To appear in *ARTS-97*,
LNCS, Springer-Verlag, May 1997.

Deductive Verification of Real-time Systems Using STeP*

Nikolaj S. Bjørner, Zohar Manna,
Henny B. Sipma and Tomás E. Uribe
nikolaj|manna|sipma|uribe@cs.stanford.edu

Computer Science Department, Stanford University
Stanford, CA. 94305

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. As an example, we present the mechanical verification of the *generalized railroad crossing* case study using the Stanford Temporal Prover, STeP.

1 Introduction

The formalism of *clocked transition systems* was introduced by Manna and Pnueli [MP96] as a simple model for real-time systems. Continuous real-time is modeled explicitly as part of the system, in the form of 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 reactive systems [KMP96].

We add modularity to this framework, using *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, provided they satisfy the same properties. Properties of each module can be proven independently and later on used to prove properties of larger systems. Properties of a module can be established using deductive verification or appropriate algorithmic techniques.

The Stanford Temporal Prover, STeP [BBC+96], is a tool for verifying linear-time temporal properties of reactive systems. STeP includes verification rules,

* 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).

verification diagrams, and automatic invariant generation methods. As an example, 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, using STeP.

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. In Section 4 we briefly describe the STeP system, including recently added modular features, verification diagrams and invariant generation. Section 5 describes the specification and verification of the generalized railroad crossing example using STeP. We conclude with a brief overview of related work in Section 6.

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*. 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*, Σ. We say that s is a *p-state* if p is satisfied by s, written $s \models p$.
- Θ: The *initial condition*, a satisfiable assertion characterizing the possible initial states. We require that Θ implies $T = 0$.
- $\mathcal{T}$: A finite set of *transitions*. Each transition τ maps each state $s \in \Sigma$ into a set of τ*-successor* states $\tau(s) \subseteq \Sigma$. Each transition τ is described by a *transition relation* $\rho_\tau(V, V')$, 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. for every transition $\tau \in \mathcal{T}$, ρ_τ implies $T' = T$.
- Π: The *time-progress condition*. An assertion over V, used to specify a global restriction over 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 Δ that satisfies the global time-progress condition Π. *tick* is the only transition 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) $s_0 \models \Theta$ and (2) for each $j \geq 0$, $s_{j+1} \in \tau(s_j)$ for some $\tau \in \mathcal{T}_T$. In this case we say that τ is *taken* at s_j. An run is a *computation* if it satisfies *time divergence*, that is, the value of T increases beyond any bound. A CTS $\mathcal{S}$ is said to be *non-Zeno* if every finite prefix of a run of $\mathcal{S}$ can be extended to a computation. $Comp(\mathcal{S})$ 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 τ is enforced by associating a clock c_τ with τ and adding the condition $c_\tau \geq l$ to τ's enabling condition. When the enabling condition for τ first becomes true, c_τ is reset to 0. Upper bounds appear as constraints on Π: if u is the upper bound on τ, then $c_\tau \leq u$ should appear as a conjunct in Π. [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* as our property specification language [MP91]. Future temporal operators include $\Box$ (always), $\mathcal{W}$ (wait-for). $\Diamond$ (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 $\Box(p \rightarrow q)$.

Given a sequence of states $s_0, s_1, \ldots$, we define $s_i \models \Box p$ iff $s_j \models p$ for all $j \geq i$; $s_i \models p \,\mathcal{U}\, q$ iff there is some $j \geq i$ such that s_j satisfies q and p is true in $s_i, s_{i+1}, \ldots, s_{j-1}$; and $s_i \models p \,\mathcal{W}\, q$ iff $s_i \models \Box p$ or $s_1 \models p \,\mathcal{U}\, q$. (These are all the temporal operators we will use in this paper—see [MP95] for more details.) A computation satisfies φ iff its initial state s_0 satisfies φ.

A *past formula* is one that contains no future temporal operators. A *state formula*, or *assertion*, is one that does not contain any temporal operators.

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 temporal formula φ is *valid* over a CTS Φ, written as $\Phi \models \varphi$, if every computation of Φ satisfies φ.

3 Clocked Transition Modules

We adapt the compositional verification methodology of [CMP94] to clocked transition systems, adding the option of synchronizing transitions.

A *clocked transition module* (CTM) $M = \langle \mathcal{V}, \Theta, \mathcal{T}, \Pi \rangle$ contains the same elements as a CTS, augmented with the following:

- A partitioning 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 τ.

