# Deductive Verification of Real-time Systems Using STeP

Nikolaj S. Bjørner, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe

*Computer Science Department, Stanford University*
*Stanford, CA. 94305*
{nikolaj|manna|sipma|uribe}@cs.stanford.edu

**Abstract**

We present a modular framework for proving temporal properties of real-time systems, based on clocked transition systems and linear-time temporal logic. We show how deductive verification rules, verification diagrams, and automatic invariant generation can be used to establish properties of real-time systems in this framework. We also discuss global and modular proofs of the branching-time property of non-Zenoness. As an example, we present the mechanical verification of the *generalized railroad crossing* case study using the Stanford Temporal Prover, STeP.

*Keywords:* real-time systems, temporal verification, modularity, receptiveness

## 1 Introduction

Many formalisms have been proposed to model real-time systems. Most are restricted to systems where time is the only variable with infinite domain (also called real-time systems with finite control). When real-time systems include software, we would like to analyze them using frameworks that can handle infinite data domains other than time. Similarly, it is generally accepted that larger and more complex systems require modular analysis techniques; however, the combination of modularity and real-time poses particular challenges. This paper explores the modular verification of infinite-control real-time systems in a general deductive setting. We show how computer-aided tools for

⋆ This research was supported in part by the National Science Foundation under grant CCR-95-27927, the Defense Advanced Research Projects Agency under NASA grant NAG2-892, ARO under grant DAAH04-95-1-0317, ARO under MURI grant DAAH04-96-1-0341, and by Army contract DABT63-96-C-0096 (DARPA).

deductive and deductive-algorithmic verification, including verification diagrams, decision procedures, and invariant generation, are used to analyze such systems.

*Clocked transition systems* were introduced by Manna and Pnueli [MP96] as a simple but general model for real-time systems. Continuous real-time is modeled explicitly as part of the system, using clock variables that increase uniformly, including a master clock that measures the passage of time. This model, in the spirit of [AL92], allows the reuse of standard deductive verification tools for discrete reactive systems [KMP96].

We add modularity to this framework, defining *clocked transition modules*. Systems often have a natural decomposition into modules, where useful properties of the entire system may be inferred from properties of the modules. Properties of the composed system will still hold if modules are replaced by new ones that satisfy the required modular properties.

Verification at the module level has several advantages: the number of verification conditions is smaller, module properties are often more intuitive, and modules may be replaced while reusing the same proofs for the global system and other modules, provided the new modules satisfy the same properties. Properties of each module can be proved independently and later on used to prove properties of larger systems. Properties of a module can be established using deductive or algorithmic techniques whenever they are applicable.

Properties verified over a real-time system may be misleading if the system description is *Zeno*, that is, it contains computation prefixes that cannot be extended to computations in which time can grow beyond any bound. Therefore, verification of real-time systems should always be preceded by a check that the system description is *non-Zeno*. This check may also be performed at the module level, in which case we prove the stronger property of *receptiveness*. We propose diagrams that reduce proofs of non-Zenoness and receptiveness to a set of first-order verification conditions.

To illustrate our methodology, we show how safety properties of the well-known *generalized railroad crossing* benchmark for real-time verification can be formally verified in a modular fashion, and how we can check that our system description is non-Zeno. For this, we use the Stanford Temporal Prover, STeP, a tool for verifying linear-time temporal properties of reactive systems [BBC$^+$96]. To facilitate the verification process, STeP includes verification rules, verification diagrams, decision procedures, and automatic invariant generation methods; we will have the opportunity to use all of these in the verification process.

**Outline:** Section 2 introduces the basic computational model and specification language. Section 3 extends this framework to account for modularity, and describes how modular verification of safety properties can be performed. Section 4 presents verification rules and verification diagrams for proving temporal safety properties, and also discusses global and modular non-Zenoness proofs. In Section 5 we briefly describe the STeP system, including modularity and invariant generation. Section 6 describes the specification and verification of the generalized railroad crossing example using STeP. We conclude with a brief overview of related work in Section 7.

## 2 Preliminaries

### 2.1 System Description: Clocked Transition Systems

Our computational model is that of *clocked transition systems* [MP96], an extension of transition systems to account for continuous real-time. They are closer to timed automata [AD94] than the previously proposed *timed transition systems* [HMP94]. The basic idea, following [AL92], is to add explicit real-valued clock variables to the system, which measure the passing of time.

A clocked transition system (CTS) $\mathcal{S} = \langle \mathcal{V}, \Theta, \mathcal{T}, \Pi \rangle$ consists of:

- $\mathcal{V}$: A finite set of *system variables*, partitioned into a set $\mathcal{D}$ of *discrete variables*, which can be of any type, and a set $\mathcal{C}$ of real-valued *clock variables*. A subset of the discrete variables may be designated as *system constants*, that is, no transition may modify these variables. One of the clock variables, $T$, is designated as the *master clock*. A *state* is a type-consistent assignment to the system variables. The set of all states is the *state-space*, $\Sigma$. An *assertion*, or *state-formula*, is a first-order formula whose free variables belong to $\mathcal{V}$. For an assertion $p$, we say that a state $s$ is a *p-state* if $p$ is satisfied by $s$, written $s \models p$.
- $\Theta$: The *initial condition*, a satisfiable assertion characterizing the possible initial states. We require that $\Theta$ implies $T = 0$.
- $\mathcal{T}$: A finite set of *transitions*. Each transition $\tau$ maps each state $s \in \Sigma$ into a set of *$\tau$-successor* states $\tau(s) \subseteq \Sigma$. Each transition $\tau$ is described by a *transition relation* $\rho_\tau(\mathcal{V}, \mathcal{V}')$, a first-order formula which refers to both unprimed and primed system variables. An unprimed system variable indicates its value in the current state $s$, while the primed version indicates the value in the next-state $s'$. Transitions in $\mathcal{T}$ are discrete and happen instantaneously; therefore we require that no transition modify the master clock, i.e. $\rho_\tau$ implies $T' = T$ for every transition $\tau \in \mathcal{T}$.
- $\Pi$: The *time-progress condition*. This is an assertion over $\mathcal{V}$, used to specify a global restriction on the progress of time.

To account for the passage of time, we add to the set of transitions a *tick* transition, whose transition relation is given by

$$\rho_{tick} : \exists \Delta \,.\, \left( \begin{array}{c} \Delta > 0 \wedge \forall t \in [0, \Delta] \,.\, \Pi(\mathcal{D}, \mathcal{C} + t) \\ \wedge \\ \mathcal{D}' = \mathcal{D} \;\wedge\; \mathcal{C}' = \mathcal{C} + \Delta \end{array} \right)$$

where $\mathcal{C}' = \mathcal{C} + \Delta$ stands for $\bigwedge_{c \in \mathcal{C}} (c' = c + \Delta)$. Thus, *tick* preserves the values of all discrete variables and uniformly increments all clocks by an amount $\Delta$ that satisfies the global time-progress condition $\Pi$. The *tick* transition is the only one that can change the master clock $T$.

The set $\mathcal{T} \cup \{tick\}$ is the *extended set of transitions*, $\mathcal{T}_T$. A *run* of a clocked transition system $\mathcal{S} : \langle \mathcal{V}, \Theta, \mathcal{T}, \Pi \rangle$ is an infinite sequence of states $\sigma : s_0, s_1, \ldots$ such that (1) $s_0 \vDash \Theta$ and (2) for each $j \geq 0$ there is some $\tau \in \mathcal{T}_T$ such that $s_{j+1} \in \tau(s_j)$. In this case we say that $\tau$ is *taken* at $s_j$. The *enabling condition* of a transition $\tau$ is $\exists \mathcal{V}' \,.\, \rho_\tau(\mathcal{V}, \mathcal{V}')$, characterizing the set of states where $\tau$ can be taken. A state $s$ is *reachable* if it appears in some run of $\mathcal{S}$. A run is a *computation* if it satisfies *time divergence*, that is, the value of $T$ increases beyond any bound. $Comp(\mathcal{S})$ is the set of all computations of $\mathcal{S}$.

In specifications of real-time systems it is common to associate lower and upper time bounds with transitions. In the CTS formalism, a lower bound $l$ on transition $\tau$ is enforced by associating a clock $c_\tau$ with $\tau$ and adding the condition $c_\tau \geq l$ to $\tau$'s enabling condition. When the enabling condition for $\tau$ first becomes true, $c_\tau$ is reset to 0. Upper bounds appear as constraints on $\Pi$: if $u$ is the upper bound on $\tau$, then $c_\tau \leq u$ appears as a conjunct in $\Pi$. [MP96] shows how a timed transition system can be translated to a CTS in this way.

Note that the generality of transition systems allows us to describe real-time systems whose discrete component is infinite- as well as finite-state. That is, the clocks need not be the only variables with an unbounded domain.

### *2.2 Specification Language: Linear-time Temporal Logic*

We use *linear-time temporal logic* (LTL) as our property specification language [MP91]. Future temporal operators include $\square$ (always), $\mathcal{W}$ (wait-for), $\diamondsuit$ (eventually), $\bigcirc$ (next), and $\mathcal{U}$ (until). Past operators include $\ominus$ (previously), $\boxminus$ (so-far), $\diamondminus$ (once), and $\mathcal{S}$ (since). We write $p \Rightarrow q$ as an abbreviation of $\square(p \rightarrow q)$. A *past formula* is one that contains no future temporal operators.

Recall that for a state $s$ and an assertion $p$, we write $s \vDash p$ if $p$ is true under the system variable assignment for state $s$. Given a sequence of states $\sigma : \; s_0, s_1, \ldots$ and temporal formulas $\varphi_1$, $\varphi_2$, $p$, $q$, we define:

$(\sigma, i) \vDash \neg\varphi$ iff $(\sigma, i) \nvDash \varphi$.
$(\sigma, i) \vDash \varphi_1 \wedge (\vee) \varphi_2$ iff $(\sigma, i) \vDash \varphi_1$ and (or) $(\sigma, i) \vDash \varphi_2$.
$(\sigma, i) \vDash \Box\, p$ iff $(\sigma, j) \vDash p$ for all $j \geq i$.
$(\sigma, i) \vDash p\, \mathcal{U}\, q$ iff there is some $j \geq i$ such that $(\sigma, j) \vDash q$ and $(\sigma, k) \vDash p$ for all $i \leq k < j$.
$(\sigma, i) \vDash p\, \mathcal{W}\, q$ iff $(\sigma, i) \vDash \Box\, p$ or $(\sigma, i) \vDash p\, \mathcal{U}\, q$.

These are all the LTL operators we will use in this paper—see [MP95] for more details.

Since clocks are system variables, temporal formulas may freely refer to them. This includes, in particular, the master clock $T$, allowing the natural expression of time-dependent properties. For example, $\Box(l \leq T \wedge T \leq u \rightarrow p)$ states that $p$ holds over the time interval $[l, u]$.

A computation $\sigma : \; s_0, s_1, \ldots$ satisfies $\varphi$ iff $(\sigma, 0)$ satisfies $\varphi$. A temporal formula $\varphi$ is *valid* over a CTS $\mathcal{S}$, written as $\mathcal{S} \vDash \varphi$, if every computation of $\mathcal{S}$ satisfies $\varphi$. In this case, we also say that $\varphi$ is $\mathcal{S}$-*valid*.

System and auxiliary variables can be divided into *rigid* and *flexible* variables. Rigid variables cannot change their value over time, whereas flexible variables can. Rigid system variables are, in effect, constants. Rigid auxiliary variables are useful for quantification across time steps.

### *2.3 Non-Zenoness*

A CTS $\mathcal{S}$ is *non-Zeno* if every finite prefix of a run of $\mathcal{S}$ can be extended to a computation; that is, it is always possible for time to grow beyond any bound.

Any system that models a real-world process must necessarily be non-Zeno, since any such process will diverge in time. A verification effort would be questionable if the system were Zeno, since such systems are not implementable.

Furthermore, the goal is often not merely to model and verify a real-world process, but to design a *controller* for a given real-world model $M$. This controller is a module $C$ such that the parallel composition of $C$ and $M$ satisfies some temporal specification. Even when $M$ is a faithful model, a badly designed $C$ may cause the combined system to be Zeno.

As a simple example, one way to satisfy a safety specification is for the controller to force time to stop whenever the safety specification is about to be violated. Consider a water-level controller description $C$ that adds the following conjunct to the time-progress condition:

$$\neg(\mathit{level} > \mathit{high}) \wedge \neg(\mathit{level} < \mathit{low}) \enspace .$$

Any system composed with this controller will trivially satisfy the specification

$$\Box(\mathit{level} \geq \mathit{low} \wedge \mathit{level} \leq \mathit{high}) \enspace .$$

Other examples may be constructed where Zeno behavior is less obvious, e.g. where a controller stops time by switching on and off infinitely often with time intervals $1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots$ Thus, having proposed a controller, a first check should be for non-Zenoness. Otherwise, verifying other properties may be of little use.

