The Generalized Railroad Crossing: A Case Study in Formal Verification of Real-Time Systems

Constance Heitmeyer* Nancy Lynch†

Abstract

A new solution to the Generalized Railroad Crossing problem, based on timed automata, invariants and simulation mappings, is presented and evaluated. The solution shows formally the correspondence between four system descriptions: an axiomatic specification, an operational specification, a discrete system implementation, and a system implementation that works with a continuous gate model.

1 Introduction

Recently, one of us (Heitmeyer) defined a benchmark problem to compare the many formal methods that exist for specifying, designing, and analyzing real-time systems and to better understand the utility of the methods for developing practical systems. The problem, which is called the Generalized Railroad (GRC) Crossing [8], is as follows:

The system to be developed operates a gate at a railroad crossing. The railroad crossing I lies in a region of interest R, i.e., $I \subseteq R$. A set of trains travel through R on multiple tracks in both directions. A sensor system determines when each train enters and exits region R. To describe the system formally, we define a gate function $g(t) \in [0,90]$, where $g(t) = 0$ means the gate is down and $g(t) = 90$ means the gate is up. We define a set $\{\lambda_i\}$ of *occupancy intervals*, where each occupancy interval is a time interval during which one or more trains are in I. The ith occupancy interval is represented as $\lambda_i = [\tau_i, \nu_i]$, where τ_i is the time of the ith entry of a train into the crossing when no other train is in the crossing and ν_i is the first time since τ_i that no train is in the crossing (i.e., the train that entered at τ_i has exited as have any trains that entered the crossing after τ_i).

Given two constants ξ_1 and ξ_2, $\xi_1 > 0$, $\xi_2 > 0$, the problem is to develop a system to operate the crossing gate that satisfies the following two properties:

Safety Property: $t \in \cup_i \lambda_i \Rightarrow g(t) = 0$ (The gate is down during all occupancy intervals.)

Utility Property: $t \notin \cup_i [\tau_i - \xi_1, \nu_i + \xi_2] \Rightarrow g(t) = 90$ (The gate is up when no train is in the crossing.)

To solve the GRC problem, real-time researchers have applied a variety of formal methods, including process algebraic [9, 3, 1], event-based [10], and logic-based approaches [19, 11]. They have also used various mechanical proof systems, including PVS [18], EVES [11], and FDR [2], to formally analyze and verify their solutions. Reference [5] describes three early efforts to solve the GRC problem.

This paper describes a new solution of the GRC based on the Lynch-Vaandrager timed automaton model [16, 15], using invariant and simulation mapping techniques [12, 15, 14]. To develop the solution, a "formal methods expert" (Lynch) and an "applications expert" (Heitmeyer) worked closely together to refine the GRC problem statement and to design and verify an implementation.

Our close collaboration was in sharp contrast to the limited interaction between the Naval Research Laboratory (NRL) group that distributed the GRC problem and the formal methods groups that developed earlier solutions. In the earlier work, the NRL group limited interaction both to encourage original solutions and to prevent some groups from having more information and thus unfair advantage over other groups. While these early efforts produced a variety of solutions and many insights into the relative strengths and weaknesses of the different formalisms, they suffered from two limitations. First, because the original problem statement was somewhat ambiguous, each group solved a slightly different problem, which caused difficulties in comparing the solutions. Second, the limited interaction meant that deficiencies in the GRC problem statement went uncorrected. Our collaboration allowed us to quickly identify and correct these deficiencies. It also led us to represent the problem and its solution in a form that is both understandable to applications experts and usable by formal methods experts for verification.

Section 2 describes our approach. Sections 3 and 4 present our highest-level problem specification, intended to be understood by applications experts, and a second operational specification, intended to be useful in formal verification. Sections 5, 6 and 7 contain our system implementation, the main correctness proof, and an extension of our solution to more realistic, continuous models. Section 8 evaluates our solution and method. An appendix provides background on our formal methods. The details of the proofs are available in the full version of the paper [6].

2 Our Approach

Formal Methods for Real-Time Systems. Applying formal methods to real-time systems involves three steps: system requirements specification, design of an implementation, and verification that the implementation satisfies the specification. This process

*Code 5546, Naval Research Laboratory, Washington, DC 20375.

†Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139. Supported by NSF grant 9225124-CCR, ONR contract N00014-91-J-1046, AFOSR contract F49620-94-1-0199, and ARPA contract N00014-92-J-4033.

has feedback loops. Once specified, the requirements must be revised when later steps expose omissions and errors. The same is true of the designed implementation.

All three steps require close collaboration between a formal methods expert and an applications expert. The role of the formal methods expert is to produce formal descriptions of both the system requirements and the selected implementation and to prove formally that the latter satisfies the former. The role of the applications expert is to work closely with the formal methods expert to identify the "real" requirements and to ensure that the specified implementation is acceptable. In our collaboration, much of the dialogue focused on the system requirements. Once the requirements specification was acceptable, defining and verifying an implementation, while labor-intensive and time-consuming, was relatively straightforward.

A system requirements specification describes all acceptable system implementations [7]. It has two parts: (1) A set of formal models describing the computer system at an abstract level, the environment (here, the trains and the gate), and the interface between them. (2) Formal statements of the properties that the system must satisfy.

In developing the GRC solution, we applied the following seven software engineering principles. The first five concern the requirements specification, the sixth concerns the implementation and its verification, and the seventh is applicable to all three steps.

1. *Avoid underspecifying system requirements.* The original problem statement lacked necessary information about the various constants. For example, the statement did not constrain the constant ξ_1. A simple analysis shows that we should assume that $\xi_1 > \gamma_{down} + \epsilon_2 - \epsilon_1$, where ϵ_2 is the maximum time and ϵ_1 the minimum time that a train requires to travel from entry into R to the crossing and γ_{down} is the maximum time needed to lower the gate.

2. *Avoid overspecifying system requirements.* For example, while the function g is an acceptable gate model, the GRC problem can be solved using a simpler, discrete model—one that represents the gate in one of four states—*up*, *going-down*, *down*, and *going-up*. Our solution uses the simpler model, but we show in Section 7 how to extend our results to the original gate model.

   For another example, the Utility Property stated above does not rule out solutions in which the last train leaves the crossing at time t but within the interval $[t, t + \xi_2]$ the gate goes first up and then down rapidly before the gate is raised for the second (and final) time. Such solutions, though not to be encouraged, should not be excluded. The essential system properties are that the gate must be down when a train is in the crossing, and that the gate must be up during the specified intervals when no train is in the vicinity. During other times, we do not care what the gate does.

3. *Make sure the specified system behavior is reasonable.* For example, suppose a train exits the crossing at time t and another train is scheduled to enter the crossing by time $t + \gamma_{up} + \gamma_{down}$. Then there is insufficient time for even one car to travel through the crossing, and thus the Utility Property fails to achieve its practical purpose. To rule out such useless activity, we modify the original problem statement to only require the gate to be raised if sufficient time, δ, exists for at least one car to travel through the crossing.

4. *Specify the system requirements as axioms rather than operationally.* In the original problem statement, both the Safety Property and the Utility Property are expressed as axioms. Each axiom describes a relationship that must hold between the two components of the system environment, namely, the trains and the crossing gate. Thus the required system properties are properties of the environment. Neither axiom mentions the computer system. Also, the two axioms are stated independently, making it easy to modify the individual properties.

   In the present study, we initially described the requirements operationally. This operational specification incorporated both the Safety and Utility Properties into a single automaton, thus losing the advantage of independence. Also, the specification was stronger than the original formulation, describing some aspects of what the computer system should do rather than just describing properties that the system needed to guarantee in the environment. Finally, the operational style of the specification was harder for applications experts to understand. Our final version of the specification, which appears in Section 3, is axiomatic. Like the original formulation, it describes the two properties as independent axioms about the environment.

5. *Provide a second, operational specification plus a formal proof that the operational specification implements the axiomatic specification.* Although it is desirable to start with an axiomatic specification, the types of proofs we do rest on operational, automaton versions of the specification and implementation. Therefore, we present a second requirements specification in terms of timed automata and prove that the operational requirements specification implements the original axiomatic specification.

   As in many applications of formal methods, we initially neglected to provide a formal proof of the correspondence between the original problem statement and the reformulation within our framework. Without such a proof, there is no assurance that the properties satisfied by the system implementation are the ones that are really required. In our case, while it was immediately obvious that the statement of the Safety Property in our operational specification was equivalent to the original statement of the Safety Property, the correspondence between the two versions of the Utility Property was not so clear.

