neg\phi \mid \phi_1 \wedge \phi_2,$$

where $p \in P$, $x, y \in C$, and $c, d \in \mathsf{N}$.

Given a state $\sigma \in \Sigma$, every state predicate ϕ evaluates to a truth value $\sigma(\phi) \in \{\mathtt{true}, \mathtt{false}\}$ in the standard way. The state σ *satisfies* ϕ, denoted by $\sigma \models \phi$, if $\sigma(\phi) = \mathtt{true}$. We write $[\![\phi]\!]$ for the set of states that satisfy the state predicate ϕ. ■

Standard abbreviations of state predicates, such as **true**, $p \Rightarrow q$, $x > 5 \vee y < 3$, and $2 < x < 5$, are defined as usual. For all state predicates ϕ and ϕ' and propositions $p \in P$, we write $\phi[p := \phi']$ for the state predicate that results from ϕ by replacing each occurrence of p with ϕ'. Analogously, for all constants $c \in \mathsf{N}$ and clocks $x, y \in C$, by $\phi[x := c]$ (or $\phi[x := y + c]$) we denote the state predicate that is obtained from ϕ by replacing each occurrence of x with c (or $y + c$, respectively). Given a set $L = \{v_1 := l_1, \ldots v_n := l_n\}$ of assignments, we write $\phi[L]$ for the state predicate that results from ϕ by simultaneously replacing each variable v_i with the corresponding expression l_i. For $\delta \in \mathsf{R}^+$ and $L = \{x := x + \delta \mid x \in C\}$, we denote $\phi[L]$ by $\phi + \delta$.

The satisfiability problem for state predicates is obviously no simpler than boolean satisfiability; neither is it harder:

Proposition 3.1 *The satisfiability problem for state predicates is NP-complete.*

Definition 3.2 (real-time program) A *guarded-command real-time program* (GCP for short) S is a triple $\langle \phi^0, \phi^\Box, \mathrm{G} \rangle$ such that:

1. ϕ^0, the *initial condition*, is a state predicate that imposes a constraint on the initial states of S.

2. $\phi^\Box$, the *invariant*, is a state predicate that imposes a constraint on all states of S. We require $\phi^\Box$ to be *stutter-closed*; that is, for all states $\sigma \in \Sigma$ and $\delta \in \mathsf{R}^+$, if $\sigma + \delta \models \phi^\Box$ then $\sigma \models \phi^\Box$.

3. G, the *program body*, is a set of guarded commands. Each *guarded command* $g \in \mathrm{G}$ is of the form $\psi \rightarrow L$, where the guard ψ is a state predicate and $L = \{v := l_v \mid v \in P \cup C\}$ is a set of simultaneous assignments such that for all propositions $p \in P$, l_p is a state predicate, and for all clocks $x \in C$, l_x is either 0 (i.e., x is reset) or x (i.e., the clock x is left unchanged). When writing guarded commands, we suppress assignments of the form $v := v$. ■

Let G be a set of guarded commands. A guarded command $g = \psi \rightarrow L$ in G is enabled on a state $\sigma \in \Sigma$ iff $\sigma \models \psi$. In a state σ, any enabled guarded command may be executed. The execution of g leads to the state $\sigma[L]$, where $\sigma[L](v) = \sigma(l_v)$ for all variables $v \in P \cup C$. Hence the guarded command g defines a partial function from Σ to Σ: if $\sigma \models \psi$, then $g(\sigma) = \sigma[L]$; otherwise, $g(\sigma)$ is undefined. The set G of guarded commands defines, then, a binary relation on the states: for all states $\sigma, \sigma' \in \Sigma$, $\mathrm{G}(\sigma, \sigma')$ iff $g(\sigma) = \sigma'$ for some guarded command $g \in \mathrm{G}$. By G^R we denote the reflexive closure of the relation G.

Given a state predicate ϕ, we define the state predicate

$$\mathtt{pre}_g(\phi) \;=\; \psi \;\wedge\; \phi[L],$$

which characterizes all states from which a state satisfying ϕ can be obtained by executing the guarded command $g = \psi \rightarrow L$. The state predicate

$$\mathtt{pre}(\phi) \;=\; \phi \;\vee\; \bigvee_{g \in \mathrm{G}} \mathtt{pre}_g(\phi)$$

characterizes the set of "G-predecessors" of all states satisfying ϕ. This is because a state $\sigma \in \Sigma$ satisfies $\mathbf{pre}(\phi)$ iff there exists a state $\sigma' \in \Sigma$ such that $G^R(\sigma, \sigma')$ and $\sigma' \models \phi$.

Definition 3.3 (run) An *initialized run* of a GCP $S = \langle \phi^0, \phi^\Box, G \rangle$ is a divergent state sequence $\bar{\sigma} = \sigma_0\sigma_1\sigma_2\ldots$ that satisfies three requirements:

1. *Initiation*: $\sigma_0 \models \phi^0$.
2. *Consecution*: For all $i \geq 0$, either $G(\sigma_i, \sigma_{i+1})$ or $\sigma_{i+1} = \sigma_i + \delta$ for some $\delta \in \mathsf{R}^+$.
3. *Invariant*: For all $i \geq 0$, $\sigma_i \models \phi^\Box$.

Any suffix of an initialized run of S is a *run* of S. We write Π_S for the set of runs of S. Two GCPs S_1 and S_2 are *equivalent* if they have the same runs (i.e., $\Pi_{S_1} = \Pi_{S_2}$). ■

Clearly, the set Π_S of runs of any GCP S is fusion-closed, suffix-closed, and divergence-safe. Furthermore, the stutter closure of the invariant $\phi^\Box$ implies the stutter closure of Π_S. Hence, Π_S is a divergence-safe real-time system.

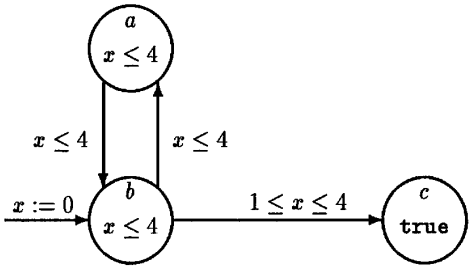
A transition of Π_S corresponds to the execution of a guarded command of the GCP S. GCPs can impose lower and upper bounds on the times of transitions. Lower bounds are enforced by the guards of the program body and upper bounds by the program invariant.