Non-Zenoness is an existential property, and cannot be expressed within linear-time temporal logic. However, it can be expressed with the following formula in the branching-time *computation tree logic* CTL (see [Eme90]):

$$\forall \epsilon > 0 \;.\; \forall t \;.\; \mathsf{AG}(T = t \;\rightarrow\; \mathsf{EF}(T \geq t + \epsilon))$$

where $\epsilon$ and $t$ are rigid auxiliary variables. That is, for every $\epsilon > 0$ and at every reachable state, where the global clock $T$ has value $t$, there is a possible future state where $T$ has increased by at least $\epsilon$. This turns out to be equivalent to

$$\exists \epsilon > 0 \;.\; \forall t \;.\; \mathsf{AG}(T = t \;\rightarrow\; \mathsf{EF}(T \geq t + \epsilon)) \enspace .$$

We discuss how branching-time reasoning can be used to prove non-Zenoness in Section 4.3. Modular verification of non-Zenoness is discussed in Sections 4.4 and 4.5.

**Controlled non-Zenoness:** With this characterization of non-Zenoness we assume that all transitions of the CTS cooperate to achieve time divergence. From a control-theoretic point of view, one could refine this notion to distinguish between controllable events, uncontrollable events, and the clock, where only the controllable events and the clock can be assumed to collaborate towards time divergence. We would then have to exchange the path modality $\mathsf{EF}$ by a game modality characterizing the states where the controller and clock can force time $T$ to grow beyond $t_0 + \epsilon$, regardless of how the uncontrollable events unfold. We will not pursue this model of non-Zenoness further here, but instead discuss the related notion of receptiveness in Section 4.4.

## 3 Clocked Transition Modules

We adapt the compositional verification methodology of [CMP94] to clocked transition systems, adding the option of synchronizing transitions. *Clocked transition modules* (CTM's) allow the system to be decomposed into interacting modules. The model allows interaction via shared variables and via synchronization of transitions. These modules can then be analyzed independently and the results inherited by the full system. Formally, a clocked transition module $M : \langle \mathcal{V}, \Theta, \mathcal{T}, \Pi \rangle$ contains the same elements as a CTS, augmented with the following:

- A partition of the set of system variables $\mathcal{V}$ into *private* ($\mathcal{V}^P$) and *shared* ($\mathcal{V}^S$) variables, where the master clock $T$ is shared.
- A label $\ell(\tau)$ for each transition $\tau$.

Clock variables can be private or shared; as with CTS's, the only requirements are that they be uniformly incremented by the *tick* transition, and that the shared master clock $T$ is never changed by a transition other than *tick*.

### 3.1 The Environment Transition

A stand-alone module is interpreted as a clocked transition system by adding an *environment transition* $\tau_E$, which can arbitrarily change the values of all shared variables except $T$. Formally, $\tau_E$ has transition relation

$$\rho_{\tau_E} : \left( \bigwedge_{u \in \mathcal{V}^P} (u' = u) \right) \ \wedge \ (T' = T) \ .$$

Given a module $M : \langle \mathcal{V}, \mathcal{T}, \Pi, \Theta \rangle$, its *associated CTS* is

$$\mathcal{S}_M : \langle \mathcal{V}^*, \mathcal{T} \cup \{\tau_E\}, \Pi, \Theta \rangle,$$

where $\mathcal{V}^*$ is the set of all the variables in our vocabulary, that is, all variables that can ever be referenced in any module that is part of our system. $Comp(M)$ is defined to be equal to $Comp(\mathcal{S}_M)$. A temporal property $\varphi$ is *valid* over a module $M$, or *modularly valid* for $M$, written $M \vDash \varphi$, iff $\mathcal{S}_M \vDash \varphi$.

Given two modules $M_1 : \langle \mathcal{V}_1, \mathcal{T}_1, \Pi_1, \Theta_1 \rangle$ and $M_2 : \langle \mathcal{V}_2, \mathcal{T}_2, \Pi_2, \Theta_2 \rangle$, their *parallel composition* $[M_1 \,||\, M_2]$ is the clocked transition module $M : \langle \mathcal{V}, \mathcal{T}, \Pi, \Theta \rangle$ defined as follows:

- $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2$, $\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2$ and $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2$.
- $\mathcal{V}^P = \mathcal{V}_1^P \cup \mathcal{V}_2^P$ and $\mathcal{V}^S = \mathcal{V}_1^S \cup \mathcal{V}_2^S$. The modules are assumed to be compatible, in that $\mathcal{V}_1^P$ and $\mathcal{V}_2$ (resp. $\mathcal{V}_2^P$ and $\mathcal{V}_1$) should be disjoint.
- $\Pi = \Pi_1 \wedge \Pi_2$ and $\Theta = \Theta_1 \wedge \Theta_2$.
- The set $\mathcal{T}$ of new transitions and their labels are defined as follows: for a transition $\tau$ in $\mathcal{T}_1$ (resp. $\mathcal{T}_2$) whose label does not appear in $\mathcal{T}_2$ (resp. $\mathcal{T}_1$), $\mathcal{T}$ includes the transition with the same label and relation

$$\rho_\tau \;\wedge\; \left( \bigwedge_{v \in \mathcal{V}_2^P} (v = v') \right) \qquad (\text{resp. } \;\rho_\tau \;\wedge\; \left( \bigwedge_{v \in \mathcal{V}_1^P} (v = v') \right))$$

  For each pair of transitions $\tau_1 \in \mathcal{T}_1$ and $\tau_2 \in \mathcal{T}_2$ that have the same label $l$, there is a corresponding transition in $[M_1 \,||\, M_2]$, also labeled $l$, with transition relation $\rho_{\tau_1} \wedge \rho_{\tau_2}$.

The following theorem relates the computations of two modules and those of their parallel composition:

**Theorem 3.1** $Comp([M_1 \,||\, M_2]) \subseteq Comp(M_1) \cap Comp(M_2)$.

**Proof:**

Consider a computation $\sigma : s_0, s_1, \ldots$ of $[M_1 \,||\, M_2]$. We show that $\sigma$ is a computation of $M_1$. The initial state $s_0$ must satisfy $\Theta_1 \wedge \Theta_2$, and hence $\Theta_1$. Consider a transition from $s_i$ to $s_{i+1}$ in $\sigma$. This transition must be due to (1) a transition in $\mathcal{T}_1$, possibly synchronized with one in $\mathcal{T}_2$, (2) a transition in $\mathcal{T}_2$, which must preserve the private variables of $M_1$, (3) the environment transition $\tau_E$ for $[M_1 \,||\, M_2]$, or (4) a *tick* transition. In case (1), the same transition in $\mathcal{T}_1$ can be taken in a computation of $M_1$. In case (2), the environment transition for $M_1$ can account for this state-change in a computation of $M_1$. In case (3), the same environment transition can take place in a computation of $M_1$, since the private variables of $M_1$ are a subset of those of $[M_1 \,||\, M_2]$. Finally, a *tick* transition for $[M_1 \,||\, M_2]$ must satisfy the time-progress condition $\Pi_1 \wedge \Pi_2$, and hence be a valid *tick* transition for $M_1$. Hence, $\sigma$ is also a computation of $M_1$.

▪

Note that while transitions with matching labels are composed synchronously, asynchronous communication is always possible using shared variables or buffered channels. Typically, the synchronizing transitions correspond to control signals that connect two modules together, as we will see in Section 6. A similar approach is used to model real-time systems in [Ost90].

**Hiding:** In practice, it is useful to distinguish transitions that will never synchronize with other modules. *Hiding* a label $l$ in a module $M$ stipulates that all transitions labeled by $l$ are internal to $M$ and cannot synchronize with other transitions. Consequently, hidden transitions cannot be disabled by the environment. Furthermore, such transitions can be assumed to preserve the private variables of all other modules.

**Controlled and External Transitions:** Discussion of receptiveness for modular non-Zenoness proofs (Section 4.4) requires that we identify which transitions are taken on the initiative of the module and which at the initiative of the environment. For this purpose, we mark each transition as `controlled` or `external`, similar to the input/output distinctions of I/O automata [GSSL94]. The `external` transitions will be controlled by the environment, and the `controlled` ones by the module.

When two modules are composed, transitions that only appear in one module inherit their marking in the new one. If a transition label appears in both modules, we require that transitions with this label should not appear marked as `controlled` in both. When two transitions with matching labels are composed, the new transition is `controlled` if one of them is `controlled`. Otherwise, both must be `external`, and the new transition is also `external`.

This marking is only defined for module transitions: the environment and *tick* transitions are not marked.

For a module $M$, we require that `external` transitions have enabling condition *true*. This property is preserved by module composition. In the following, when we write $[M_1 \,||\, M_2]$ we assume that modules $M_1$ and $M_2$ are well-formed and compatible, so they can be composed.

## 4 Deductive Verification

### *4.1 Verification of Safety Properties*

A *safety property* is one that can be expressed by a formula of the form $\Box\, p$, for a past temporal formula $p$. This includes *invariances*, where $p$ is an assertion, and *wait-for* properties, of the form $p \Rightarrow q\, \mathcal{W}\, r$ for assertions

$$
\begin{array}{ll}
\multicolumn{2}{l}{\text{For CTM } M \text{ and assertions } \varphi,\, p,} \\
\text{I1.} & \varphi \rightarrow p \\
\text{I2.} & \Theta \rightarrow \varphi \\
\text{I3.} & \{\varphi\}\, \tau\, \{\varphi\} \quad \text{for each } \tau \in \mathcal{T}_T \\
\hline
 & M \vDash \Box\, p
\end{array}
$$

Fig. 1. Invariance rule INV

$$
\begin{array}{ll}
\multicolumn{2}{l}{\text{For CTM } M \text{ and assertions } p,\, q,\, \varphi,\, r,} \\
\text{W1.} & \varphi \rightarrow q \\
\text{W2.} & p \rightarrow r \vee \varphi \\
\text{W3.} & \{\varphi\}\, \tau\, \{r \vee \varphi\} \quad \text{for each } \tau \in \mathcal{T}_T \\
\hline
 & M \vDash\ p \Rightarrow q\, \mathcal{W}\, r
\end{array}
$$

Fig. 2. Waiting-for rule WAIT

$p$, $q$ and $r$. Many useful specifications that are progress properties in their untimed version correspond to safety properties when a time bound is added [MP96]. For example, showing that a system reaches and maintains $p$ within time $\delta$ is expressed as the invariance $\Box(T \geq \delta \rightarrow p)$.

Many verification rules for standard transition systems [MP95] can be reused for the verification of clocked transition systems, since the basic computational model and specification language are virtually the same. The *invariance rule* INV and the *wait-for rule* WAIT, shown in Figures 1 and 2, can be used to prove simple modular safety properties. These rules reduce the modular validity of a temporal formula to the general validity of a set of first-order *verification conditions*. The triple $\{\varphi\}\, \tau\, \{\psi\}$ stands for the formula

$$(\varphi(V) \wedge \rho_\tau(V, V')) \rightarrow \psi(V')\,.$$

Rules INV and WAIT generate $(N + 2)$ verification conditions for a CTM with $N$ transitions (including the *tick* and $\tau_E$ transitions).

We say that an assertion $p$ is *inductive* for a module $M$ if $\Box\, p$ can be proved using rule INV with $\varphi$ equal to $p$ (that is, $p$ holds initially and is preserved by all transitions). If these verification conditions can be proved assuming a set of properties $S$, we say that $p$ is *inductive relative to* $S$.

In the case of the WAIT rule, recall that $p \Rightarrow q\, \mathcal{W}\, r$ is an abbreviation of

$\Box(p \rightarrow q \mathcal{W} r)$. The intermediate assertion $\varphi$ strengthens $q$; it should hold whenever $p$ is true, and continue to hold until $r$ becomes true.

These rules are sound for proving safety properties of CTM's [MP96]. Furthermore, [KMP96] shows that these rules are also complete (relative to the underlying first-order reasoning) for non-Zeno CTS's.

A consequence of Theorem 3.1 is that any linear-time temporal property which holds for a module $M_1$ will also hold for $[M_1 \,||\, M_2]$ for any $M_2$:

**Corollary 4.1** *If* $M_1 \vDash \varphi$ *for an LTL property* $\varphi$*, then* $[M_1 \,||\, M_2] \vDash \varphi$ *for any module* $M_2$*.*

We will only use this fact in the case of safety properties. A module can satisfy non-trivial progress properties only if restrictions on the environment are made, or transitions hidden to avoid a deadlocking environment. Assumption-guarantee proof systems [AL92,Cha93] are essential for these applications.