6. *Provide a formal model for the implementation and a proof that it implements the operational specification.* The implementation should be described using the same model that is used for the operational specification, or at least one that is compatible. The proof that the implementation meets the specification can be done using a variety of methods, either by hand, as in this paper, or with computer assistance.

7. *Express the system requirements specification, the implementation, and the formal proofs so that they are understandable to applications experts.* If the requirements specification and the description of the implementation are difficult to understand, the applications expert cannot be confident that the right requirements have been specified and that the implementation is acceptable. The same holds for the formal proofs: the applications expert must be able to understand the proofs. This gives him/her a deep understanding of how and why the system works and how future changes are likely to affect system behavior. To increase their understandability, both the formal specifications and the proofs should be based on standard models such as automaton models, standard notations, and standard proof techniques such as invariants and simulation mappings. To the extent feasible, applications experts should not be required to learn new notations or proof techniques.

The Formal Framework. The formal method we used to specify the GRC problem and to develop and verify a solution represents both the computer system and the system environment as *timed automata*, according to the definitions of Lynch and Vaandrager [16, 15]. A timed automaton is a very general automaton, i.e., a labeled transition system. It is not finite-state: for example, the state can contain real-valued information, such as the current time or the position of a train or crossing gate. This makes timed automata suitable for modeling not only computer systems but also real-world entities such as trains and gates. We base our work directly on an automaton model rather than on any particular specification language, programming language, or proof system, in order to obtain the greatest flexibility in selecting specification and proof methods. The formal definition of a timed automaton appears in the Appendix.

The timed automaton model supports description of systems as collections of timed automata, interacting by means of common actions. In our example, we define separate timed automata for the trains, the gate, and the computer system; the common actions are sensors reporting the arrival of trains and actuators controlling the raising and lowering of the gate.

An important special case of the model, describable in a particularly simple way, is the MMT automaton model [17], developed by Merritt, Modugno and Tuttle. An MMT automaton consists of a collection of "tasks" (i.e., "processes") sharing common data, where each task has an upper bound and a lower bound on the time between its events. This special case is sufficient for describing several of our components, in particular, the trains and the discrete version of the gate. Our other components, e.g., the computer system, cannot be expressed in the MMT style, so we describe them directly in terms of the general model.

Applying Formal Methods to GRC. Our solution contains four system descriptions: *AxSpec*, the axiomatic requirements specification; *OpSpec*, the operational requirements specification; *SystImpl*, the discrete system implementation; and *SystImpl'*, a system implementation with a continuous gate model. Figure 1 illustrates the four descriptions and how they are related.

The top-level requirements specification *AxSpec* contains timed automata describing the computer system and its environment (the trains and gate) and axioms expressing the Safety and Utility Properties. The Safety Property states that if a train is in the crossing, the gate must be down. The Utility Property states that the gate is up unless a train is in the vicinity. Formally, these axioms are properties added to the composition of three timed automata: *Trains*, *Gate*, and *CompSpec*, a trivial specification of the computer system. Figure 2 illustrates *AxSpec*.

Next, because it is easier to use in proving correctness, we produce a second, more operational requirements specification in the form of a timed automaton *OpSpec*. We show that *OpSpec* implements *AxSpec*.

Next, we describe our computer system implementation as a timed automaton *CompImpl*. Correctness means that *CompImpl*, when it interacts with *Trains* and *Gate*, guarantees the Safety and Utility Properties. To show this, we prove that *SysImpl*, the composition of *CompImpl*, *Trains* and *Gate*, provides the same view to the environment components, *Trains* and *Gate*, as the operational specification *OpSpec*. This part of the proof follows well-established, stylized invariant and simulation mapping methods, which is why we moved from the axiomatic style of specification to the operational style. All these proofs can be verified using current mechanical proof technology.

In both specification automata, *AxSpec* and *OpSpec*, and in the implementation automaton *SysImpl*, time is built into the state. Time information consists of the current time plus some deadline infor-

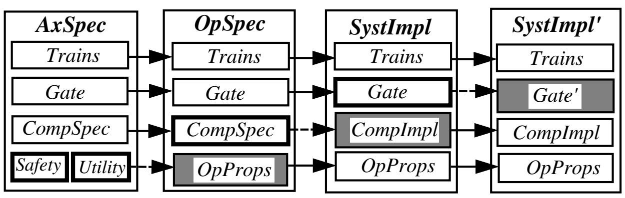

Figure 1: The four system descriptions and how they are related. In *OpSpec*, *OpProps* incorporates the Safety and Utility properties into the automaton that results from composing *Trains*, *Gate*, and *CompSpec*.

mation, such as the earliest and latest times that a train that has entered R will actually enter the crossing. The correctness proof proceeds by first proving by induction some invariants about the reachable states of *SysImpl*. The main work in the proof of the Safety Property is done by means of these invariants. An interesting feature of the proofs is that the invariants involve time deadline information.