Example 3.1 Consider the GCP $S_1 = \langle \phi_1^0, \phi_1^\Box, G_1 \rangle$, where ℓ is a ternary variable (short for two boolean variables) that ranges over $\{a, b, c\}$, and x is a clock.

$$\begin{aligned}
\phi_1^0 &= (\ell = b \wedge x = 0);\\
\phi_1^\Box &= (\ell = a \wedge x \leq 4) \vee (\ell = b \wedge x \leq 4)\\
&\quad \vee (\ell = c);\\
G_1 &= \{\ell = a \rightarrow \ell := b,\\
&\quad \ell = b \rightarrow \ell := a,\\
&\quad (\ell = b \wedge x \geq 1) \rightarrow \ell := c\}.
\end{aligned}$$

This program starts with ℓ set to b and x set to 0. It then switches the value of ℓ between a and b arbitrarily (but finitely) often, until ℓ is assigned c at some point between time 1 and 4. The upper bound of 4 for this transition is imposed by the invariant. Once ℓ is set to c, the program does nothing but let time pass. ■

The real-time program S_1 of Example 3.1 can be graphically represented as follows:



This graph can be viewed as the transition diagram of a timed automaton [AD90] — a finite automata with a finite set of real-valued clocks (the reader familiar with timed automata will notice that we use a nonstandard variety that (1) permits clock constraints on both locations and transitions and (2) is interpreted over weakly monotonic time [AH91]). To make sure that a timed automaton defines a divergence-safe real-time system, we assume that all states are (Büchi) accepting. The resulting *timed safety automata* are defined in the full paper. There we also translate every GCP into a timed safety automaton, and every timed safety automaton into a GCP (with new propositions designating the locations of the automaton).

The real-time program of Example 3.1 can be executed in a stepwise fashion: start with $\ell = b \wedge x = 0$ and at any step, either pick a guarded command that is enabled or advance time without violating the invariant.

Definition 3.4 (stepwise execution) Let $S = \langle \phi^0, \phi^\Box, G \rangle$ be a GCP. For all states $\sigma, \sigma' \in \Sigma$, let $\sigma \triangleright_S \sigma'$ if $\sigma' \models \phi^\Box$ and there is some $\delta \in \mathsf{R}^+$ such that $\sigma + \delta \models \phi^\Box$ and $G^R(\sigma + \delta, \sigma')$. A *step sequence* of S is an infinite sequence $\hat{\sigma} = \sigma_0\sigma_1\sigma_2\ldots$ of states $\sigma_i \in \Sigma$ such that (1) $\sigma_0 \models \phi^0 \wedge \phi^\Box$ and (2) $\sigma_i \triangleright_S \sigma_{i+1}$ for all $i \geq 0$. We write S^{div}_{step} for the set of suffixes of divergent step sequences of S. ■

A GCP S can be executed step by step by beginning in an initial state (i.e., a state that satisfies both the initial condition and the invariant) and iterating the relation $\triangleright_S$. However, it is possible to write down a real-time program during whose execution such a simple-minded interpreter could paint itself into a corner and arrive at a state from which it cannot let time advance ever.

Example 3.2 Consider the GCP $S_2 = \langle \phi_2^0, \phi_2^\Box, G_2 \rangle$:

$$\begin{aligned}
\phi_2^0 &= (\ell = b \wedge x = 0);\\
\phi_2^\Box &= (\ell = a \wedge x \leq 5) \vee (\ell = b \wedge x \leq 5)\\
&\quad \vee (\ell = c);\\
G_2 &= \{\ell = a \rightarrow \ell := b,
\end{aligned}$$

$$\ell = b \;\rightarrow\; \ell := a,$$
$$(\ell = b \wedge 1 \leq x \leq 4) \;\rightarrow\; \ell := c\}.$$

It is easy to check that S_2 is equivalent to the program S_1 of Example 3.1. However, when executing S_2, a stepwise interpreter may decide to let time pass beyond 4 in a state with $\ell = a$ or $\ell = b$, in which case it enters a "Zeno" run that must converge before time 5. ■

Thus, the relation $\triangleright_S$ extracted directly from the text of a program $S = \langle \phi^0, \phi^\Box, G \rangle$ is in general not equal to its successor relation $\triangleright_{\Pi_S}$, which is defined on the model. We wish to avoid programs for which the two relations differ, because for programs S with $S^{div}_{step} = \Pi_S{}^{div}_{step}$ we can use the state predicates ϕ^0, $\phi^\Box$, and **pre**, which are extracted from the program text, to compute Π_S (which is the subset of state sequences in $\Pi_{S\,ex} = \Pi_S{}^{div}_{step}$).

Definition 3.5 (nonZeno) A GCP $S = \langle \phi^0, \phi^\Box, G \rangle$ is *nonZeno*[1] if

1. the initial condition implies the invariant (i.e., the state predicate $\phi^0 \Rightarrow \phi^\Box$ is valid);
2. the invariant is preserved by all guarded commands (i.e., the state predicate $\phi^\Box \Rightarrow \neg\mathbf{pre}(\neg\phi^\Box)$ is valid); and
3. any finite prefix of a run of S is a prefix of a divergent run of S.

The GCP S is *strongly nonZeno* if both S and $S' = \langle \phi^\Box, \phi^\Box, G \rangle$ are nonZeno. ■

The third condition guarantees that every nonZeno GCP S can be executed by iterating the relation $\triangleright_S$, beginning with any state that satisfies the initial condition:

Proposition 3.2 *For all* GCPs S, *if* S *is nonZeno then* $S^{div}_{step} = \Pi_S{}^{div}_{step}$.

In what follows, we assume that all GCPs we consider are nonZeno (i.e., "executable"). In the last section, we show that we can automatically convert any given real-time program into an equivalent nonZeno one.

4 Real-time Logics

In this section we compare two branching-time logics as specification formalisms for real-time systems.

[1] The term is taken from [AL91], where it is used in a different context for the same concept, namely, machine closure with respect to the divergence of time [Hen91].

4.1 The timed μ-calculus

We introduce a dense-time μ-calculus Tμ with clocks. While a discrete-time μ-calculus can rely on a standard unary next-time operator [Eme91], there is no unique "next time" when time is modeled by the real numbers. Instead, we use a binary *next* operator $\triangleright$ that can be viewed as a "single-step until" operator of temporal logic. The formulas of Tμ are built from state predicates by boolean operators, the next operator $\triangleright$, a reset operator for clocks, and a least-fixpoint operator. The *reset* operator "$z.$" binds and resets the clock variable z; it is inspired by the freeze quantifier of real-time logics [AH89].