Note that $Comp(M_1)$ may be a strict superset of $Comp([M_1 \,||\, M_2])$, so safety properties of $[M_1 \,||\, M_2]$ may not hold modularly for $M_1$ or $M_2$. Note also that the inclusion in Theorem 3.1 can be strict, due to the presence of synchronizing transitions. For example, consider a module $M_1$ (resp. $M_2$) containing a single transition $\tau_1$ (resp. $\tau_2$) that increments a private variable $x_1$ (resp. $x_2$), where $\tau_1$ and $\tau_2$ have the same label and $x_1, x_2$ are initially 0. Then $\langle x_1, x_2\rangle : \langle 0,0\rangle, \langle 1,0\rangle, \langle 1,1\rangle, \ldots$ is a prefix of a computation in both $Comp(M_1)$ and $Comp(M_2)$. However, this cannot happen in a computation of $[M_1 \,||\, M_2]$, since synchronization requires that $x_1$ and $x_2$ must increase at the same time.

In this example, note that $\tau_1$ can change $x_2$ in a computation of $M_1$. This could not be the case if $\tau_1$ were hidden in $M_1$.

### *4.2 Verification Diagrams*

Proofs of temporal system specifications can often be conveniently presented using *verification diagrams* [MP94]. A verification diagram $\mathcal{D}$ is a directed graph whose nodes are labeled by formulas and edges labeled by transitions. Node labels summarize sets of states, and edges represent possible state transitions between them. Every edge $e$ is labeled by a transition $\tau(e)$. A *terminal* node in $\mathcal{D}$ is one with no outgoing edges. $Nodes(\mathcal{D})$ is the set of nodes in $\mathcal{D}$, and $Term(\mathcal{D})$ is the set of terminal nodes.

With a verification diagram $\mathcal{D}$ and system $\mathcal{S}$ we associate a set of $\mathcal{S}$*-verification conditions*. When these are valid we say the diagram $\mathcal{D}$ is $\mathcal{S}$-valid. Similarly, we associate a set of *$\varphi$-verification conditions* with a diagram $\mathcal{D}$ and temporal formula $\varphi$. When these are valid, the $\mathcal{S}$-validity of the diagram implies the $\mathcal{S}$-validity of $\varphi$.

We will use diagrams to prove *wait-for properties*, of the form $p \Rightarrow q \mathcal{W} r$ for assertions $p$, $q$ and $r$. For convenience, we introduce a special class of diagrams for these properties:

**Definition 4.2 (Wait-for Diagrams)** *A* wait-for diagram *is a verification diagram $\mathcal{D}$, where each node $n$ is labeled by the formula $\varphi_n$. With each non-terminal node $n \in Nodes(\mathcal{D})$ and every transition $\tau$ in $\mathcal{S}$, we associate the $\mathcal{S}$-verification condition:*

$$\{\varphi_n\} \ \tau \ \{\varphi_n \ \vee \ \left( \bigvee_{m \in \tau(n)} \varphi_m \right) \} \ .$$

*The $\varphi$-verification conditions for $\mathcal{D}$ necessary to prove $\varphi : p \Rightarrow q \mathcal{W} r$ are:*

| | | |
|---|---|---|
| $W1.$ | $p \rightarrow \bigvee_{n \in Nodes(\mathcal{D})} \varphi_n$ | *All p-states are somewhere in the diagram* |
| $W2.$ | $\bigwedge_{n \notin Term(\mathcal{D})} (\varphi_n \rightarrow q)$ | *The non-terminal nodes in $\mathcal{D}$ are q-states* |
| $W3.$ | $\bigwedge_{n \in Term(\mathcal{D})} (\varphi_n \rightarrow r)$ | *The terminal nodes imply r* |

Thus, a wait-for diagram is a graphical alternative to rule WAIT of Figure 2, with the disjunction $\bigvee_n \varphi_n$ over the non-terminal nodes $\{n\}$ replacing the intermediate assertion $\varphi$ of rule WAIT.

Like the verification rules of Section 4.1, these diagrams are sound and complete for establishing wait-for properties of non-Zeno CTS's [KMP96]. We will see examples of wait-for diagrams in Section 6.4. STeP also supports diagrams for response and invariance properties. Furthermore, arbitrary linear-time temporal formulas can be verified in STeP using the *generalized verification diagrams* introduced in [BMS95].

### *4.3 Non-Zenoness Diagrams*

As discussed in Section 2.3, any realistic real-time system specification should be non-Zeno. Furthermore, the possibility of Zeno systems compromises the completeness of our diagrams and rules [KMP96]. We will use a special class of verification diagrams to verify non-Zenoness. Reflecting the existential nature of the property, the verification conditions associated with these diagrams use the notation of *possibility triples*:

**Definition 4.3 (Possibility triples)** *With a* possibility triple

$$\{\varphi\} \;\; \tau \; \exists \, \{\psi\}$$

*we associate the verification condition*

$$\varphi(V) \;\; \rightarrow \;\; \exists V' \; . \; \rho_\tau(V, V') \; \wedge \; \psi(V') \;\; .$$

A possibility triple $\{\varphi\}\,\tau\,\exists\,\{\psi\}$ is valid iff at every $\varphi$-state it is possible for transition $\tau$ to reach a $\psi$-state.

**Definition 4.4 (Non-Zenoness Diagrams)** *A* non-Zenoness diagram *is a directed graph $\mathcal{D}$ where each node $n$ is labeled by an assertion $\varphi_n$ and a ranking function $\delta_n$, whose range is a well-founded domain. The diagram also contains two distinguished constants $t_0$ and $\epsilon$. We associate the following first-order verification conditions with a non-Zenoness diagram $\mathcal{D}$:*

**Well-formedness:** $t_0$ *is a fresh Skolem constant;* $\epsilon$ *can be expressed as a function of system constants and is greater than 0.*

**Initiality:** $(T = t_0) \;\; \rightarrow \displaystyle\bigvee_{n \in Nodes(\mathcal{D})} \varphi_n \;\; .$

**Progress:** *For every node $n$ in $\mathcal{D}$, either (i) some outgoing edge is labeled by a transition that is enabled and can decrease the well-founded order, or (ii)* `tick` *is enabled and can advance time beyond $t_0 + \epsilon$. Formally,*

$$\begin{array}{c} \{\varphi_n\} \;\; \texttt{tick} \;\; \exists\{T \geq t_0 + \epsilon\} \\ \vee \\ \displaystyle\bigvee_{e_m = \langle n, m\rangle \in Out(n)} \{\varphi_n \wedge \delta_n = u\} \;\;\; \tau(e_m) \;\;\; \exists\,\{\varphi_m \wedge \delta_m \prec u\}\,, \end{array}$$

*where $u$ is a auxiliary rigid variable, and $Out(n)$ is the set of outgoing edges from $n$.*

**Proposition 4.5** *An $\mathcal{S}$-valid non-Zenoness diagram establishes that $\mathcal{S}$ is non-Zeno.*

**Justification:** Consider an $\mathcal{S}$-valid non-Zenoness diagram $\mathcal{D}$. The **Initiality** condition ensures that every reachable state where $T = t_0$ is contained in a diagram node. By the **Progress** conditions, at every node in $\mathcal{D}$ a transition can be taken that advances time beyond $\epsilon$ or decreases the well-founded order, moving to another node in $\mathcal{D}$. Following such transitions must eventually advance time beyond $\epsilon$, since otherwise well-foundedness would be violated. Thus, since $\epsilon$ is fixed and greater than 0, the system is itself non-Zeno. ▪

### *4.4 Modular non-Zenoness: Receptiveness*

It would be preferable to establish non-Zenoness for modules and then infer the non-Zenoness of their composition, as we propose in Section 4.1 for the case of safety properties. Unfortunately, unlike safety properties, non-Zenoness is not closed under parallel composition: the composition of two modules may fail to be non-Zeno even if each module is non-Zeno. [1]

A condition that implies non-Zenoness and is preserved under parallel composition is *receptiveness* [Dil89,AH97]. Informally, a module is receptive if it is non-Zeno whenever it is composed with a reasonably cooperative environment—one that does not maliciously prevent the advance of time.

More formally, a module $M$ is receptive if at every reachable state $M$ can construct a computation in which time diverges or only finitely many steps are taken by $M$. Similarly to [Dil89,GSSL94,AH97], this is represented by a game between the module and its environment, called the *receptiveness game*.

Recall that module transitions are marked as `controlled` or `external`. Starting at a state $s$, the game constructs a run of $\mathcal{S}_M$ as follows: at each step, the environment proposes to take the environment transition $\tau_E$, an `external` transition, or the *tick* transition with an increment $\Delta_{env} > 0$. The module proposes a `controlled` transition, which should be enabled, or a $\Delta_{mod} > 0$ by which the *tick* transition can be taken. The next-state is chosen as follows:

- If the environment proposed $\tau_E$ or an `external` transition, this transition is taken, and the move counts as an *environment move*. The environment chooses the next-state as well.
- If the environment proposed *tick* and the module proposed a `controlled` transition $\tau$, then $\tau$ is taken. The module chooses the next-state values of the private variables, and the environment then chooses those of the shared variables, consistent with $\tau$. The move is counted as a *module move*.
- The last case is where both have proposed a *tick* transition. Then *tick* is taken, with duration $\min\{\Delta_{env}, \Delta_{mod}\}$. This is considered a module move if $\Delta_{mod} \leq \Delta_{env}$, and an environment move otherwise.

[1] This does not contradict Corollary 4.1, since non-Zenoness is not an LTL property, as we discuss in Section 2.3.

At any given point, the module loses the game if it cannot propose a move, that is, no `controlled` or *tick* transition is enabled. The module wins the game over an infinite run if either (1) time advances beyond any bound, or (2) the module only performs a finite number of moves.

Note that the environment can always make a move: $\tau_E$ and `external` transitions are always enabled, and it is not required to choose an enabled *tick*.

A module *strategy* is a function that maps every reachable state to a suitable `controlled` transition, choosing next-state values for the private variables, or a $\Delta_{mod}$ increment for the *tick* transition. The strategy determines the module's proposed move at each step of the receptiveness game.

**Definition 4.6 (Receptiveness)** *A module $M$ is* receptive *if it has a strategy that wins against any environment, starting the receptiveness game at any reachable state. We call this a* winning strategy *for $M$.*

A winning strategy must be defined at every reachable state.

**Proposition 4.7** *(a) If $M$ is receptive, then its associated CTS $\mathcal{S}_M$ is non-Zeno. (b) If $M_1$ and $M_2$ are receptive, then $[M_1 \,||\, M_2]$ is receptive.*

**Justification:** (a) If $M$ is receptive, it has a winning strategy, in particular, against an environment that always lets $M$ move—one that always proposes *tick*, where $\Delta_{env} \geq \Delta_{mod}$ if $M$ also proposes *tick*. For any reachable state, this game generates a run of $\mathcal{S}_M$ where time diverges, so $\mathcal{S}_M$ is non-Zeno. Note that the `controlled` and *tick* transitions should thus, on their own, ensure time divergence.

(b) Assume that $M_1$ and $M_2$ are receptive. The winning strategy for $M = [M_1 \,||\, M_2]$ is constructed from winning strategies for $M_1$ and $M_2$ as follows:

Consider a state $s$ reachable in a run of $M$. The proof of Theorem 3.1 applies to runs as well, so $s$ is a reachable state for both $M_1$ and $M_2$. We consider two cases, depending on what each strategy proposes at $s$:

(i) At least one of the modules, say $M_1$, proposes a `controlled` transition $\tau$. Since $\tau$ can only synchronize with `external` transitions of $M_2$, all transitions corresponding to $\tau$ in $M$ can be taken, and any one can be chosen. In this case, $M_2$ sees an environment move that takes $\tau_E$ if $\tau$ does not synchronize, or an `external` transition of $M_2$ otherwise.

(ii) $M_1$ proposes *tick* with $\Delta_1$ and $M_2$ proposes *tick* with $\Delta_2$. In this case, a *tick* with $\min\{\Delta_1, \Delta_2\}$ can be taken, since it will satisfy the progress conditions for both modules. If $\Delta_1 = \Delta_2$, both modules see a module move in their respective games. Otherwise, the module that proposed the shorter duration will see a module move, and the other will see an environment move, namely a shorter *tick*.