Next, we show a "simulation mapping" between the states of *SysImpl* and *OpSpec*, again by induction; this is enough to prove the Utility Property. Like the invariants, the simulations also involve time deadline information, in particular, they include inequalities between time deadlines.

Finally, we observe that our main proofs yield a weaker result that what we really want. Namely, we have worked with abstract, discrete models of the trains and gate rather than with realistic models that allow continuous behavior. And we have only shown that the "admissible timed traces", i.e., the sequences of externally visible actions, together with their times of occurrence, are preserved, rather than all aspects of the environment's behavior. We conclude by showing that we have not lost any generality by proving the weaker results. In particular, preservation of admissible timed traces actually implies preservation of all aspects of the environment's behavior. Further, the results extend to *SystImpl′*, a system implementation with a more realistic environment model. Both extensions are obtained as corollaries of the results for admissible timed traces of the discrete model, using general results about composition of timed automata.

3 Axiomatic Specification

We first define two timed automata, *Trains* and *Gate*, which are abstract representations of the trains and the gate. These two components do not interact directly. We then define a trivial automaton *CompSpec*, which interacts with both *Trains* and *Gate* via actions representing sensors and actuators. *CompSpec* describes nothing more than the computer system's interface with the environment. *AxSpec* is obtained by composing these three automata and then imposing the Safety and Utility Properties on the composition; see Figure 2. Formally, the two properties are restrictions on the executions of the composition. The Safety Property is just a restriction on the states that occur in the execution, while the Utility Property is a more complex temporal condition.

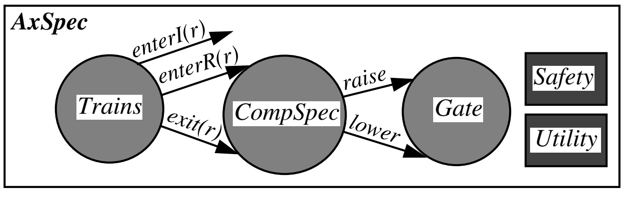

Figure 2: *AxSpec* is the composition of *Trains*, *Gate*, and *CompSpec*, constrained by the Safety and Utility properties.

Parameters and Other Notation. We use the notation r, r', etc. to denote trains, I to denote the railroad crossing, R to denote the region from where a train passes a sensor until it exits the crossing, and P to denote the portion of R prior to the crossing. We define some positive real-valued constants:

- ϵ_1, a lower bound on the time from when a train enters R until it reaches I.
- ϵ_2, an upper bound on the time from when a train enters R until it reaches I.
- δ, the minimum useful time for the gate to be up. (For example, this might represent the minimum time for a car to pass through the crossing safely.)
- γ_{down}, an upper bound on the time to lower the gate completely.
- γ_{up}, an upper bound on the time to raise the gate completely.
- ξ_1, an upper bound on the time from the start of lowering the gate until some train is in I.
- ξ_2, an upper bound on the time from when the last train leaves I until the gate is up (unless the raising is interrupted by another train getting "close" to I).
- β, an arbitrarily small constant used to take care of some technical race conditions.[1]

We need some restrictions on the values of the various constants:

1. $\epsilon_1 \leq \epsilon_2$.
2. $\epsilon_1 > \gamma_{down}$. (The time from when a train arrives until it reaches the crossing is sufficiently large to allow the gate to be lowered.)
3. $\xi_1 \geq \gamma_{down} + \beta + \epsilon_2 - \epsilon_1$. (The time allowed between the start of lowering the gate and some train reaching I is sufficient to allow the gate to be lowered in time for the fastest train, and then to accommodate the slowest train. The time γ_{down} is needed to lower the gate in time for the fastest train, but the slowest train could take an additional time $\epsilon_2 - \epsilon_1$. The β is a technicality.)
4. $\xi_2 \geq \gamma_{up}$. (The time allowed to raise the gate is sufficient.)