Let V be a set of formula variables and let C' be a new set of clocks (i.e., $C \cap C' = \emptyset$).

Definition 4.1 (syntax of Tμ) The *formulas* φ of the timed μ-calculus Tμ are inductively defined as follows:

$$\varphi \;::=\; X \mid p \mid x + c \leq y + d \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \triangleright \varphi_2 \mid z.\varphi \mid \mu X.\varphi(X),$$

where $X \in V$, $p \in P$, $x, y \in C \cup C'$, $z \in C'$, and $c, d \in \mathbb{N}$. A Tμ-formula φ is *closed* if

1. every occurrence of a clock $z \in C'$ in φ is bound (i.e., it appears within the scope of a reset operator $z.$);
2. every occurrence of a formula variable X in φ is bound (i.e, it appears within the scope of a least-fixpoint operator $\mu X.$) and positive (i.e., it appears within an even number of negations from the operator $\mu X.$ that binds X).

We restrict ourselves to closed formulas of Tμ. ■

Note that we permit free occurrences of the clocks in C, which are given an interpretation in every state $\sigma \in \Sigma$; these variables can be used to refer to the clocks of a real-time program. Typical abbreviations include $\nu X.\varphi$ for $\neg\mu X.\neg\varphi(\neg X)$ and $x \leq 5$ for $y.(x \leq y + 5)$. Note that all state predicates are formulas of Tμ.

The formulas of Tμ are interpreted over states with respect to a given premodel. An environment $\mathcal{E}$ is a function that assigns to every formula variable $X \in V$ a set $\mathcal{E}(X) \subseteq \Sigma$ of states — those in which the formula X holds — and to every new clock $z \in C'$ a nonnegative real $\mathcal{E}(z) \in \mathbb{R}^+$. We write $\mathcal{E}[v := l]$ for the environment that agrees with the environment $\mathcal{E}$ on all variables in $V \cup C'$ except for v, which is assigned the value l. The environment $\mathcal{E} + \delta$, for $\delta \in \mathbb{R}^+$, agrees with the environment $\mathcal{E}$ on all formula variables and assigns the value $\mathcal{E}(z) + \delta$ to each clock $z \in C'$.

Definition 4.2 (semantics of Tμ) Given a premodel Π, a state $\sigma \in \Sigma$ *satisfies* the (closed) Tμ-formula φ in Π, denoted by $\sigma \models_\Pi \varphi$, if $\sigma \models_{\Pi,\mathcal{E}} \varphi$ for all environments $\mathcal{E}$:

$\sigma \models_{\Pi,\mathcal{E}} X$ iff $\sigma \in \mathcal{E}(X)$;

$\sigma \models_{\Pi,\mathcal{E}} p$ iff $\sigma(p) = \mathtt{true}$;

$\sigma \models_{\Pi,\mathcal{E}} x + c \leq y + d$ iff $f(x) + c \leq f(y) + d$, where $f(x) = \sigma(x)$ if $x \in C$ and $f(x) = \mathcal{E}(x)$ if $x \in C'$;

$\sigma \models_{\Pi,\mathcal{E}} \neg\varphi$ iff $\sigma \not\models_{\Pi,\mathcal{E}} \varphi$;

$\sigma \models_{\Pi,\mathcal{E}} \varphi_1 \vee \varphi_2$ iff $\sigma \models_{\Pi,\mathcal{E}} \varphi_1$ or $\sigma \models_{\Pi,\mathcal{E}} \varphi_2$;

$\sigma \models_{\Pi,\mathcal{E}} \varphi_1 \triangleright \varphi_2$ iff for some state $\sigma' \in \Sigma$ and $\delta \in \mathsf{R}^+$, $\sigma \triangleright^{\delta}_{\Pi} \sigma'$ and $\sigma' \models_{\Pi,\mathcal{E}+\delta} \varphi_2$, and for all $0 \leq \delta' \leq \delta$, $\sigma + \delta' \models_{\Pi,\mathcal{E}+\delta'} \varphi_1 \vee \varphi_2$;

$\sigma \models_{\Pi,\mathcal{E}} z.\varphi$ iff $\sigma \models_{\Pi,\mathcal{E}[z:=0]} \varphi$;

$\sigma \models_{\Pi,\mathcal{E}} \mu X.\varphi$ iff
$\sigma \in \bigcap\{\Sigma' \subseteq \Sigma \mid \{\sigma' \in \Sigma \mid \sigma' \models_{\Pi,\mathcal{E}[X:=\Sigma']} \varphi\} \subseteq \Sigma'\}$.

A Tμ-formula φ is *satisfiable* if $\sigma \models_\Pi \varphi$ for some real-time system Π and some state $\sigma \in \Sigma$. ■

Instead of the standard unary next-time operator used in μ-calculi with discrete models [Koz83, Eme91], Tμ relies on the binary next operator $\triangleright$ (a similar operator has been proposed for a dense-time extension of Hennessy-Milner logic [HLY91]). The formula $\varphi_1 \triangleright \varphi_2$ holds in a state σ of a premodel iff there is a state sequence from σ along which $\varphi_1 \vee \varphi_2$ holds *until the next transition is taken*, after which φ_2 becomes true. The next transition may be arbitrarily close in time or arbitrarily far away. The requirement that the disjunction $\varphi_1 \vee \varphi_2$ holds in all intermediate states — rather than φ_1, as is standard for until operators — is caused by density of the time domain. For example, we want the formula $(x \leq 5) \triangleright (x > 5)$ to be satisfied in a state σ with $\sigma(x) = \delta \leq 5$ iff time may advance by more than $5 - \delta$.

Example 4.1 Consider the formula

$$\varphi \;=\; x.(p \triangleright (q \wedge y.\ x \leq y + 2)).$$

It asserts that either the system is in a state in which q holds, or p holds and a q-state can be reached in a single step within time 2. ■

4.2 Timed computation tree logic

Many important system properties find a natural expression in temporal logic.We now review the real-time temporal logic TCTL [Alu91], which extends the branching-time logic CTL [CE81] with clocks. The formulas of TCTL are built from state predicates by boolean connectives, the two temporal operators $\exists\mathcal{U}$ (*possibly*) and $\forall\mathcal{U}$ (*inevitably*), and the reset operator for clocks. Intuitively, the formula $\varphi_1 \exists\mathcal{U} \varphi_2$ holds in state σ iff along some state sequence from σ, φ_2 becomes true and φ_1 is true in all intermediate states; the formula $\varphi_1 \forall\mathcal{U} \varphi_2$ asserts that along every state sequence from σ, φ_1 is true until φ_2 becomes true.