In all cases, the new strategy selects a valid move, which is a module move for at least one of the modules, and a module move or an environment move for the other, where the module moves follow the respective winning strategies. Environment moves in the game for $M$ are environment moves in the games for $M_1$ and $M_2$. Now assume, for a contradiction, that $M$ loses the game. Since $M$ always has a valid move, this must be an infinite run, where time does not diverge and $M$ moves infinitely often. This means that at least one module moves infinitely often as well, and thus loses its own game, a contradiction. ■

### *4.5 Receptiveness Diagrams*

To show that a module is receptive we use receptiveness diagrams. They are similar to the non-Zenoness diagrams of Section 4.3, but we now make a distinction between transitions taken by the module and transitions taken by the environment.

**Definition 4.8 (Receptiveness Diagrams)** *A* receptiveness diagram *for a module $M$ is a directed graph $\mathcal{D}$, where each node $n$ is labeled by an assertion $\varphi_n$ and a ranking function $\delta_n$, which only depends on private variables of $M$ and whose range is a well-founded domain. The diagram contains two distinguished constants, $t_0$ and $\epsilon$. We associate the following first-order verification conditions with a receptiveness diagram $\mathcal{D}$:*

**Well-formedness:** *$t_0$ is a fresh Skolem constant; $\epsilon$ can be expressed as a function of system constants and is greater than 0.*

**Initiality:** $(T = t_0) \;\rightarrow \bigvee_{n \in Nodes(\mathcal{D})} \varphi_n \;.$

**Module:** *For every node $n$ in $\mathcal{D}$, either (i) some outgoing edge is labeled by* `tick` *or a* `controlled` *transition, which is enabled and can decrease the well-founded order, or (ii)* `tick` *is enabled and can advance time beyond $t_0 + \epsilon$. Formally,*

$$\begin{array}{c} \{\varphi_n\} \;\; \texttt{tick} \;\; \exists\{T \geq t_0 + \epsilon\} \\ \vee \\ \displaystyle\bigvee_{e_m = \langle n,m\rangle \in COut(n)} \{\varphi_n \wedge \delta_n = u\} \;\; \tau(e_m) \;\; \exists\,\{\varphi_m \wedge \delta_m \prec u\}\,, \end{array}$$

*where $COut(n)$ is the set of outgoing edges from $n$ labeled by* `controlled` *or* `tick` *transitions.*

**Environment:** *For every node $n$ and* `tick`*, environment or* `external` *transition $\tau$, either the assertion $\varphi_n$ is preserved by $\tau$, or $\tau$ leads to one of the nodes in $\tau(n)$ (the set of nodes reachable from $n$ by an edge labeled by $\tau$) without increasing the well-founded order. Formally,*

$$\{\varphi_n \ \wedge \ \delta_n = u\} \quad \tau \quad \{(\varphi_n \wedge \delta_n \preceq u) \ \vee \ \left( \bigvee_{m \in N(\tau,n)} (\varphi_m \wedge \delta_m \preceq u \right) \} \ ,$$

*where $N(\tau, n)$ denotes the set of nodes $m$ such that there is an edge from $n$ to $m$ labeled by $\tau$.*

A receptiveness diagram is *$M$-valid* if all of its verification conditions are valid.

**Proposition 4.9** *An $M$-valid receptiveness diagram establishes that $M$ is receptive.*

**Justification:** Assume we have an $M$-valid receptiveness diagram $\mathcal{D}$. The winning strategy for $M$ is as follows. By the **Initiality** condition for $\mathcal{D}$, the game starts at some node in the diagram. Whenever a node $n$ is reached, if *tick* can advance time beyond $t_0 + \epsilon$, then such a time-increment $\Delta_{mod}$ is chosen. Otherwise, the **Module** conditions for $\mathcal{D}$ ensure that *tick* or a `controlled` transition, which can be taken, labels an outgoing edge from $n$. The strategy chooses such a transition (where *tick* is chosen with the maximum possible $\Delta_{mod}$), choosing values for the private variables that guarantee a successor node where the ranking is decreased. This is possible because the ranking functions only depend on private variables, so a suitable next-state will be reached regardless of the environment's choice for the shared variables.

By the **Environment** conditions on $\mathcal{D}$, any move by the environment must stay within $\mathcal{D}$, so the game will always stay within $\mathcal{D}$, unless time progresses beyond $t_0 + \epsilon$. This is a winning strategy, since by the ranking functions, the only way that the game can be played over an infinite run without advancing time past $t_0 + \epsilon$ is if the module moves only finitely many times. ▟

When all the strongly connected components in a receptiveness diagram $\mathcal{D}$ only contain environment or `external` transitions, it is clear that a game where the module moves infinitely often cannot remain forever within the diagram, since only finitely many transitions can be taken outside a strongly connected component, before an infinite computation eventually ends up inside one. In this case, we will associate a default well-founded ordering with $\mathcal{D}$ as follows: sort $\mathcal{D}$ topologically into its MSCS (maximally strongly connected components) $S_0, \ldots, S_N$, where nodes in $S_i$ are not reachable from nodes in $S_j$ if $i > j$. The well-founded domain is the finite set $\{0, \ldots, N\}$ ordered by $<$. For each node $n$, if $n \in S_i$ then $\delta_n \stackrel{\text{def}}{=} i$.

## 5 The STeP System

The Stanford Temporal Prover, STeP, is a tool for the deductive and algorithmic verification of reactive and real-time systems [BBC⁺96,BBC⁺95]. Untimed reactive systems are expressed by fair transition systems; real-time systems are expressed by clocked transition systems. Specifications are given in linear-time temporal logic.

STeP implements *verification rules* and *verification diagrams*, including rules INV and WAIT of Section 4.1, the wait-for diagrams of Section 4.2, and the non-Zenoness diagrams of Section 4.3. [2] A collection of decision procedures for built-in theories including integers, reals, datatypes and equality, is combined with propositional and first-order reasoning to simplify verification conditions, proving many of them automatically [Bjø98]. For those which cannot be established automatically, an interactive Gentzen-style theorem prover is available. STeP also includes a model checker, which can automatically establish or refute linear-time temporal properties of finite-state systems.

Features such as parameterization and the *tick* transition introduce quantifiers in verification conditions. Fortunately, the required quantifier instantiations are often "obvious" in that they use instances that can be provided by the decision procedures themselves. Accordingly, we have developed an integration of first-order reasoning and decision procedures that can automatically discharge many verification conditions that would otherwise require the use of the interactive prover [BSU97,Bjø98].

By allowing systems expressed by first-order transition relations and using a rich assertion language that includes constraints over the reals, STeP can support the real-time framework described in Section 2.

### *5.1 Generation of Invariants*

STeP provides tools for automatic generation of invariants based on the static analysis of transition systems [BBM97]. These invariants can then be used as auxiliary properties in deductive verification and model checking. We now briefly describe forward propagation, one of the methods presented in [BBM97], and an application to the real-time case that automatically generates useful auxiliary properties in Section 6.

The *strongest postcondition* $post(\tau, \varphi)$ of a formula $\varphi$ relative to a transition $\tau$ is the formula $\exists V_0 \, . \, (\rho_\tau(V_0, V) \wedge \varphi(V_0))$, describing the set of states reachable

[2] Receptiveness diagrams have not been implemented yet.

from a $\varphi$-state by taking $\tau$. In forward propagation we simulate the system by executing transitions, starting from the initial states, until we reach a fixpoint. Such an execution step can be expressed by

$$\begin{aligned}\mathcal{F}(X) &\stackrel{\text{def}}{=} \Theta \vee post(\mathcal{T}_T, X) \\ &\stackrel{\text{def}}{=} \Theta \vee \left( \bigvee_{\tau \in \mathcal{T}_T} \exists V_0 \,.\, X(V_0) \wedge \rho_\tau(V_0, V) \right) .\end{aligned}$$

The operator $\mathcal{F}$ is a predicate transformer, taking a predicate $X$ and producing the new predicate $\Theta \vee post(\mathcal{T}_T, X)$. The least fixpoint of $\mathcal{F}$: $\mu X.\mathcal{F}(X)$, which is equal to $\bigvee_{i=0}^{\infty} \mathcal{F}^i(\mathit{false})$, characterizes the set of reachable states, and is thus the strongest possible system invariant. However, expressing this fixpoint can require quantification over auxiliary variables, resulting in formulas that are not very useful in practice.

An alternative, weaker way to establish invariants is to compute

$$\mathcal{F}(\mathit{true}) = \Theta \vee \left( \bigvee_{\tau \in \mathcal{T}_T} \exists V_0 \,.\, \rho_\tau(V_0, V) \right) .$$

$\mathcal{F}(\mathit{true})$ characterizes the set of states reachable by one execution step, starting from the entire state space. In [BLS96], these are called *reaffirmed invariants*, and are generated for the parallel composition of untimed sequential programs. In [SDB96] they are used for hardware verification.

To obtain succinct and useful invariants, it is desirable to eliminate the existential quantifiers. Both [BLS96] and [SDB96] apply heuristics to approximate $\mathcal{F}(\mathit{true})$ tailored towards the class of systems they consider. Similarly, in STeP we use the resident simplifier to eliminate redundant quantifiers in the generated invariant and approximate $post(tick, \mathit{true})$ to $\Pi(\mathcal{D}, \mathcal{C})$, justified as follows:

$$post(tick, \mathit{true}) = \exists \mathcal{D}_0, \mathcal{C}_0, \Delta \,.\, \left( \begin{array}{c} \Delta > 0 \wedge \forall t \in [0, \Delta] \,.\, \Pi(\mathcal{D}_0, \mathcal{C}_0 + t) \\ \wedge \\ \mathcal{D} = \mathcal{D}_0 \;\wedge\; \mathcal{C} = \mathcal{C}_0 + \Delta \end{array} \right)$$

which, by taking $\mathcal{D}_0 = \mathcal{D}$ and $\mathcal{C}_0 = \mathcal{C} - \Delta$, is equivalent to

$$\exists \Delta \,.\, (\Delta > 0 \;\wedge\; \forall t \in [0, \Delta] \,.\, \Pi(\mathcal{D}, \mathcal{C} - \Delta + t))$$

which implies $\Pi(\mathcal{D}, \mathcal{C})$.

### *5.2 Modules in STeP*

A transition system module in STeP contains a *name*, an *interface declaration*, and a *body*. The body contains a list of transitions and, in the case of clocked transition systems, a time-progress condition.

In the STeP system specifications below, communication between modules occurs in the form of synchronizing transitions, so we will not need shared variables. Variables declared as `local` are private variables. Variables declared as `in` cannot be changed by any module, and are treated as constants.

For convenience, transitions can be described using guarded commands. These contain an enabling condition $\varphi$, tagged by `enable`, and a set of variable assignments of the form `x := e`, tagged by `assign`. This adds the conjuncts $\varphi$ and $\mathtt{x}' = \mathtt{e}$ to the corresponding transition relation. Of the variables not mentioned in an assignment statement, shared variables can be modified arbitrarily, whereas all other variables are unchanged.

A STeP specification is a set of *axioms* and *properties*. Properties are proof obligations, whereas axioms represent additional assumptions or previously proved properties. We write `[M]|=p` to indicate that module `M` satisfies property `p`.

## 6 Example: Generalized Railroad Crossing

We illustrate our methodology by verifying the generalized railroad crossing (GRC) benchmark problem, originally proposed in [HJL93]. Our system description closely follows the operational specification presented in [HL94]. The goal is to design and verify a controller that lowers and raises a gate such that (a) the system is "safe:" the gate is down if a train is passing the intersection, and (b) the system is "useful:" the gate is up if there is no good reason for it to be down.

### *6.1 System description*

The GRC system contains $N$ parallel railroad tracks protected by a gate and a gate controller. Each track is divided into three regions: $I$ (intersection), $P$ (an interval preceding the intersection), and *notHere* (everywhere else). The gate can be in any of four states: *down*, *up*, *goingDown*, and *goingUp*. Initially all trains are *notHere* and the gate is in state *up*. Each track is equipped with two sensors: one located at the beginning of the $P$-region, triggered when the

```
Clocked Transition Module Trains
in minTimeToI : real where minTimeToI > 0
in maxTimeToI : real where maxTimeToI > 0
type trainStatus = {notHere, P, I}

local trains: array[1..N] of trainStatus
          where Forall i:[1..N].(trains[i] = notHere)
local firstEnter: array[1..N] of real
local lastEnter : array[1..N] of real

Progress
  Forall i:[1..N].(trains[i] = P --> T <= lastEnter[i])

Transition trainsEnter[i:[1..N]] :
     enable   trains[i] = notHere
     assign   trains[i]      :=  P,
              firstEnter[i]  :=  T + minTimeToI,
              lastEnter[i]   :=  T + maxTimeToI

Transition enterI[i:[1..N]] :
     enable   trains[i] = P /\ T >= firstEnter[i]
     assign   trains[i] := I

Transition trainsExit[i:[1..N]] :
     enable   trains[i] = I
     assign   trains[i] := notHere
```

Fig. 3. The `Trains` clocked transition module

front of the train enters, and one at the end of the $I$-region, triggered when the train completely leaves the intersection. The gate accepts two commands, *lower* and *raise*, with the obvious effects.