Trains. We model the *Trains* component as an MMT automaton with no input or internal actions, and three types of outputs, $enterR(r)$, $enterI(r)$, and $exit(r)$, for each train r. The state consists of a *status* component for each train, just saying where it is.

State:
for each train r:
$r.status \in \{not\text{-}here, P, I\}$, initially $not\text{-}here$

The state transitions are described by specifying the "preconditions" under which each action can occur and the "effect" of each action. s denotes the state before the event occurs and s' the state afterwards.

Transitions:

$enterR(r)$
Precondition:
$s.r.status = not\text{-}here$
Effect:
$s'.r.status = P$

[1] These arise because the model allows more than one event to happen at the same real time.

$enterI(r)$
Precondition:
$s.r.status = P$
Effect:
$s'.r.status = I$

$exit(r)$
Precondition:
$s.r.status = I$
Effect:
$s'.r.status = not\text{-}here$

In this automaton (and for all other MMT automata in this paper), we make each non-input action a task by itself. We only specify trivial bounds (that is, $[0, \infty]$) for the $enterR(r)$ and $exit(r)$ actions. For each $enterI(r)$ action, we use bounds $[\epsilon_1, \epsilon_2]$. This means that from the time when any train r has reached R, it is at least time ϵ_1 and at most time ϵ_2 until the train reaches I.

We use the general construction described in the Appendix to convert this automaton to a timed automaton. This construction involves adding some components to the state – a current time component *now*, and *first* and *last* components for each task, giving the earliest and latest times at which an action of that task can occur once the task is enabled. The transition relation is augmented with conditions to enforce the bound assumptions; that is, an event cannot happen before its *first* time, and time cannot pass beyond any *last* time. In this case, only the state components *now* and $first(enterI(r))$ and $last(enterI(r))$ for each r contain nontrivial information, so we ignore the other cases. Applying this construction yields the timed automaton with the same actions and the following states and transitions.

State:
now, a nonnegative real, initially 0
for each train r:
$r.status \in \{not\text{-}here, P, I\}$, initially $not\text{-}here$
$first(enterI(r))$, a nonnegative real, initially 0
$last(enterI(r))$, a nonnegative real or ∞, initially ∞

Transitions:

$enterR(r)$
Precondition:
$s.r.status = not\text{-}here$
Effect:
$s'.r.status = P$
$s'.first(enterI(r)) = now + \epsilon_1$
$s'.last(enterI(r)) = now + \epsilon_2$

$enterI(r)$
Precondition:
$s.r.status = P$
$now \geq s.first(enterI(r))$
Effect:
$s'.r.status = I$
$s'.first(enterI(r)) = 0$
$s'.last(enterI(r)) = \infty$

$exit(r)$
Precondition:
$s.r.status = I$
Effect:
$s'.r.status = not\text{-}here$

$\nu(\Delta t)$
Precondition:
for all r, $s.now + \Delta t \leq s.last(enterI(r))$
Effect:
$s'.now = s.now + \Delta t$

Gate. We model the gate as another MMT automaton, this one with inputs *lower* and *raise* and outputs *down* and *up*. The time bounds are *down*: $[0, \gamma_{down}]$, and *up*: $[0, \gamma_{up}]$, where γ_{up} and γ_{down} are upper bounds on the time required for the gate to be raised and lowered. To build time into the state, the state components *now*, $last(up)$, and $last(down)$ are added to produce the following states and transitions.

State:
$status \in \{up, down, going\text{-}up, going\text{-}down\}$, initially up
now, a nonnegative real, initially 0
$last(down)$, a nonnegative real or ∞, initially ∞
$last(up)$, a nonnegative real or ∞, initially ∞

Transitions:

$lower$
Effect:
if $s.status \in \{up, going\text{-}up\}$ then
$s'.status = going\text{-}down$
$s'.last(down) = now + \gamma_{down}$
$s'.last(up) = \infty$

$raise$
Effect:
if $s.status \in \{down, going\text{-}down\}$ then
$s'.status = going\text{-}up$
$s'.last(up) = now + \gamma_{up}$
$s'.last(down) = \infty$

$down$
Precondition:
$s.status = going\text{-}down$
Effect:
$s'.status = down$
$s'.last(down) = \infty$

up
Precondition:
$s.status = going\text{-}up$
Effect:
$s'.status = up$
$s'.last(up) = \infty$

$\nu(\Delta t)$
Precondition:
$s.now + \Delta t \leq s.last(up)$
$s.now + \Delta t \leq s.last(down)$
Effect:
$s'.now = s.now + \Delta t$

CompSpec. We model the computer system interface as a trivial MMT automaton *CompSpec* with inputs $enterR(r)$ and $exit(r)$ for each train r and outputs *lower* and *raise*. *CompSpec* receives sensor information when a train arrives in the region R and when it leaves the crossing I. It does *not* have an input action $enterI(r)$; this expresses the assumption that no sensor informs the system when a train actually enters the crossing. *CompSpec* has just a single state. Inputs and outputs are always enabled and cause no state change. There are no timing requirements.