Definition 4.3 (syntax of TCTL) The *formulas* of TCTL are inductively defined as follows:

$$\varphi \;::=\; p \mid x + c \leq y + d \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \exists\mathcal{U} \varphi_2 \mid \varphi_1 \forall\mathcal{U} \varphi_2 \mid z.\varphi,$$

where $p \in P$, $x, y \in C \cup C'$, $z \in C'$, and $c, d \in \mathsf{N}$. A TCTL-formula φ is *closed* if every occurrence of a clock $z \in C'$ in φ is bound. We restrict ourselves to closed formulas of TCTL. ■

Typical abbreviations include $\forall\Diamond\varphi$ and $\exists\Box\varphi$ for $\mathtt{true}\forall\mathcal{U}\varphi$ and $\neg\forall\Diamond\neg\varphi$, respectively. The formulas of TCTL are interpreted over states with respect to a given premodel. In the case of TCTL, an environment $\mathcal{E}$ is a function that assigns to every new clock $z \in C'$ a nonnegative real value $\mathcal{E}(z) \in \mathsf{R}^+$.

Definition 4.4 (semantics of TCTL) Given a premodel Π, a state $\sigma \in \Sigma$ *satisfies* the (closed) TCTL-formula φ in Π, denoted by $\sigma \models_\Pi \varphi$, if $\sigma \models_{\Pi,\mathcal{E}} \varphi$ for all environments $\mathcal{E}$:

$\sigma \models_{\Pi,\mathcal{E}} p$ iff $\sigma(p) = \mathtt{true}$;

$\sigma \models_{\Pi,\mathcal{E}} x + c \leq y + d$ iff $f(x) + c \leq f(y) + d$, where $f(x) = \sigma(x)$ if $x \in C$ and $f(x) = \mathcal{E}(x)$ if $x \in C'$;

$\sigma \models_{\Pi,\mathcal{E}} \neg\varphi$ iff $\sigma \not\models_{\Pi,\mathcal{E}} \varphi$;

$\sigma \models_{\Pi,\mathcal{E}} \varphi_1 \vee \varphi_2$ iff $\sigma \models_{\Pi,\mathcal{E}} \varphi_1$ or $\sigma \models_{\Pi,\mathcal{E}} \varphi_2$;

$\sigma \models_{\Pi,\mathcal{E}} \varphi_1 \exists\mathcal{U} \varphi_2$ iff for some state sequence $\bar{\sigma} \in \Pi$ with $\sigma_0 = \sigma$, there exists an $n \geq 0$ such that $\sigma_n \models_{\Pi,\mathcal{E}+\tau_n} \varphi_2$ and for all $0 \leq i < n$ and $0 \leq \delta \leq \delta_i$, $\sigma_i + \delta \models_{\Pi,\mathcal{E}+\tau_i+\delta} \varphi_1 \vee \varphi_2$;

$\sigma \models_{\Pi,\mathcal{E}} \varphi_1 \forall\mathcal{U} \varphi_2$ iff for all state sequences $\bar{\sigma} \in \Pi$ with $\sigma_0 = \sigma$, there exist $n \geq 0$ and $0 \leq \delta \leq \delta_n$ such that $\sigma_n + \delta \models_{\Pi,\mathcal{E}+\tau_n+\delta} \varphi_2$ and for all $0 \leq i < n$ and $0 \leq \delta \leq \delta_i$, $\sigma_i + \delta \models_{\Pi,\mathcal{E}+\tau_i+\delta} \varphi_1 \vee \varphi_2$;

$\sigma \models_{\Pi,\mathcal{E}} z.\varphi$ iff $\sigma \models_{\Pi,\mathcal{E}[z:=0]} \varphi$.

A TCTL-formula φ is *satisfiable* if $\sigma \models_\Pi \varphi$ for some real-time system Π and some state $\sigma \in \Sigma$. ■

A few comments regarding our version of TCTL are in order. First, the time-bounded temporal operators used in [ACD90] can be expressed in TCTL. Consider, for instance, the formula $\varphi_1 \exists\mathcal{U}_{\leq 2} \varphi_2$, which is true in a state σ iff some state sequence from σ has a finite prefix whose last state is within time 2 of σ and satisfies φ_2, and all intermediate states satisfy $\varphi_1 \vee \varphi_2$. This requirement is expressible in TCTL as

$$z.(\varphi_1 \exists\mathcal{U}(z \leq 2 \wedge \varphi_2)).$$

Thus we may use time-bounded temporal operators as abbreviations, such as $\forall\Diamond_{>5}\varphi$ for $z.\forall\Diamond(z > 5 \wedge \varphi)$. Second, in our version of TCTL freeze quantifiers on static time variables are replaced by reset operators on dynamic clock variables. Both approaches are equivalent [Alu91], but the clock-reset style fits in better with our definition of real-time programs. Third, we have not added the next operator $\triangleright$ of Tμ to TCTL. It is not hard to see that if the set $P \cup C$ of propositions and clocks is, as we have assumed, finite, then the next operator is definable in terms of the operator $\exists\mathcal{U}$.

4.3 Expressiveness of Tμ versus TCTL

We now compare the expressive power of the timed μ-calculus Tμ with the expressive power of the timed computation tree logic TCTL over premodels and real-time systems.

Definition 4.5 (characteristic set) A state $\sigma \in \Sigma$ of a premodel Π is *reachable* in Π if σ occurs on some state sequence of Π. We write Σ_Π for the reachable states of the premodel Π.

Given a premodel Π and a (closed) formula φ of Tμ or TCTL, the *characteristic set* $[\![\varphi]\!]_\Pi$ of φ in Π is the set of reachable states that satisfy φ in Π:

$$[\![\varphi]\!]_\Pi = \{\sigma \in \Sigma_\Pi \mid \sigma \models_\Pi \varphi\}. \blacksquare$$

Throughout this section, when interpreting a formula of Tμ or TCTL over a premodel Π, we sometimes assume that instead of closed subformulas in a particular language, we are given characteristic sets in Π (i.e., language-independent sets of reachable states). If $s \subseteq \Sigma_\Pi$ is a characteristic set in the premodel Π, then for all environments $\mathcal{E}$ and states $\sigma \in \Sigma$, $\sigma \models_{\Pi,\mathcal{E}} s$ iff $\sigma \in s$.