We model this system as the parallel composition of three clocked transition modules:

**Trains:** The `Trains` module, shown in Figure 3, has three transitions parameterized by `i:[1..N]`, illustrated in Figure 4. Constants `minTimeToI` and `maxTimeToI` are the minimum and maximum time it may take a train to reach the intersection from the time it enters $P$. The lower bound is reflected in the enabling condition of `enterI` and the upper bound is included in the progress condition: when a train is in $P$, time cannot advance beyond this upper bound.

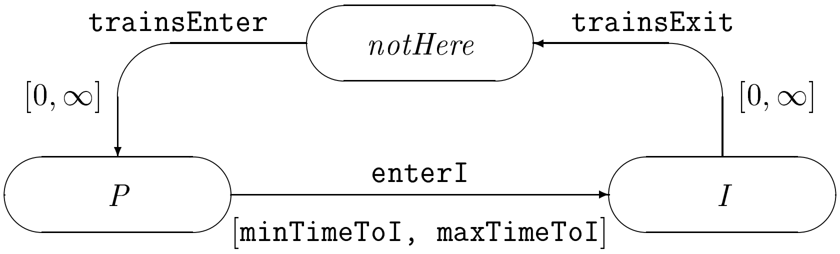



Fig. 4. Transitions for the `Trains` module

**Gate:** The `Gate` module, shown in Figure 5, has four transitions, illustrated in Figure 6. Constants `gateRiseTime` and `gateDownTime` represent the times to fully raise and lower the gate. The time-progress condition states that the gate can stay at most this long in a moving state, after which the transition `isUp` or `isDown` must be taken; otherwise, time cannot advance.

**Controller:** The composition `[Gate||Trains]` is neither "safe" nor "useful:" the gate can be arbitrarily raised or lowered regardless of the presence of trains. A controller is needed to constrain the behavior of the gate, based on observation of the trains. Such a module is shown in Figure 7. It has parameterized transitions `trainsEnter` and `trainsExit` that, upon module composition, synchronize with their counterparts in the `Trains` module, reflecting the modeling assumption that the communication between sensor and controller is instantaneous. Similarly, transitions `lower` and `raise` in `Controller` synchronize with the corresponding transitions in the `Gate` module. We will prove that the system `[Gate||Controller||Trains]` is safe and useful.

Note that the modules have no shared variables, but communicate only through synchronous transitions.

To relate the presence of trains with the behavior of the gate, `Controller` uses the local array `trainHere[i]` to record the trains at each track, while `gs` records the status of the gate. Transitions `trainsEnter` and `trainsExit` in the controller have no enabling conditions, since the controller cannot control when they are taken. In contrast, transitions `raise` and `lower` are determined by the controller. Since there is a delay between lowering the gate and the gate being down, the controller has to plan when to lower it. Therefore, when the sensor is triggered, the controller estimates the earliest possible time the train will be in the intersection, by setting `schedTime` to `T` (the current time) `+ conMinI`. For the system to be safe, we must assume that this estimate is conservative, that is, `conMinI` is smaller than the actual minimum time, `minTimeToI`, that a train may take to reach the intersection from the time it triggers the sensor. This assumption is expressed by axiom `ACT1` of Figure 8. Note that `conMinI` is an internal system constant of the controller, while `minTimeToI` represents the actual value as exhibited by the trains.

```
Clocked Transition Module Gate

in gateRiseTime : real where gateRiseTime > 0
in gateDownTime : real where gateDownTime > 0

type gateStatus = {up, down, goingUp, goingDown}
local gate: gateStatus where gate = up
local lastDown, lastUp: real

Progress ( (gate = goingUp   --> T <= lastUp) /\
           (gate = goingDown --> T <= lastDown) )

Transition lower :
     assign   (gate,lastDown) :=
              if gate = up \/ gate = goingUp
                 then (goingDown,T + gateDownTime)
                 else  (gate,lastDown)

Transition raise :
     assign   (gate,lastUp) :=
              if gate = down \/ gate = goingDown
                 then (goingUp,T + gateRiseTime)
                 else  (gate,lastUp)

Transition isUp :
     enable   gate = goingUp
     assign   gate  :=  up

Transition isDown :
     enable   gate = goingDown
     assign   gate  :=  down
```

Fig. 5. The `Gate` clocked transition module

The first conjunct of the `Controller` time-progress condition ensures that the gate is lowered in time, according to the controller's estimate. Again, this estimate must be conservative, as stated by axiom `AGC1`. To ensure that the gate is not raised too early, the enabling condition of `raise` requires that there be no trains in the intersection. In fact, it states that the gate cannot be raised if another train approaches and there is no time to raise the gate, let one automobile pass, and lower the gate again. Constant `gammaUp` is a conservative estimate of the time needed to raise the gate, and `carPassingTime` is the minimum time for an automobile to pass the intersection. These controller restrictions on the gate behavior ensure a safe intersection.

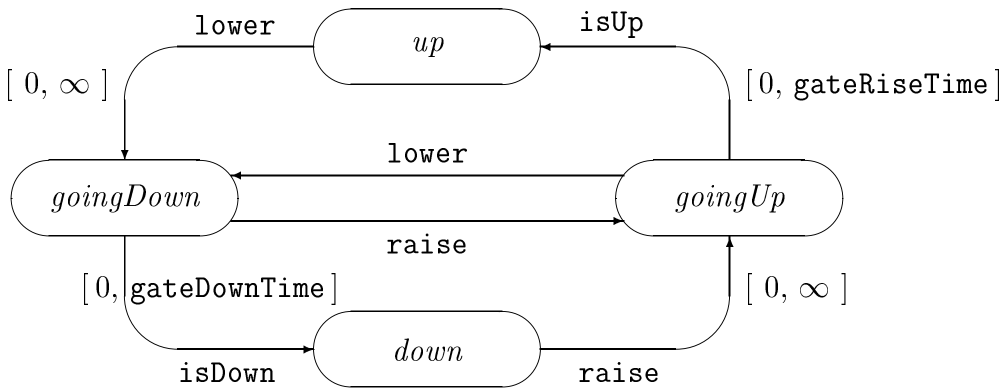



Fig. 6. Transitions for the `Gate` module

Further controller restrictions on the gate behavior are necessary to make the system useful. First, the gate should not be lowered when it is not necessary. This restriction is ensured by the second conjunct of the enabling condition, where `beta` is a positive constant that gives the controller some time-interval to lower the gate. Second, the gate should not stay down without reason. This restriction is enforced by the second conjunct of the controller's time-progress condition, which states that the controller can keep the gate down only if there is a train that is sufficiently near. There is no corresponding slack variable `beta'` for raising the gate, since we do not model urgency in raising the gate.

### *6.2 Modular Analysis*

Before proving properties over the entire system, we analyze the modules individually and in various combinations. By Theorem 3.1 of Section 4, properties proved for individual modules, or pairwise compositions of them, will also hold for the entire system, and can thus be used as axioms when verifying properties of the complete system.

Figure 9 lists some properties proved valid over single or pairwise combinations of modules. Property `GC1` states to what extent the controller knows the status of the gate (it cannot distinguish between `goingUp` and `up`). Property `GC2` states that whenever the controller has given the command *lower*, the gate is guaranteed to be down within `gateDownTime`. Property `GC3` states the main property of the controller, namely that the gate will be down before a train is expected to enter the intersection. Property `TC1` states that the status of the train is correctly represented internally in the controller. Property `TC2` states that the controller's estimate of the expected arrival time is conservative. All these properties were proved with STeP using rule INV. Many of the resulting verification conditions were proved automatically by STeP's decision procedures, and the rest required straightforward use of the interactive prover.

```
Clocked Transition Module Controller

in conMinI : real where conMinI > 0
in gammaDown: real where gammaDown > 0
in gammaUp : real where gammaUp > 0
in carPassingTime: real where carPassingTime > 0
in beta : real where beta > 0

type gstatus = {gUp,gDown}
local gs: gstatus where gs = gUp
local schedTime: array[1..N] of real
local trainHere: array[1..N] of bool
    where Forall i:[1..N]. !trainHere[i]

Progress (gs = gUp --> Forall i:[1..N].
  (trainHere[i] --> T < schedTime[i] - gammaDown)) /\
  (gs = gDown --> Exists i:[1..N]. (trainHere[i] /\
  schedTime[i] <= T + gammaUp + carPassingTime + gammaDown))

Transition trainsEnter [i:[1..N]] :
  assign schedTime[i] := T + conMinI,
         trainHere[i] := true

Transition lower :
  enable gs = gUp /\ Exists i:[1..N].(trainHere[i] /\
           schedTime[i] <= T + gammaDown + beta)
  assign gs := gDown

Transition trainsExit [i:[1..N]] :
  assign trainHere[i] := false

Transition raise :
  enable gs = gDown /\ Forall i:[1..N].(trainHere[i] -->
    T + gammaUp + carPassingTime + gammaDown < schedTime[i])
  assign gs := gUp
```

Fig. 7. The Controller clocked transition module

```
AXIOM AC1 (feasibility): gammaDown < conMinI
AXIOM ACT1 (conservative estimate): conMinI < minTimeToI
AXIOM AGC1 (conservative est.): gateDownTime < gammaDown
AXIOM AGC2 (conservative est.): gateRiseTime < gammaUp
```

Fig. 8. Axioms relating system constants

```
PROPERTY G1: [Gate] |= gate = goingDown ==> T <= lastDown
PROPERTY G2: [Gate] |= gate = goingUp ==> T <= lastUp

PROPERTY T1: [Trains] |=
  Forall i:[1..N].(T < firstEnter[i] ==> trains[i] != I)
PROPERTY T2: [Trains] |=
  Forall i:[1..N].(trains[i] = P ==>
        firstEnter[i] <= T + minTimeToI /\
        T <= lastEnter[i] /\
        lastEnter[i] - firstEnter[i] = maxTimeToI - minTimeToI)

PROPERTY C1: [Controller] |= gs = gUp ==>
  Forall i:[1..N].(trainHere[i] --> T < schedTime[i] - gammaDown)

PROPERTY GC1: [Gate || Controller] |=
  gs = gDown <==> (gate = goingDown \/ gate = down)
PROPERTY GC2: [Gate || Controller] |=
  gs = gDown ==> lastDown <= T + gateDownTime
PROPERTY GC3: [Gate || Controller] |= gs = gDown ==>
  Forall i:[1..N].(trainHere[i] --> lastDown < schedTime[i])

PROPERTY TC1: [Controller || Trains] |=
  Forall i:[1..N].(trainHere[i] <==> trains[i] != notHere)
PROPERTY TC2: [Controller || Trains] |=
  Forall i:[1..N].(trainHere[i] ==> schedTime[i] < firstEnter[i])
```

Fig. 9. Modularly valid auxiliary properties

Properties G1 and G2 were generated by the automatic invariant generation method described in Section 5.1.

### *6.3 Specification*

The informal requirements that the system must be "safe" and "useful" must be precisely formulated as formulas in temporal logic. The requirement of safety, that the gate is down whenever there is a train in the intersection, is straightforward to express:

```
PROPERTY safety: [Gate || Controller || Trains] |=
  (Exists i:[1..N].trains[i] = I) ==> gate = down
```

The "usefulness" (or *utility*) requirement states that the gate should be up when there is no good reason for it to be down, and is more difficult to formalize. In [HL94] this requirement is formulated as follows (adapted to our notation and naming of constants): For a state $s_i$ in a computation $\sigma : s_0, s_1, \ldots, s_i, \ldots$ such that $s_i[gate] \neq up$, where $s_i[x]$ is the value of variable $x$ in state $s_i$, one of the following conditions must hold:

(i) A train has passed recently: there exists $j \leq i$ such that $s_j[trains[k]] = I$ for some $k \leq N$ and $T_j \geq T_i - maxPost$, where $T_i$ is the value of the master clock in state $s_i$ and *maxPost* is the maximum time we allow for the gate to move up.

(ii) A train will arrive soon: there exists $j \geq i$ such that $s_j[trains[k]] = I$ for some $k \leq N$ and $T_j \leq T_i + maxPre$, where *maxPre* is the maximum time we allow for the gate to lower before a train arrives at the intersection.