Transitions:

$enterR(r)$ Effect: none	$exit(r)$ Effect: none
$lower$ Precondition: $true$ Effect: none	$raise$ Precondition: $true$ Effect: none

AxSpec. To get the full specification, the three MMT automata given above, *Trains*, *Gate* and *CompSpec*, are composed yielding a new MMT automaton. We then add constraints to express the correctness properties in which we are interested. Formally, these constraints are axioms about an *admissible timed execution* α of the composition automaton:

1. **Safety Property**

   All the states in α satisfy the following condition:
   If $Trains.r.status = I$ for any r, then $Gate.status = down$.

2. **Utility Property**

   If s is a state in α with $s.Gate.status \neq up$, then at least one of the following conditions holds.

   (a) There exists s' preceding (or equal to) s in α with $s'.Trains.r.status = I$ for some r and $s'.now \geq s.now - \xi_2$.

   (b) There exists s' following (or equal to) s in α with $s'.Trains.r.status = I$ for some r and $s'.now \leq s.now + \xi_1$.

   (c) There exist two states s' and s'' in α, with s' preceding or equal to s, s'' following or equal to s, $s'.Trains.r.status = I$ for some r, $s''.Trains.r.status = I$ for some r, and $s''.now - s'.now \leq \xi_1 + \xi_2 + \delta$.

The Safety and Utility properties are stated independently. The Safety Property is an assertion about all states reached in α, saying that each satisfies the critical safety property. In contrast, the Utility Property is a temporal property with a somewhat more complicated structure, which says that if the gate is not up, then either there is a recent preceding state or an imminent following state in which a train is in I. The third condition takes care of the special case where there is both a recent state and an imminent state in which some train is in I; although these states are not quite as recent or imminent as required by the first two cases, there is insufficient time for a car to pass through the crossing.

4 Operational Spec

Unlike *AxSpec*, which consists of a timed automaton together with some axioms that restrict the automaton's executions, the operational specification *OpSpec* is simply a timed automaton – all required properties are built into the automaton itself as restrictions on the state set and on the actions that are permitted to occur. As a result, *OpSpec* is probably harder for an application expert to understand than *AxSpec*. But it is easier to use in proofs (at least for the style of verification we are using). Thus we regard *OpSpec* as an intermediate specification rather than a true problem specification; we only require that *OpSpec* implement *AxSpec*, not necessarily vice versa, and that all implementations satisfy *OpSpec*.

The two specifications are also different in another respect: while *AxSpec* preserves the independence of the Safety and Utility Properties, *OpSpec* does not. When a collection of separate properties are specified by an automaton, the properties usually become intertwined.

To obtain *OpSpec*, we first compose *Trains*, *Gate*, and *CompSpec*, and then incorporate the Safety and Utility Properties into the automaton itself. Formally, the modified automaton is obtained from the composition by restricting it to a subset of the state set, then adding some additional state components, and finally modifying the definitions of the steps to describe their dependence on and their effects on the new state components. Although the composition of the three component automata is an MMT automaton, the modified version is not – it is a timed automaton.

First, to express the Safety Property, we restrict the states to be those states of the composition that satisfy the following invariant: "If $Trains.r.status = I$ for any r, then $Gate.status = down$."