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* τ_E, which can arbitrarily change the values of all shared variables except T. Formally, τ_E has transition relation

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

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 (i.e. all variables ever available to other modules). $Comp(M)$ is defined to be equal to $Comp(\mathcal{S}_M)$. A temporal property φ is *valid* over a module M, or *modularly valid* for M, written $M \models \varphi$, iff $\mathcal{S}_M \models \varphi$.

3.2 Module Composition

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 τ 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 $\rho_\tau \wedge \bigwedge_{v \in \mathcal{V}_2^P} (v = v')$ (resp. $\rho_\tau \wedge \bigwedge_{v \in \mathcal{V}_1^P} (v = v')$). 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 $\rho_{\tau_1} \wedge \rho_{\tau_2}$.

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 5. 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 external 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.

3.3 Verification of Safety Properties

A *safety property* is one expressible 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 p, q and r. Many useful specifications which 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 δ 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. We will use the *invariance rule* INV and the *wait-for rule* WAIT, shown in Figure 1, to prove several 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*. $\{\varphi\}\,\tau\,\{\psi\}$ 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 τ_E transitions).

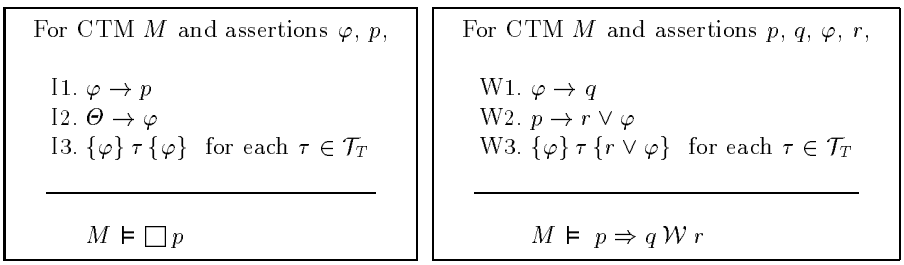

Fig. 1. Invariance rule INV and Waiting-for rule WAIT

We say that an assertion p is *inductive* for a module M if $\Box p$ can be proved using rule INV with φ 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.

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.

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

Theorem 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 σ is a computation of M_1. The initial state s_0 must satisfy $\Theta_1 \wedge \Theta_2$, and hence Θ_1. Consider a transition from s_i to s_{i+1} in σ. 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 τ_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, σ is also a computation of M_1. □

A consequence of Theorem 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. In this paper, we will only use this fact in the case of safety properties. Establishing non-trivial progress properties of a module requires extra restrictions on the environment and hiding transitions to avoid a deadlocking environment. Assumption-guarantee proof systems [AL92, Cha93] are essential for these applications.

Note that $Comp(M_1)$ will be, in general, a strict subset 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 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 τ_1 (resp. τ_2) that increments a private variable x_1 (resp. x_2), where τ_1 and τ_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 τ_1 can change x_2 in a computation of M_1. This could not be the case if τ_1 were hidden in M_1.

Finally, note that the composition of two modules may fail to be non-Zeno even if each module is non-Zeno. This is not a contradiction, since non-Zenoness is not a linear-time temporal property. Any realistic real-time system specification should be non-Zeno; furthermore, the possibility of non-Zeno systems compromises the completeness of our rules [KMP96]. We very briefly sketch how branching-time reasoning can be used to prove non-Zenoness in Section 5.5.

4 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.[2]

STeP implements *verification rules* and *verification diagrams*, including rules INV and WAIT of Section 3.3. 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. 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].

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.

4.1 Verification Diagrams

Proofs of temporal system specifications can often be conveniently presented using *verification diagrams* [MP94]. These are directed graphs 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. We write $m \in \tau(n)$ if transition τ connects node n to m. 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 φ-*verification conditions* with a diagram $\mathcal{D}$ and temporal formula φ. When these are valid, the $\mathcal{S}$-validity of the diagram implies the $\mathcal{S}$-validity of φ.

In this paper we will only use diagrams to prove wait-for properties, of the form $p \Rightarrow q\mathcal{W}r$. Thus, we only introduce a special class of diagrams for these properties:

[2] To obtain a copy of STeP, send e-mail to `step-request@cs.stanford.edu`.

Definition 2. A *wait-for* diagram is a verification diagram $\mathcal{D}$, where each node n is labeled by the formula φ_n. For each non-terminal node $n \in Nodes(\mathcal{D})$ and every transition τ in $\mathcal{S}$, we associate the $\mathcal{S}$-verification condition:

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

The φ-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

Like the verification rules of Section 3.3, 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 5.4. STeP also supports diagrams for response and invariance properties. Furthermore, arbitrary temporal formulas can be verified in STeP using the *generalized verification diagrams* introduced in [BMS95].