(iii) There is not enough time between the passage of two trains to let an automobile pass: there exist $n \leq i,\ m \geq i$, such that $s_n[trains[k]] = I$, $s_m[trains[\ell]] = I$ for some $k, \ell \leq N$, and $T_m - T_n \leq maxPre + maxPost + carPassingTime$, where *carPassingTime* is the minimum time required for one automobile to pass over the tracks. Figure 10 illustrates this case.

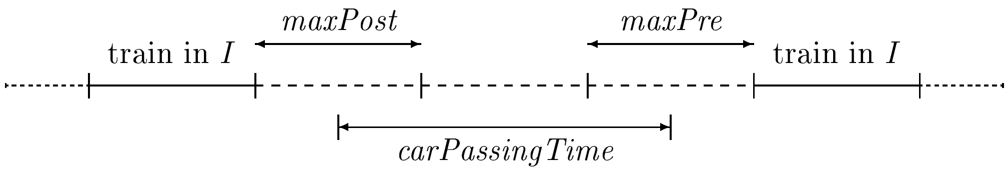



Fig. 10. An interval where the gate should stay down

We reformulate this property as the two *wait-for* properties of Figure 11, where `Awaits` stands for the $\mathcal{W}$ (wait-for) operator. Specification variables `maxPre` and `maxPost`, also defined in Figure 11, are bounds on the time the gate is not up before and after a train is in the intersection. To ensure that there exists a controller such that the overall system satisfies the specification, their values must be sufficiently large, as expressed by axioms `ASP1` and `ASP2` in Figure 12. Axiom `ASP1` states that we cannot require the gate to be up within the time it takes to raise the gate. Similarly, axiom `ASP2` states `maxPre` should be larger than the time it takes to lower the gate, as expressed by `gateDownTime`. To account for the uncertainty in how long it takes the train to reach the intersection we add the term `maxTimeToI - minTimeToI`; the controller must assume the shortest time to decide when to lower the gate, but the train may actually arrive at the latest possible time. The term `margin` accounts for the fact that the controller does not have access to the precise values of `gateDownTime` and `minTimeToI` and therefore must make conservative estimates in `gammaDown` and `conMinI`, thus increasing the time the gate is down. Finally, axiom `ASP4` limits how conservative these estimates can be by requiring that `margin` does not exceed the time it takes a car to cross the intersection.

```
value maxPre, maxPost: real
value t: real

PROPERTY utility1: [Gate || Controller || Trains] |=
gate = goingDown /\ t = T ==>
  T <= t + maxPre Awaits Exists i:[1..N].trains[i] = I

PROPERTY utility2: [Gate || Controller || Trains] |=
t = T /\ (Forall i:[1..N] .trains[i] != I) ==>
  (T <= t + maxPre + carPassingTime + maxPost)
  Awaits
  (T <= t + maxPost /\ gate = up) \/
  (T <= t + maxPre + carPassingTime + maxPost /\
        Exists i:[1..N].trains[i] = I)
```

Fig. 11. Specification of the utility property

```
AXIOM ASP1: maxPost > gammaUp
AXIOM ASP2: maxPre >
        gateDownTime + maxTimeToI - minTimeToI + margin
AXIOM ASP3: margin > (gammaDown - gateDownTime)
                + (minTimeToI - conMinI) + beta
AXIOM ASP4: margin <= carPassingTime
```

Fig. 12. Axioms relating specification variables to system constants

Property `utility1` states that if the gate is going down at time `t`, then a train will enter the intersection within time `maxPre`. This forbids spurious or premature lowering of the gate. Property `utility2` states that if there are no trains in the intersection at time `t`, then the gate is up within time `maxPost` or another train approaches and will reach the intersection within time `maxPre + carPassingTime + maxPost`. This forbids keeping the gate down unnecessarily.

### *6.4 Verification*

**Property `safety`:** The property `safety` was proved with STeP using the INV verification rule. The property is inductive relative to the previously proved modular properties shown in Figure 9; therefore, no auxiliary assertion $\varphi$ had to be constructed. Of the 10 verification conditions, only two required non-trivial user interaction to be proved.

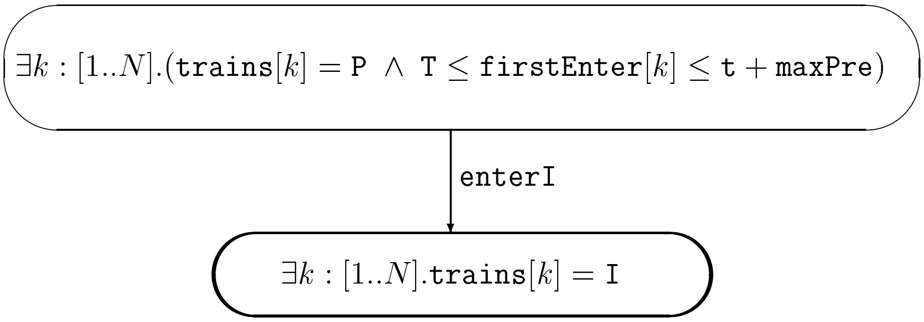



Fig. 13. Wait-for verification diagram for `utility1`

**Property `utility1`:** The *wait-for* verification diagram of Figure 13 was used to prove `utility1`. The boldface boundary indicates a terminal node. This diagram generates eight verification conditions: the five $\mathcal{S}$-verification conditions were established automatically. An additional invariant is required to prove that the antecedent of `utility1` implies the formula in the top-most node. This invariant is summarized in `gateInv` below, proved using INV. The other two $\varphi$-verification conditions were proved with trivial user interaction.

```
PROPERTY gateInv: [Trains || Controller || Gate] |=
  gate = goingDown ==> Exists i:[1..N].
    (trains[i] = P /\ firstEnter[i] - gammaDown - beta <= T)
```

**Property `utility2`:** The *wait-for* verification diagram of Figure 14 was used to prove `utility2`. It generates 36 verification conditions, all of which were proved valid with the help of the previously proved invariants. Twelve of the verification conditions were proved automatically, and the rest required some user interaction.

### *6.5 Proving Non-Zenoness*

Before we discuss receptiveness in Section 6.6, we give a global non-Zenoness proof, considering the composition of all three modules.

A non-Zenoness diagram for `[Gate || Controller || Trains]` is given in Figure 15.[3] This diagram uses *encapsulation conventions* [MP94]: a compound node labeled by an assertion $f$ adds $f$ as a conjunct to each one of its subnodes, and an edge arriving at (or departing from) a compound node represents a set of edges that arrive at (or depart from) each of its subnodes.

[3] Quantifiers in STeP have maximal scope, so some parentheses are omitted.

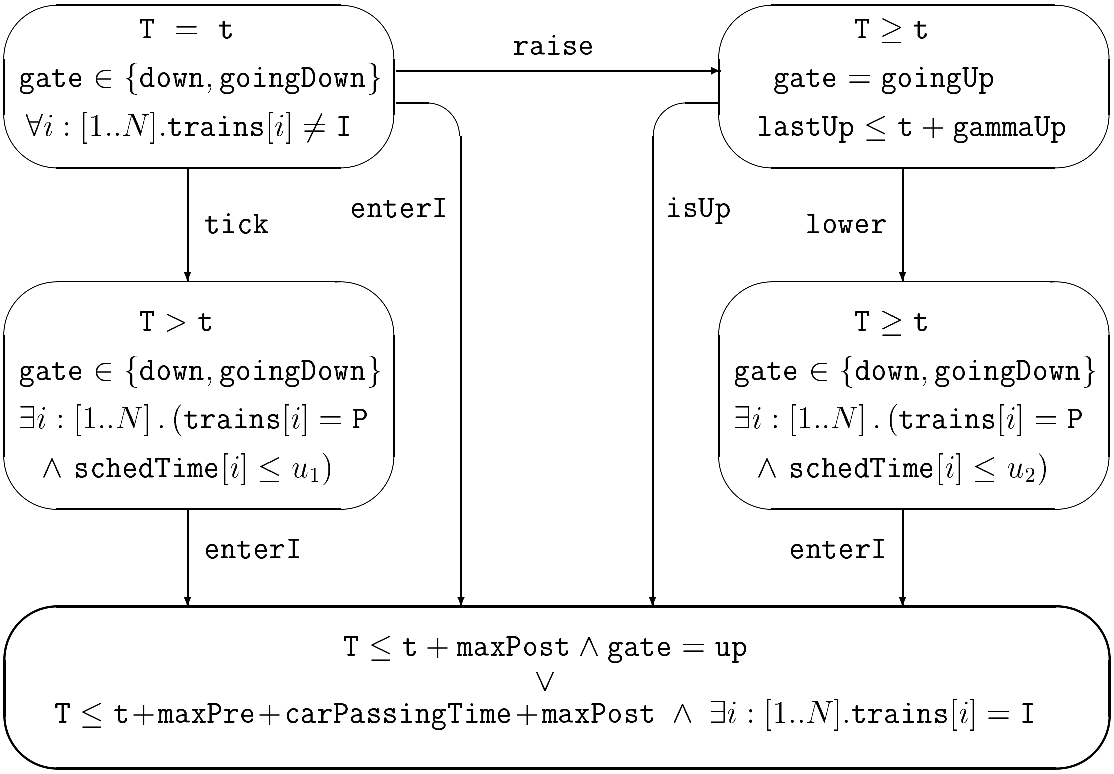
The ranking functions $\delta_i$ associated with nodes $n_i$ are given by:

$$\delta_i \stackrel{\text{def}}{=} (i, \emptyset, 0) \qquad \text{for } i = 0, 3, \ldots, 7, \text{ and}$$
$$\delta_i \stackrel{\text{def}}{=} (1, \{k : [1..N] \mid \text{urgent}(k)\}, i) \qquad \text{for } i = 1, 2.$$

The range of the $\delta_i$'s is the set $\mathcal{N} \times 2^{[1..N]} \times \mathcal{N}$, a well-founded domain using, as the well-founded order, the lexicographic extension $\langle <, \subset, < \rangle$ of the $<$-ordering on $\mathcal{N}$ and the subset ordering $\subset$. The predicate urgent($i$) holds if $\texttt{trains}[i] = \texttt{P}$ and $t_0 + \epsilon > \texttt{lastEnter}[i]$: an *urgent* train must enter I before time can advance beyond $t_0 + \epsilon$.

STeP was used to check this non-Zenoness diagram, generating the associated verification conditions and proving their system validity.

Given our interleaving model of computation, it is essential that at any reachable state there is only a finite number of urgent trains. Otherwise, an infinite number of events would have to happen in a bounded time interval. Other computational models, such as those based on partial orders [PPH97], can account for event independence, and do not need the restriction to a finite number of trains.