Second, the time-bound restrictions expressed by the Utility Property are encoded as restrictions on the steps. The strategy is similar to that used to encode MMT time bound restrictions into the steps of a timed automaton – it involves adding explicit deadline components. We describe the modifications in two pieces:

1. *The time from when the gate starts going down until some train enters I is bounded by ξ_1.* To express this restriction formally, we add to the state of the composed system a new deadline $last_1$, representing the latest time in the future that a train is guaranteed to enter I. Initially, this is set to ∞, meaning that there is no such scheduled requirement. To add this new component to *OpSpec*, we include the following new effects in two of the actions:

Transitions:

$lower$
Effect:
if $s.Gate.status \in \{up, going\text{-}up\}$
and $s.last_1 = \infty$ then
$s'.last_1 = now + \xi_1$

$enterI(r)$
Effect:
$s'.last_1 = \infty$

Also added is a new precondition: the time-passage action cannot cause time to pass beyond $last_1$. This means that whenever the gate starts moving down, some train must enter I within time ξ_1. The new effect being added to the $lower$ action just "schedules" the arrival of a train in I.

2. *From when the crossing becomes empty, either the time until the gate is up is bounded by ξ_2 or else the time until a train is in I is bounded by $\xi_2 + \delta + \xi_1$.* Again, we express the condition by adding deadlines, only this time the situation is trickier since two alternative bounds exist rather than just one. We add two new components, $last_2(up)$ and $last_2(I)$, both initially ∞. The first represents a milestone to be noted – whether the gate reaches the *up* position by the designated time – rather than an actual deadline. In contrast, the second represents a real deadline – a time by which a new train must enter I *unless* the gate reached the *up* position by the milestone time $last_2(up)$. To include these new components in *OpSpec*, we add the following effects to three of the actions:

Transitions:

$exit(r)$
  Effect:
    if $s.Trains.r'.status \neq I$ for all $r' \neq r$ then
      $s'.last_2(up) = now + \xi_2$
      $s'.last_2(I) = now + \xi_2 + \delta + \xi_1$

up
  Effect:
    if $now \leq s.last_2(up)$ then
      $s'.last_2(up) = \infty$
      $s'.last_2(I) = \infty$

$enterI(r)$
  Effect:
      $s'.last_2(up) = \infty$
      $s'.last_2(I) = \infty$

Also, as with $last_1$, an implicit precondition is placed on the time-passage action, saying that time cannot pass beyond $last_2(I)$. But because $last_2(up)$ is just a milestone to be recorded, no such limitation is imposed for time passing beyond $last_2(up)$.

We show that *OpSpec* implements *AxSpec* in the following sense:

Lemma 4.1 *For any admissible timed execution α of OpSpec, there is an admissible timed execution α' of AxSpec such that $\alpha' | Trains \times Gate = \alpha | Trains \times Gate$. (This is the same as saying that α satisfies the two properties given explicitly for AxSpec.)*

Note that the relationship between *OpSpec* and *AxSpec* is only one-way: there are admissible timed executions of *AxSpec* that have no executions of *OpSpec* yielding the same projection. Consider, for example, the following example. Suppose that after I becomes empty, the system does a very rapid *raise*, *lower*, *raise*. These could conceivably all happen within time ξ_2 after the previous time there was a train in I, which would make this "waffling" behavior legal according to *AxSpec*. However, when this *lower* occurs, there is no following entry of a train into I, which means that this does not satisfy *OpSpec*.

5 Implementation

To describe our implementation *SysImpl*, we use the same *Trains* and *Gate* automata but replace the *CompSpec* component in *OpSpec* and *AxSpec* with a new component *CompImpl*, a computer system implementation. *CompImpl* is a timed automaton with the same interface as *CompSpec*. It keeps track of the trains in R together with the earliest possible time that each might enter I. (This time could be in the past.) It also keeps track of the latest operation that it has performed on the gate and the current time.

State:
  for each train r:
    $r.status \in \{not\text{-}here, R\}$, initially $not\text{-}here$
    $r.sched\text{-}time$, a nonneg. real number or ∞, initially ∞
  $gate\text{-}status \in \{up, down\}$, initially up
  now, initially 0

Transitions:

$enterR(r)$
  Effect:
    $s'.r.status = R$
    $s'.r.sched\text{-}time = now + \epsilon_1$

$exit(r)$
  Effect:
    $s'.r.status = not\text{-}here$
    $s'.r.sched\text{-}time = \infty$

$lower$
  Precondition:
    $s.gate\text{-}status = up$
    $\exists r : s.r.sched\text{-}time \leq now + \gamma_{down} + \beta$
  Effect:
    $s'.gate\text{-}status = down$

$raise$
  Precondition:
    $s.gate\text{-}status = down$
    $\nexists r : s.r.sched\text{-}time \leq now + \gamma_{up} + \delta + \gamma_{down}$
  Effect:
    $s'.gate\text{-}status = up$