4.2 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 5.

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

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

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 $\bigvee_{i=0}^{\infty} \mathcal{F}^i(false)$, 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

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

$\mathcal{F}(true)$ 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 $\mathcal{F}(true)$ 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 $post(tick, true)$ to $\Pi(D, C)$, justified as follows:

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

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

4.3 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 φ, tagged by `enable`, and a set of variable assignments of the form `x := e`, tagged by `assign`. This adds the conjuncts φ 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`.

5 Example: Generalized Railroad Crossing

We illustrate our methodology by verifying the generalized railroad crossing (GRC), 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.

5.1 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 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 and Gate: The `Trains` module, shown in Figure 2, has three transitions parameterized by `i:[1..N]`, illustrated in Figure 3. 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.

The `Gate` module, shown in Figure 4, has four transitions, illustrated in Figure 5. 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 6. 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.

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

```
Clocked Transition Module Trains

in minTimeToI : rat where minTimeToI > 0
in maxTimeToI : rat 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 rat
local lastEnter : array[1..N] of rat

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. 2. The `Trains` clocked transition module

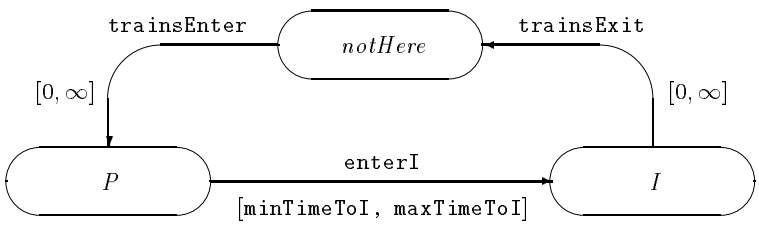

Fig. 3. Transitions for the `Trains` module

```
Clocked Transition Module Gate

in gateRiseTime : rat where gateRiseTime > 0
in gateDownTime : rat where gateDownTime > 0

type gateStatus = {up, down, goingUp, goingDown}

local gate: gateStatus where gate = up
local lastDown, lastUp: rat

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

Transition lower :
  enable gate = up \/ gate = goingUp
  assign gate := goingDown,
         lastDown := T + gateDownTime

Transition raise :
  enable gate = down \/ gate = goingDown
  assign gate := goingUp,
         lastUp := T + gateRiseTime

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

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

Fig. 4. The `Gate` clocked transition module

`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 (partially) determined by the controller.[3] 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, in `schedTime`, when a train will be in the intersection. For the system to be safe, we must assume that this estimate is conservative, as stated by axiom `ACT1` in Figure 7.

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

[3] Note that the corresponding gate transitions have their own enabling conditions, so the gate may possibly block these transitions.

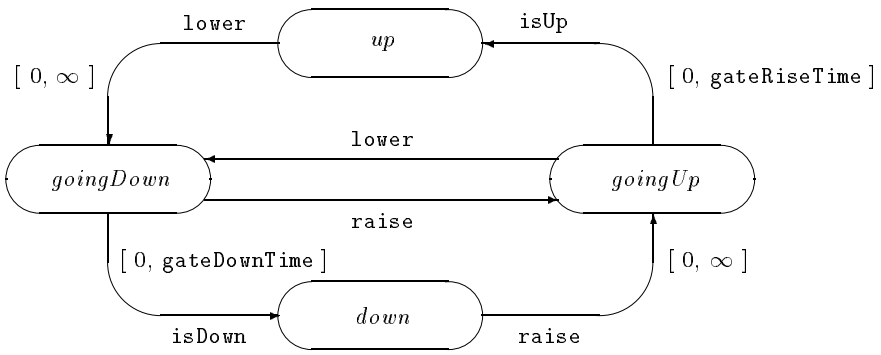

Fig. 5. Transitions for the `Gate` module

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. `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.

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, 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, which is enforced by the second conjunct of the controller's time-progress condition: it states that the controller can keep the gate down only if there is a train that is sufficiently near.