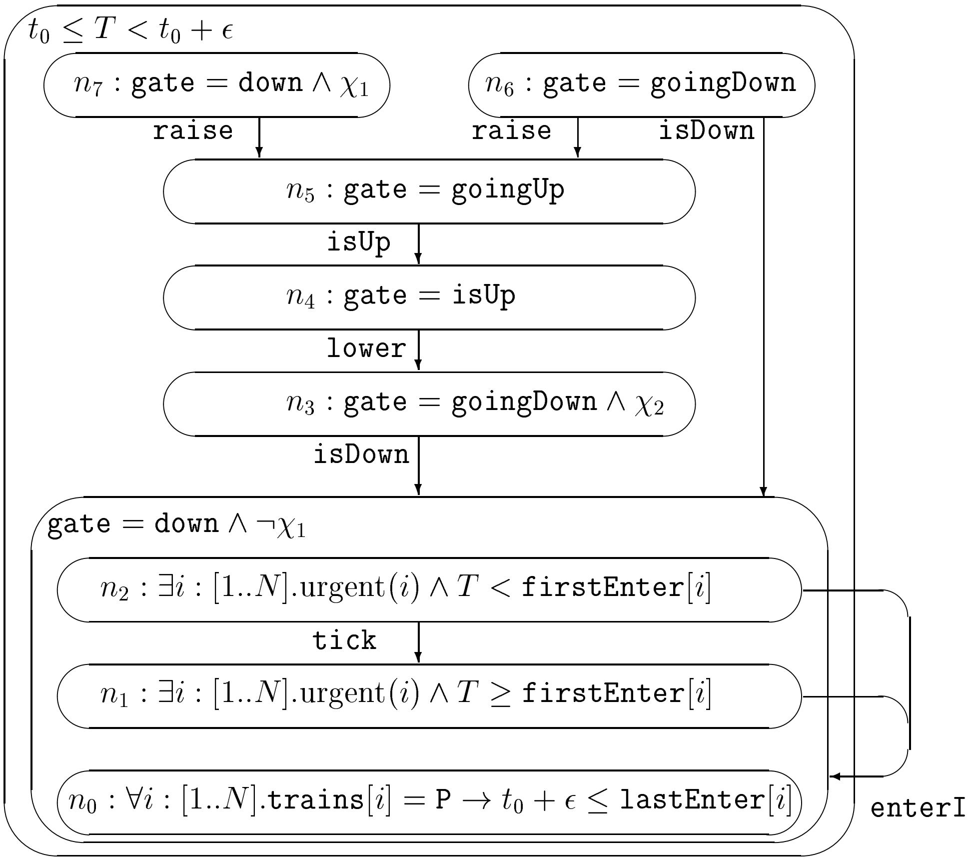



$u_1 = \texttt{t} + \texttt{gammaUp} + \texttt{carPassingTime} + \texttt{gammaDown}$
$u_2 = \texttt{t} + \texttt{gammaUp} + \texttt{beta} + \texttt{gammaDown}$

Fig. 14. Wait-for verification diagram for `utility2`

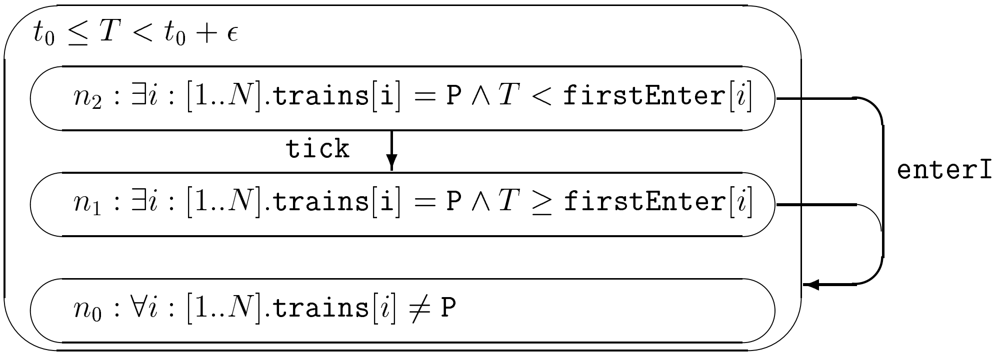
### 6.6 *Proving Receptiveness*

In the spirit of modularity, we now sketch, more informally, a modular non-Zenoness proof for our GRC specification, by showing that the individual modules are receptive and then inferring non-Zenoness for the combined system using Proposition 4.7. As we will see, the combined modular proof effort is more costly than the global non-Zenoness proof above. The advantage of a modular justification is that modules can be replaced by other receptive modules, while still ensuring the non-Zenoness of the entire system. (However, this justification has not yet been mechanically checked using STeP.)




$$
\begin{aligned}
\chi_1 &: \forall i : [1..N]\,.\,\mathtt{trainHere}[i] \rightarrow \\
&\qquad \mathtt{schedTime}[i] > T + \mathtt{gammaUp} + \mathtt{carPassingTime} + \mathtt{gammaDown} \\
\chi_2 &: \exists i : [1..N]\,.\,\mathtt{trainHere}[i] \wedge \mathtt{schedTime}[i] \le t_0 + \epsilon + \mathtt{gammaDown} \\
\mathrm{urgent}(i) &: \mathtt{trains}[i] = \mathtt{P} \wedge t_0 + \epsilon > \mathtt{lastEnter}[i] \\
\epsilon &: \min\{\mathtt{beta},\ \mathtt{gammaUp},\ \mathtt{maxtimetoI} - \mathtt{mintimetoI},\ \mathtt{gammaDown}\}
\end{aligned}
$$

Fig. 15. Global non-Zenoness diagram




$\delta(n_i) = (|\{k : [1..N] \mid \mathtt{trains[k]} = \mathtt{P}\}|, i)$

Fig. 16. Receptiveness diagram for the `Trains` module

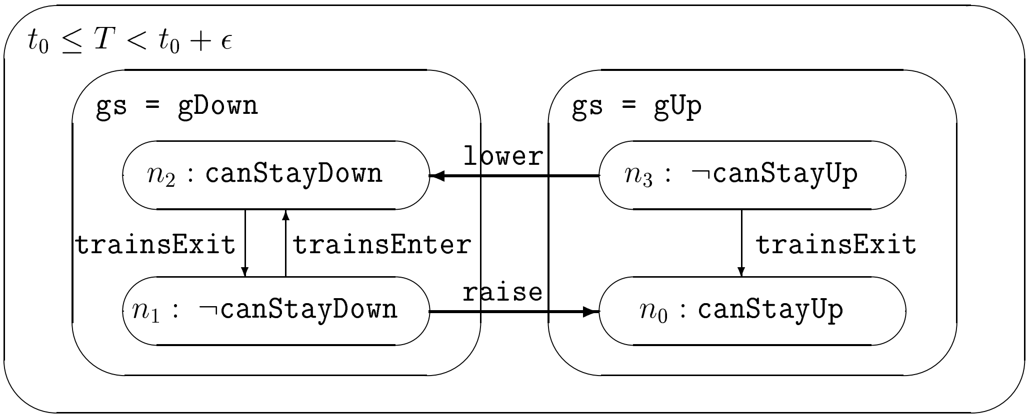
For the `Trains` module, all transitions are `controlled`. For the `Gate` module, `lower` and `raise` are `external` (decided by the `Controller` module), while `isUp` and `isDown` are `controlled`. For the `Controller` module, `lower` and `raise` are `controlled`, while `trainsEnter` and `trainsExit` are `external` (decided by the `Trains` module).

We present receptiveness diagrams to show that the `Trains`, `Controller` and `Gate` modules are receptive. In our diagrams, we will use bold edges to indicate `controlled` or *tick* transitions, which can be chosen by the module, and single edges for `external` ones.

The diagram in Figure 16 represents a proof that the `Trains` module is receptive. The ranking functions for nodes $n_2$ and $n_1$ count the number of trains that still have to enter the intersection (that is, must perform a transition) before time can be advanced freely, since taking transition `enterI` is forced by the progress condition. Note that the module may choose whether to take any transition that is not forced by the progress condition; in particular, it can choose not to have a train enter $P$.

The diagram in Figure 17 represents a proof that the `Controller` module is receptive. In node $n_3$ the module can always take the `lower` transition, since $\epsilon \leq \mathtt{beta}$, and the resulting state will satisfy `canStayDown` (the formula $\neg\mathtt{canStayDown}$ is equivalent to $\chi_1$ used in Figure 15). In $n_2$ the module can let time advance beyond $t_0 + \epsilon$, justified by the existence of at least one train that is close enough, and that will stay close enough until it exits. In $n_1$ the module can always take the `raise` transition, since $\neg\mathtt{canStayDown}$ is its enabling condition. The resulting state will satisfy `canStayUp`, since $\epsilon \leq \mathtt{gammaUp} + \mathtt{carPassingTime}$. Finally, node $n_0$ allows time to advance beyond $t_0 + \epsilon$. The absence of an environment edge $\langle n_0, n_3 \rangle$ is justified by $\epsilon \leq \mathtt{conMinI} - \mathtt{gammaDown}$: whenever a train enters time will advance beyond $t_0 + \epsilon$ before the controller has to lower the gate.

The only non-trivial MSCS in the diagram is $\{n_1, n_2\}$, which contains only `external` transitions, so we can use the default well-founded order.




canStayUp: $\forall i : [1..N]\,.\,\texttt{trainHere}[i] \rightarrow t_0 + 2\epsilon < \texttt{schedTime}[i] - \texttt{gammaDown}$

canStayDown: $\exists i : [1..N]\,.\,(\texttt{trainHere}[i] \wedge$
$\quad \texttt{schedTime}[i] \leq T + \texttt{gammaUp} + \texttt{carPassingTime} + \texttt{gammaDown})$

$\epsilon : \min\{\texttt{conMinI} - \texttt{gammaDown},\ \texttt{beta},\ \texttt{gammaUp} + \texttt{carPassingTime}\}/2$

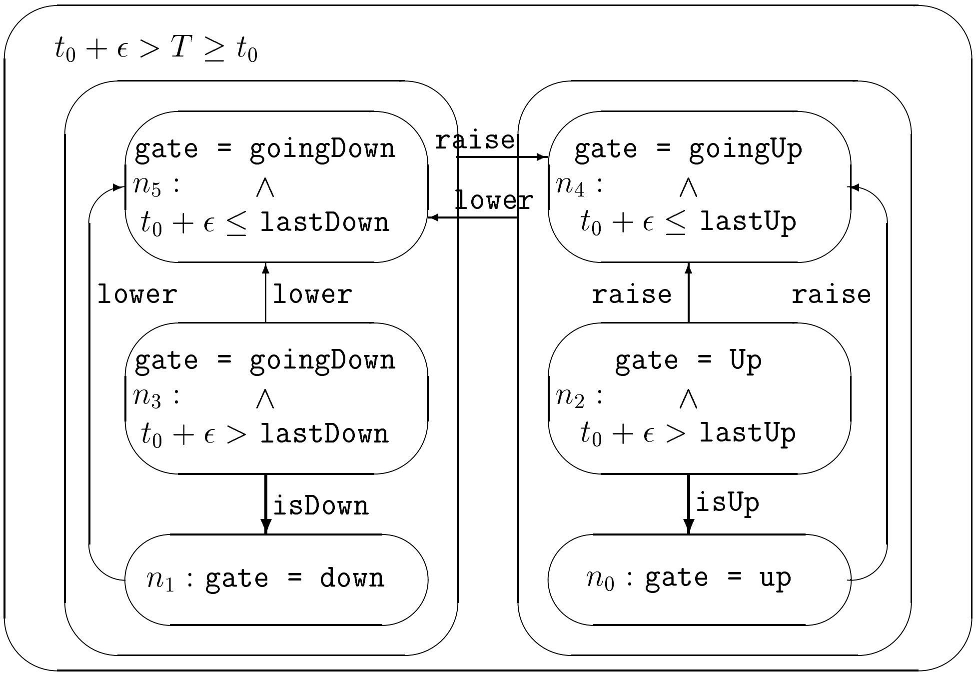
Fig. 17. Receptiveness diagram for the `Controller` module

The diagram in Figure 18 proves that the `Gate` module is receptive. The states where the module cannot let time advance beyond $t_0 + \epsilon$ are covered by nodes $n_2$ and $n_3$. However, from these nodes, transitions `isUp` and `isDown`, respectively, are guaranteed to be enabled and can take the module to a node where time can be advanced. In all other nodes the module can let time advance beyond $t_0 + \epsilon$. As regards the environment condition, the absence of environment edges $\langle n_1, n_3\rangle$ and $\langle n_0, n_2\rangle$ is justified by our choice to have $\epsilon$ smaller than the minimum of `gateRiseTime` and `gateDownTime`. This choice of $\epsilon$ guarantees that whenever a `raise` or `lower` transition is taken while $T \geq t_0$, the module is not forced by the progress condition to take the `isUp` or `isDown` transition before $t_0 + \epsilon$.

## 7 Related Work

### *7.1 The Railroad Crossing Benchmark*

The GRC and similar examples have been analyzed using many different specification techniques and tools. The first machine-verified proof of safety and utility properties of a railroad crossing controller appeared in [Sha93]. A linear-time model for real-time specifications was encoded in the higher-order specification language of the PVS system [ORR+96], and formal proofs were conducted using the PVS proof-assistant. PVS was also used in [Ska94] to verify




$$\delta(n_0) = \delta(n_1) = 1,\ \delta(n_2) = \delta(n_3) = 2,\ \delta(n_4) = \delta(n_5) = 0$$

$$\epsilon = \min\{\texttt{gateRiseTime}, \texttt{gateDownTime}\}/2$$

Fig. 18. Receptiveness diagram for the `Gate` module

a railroad crossing system formulated in the Duration Calculus [CHR92]. The work of [AB96] uses Timed Statecharts [MMP92] to encode real-time systems, and the HOL prover [GM93] to conduct a formal proof. The version of the railroad controller we present appeared in [HL94], which reports preliminary experience with PVS and the Larch prover [GG91] to obtain a formally checked proof.

When numerical values are substituted for some of the symbolic parameters in the problem formulation, it is possible to obtain systems suitable for automatic real-time model checkers such as HyTech [HH95], Kronos [DOTY96], and Uppaal [BLL+96]; see, for instance, [DY95]. These tools are based on representing bisimilar regions of the reachable state-space using linear constraints over the reals. The constraint logic programming language CLP($\mathcal{R}$) includes built-in solvers for constraints over the reals. CLP($\mathcal{R}$) is used in [Urb96] to check a hybrid solution to the GRC.