Definition 4.6 (expressiveness) We say that logic $\mathcal{A}$ is *as expressive in the weak (strong) sense as* logic $\mathcal{B}$ if for every $\mathcal{B}$-formula φ_B there is an $\mathcal{A}$-formula φ_A such that $[\![\varphi_A]\!]_\Pi = [\![\varphi_B]\!]_\Pi$ for all real-time systems (premodels) Π. ■

In the untimed case, the branching-time logic CTL is strictly less expressive than the propositional μ-calculus [EC80]. We show that in the dense-time case, the expressive powers of TCTL and the Tμ are incomparable, both over premodels and real-time systems. The following proposition is instrumental for characterizing the expressive power of the timed μ-calculus: Tμ cannot distinguish between premodels with the same safety closure.

Proposition 4.1 *For all premodels* Π_1 *and* Π_2 *and* Tμ*-formulas* φ*, if* $\overline{\Pi}_1 = \overline{\Pi}_2$ *then* $[\![\varphi]\!]_{\Pi_1} = [\![\varphi]\!]_{\Pi_2}$.

We apply Proposition 4.1 to two examples. First, consider the TCTL-formula $\forall\Box z.\forall\Diamond(z = 1)$. This formula is satisfied in a state σ iff time diverges along all state sequences from σ; it is satisfied by all states in a given premodel Π iff Π is a real-time system. From Proposition 4.1 it follows that this property cannot be expressed in Tμ in the strong sense: suppose that there is some Tμ-formula φ asserting that a premodel Π is a real-time system, then by Proposition 4.1, $\overline{\Pi}$ is also a real-time system, which is a contradiction (no real-time system is safe).

Second, consider the real-time system Π_2 of Example 2.1, which eventually changes the value of the proposition p from `false` to `true`. Clearly, $\overline{\Pi}_2$ contains a state sequence $\bar{\sigma}$ such that $\sigma_i(p) = \texttt{false}$ for all $i \geq 0$. Thus, while all states satisfy the TCTL-formula $\forall\Diamond p$ in Π_2, this it not the case for $\overline{\Pi}_2$. From Proposition 4.1 it follows that this property cannot be expressed in Tμ in the weak sense.

Hence, we have one direction of Theorem 4.1. The other direction can be shown similar to the standard argument that CTL is strictly less expressive than the propositional μ-calculus [EC80, EH86].[2]

Theorem 4.1 Tμ *is not as expressive (in both the weak and the strong sense) as* TCTL *and vice versa.*

Nevertheless, there are some encouraging results. The following proposition shows that the temporal operator $\exists\mathcal{U}$ is in the strong sense expressible as a least fixpoint. In fact, the resulting Tμ-formula is similar to the translation of untimed $\exists\mathcal{U}$ into the μ-calculus.

Proposition 4.2 *For all premodels* Π *and characteristic sets* $s_1, s_2 \subseteq \Sigma_\Pi$,

$$[\![s_1 \exists\mathcal{U} s_2]\!]_\Pi = [\![\mu X.(s_2 \vee (s_1 \triangleright X))]\!]_\Pi.$$

[2] A Tμ-formula that in the strong sense is not expressible in TCTL is given in Section 5.2.

Our argument for Theorem 4.1 showed that the temporal operator $\forall\mathcal{U}$ cannot be expressed as a Tμ-formula. Thus, while we can use the fixpoint characterization for $\exists\mathcal{U}$ (Proposition 4.2) to compute possibility properties ($\exists\Diamond$) of real-time systems, in general we cannot use a fixpoint approach to compute inevitability properties ($\forall\Diamond$). However, the real-time system Π_2 from Example 2.1, which we used to distinguish between Tμ and TCTL in the weak sense, is not divergence-safe. We now show in two steps that *for divergence-safe real-time systems* there is a fixpoint characterization of inevitability.

In the first step, we observe that time-bounded inevitability ($\forall\Diamond_{\leq c}$) is in the weak sense expressible by the (negated) possibility operator $\exists\mathcal{U}$. Intuitively, this is because in all divergent state sequences a state σ is inevitable within time c iff it is not possible that more than time c passes without σ occurring (which need not be the case for nondivergent sequences).

Proposition 4.3 *For all real-time systems* Π, *characteristic sets* $s_1, s_2 \subseteq \Sigma_\Pi$, *and integer constants* $c \in \mathsf{N}$,

$$[\![s_1 \forall\mathcal{U}_{\leq c}\, s_2]\!]_\Pi \;=\; [\![\neg z.((\neg s_2)\exists\mathcal{U}(\neg(s_1 \vee s_2) \vee z > c))]\!]_\Pi.$$

Put together, Propositions 4.2 and 4.3 imply that over real-time systems, the TCTL-formula $s_1 \forall\mathcal{U}_{\leq c}\, s_2$ is expressible in Tμ. In the second step, we show that for divergence-safe real-time systems, unbounded inevitability is expressible in terms of time-bounded inevitability:

Proposition 4.4 *For all divergence-safe real-time systems* Π, *characteristic sets* $s_1, s_2 \subseteq \Sigma_\Pi$, *and integer constants* $c \geq 1$,

$$[\![s_1 \forall\mathcal{U} s_2]\!]_\Pi \;=\; [\![\mu X.(s_2 \vee (s_1 \forall\mathcal{U}_{\leq c}\, X))]\!]_\Pi.$$

Propositions 4.2, 4.3 and 4.4 prescribe a method for computing, over divergence-safe real-time systems, all TCTL-formulas as fixpoints of Tμ. Hence we conclude the following theorem.

Theorem 4.2 *Let* φ *be a* TCTL*-formula. If* Π *is a divergence-safe real-time system, then there is a* Tμ*-formula* φ' *such that* $[\![\varphi]\!]_\Pi = [\![\varphi']\!]_\Pi$.

Furthermore, Propositions 4.2 and 4.3 imply that the timed μ-calculus is undecidable. The satisfiability problem for TCTL is known to be undecidable, because TCTL can encode the halting problem for Turing machines [Alu91]. By inspecting the proof we observe that already bounded temporal operators and unbounded $\exists\Diamond$ suffice for the encoding. The following result follows from our reduction of TCTL-formulas to Tμ.

Theorem 4.3 *The satisfiability problem for* Tμ *is undecidable.*

We are, however, not interested in general satisfiability, but rather satisfiability in a given real-time system. This question can be answered by model checking.