5.2 Modular Analysis

Before proving properties over the entire system, we analyze the modules individually and in various combinations. By Theorem 1 of Section 3.3, 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 8 lists some properties proved valid over single or pairwise combinations of modules. `GC1` states to what extent the controller knows the status of the gate (it cannot distinguish between `goingUp` and `up`). `GC2` states that whenever the controller has given the command *lower*, the gate is guaranteed to be down within `gateDownTime`. `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

```
Clocked Transition Module Controller

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

type gstatus = {gUp,gDown}

local gs: gstatus where gs = gUp
local schedTime: array[1..N] of rat
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. 6. The `Controller` clocked transition module

```
AXIOM AC1 (feasibility): gammaDown < conMinI

AXIOM ACT1 (conservative estimate): (conMinI < minTimeToI)

AXIOM AGC1 (conservative estimate): (gateDownTime < gammaDown)
AXIOM AGC2 (conservative estimate): (gateRiseTime < gammaUp)
```

Fig. 7. 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. 8. Modularly valid auxiliary properties

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. Properties `G1` and `G2` were generated by the automatic invariant generation method described in Section 4.2.

5.3 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:

1. 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.
2. 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.
3. 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 time required for one automobile to pass over the tracks. Figure 9 illustrates this case.

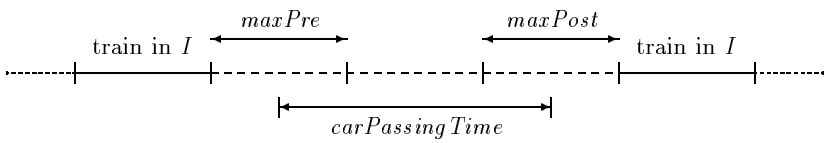

Fig. 9. An interval where the gate should stay down

We reformulate this property as the two *wait-for* properties of Figure 10, where `Awaits` stands for the $\mathcal{W}$ (wait-for) operator. Specification variables `maxPre` and `maxPost`, also defined in Figure 10, 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 11. `maxPre` includes the term `maxTimeToI - conMinI` because the controller must ensure that the gate is down before a train's earliest estimated arrival time, but the train may not actually arrive until the latest possible time. The positive term `margin` gives the controller some freedom in making conservative estimates. Property `utility1` states that if the gate is going down then a train will enter the intersection within time `maxPre`. This forbids spurious or premature lowerings of the gate. Property `utility2` states that if there are no trains in the intersection, 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.

```
value MaxPre, maxPost: rat

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. 10. Specification of the utility property

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

Fig. 11. Axioms relating specification variables to system constants

5.4 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 8; therefore, no auxiliary assertion φ had to be constructed. Of the 10 verification conditions, only two required non-trivial user guidance to be proved.

Property `utility1`: The *wait-for* verification diagram of Figure 12 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 or with trivial user interaction; one of the three φ-verification conditions required one additional invariant, `gateInv`, proved using rule INV. The other two φ-verification conditions were proved with trivial user interaction.