### *7.2 Real-time Verification Frameworks*

A related deductive framework for the analysis of real-time systems is presented in [HK94]. That paper provides a bisimulation-based translation from a class of real-time systems and specifications to untimed versions that are amenable to standard verification methods for infinite-state reactive systems. [HK94] also studies methods for translating liveness properties into safety formulas when progress is guaranteed within a computable bound.

The clocked transition system approach using explicit clock variables can be traced to [AL92]. There, system specifications in the Temporal Logic of Actions (TLA) are extended by a clock variable $now$. Since systems and specifications are modeled in the same logic, the real-time TLA system description includes additional conjunctions constraining real-time behaviors of transitions, and requiring non-Zenoness. Modularity and assumption-guarantee specifications are handled in a version of TLA that mixes classical and intuitionistic logical connectives. Proof rules for the modular verification of timed transition systems are presented in [Cha93] and [CMP94].

Systems driven by a digital clock where time advances by discrete intervals of fixed size can be analyzed using automatic model checking tools. In [CC95] the CTL branching-time logic is extended with metric temporal operators, and the computational model is enhanced with *timed transition graphs*. In [Ost97] the StateTime visual toolset allows the designer to encode a discrete-event real-time system, using a statechart graphical interface with debugging utilities. The computational model is that of *timed transition modules* (TTM's), where time is incremented by discrete amounts. TTM's are translated into standard fair transition systems; STeP's model checker and theorem-proving utilities are then used to verify temporal specifications.

### *7.3 Proving Non-Zenoness*

Modular conditions for ensuring non-Zenoness are discussed in [AL92], [LV92], [GSSL94] and [AH97]. Like the one we use, they are variations on the sufficient, but not necessary, modular condition of receptiveness, introduced in [Dil89] in the general context of open circuit components.

Our non-Zenoness diagrams are inspired by the *possibility diagrams* used in [KMP98]. Although the property itself is not expressible in LTL, another approach to proving non-Zenoness within an LTL framework is reported in [Lam95]. The validity of possibility properties over a particular system is reduced to the validity of *closure formulas*, based on TLA formulas that describe the system itself.

## 8 Conclusions

We have presented a machine-checked modular verification of a real-time control system, using clocked transition modules as the computational model and temporal logic to capture the requirements. The STeP verification system was used to verify the specified properties and to check that the system is well-specified, that is, that it is non-Zeno.

To verify non-Zenoness and the related modular property of receptiveness, we proposed non-Zenoness and receptiveness diagrams, which reduce the proof of such properties to that of first-order verification conditions. These diagrams provide an intuitive, visual representation of the corresponding proofs.

The proofs of the safety and utility properties were simplified by automatically generated auxiliary invariants. Diagrams were also used to reduce the verification of these temporal properties to proofs of first-order verification conditions. Many of these verification conditions were proven automatically by STeP's decision procedures, while the rest required some user interaction to prove their validity.

Future work aims at increasing the use of model checking techniques in the verification of real-time systems by using abstraction [CU98,Uri98], and enhancing the verification support, including machine-guided construction of verification diagrams [Sip98] and more powerful decision procedures [Bjø98].

**Acknowledgements:** We thank Uri Lerner and the anonymous reviewers for their feedback and comments.

## References

[AB96] J. Armstrong and L. Barroca. Specification and verification of reactive system behaviour: The railroad crossing example. *Real-Time Systems*, 10:143–178, 1996.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AH96] R. Alur and T.A. Henzinger, editors. *Proc. 8$^{th}$ Intl. Conference on Computer Aided Verification*, vol. 1102 of *LNCS*. Springer-Verlag, July 1996.

[AH97] R. Alur and T.A. Henzinger. Modularity for timed and hybrid systems. In *Proc. 8th Intl. Conference on Concurrency Theory*, vol. 1243 of *LNCS*, pages 74–88. Springer-Verlag, 1997.

[AHS96] R. Alur, T.A. Henzinger, and E.D. Sontag, editors. *Hybrid Systems III: Verification and Control*, vol. 1066 of *LNCS*. Springer-Verlag, 1996.

[AL92] M. Abadi and L. Lamport. An old-fashioned recipe for real time. In de Bakker et al. [dHdR92], pages 1–27.

[BBC+95] N.S. Bjørner, A. Browne, E.S. Chang, M. Colón, A. Kapur, Z. Manna, H.B. Sipma, and T.E. Uribe. STeP: The Stanford Temporal Prover, User's Manual. Technical Report STAN-CS-TR-95-1562, Computer Science Department, Stanford University, November 1995.

[BBC+96] N.S. Bjørner, A. Browne, E.S. Chang, M. Colón, A. Kapur, Z. Manna, H.B. Sipma, and T.E. Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In Alur and Henzinger [AH96], pages 415–418.

[BBM97] N.S. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, February 1997. Preliminary version appeared in 1$^{st}$ *Intl. Conf. on Principles and Practice of Constraint Programming*, vol. 976 of LNCS, pp. 589–623, Springer-Verlag, 1995.

[Bjø98] N.S. Bjørner. *Integrating Decision Procedures for Temporal Verification*. PhD thesis, Computer Science Department, Stanford University, November 1998.

[BLL+96] J. Bengtsson, K. Larson, F. Larsson, P. Pettersson, and W. Yi. Uppaal — a Tool Suite for Automatic Verification of Real-Time Systems. In Alur et al. [AHS96], pages 232–243.

[BLS96] S. Bensalem, Y. Lakhnech, and H. Saidi. Powerful techniques for the automatic generation of invariants. In Alur and Henzinger [AH96], pages 323–335.

[BMS95] A. Browne, Z. Manna, and H.B. Sipma. Generalized temporal verification diagrams. In *15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, vol. 1026 of *LNCS*, pages 484–498. Springer-Verlag, 1995.

[BSU97] N.S. Bjørner, M.E. Stickel, and T.E. Uribe. A practical integration of first-order reasoning and decision procedures. In *Proc. of the* 14$^{th}$ *Intl. Conference on Automated Deduction*, vol. 1249 of *LNCS*, pages 101–115. Springer-Verlag, July 1997.

[CC95] S. Campos and E.M. Clarke. Real-time symbolic model checking for discrete time models. In T. Rus and C. Rattray, editors, *Theories and Experiences for Real-Time System Development.*, AMAST. World Scientific Publishing Company, 1995.

[Cha93] E.S. Chang. *Compositional Verification of Reactive and Real-Time Systems.* PhD thesis, Computer Science Department, Stanford University, Stanford, California, 1993. Tech. Report STAN-CS-TR-94-1522.

[CHR92] Z. Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1992.

[CMP94] E.S. Chang, Z. Manna, and A. Pnueli. Compositional verification of real-time systems. In *Proc. 9th IEEE Symp. Logic in Comp. Sci.*, pages 458–465. IEEE Computer Society Press, 1994.

[CU98] M.A. Colón and T.E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In A.J. Hu and M.Y. Vardi, editors, *Proc.* $10^{th}$ *Intl. Conference on Computer Aided Verification*, vol. 1427 of *LNCS*, pages 293–304. Springer-Verlag, July 1998.

[dHdR92] J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rosenberg, editors. *Proceedings of the REX Workshop "Real-Time: Theory in Practice"*, vol. 600 of *LNCS*. Springer-Verlag, 1992.

[Dil89] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* MIT Press, 1989.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In Alur et al. [AHS96], pages 208–219.

[DY95] C. Daws and S. Yovine. Verification of multirate timed automata with KRONOS: two examples. Technical Report Spectre-95-06, VERIMAG, April 1995.

[Eme90] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, vol. B, pages 995–1072. Elsevier Science Publishers (North-Holland), 1990.

[GG91] S.J. Garland and J.V. Guttag. A guide to LP, the Larch Prover. Technical Report 82, Digital Equipment Corportation, SRC, December 1991.

[GM93] M. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[GSSL94] R. Gawlick, R. Segala, J.F. Søgaard-Andersen, and N.A. Lynch. Liveness in timed and untimed systems. In S. Abiteboul and E. Shamir, editors, *Proc. 21st Intl. Colloq. Aut. Lang. Prog.*, vol. 820 of *LNCS*. Springer-Verlag, 1994.

[HH95] T.A. Henzinger and P. Ho. HyTech: The Cornell hybrid technology tool. In *Hybrid Systems II*, vol. 999 of *LNCS*, pages 265–293. Springer-Verlag, 1995.

[HJL93] C. Heitmeyer, R.D. Jeffords, and B. Labaw. A benchmark for comparing different approaches for specifying and verifying real-time systems. In *Proc. Tenth Intl. Workshop on Real-Time Operating Systems and Software*, 1993.

[HK94] T.A. Henzinger and P.W. Kopke. Verification methods for the divergent runs of clock systems. In H. Langmaack, W.P. de Roever, and J. Vytopil, editors, *Proc. Third Intl. Symposium on Formal Techniques in Real Time and Fault Tolerant Systems*, vol. 863 of *LNCS*, pages 351–372. Springer-Verlag, September 1994.

[HL94] C. Heitmeyer and N. Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *Proc. IEEE Real-Time Systems Symposium*, pages 120–131. IEEE Press, December 1994.

[HMP94] T.A. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for timed transition systems. *Inf. and Comp.*, 112(2):273–337, August 1994.

[KMP96] Y. Kesten, Z. Manna, and A. Pnueli. Verifying clocked transition systems. In Alur et al. [AHS96], pages 13–40.

[KMP98] Y. Kesten, Z. Manna, and A. Pnueli. Verification of clocked and hybrid systems. In G. Rozenberg and F.W. Vaandrager, editors, *Lectures on Embedded Systems*, vol. 1494 of *LNCS Tutorial*, pages 4–73. Springer, Heidelberg, 1998.

[Lam95] L. Lamport. Proving possibility properties. Technical Report 137, Digital Equipment Corporation, Systems Research Center, July 1995.

[LV92] N. Lynch and F.W. Vaandrager. Forward and backward simulations for timing-based systems. In de Bakker et al. [dHdR92], pages 397–446.

[MMP92] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In de Bakker et al. [dHdR92], pages 447–484.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1991.

[MP94] Z. Manna and A. Pnueli. Temporal verification diagrams. In M. Hagiya and J.C. Mitchell, editors, *Proc. International Symposium on Theoretical Aspects of Computer Software*, vol. 789 of *LNCS*, pages 726–765. Springer-Verlag, 1994.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[MP96] Z. Manna and A. Pnueli. Clocked transition systems. Technical Report STAN-CS-TR-96-1566, Computer Science Department, Stanford University, April 1996.

[ORR+96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Alur and Henzinger [AH96], pages 411–414.

[Ost90] J.S. Ostroff. *Temporal Logic of Real-Time Systems.* Advanced Software Development Series. Research Studies Press (John Wiley & Sons), Taunton, England, 1990.

[Ost97] J.S. Ostroff. A visual toolset for the design of real-time discrete event systems. *IEEE Trans. on Control Systems Technology*, May 1997.

[PPH97] D.A. Peled, V. Pratt, and G.J. Holzmann, editors. *Workshop on Partial Order Methods in Verification (Princeton: 1996)*, vol. 29 of *DIMACS series in discrete mathematics and theoretical computer science*, July 1997.

[SDB96] J.X. Su, D.L. Dill, and C.W. Barrett. Automatic generation of invariants for processor verification. In *1st Intl. Conf. on Formal Methods in Computer-Aided Design*, vol. 1166 of *LNCS*, pages 377–388. Springer-Verlag, November 1996.

[Sha93] N. Shankar. Verification of real-time systems using PVS. In C. Courcoubetis, editor, *Proc.* $5^{th}$ *Intl. Conference on Computer Aided Verification*, vol. 697 of *LNCS*, pages 280–291. Springer-Verlag, June 1993.

[Sip98] H.B. Sipma. *Diagram-based Verification of Discrete, Real-time and Hybrid Systems*. PhD thesis, Computer Science Department, Stanford University, December 1998.

[Ska94] J.U. Skakkebæk. *A Verification Assistant for a Real-Time Logic*. PhD thesis, Technical University of Denmark, 1994.

[Urb96] L. Urbina. The generalized railroad crossing: Its symbolic analysis in CLP($\mathcal{R}$). In *Principles and Practice of Constraint Programming*, vol. 1118 of *LNCS*, pages 564–567, August 1996.

[Uri98] T.E. Uribe. *Abstraction-based Deductive-Algorithmic Verification of Reactive Systems*. PhD thesis, Computer Science Department, Stanford University, December 1998.