5 Symbolic Model Checking

The constructive proof of Theorem 4.2 provides a method for symbolic model checking of TCTL-properties over divergence-safe real-time systems, provided that the next relation $\triangleright$ and all fixpoints are computable. In this section we describe such an algorithm for the symbolic model checking of real-time systems that are given by guarded-command real-time programs (or timed safety automata).

Definition 5.1 (correctness) A real-time system Π is *correct* with respect to the TCTL-formula φ if $\sigma \models_\Pi \varphi$ for all reachable states $\sigma \in \Sigma_\Pi$. ■

Let $S = \langle \phi^\Box, \phi^\Box, G\rangle$ be a nonZeno GCP (without initial condition) and let φ be a formula of TCTL. The program S meets the specification φ iff Π_S is correct with respect to φ; that is, φ holds in all states that can be reached by executing S. We show by construction that for any TCTL-formula φ, the set $[\![\varphi]\!]_{\Pi_S}$ of states that are reachable in Π_S and satisfy φ, can be defined by a state predicate $|\varphi|$, the *characteristic predicate* of φ in Π_S; that is, $[\![|\varphi|]\!] = [\![\varphi]\!]_{\Pi_S}$. Now let S′ be a GCP that is obtained from S by introducing an initial condition ϕ^0 that implies $\phi^\Box$. We can verify that the real-time system $\Pi_{S'}$ is correct with respect to the specification φ by first computing the characteristic predicate $|\forall\Box\varphi|$ in Π_S and then checking the validity of the state predicate

$$\phi^0 \Rightarrow |\forall\Box\varphi|.$$

We compute characteristic predicates inductively and use the Tarski-Knaster Theorem to approximate fixpoints in the standard way:

Algorithm 5.1 (symbolic model checking)

Input: a GCP $S = \langle \phi^0, \phi^\Box, G\rangle$ and a TCTL-formula φ.

Output: the state predicate $|\varphi|$.

$|p|$ is $\phi^\Box \wedge p$;

$|x + c \leq y + d|$ is $\phi^\Box \wedge x + c \leq y + d$;

$|\neg\varphi|$ is $\neg|\varphi|$;

$|\varphi_1 \vee \varphi_2|$ is $|\varphi_1| \vee |\varphi_2|$;

$|\varphi_1 \triangleright \varphi_2|$ is $\exists \delta \in \mathsf{R}^+.((\phi^\Box \wedge \mathbf{pre}(|\varphi_2|)) + \delta \wedge \forall \delta' \in \mathsf{R}^+.(\delta' < \delta \Rightarrow (|\varphi_1| \vee |\varphi_2|) + \delta'))$;

$|z.\varphi|$ is $|\varphi|[z := 0]$;

$|\varphi_1 \exists\mathcal{U} \varphi_2|$ is $\bigvee_{i \geq 0} \psi_i$, where ψ_0 is $|\varphi_2|$ and ψ_{i+1} is $|\psi_i \vee (|\varphi_1| \triangleright \psi_i)|$;

$|\varphi_1 \forall\mathcal{U} \varphi_2|$ is $\bigvee_{i \geq 0} \psi_i$, where ψ_0 is $|\varphi_2|$ and ψ_{i+1} is $|\psi_i \vee \neg z.((\neg\psi_i)\exists\mathcal{U}(\neg(|\varphi_1| \vee \psi_i) \vee z > 1^3))|$. ■

It is not obvious from this abstract description of Algorithm 5.1 that the metaexpression used to define $|\varphi_1 \triangleright \varphi_2|$ is equivalent to a state predicate for all state predicates $|\varphi_1|$ and $|\varphi_2|$. To prove the correctness of Algorithm 5.1 we must show that for any GCP $S = \langle \phi^\Box, \phi^\Box, G \rangle$ and TCTL-formula φ, (1) the algorithm terminates, (2) the output $|\varphi|$ is a state predicate, and (3) if S is nonZeno, then $[\![|\varphi|]\!] = [\![\varphi]\!]_{\Pi_S}$.

The correctness proof makes use of the region graph construction on which the graph-theoretic algorithm for model checking is based [ACD90]. First we show that given an input formula φ of TCTL, all intermediate results of the algorithm are state predicates whose largest integer constant K is bounded by the largest constants in the program S and the specification φ. Let Φ_K be a minimal set of state predicates such that every state predicate without constants larger than K is equivalent to some predicate in Φ_K. Every state predicate in Φ_K symbolically represents the union of any number of regions in the region graph. Then Φ_K is finite. Consequently, all fixpoint computations must converge within a finite number of iterations and Algorithm 5.1 must terminate. The partial correctness of the algorithm follows from the results of Section 4.

Theorem 5.1 *For every nonZeno* GCP $S = \langle \phi^\Box, \phi^\Box, G \rangle$ *and* TCTL*-formula* φ*, Algorithm 5.1 computes a state predicate* ϕ *such that* $[\![\phi]\!] = [\![\varphi]\!]_{\Pi_S}$.

Corollary 5.1 *For every strongly nonZeno* GCP $S = \langle \phi^0, \phi^\Box, G \rangle$*, the real-time system* Π_S *is correct with respect to the* TCTL*-formula* φ *iff the state predicate* $\phi^0 \Rightarrow |\forall\Box\varphi|$ *is valid (where the state predicate* $|\forall\Box\varphi|$ *is computed by Algorithm 5.1).*

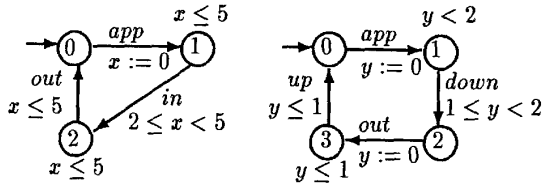
The practicality of the method depends on the particular representation of state predicates and on the computation of the next relation $\triangleright$. For instance, Algorithm 5.1 can be used to perform model checking directly on the exponential region graph. In fact, in the worst case we can do no better, because the problem of whether a GCP meets a TCTL-specification is PSPACE-complete [ACD90]. The importance of the suggested method, however, resides in the fact that in many cases we may avoid the costly construction of the region graph by representing unions of regions symbolically as state predicates and by using the predicate transformer **pre**.

It is straightforward to generalize Algorithm 5.1 for checking any formula of Tμ [EL86, CS91].

[3] Or any positive integer. This choice may effect the number of iterations necessary.