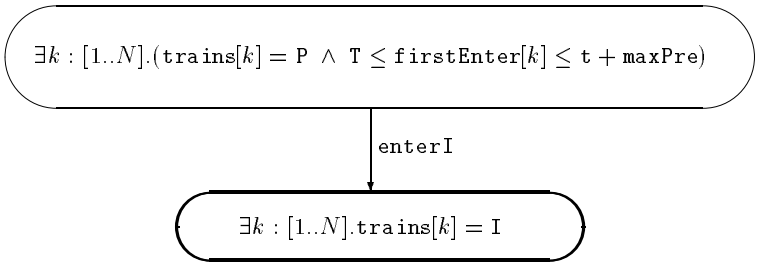

Fig. 12. Wait-for verification diagram for `utility1`.

```
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 13 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.

5.5 Non-Zenoness

The above verification would be questionable if the system were Zeno. A complete verification of the system should therefore address non-Zenoness. Non-Zenoness can be established with the following CTL [Eme90] property[4]

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

STeP was used to generate the verification conditions required to establish this CTL property for the railroad crossing. It suffices to take ϵ to be the smallest positive constant used in the system specification. The proof then conducts a comprehensive case analysis over the combined system, including well-founded induction on the number of trains that appear in the P interval and may prevent the clock from progressing beyond $t+\epsilon$. The full details are beyond the page-limit of this paper and will be reported elsewhere.

[4] A different approach to proving non-Zenoness is reported in [Lam95].

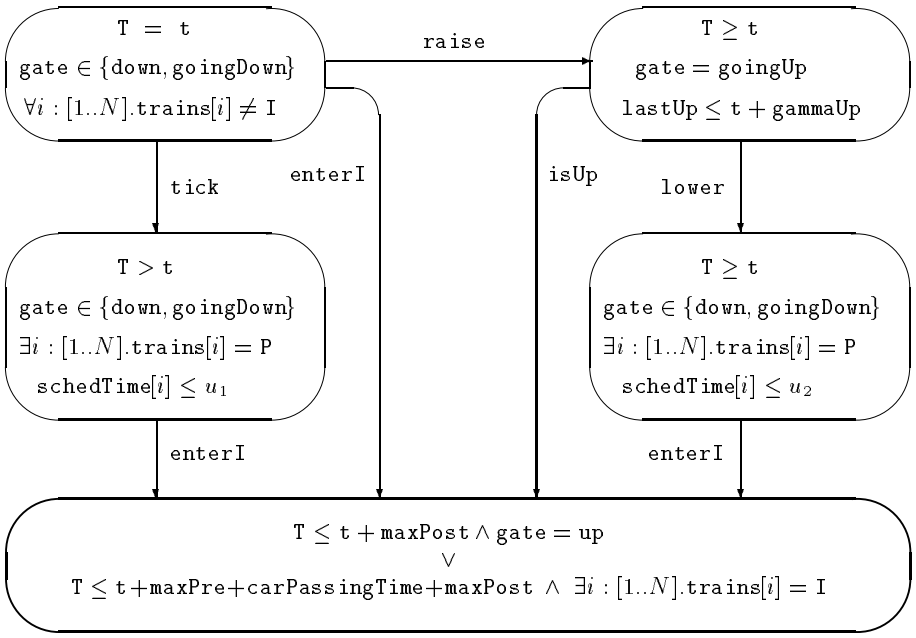

Fig. 13. Wait-for verification diagram for `utility2`

6 Related Work

6.1 The Railroad Crossing Benchmark

The GRC and similar examples have been previously analyzed using 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, and formal proofs were conducted using the PVS proof-assistant. PVS was also used in [Ska94] to verify a railroad crossing system formulated in the Duration Calculus [CHR92]. [AB96] use Timed Statecharts [MMP92] to encode real-time systems, and the HOL prover 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 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+95]. 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.

6.2 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 CTS 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 tools for symbolic and explicit-state model checking. 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.

Acknowledgements: We thank Uri Lerner for his 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. *Theor. Comp. Sci.*, 126:183–235, 1994.

[AL92] M. Abadi and L. Lamport. An old-fashioned recipe for real time. In *Proceedings of the REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *LNCS*, pages 1–27. Springer-Verlag, 1992.

[BBC+95] N.S. Bjørner, A. Browne, E. 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. 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 *Proc. 8^{th} Intl. Conference on Computer Aided Verification*, volume 1102 of *LNCS*, pages 415–418. Springer-Verlag, July 1996.

[BBM97] N.S. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theor. Comp. Sci.*, 173(1):49–87, February 1997.

[BLL+95] J. Bengtsson, K.G. Larson, F. Larsson, P. Pettersson, and W. Yi. UPPAAL – a Tool Suite for Automatic Verification of Real-Time Systems. In *Proc. of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems*, volume 1066 of *LNCS*, pages 232–243, 1995.

[BLS96] S. Bensalem, Y. Lakhnech, and H. Saidi. Powerful Techniques for the Automatic Generation of Invariants. In R. Alur and T. Henzinger, editors, *Proc. 8^{th} Intl. Conference on Computer Aided Verification*, volume 1102 of *LNCS*, pages 323–335. Springer-Verlag, 1996.

[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*, volume 1026 of *LNCS*, pages 484–498, 1995.

[BSU97] N.S. Bjørner, M.E. Stickel, and T.E. Uribe. A practical combination of first-order reasoning and decision procedures. In *14th Intl. Conference on Automated Deduction*, LNCS. Springer-Verlag, July 1997. To appear.

[CC95] S. Campos and E. Clarke. Real-time symbolic model checking for discrete time models. In *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.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III*, volume 1066 of *LNCS*, pages 208–219. Springer-Verlag, 1996.

[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*, volume B, pages 995–1072. Elsevier Science Publishers (North-Holland), 1990.

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

[HJL93] C. Heitmeyer, R. 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. Kopke. Verification methods for the divergent runs of clock systems. In *FTRTFT 94: Formal Techniques in Real-time and Fault-tolerant Systems*, volume 863 of *LNCS*, pages 351–372. Springer-Verlag, 1994.

[HL94] C. Heitmeyer and N. Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *Proc. ICCC Real-Time Systems Symposium*, pages 120–131. IEEE Press, 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 *Hybrid Systems III*, volume 1066 of *LNCS*, pages 13–40. Springer-Verlag, 1996.

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

[MMP92] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In *Proceedings of the REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *LNCS*, pages 447–484. Springer-Verlag, 1992.

[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 *Proc. Int. Symp. on Theoretical Aspects of Computer Software*, volume 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.

[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. To appear.

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

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

[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*, volume 1118 of *LNCS*, pages 564–567, August 1996.