5.1 A verification example

Consider the gate controller of a railroad crossing. The system consists of two parallel processes, namely, the train and the controller:



The train is described by a timed safety automaton with three states, which we encode using the ternary variable S: $S = 1$ if the train is approaching the railroad crossing; $S = 2$ if the train is crossing; $S = 0$ otherwise. When the train approaches, it sends the signal *app* at least 2 minutes before it enters the crossing. The event *out* marks the exit of the train from the crossing. Furthermore, we know that the train leaves the crossing at most 5 minutes after it signals *app*.

The controller is described by a timed safety automaton with four states, which we encode using the variable R: $R = 0$ if the controller is waiting for the train to arrive; $S = 1$ if the train is approaching; $S = 2$ if the gate is down; $S = 3$ if the train has left. Whenever the controller receives the signal *app* from the train, it closes the gate. This event takes at least 1 minute but less than 2 minutes. Then the controller waits for the train to send the signal *out*. When the signal arrives, the controller responds by opening the gate within 1 minute.

The entire system can be described by the following strongly nonZeno GCP $S = \langle \phi^0, \phi^\Box, G \rangle$, which represents the parallel composition of the two timed automata:

$$
\begin{aligned}
\phi^0 &= (S = 0 \wedge x = 0 \wedge R = 0 \wedge y = 0);\\
\phi^\Box &= (S = 0 \vee ((S = 1 \vee S = 2) \wedge x \leq 5)) \wedge\\
&\quad (R = 0 \vee (R = 1 \wedge y < 2) \vee\\
&\quad R = 2 \vee R = 3 \wedge y \leq 1));\\
G &= \{(S = 0 \wedge R = 0) \rightarrow_{app} S, R := 1; x, y := 0,\\
&\quad (S = 1 \wedge 2 < x \leq 5) \rightarrow_{in} S := 2,\\
&\quad (R = 1 \wedge 1 \leq y < 2) \rightarrow_{down} R := 2,
\end{aligned}
$$

$$(S = 2 \wedge x \leq 5 \wedge R = 2) \rightarrow_{out}$$
$$S := 0; R := 3; y := 0,$$
$$(R = 3 \wedge y \leq 1) \rightarrow_{up} R := 0\}.$$

We want to verify that the gate is never closed for more than 5 minutes. This property is expressed by the TCTL-formula

$$\varphi = \forall\Box(R = 2 \Rightarrow \forall\Diamond_{\leq 5} R = 0),$$

which over divergence-safe real-time systems is equivalent to the Tμ-formula

$$\neg\mu Y.((R = 2 \wedge z.\mu X.(z > 5 \vee ((R \neq 0) \triangleright X))) \vee (\mathbf{true} \triangleright Y)).$$

The state predicate that defines the innermost least fixpoint in this formula is computed iteratively as $X = \bigvee_i X_i$, where $X_0 = \phi^{\Box} \wedge z > 5$ and $X_{i+1} = X_i \vee ((R \neq 0) \triangleright X_i)$. We have:

$$\begin{aligned}
X_1 &= X_0 \vee R \neq 0 \triangleright X_0 \\
&= \phi^{\Box} \wedge z > 5 \vee \\
&\quad (S = 0 \vee (S = 1 \vee S = 2) \wedge x \leq 5 \wedge z > x) \\
&\quad \wedge (R = 1 \wedge y < 2 \wedge z > y + 3 \vee R = 2 \\
&\quad \vee R = 3 \wedge y \leq 1 \wedge z > y + 4); \\
X_2 &= X_1 \vee R \neq 0 \triangleright X_1 \\
&= X_1 \vee S = 0 \wedge R = 1 \wedge y < 2 \vee \\
&\quad (S = 1 \vee S = 2) \wedge R = 1 \wedge x \leq 5 \\
&\quad \wedge y < 2 \wedge z > x \wedge y \geq x - 4 \\
&\quad \vee S = 2 \wedge R = 2 \wedge x \leq 5 \wedge z > x - 1 \\
X_3 &= X_2 \vee R \neq 0 \triangleright X_2; \\
&= X_2 \vee x \leq 5 \wedge z > x - 1 \wedge (S = 1 \wedge R = 2 \\
&\quad \vee S = 2 \wedge R = 1 \wedge y < 2 \wedge y \geq x - 4) \\
X_4 &= X_3 \vee R \neq 0 \triangleright X_3; \\
&= X_3 \vee S = 1 \wedge R = 1 \wedge x \leq 5 \\
&\quad \wedge z > x - 1 \wedge y < 2 \wedge y \geq x - 4.
\end{aligned}$$

We find that $X = X_5 = X_4$ and the computation terminates. Next we compute the reset operator:

$$\begin{aligned}
z.X &= X[z := 0] \\
&= (S = 0 \vee (S = 1 \vee S = 2) \wedge x < 1) \\
&\quad \wedge (R = 1 \wedge y < 2 \vee R = 2).
\end{aligned}$$

Now we compute the outermost least fixpoint iteratively as $Y = \bigvee_i Y_i$, where $Y_0 = (R = 2 \wedge z.X)$ and $Y_{i+1} = Y_i \vee (\mathbf{true} \triangleright Y_i)$. This computation converges in the second iteration with $Y = Y_2 = Y_1$:

$$\begin{aligned}
Y &= Y_0 \vee (S = 0 \vee (S = 1 \vee S = 2) \wedge \\
&\quad x < 1 \wedge x < y) \wedge R = 1 \wedge y < 2.
\end{aligned}$$

Finally, it easy to check that the initial condition ϕ^0 implies $\varphi = \neg Y$ as required.

5.2 Symbolic nonZenoness analysis

The invariant $\phi^{\Box}$ and the predicate transformer **pre** used in Algorithm 5.1 provide the correct next relation $\triangleright$ only for nonZeno GCPs. Recall, for instance, the Zeno GCP S_2 from Example 3.2. Using Algorithm 5.1 with the invariant $\phi_2^{\Box}$ of S_2, we find, mistakenly, that the state $\ell = a \wedge x = 5$ is reachable from the initial state of S_2; that is,

$$(\ell = b \wedge x = 0) \Rightarrow |\exists\Diamond(\ell = a \wedge x = 5)|$$

is valid. Rather we ought to use the stronger invariant $\phi_1^{\Box}$ of the equivalent nonZeno GCP S_1 from Example 3.1. In fact, ideally we would like to convert S_2 automatically into S_1 before extracting the invariant and the **pre** operator. We show that we can apply the timed μ-calculus and Algorithm 5.1 to do precisely that; namely, to (1) check if a given GCP is nonZeno and, if not, to (2) convert it into an equivalent nonZeno GCP. Then we may prove properties of the corresponding real-time system by using $\phi^{\Box}$ and **pre** of the nonZeno program.

Proposition 5.1 *A* GCP $S = (\phi^0, \phi^{\Box}, G)$ *is nonZeno iff the state predicate* $\phi^0 \Rightarrow |\forall\Box\exists\Diamond_{=1}\mathbf{true}|$ *is valid (where the state predicate* $|\forall\Box\exists\Diamond_{=1}\mathbf{true}|$ *is computed by Algorithm 5.1).*

To convert a Zeno program into an equivalent nonZeno one, we need to find all states from which time cannot diverge. This Zeno condition is expressed by the negation of the temporal formula

$$\varphi_{NZ} = \exists\Box\Diamond_{=1}\mathbf{true},$$

which is not a TCTL-formula, but rather a formula of TCTL*. In the full paper, we define the real-time logic TCTL* as an extension of TCTL analogous to the extension CTL* [EH86] of CTL. The formula φ_{NZ} is expressible in Tμ in the strong sense, as $\nu X.\exists\Diamond_{=1}X$. Strong expressibility allows us to compute the characteristic predicate of the formula φ_{NZ} with Algorithm 5.1 by using the invariant and **pre** operator of a Zeno program. Once the set of "bad" (Zeno) states is isolated, it is easy to strengthen the initial condition, the invariant, and the guards appropriately to avoid these states. The method is demonstrated in the full paper.

References

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proc. 5th IEEE Symp. on Logic in Computer Science*, IEEE Computer Society Press, 1990, pp. 414–425.

[AD90] R. Alur and D. Dill. Automata for modeling real-time systems. In *Proc. 17th ICALP*, Lecture Notes in Computer Science 443, Springer-Verlag, 1990, pp. 322–335.

[ADS86] B. Alpern, A. Demers, and F. Schneider. Safety without stuttering. *Information Processing Letters* 23(4), 1986, pp. 177–180.

[AH89] R. Alur and T. Henzinger. A really temporal logic. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, IEEE Computer Society Press, 1989, pp. 164–169.

[AH91] R. Alur and T. Henzinger. Logics and models of real time: a survey. In *Real-Time: Theory in Practice*, Lecture Notes in Computer Science 600, Springer-Verlag, 1991.

[AL91] M. Abadi and L. Lamport. An old-fashioned recipe for real time. In *Real-Time: Theory in Practice*, Lecture Notes in Computer Science 600, Springer-Verlag, 1991.

[Alu91] R. Alur. *Techniques for Automatic Verification of Real-time Systems.* PhD thesis, Dept. of Computer Science, Stanford University, 1991.

[BCD*90] J. Burch, E. Clarke, D. Dill, L. Hwang, and K. McMillan. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th IEEE Symp. on Logic in Computer Science*, IEEE Computer Society Press, 1990, pp. 428–439.

[Bry86] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers* C-35(8), 1986.

[CBG*91] E. Clarke, J. Burch, O. Grumberg, D. Long, and K. McMillan. Automatic verification of sequential circuit designs. Presented at *Royal Society of London*, 1991.

[CBM89] O. Coudert, C. Berthet, and J. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proc. 1st Workshop on Computer-Aided Verification*, Lecture Notes in Computer Science 407, Springer-Verlag, 1989.

[CE81] E. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proc. Workshop on Logic of Programs*, Lecture Notes in Computer Science 131, Springer-Verlag, 1981.

[CES86] E. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. on Programming Languages and Systems* 8(2), 1986, pp. 244–263.

[CG87] E. Clarke and O. Grumberg. Research on automatic verification of finite-state concurrent systems. In *Annual Review of Computer Science*, vol. II (J. Traub, B. Grosz, B. Lampson, and N. Nilsson, eds.), Annual Reviews, 1987. pp. 269–290.

[CS91] R. Cleaveland and B. Steffen. A linear-time model checking algorithm for the alternation-free modal μ-calculus. In *Proc. 3rd Workshop on Computer-Aided Verification*, Lecture Notes in Computer Science, Springer-Verlag, 1991.

[EC80] E.A. Emerson and E. Clarke. Characterizing correctness properties of parallel programs as fixpoints. In *Proc. 7th ICALP*, Lecture Notes in Computer Science 85, Springer-Verlag, 1980, pp. 169–181.

[EH86] E.A. Emerson and J. Halpern. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM* 33(1), 1986, pp. 151–178.

[EL86] E.A. Emerson and C. Lei. Efficient model checking in fragments of the propositional μ-calculus. In *Proc. 1st IEEE Symp. on Logic in Computer Science*, IEEE Computer Society Press, 1986, pp. 267–278.

[Eme91] E.A. Emerson. Real time and the μ-calculus. In *Real-Time: Theory in Practice*, Lecture Notes in Computer Science 600, Springer-Verlag, 1991.

[EMSS89] E.A. Emerson, A. Mok, A.P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. Presented at *Proc. 1st Workshop on Computer-Aided Verification*, 1989.

[Hen91] T. Henzinger. *The Temporal Specification and Verification of Real-time Systems.* PhD thesis, Dept. of Computer Science, Stanford University, 1991.

[HLY91] U. Holmer, K. Larsen, and W. Yi. Deciding properties of regular real timed processes. In *Proc. 3rd Workshop on Computer-Aided Verification*, Lecture Notes in Computer Science, Springer-Verlag, 1991.

[HMP91] T. Henzinger, Z. Manna, and A. Pnueli, Temporal proof methodologies for real-time systems. In *Proc. 18th ACM Symp. on Principles of Programming Languages*, ACM Press, 1991, pp. 353–366.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science* 27(3), 1983, pp. 333–354.

[Lew90] H. Lewis. A logic of concrete time intervals. In *Proc. 5th IEEE Symp. on Logic in Computer Science*, IEEE Computer Society Press, 1990, pages 380–389.

[NS91] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *Proc. 3rd Workshop on Computer-Aided Verification*, Lecture Notes in Computer Science, Springer-Verlag, 1991.

[NSY91] X. Nicollin, J. Sifakis, and S. Yovine. From ATP to timed graphs and hybrid systems. In *Real-Time: Theory in Practice*, Lecture Notes in Computer Science 600, Springer-Verlag, 1991.

[QS81] J. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int. Symp. in Programming*, Lecture Notes in Computer Science 137, Springer-Verlag, 1981, pp. 337–351.

[Sif82] J. Sifakis. A unified approach for studying the properties of transition systems. *Theoretical Computer Science* 18, 1982.