$\nu(\Delta t)$
  Precondition:
    $t = s.now + \Delta t$
    if $s.gate\text{-}status = up$ then
      $t < s.r.sched\text{-}time - \gamma_{down}$ for all r
    if $s.gate\text{-}status = down$ then
      $\exists r : s.r.sched\text{-}time \leq s.now + \gamma_{up} + \delta + \gamma_{down}$
  Effect:
    $s'.now = t$

Observe that the fact that $CompImpl.gate\text{-}status = up$ does not mean that $Gate.status = up$ but just that $Gate.status \in \{up, going\text{-}up\}$. A similar remark holds for $CompImpl.gate\text{-}status = down$. Note that $r.sched\text{-}time$ keeps track of the earliest time that train r might enter I. The system lowers the gate if the gate is currently up (or going up) and some train might soon arrive in I. Here "soon" means by the time the computer system can lower the gate plus a little bit more – this is where we consider the technical race condition mentioned earlier. The system raises the gate if the gate is currently down (or going down) and no train can soon arrive in I. This time, "soon" means by the time the gate can be raised plus the time for a car to pass through the crossing plus the time for the system to lower the gate. The system allows time to pass subject to two conditions. First, if *gate-status* = *up*, then real time is not allowed to reach a time at which it is necessary to lower the gate. Second, if *gate-status* = *down* and the gate should be raised, then time cannot increase at all (until the gate is raised).

The full system implementation, *SysImpl*, is just the composition of the *Trains*, *Gate* and *CompImpl* components.

6 Correctness Proof

The main correctness proof shows that every admissible execution of *SysImpl* projects on the external world like some admissible execution of *OpSpec*.

We first state a collection of invariants, leading to a proof of the safety property. All are proved by induction on the length of an execution. The first invariant says that if a train is in the region and the gate is either up or going up, then the train must still be far from the crossing.

Lemma 6.1 *In all reachable states of SysImpl, if Trains.r.status = P and Gate.status* $\in \{up, going\text{-}up\}$*, then* $Trains.first(enterI(r)) > now + \gamma_{down}$.

The second invariant says that if a train is nearing I and the gate is going down, then the gate is nearing the *down* position. In particular, the earliest time at which the train might enter I is strictly after the latest time at which the gate will be down.

Lemma 6.2 *In all reachable states of SysImpl, if Trains.r.status = P and Gate.status = going-down, then* $Trains.first(enterI(r)) > Gate.last(down)$.

These invariants yield the main safety result:

Lemma 6.3 *In all reachable states of SysImpl, if Trains.r.status = I for any r, then Gate.status = down.*

To show the Utility Property, we present the simulation mapping from *SysImpl* to *OpSpec*. Specifically, if s and u are states of *SysImpl* and *OpSpec*, respectively, then we define s and u to be related by relation f provided that:

1. $u.now = s.now$.
2. $u.Trains = s.Trains$.[2]
3. $u.Gate = s.Gate$.
4. $u.last_1 \geq min\{s.Trains.last(enterI(r))\}$.
5. Either $u.last_2(I) \geq min\{s.Trains.last(enterI(r))\}$, or $u.last_2(up) \geq now + \gamma_{up}$ and the *raise* precondition holds in s, or $u.last_2(up) \geq s.Gate.last(up)$ and $s.Gate.status =$ *going-up*.

[2]By this we mean that the entire state of the *Trains* automaton, including the time components, is preserved.

The first three parts of the definition are self-explanatory. The last two parts provide connections between the time deadlines in the specification and implementation. In the typical style for this approach, the connections are expressed as inequalities. The fourth condition bounds the latest time for some train to enter I, a bound mentioned in the specification, in terms of the actual time it could take in the implementation, namely, the minimum of the latest times for all the trains in P. The fifth condition is slightly more complicated – it bounds the time for *either* some train to enter I or the gate to reach the up position. There are two cases for the gate reaching the up position – one in which the gate has not yet begun to rise and the other in which it has.

Theorem 6.4 *f is a simulation mapping from SysImpl to OpSpec.*

Proof: We show the three conditions required for a simulation mapping, as defined in the Appendix. ■

Theorems 6.4 and A.1 together imply that all admissible timed traces of *SysImpl* are admissible timed traces of *OpSpec*. This is not quite what we need. However, we can obtain the needed correspondence between *SysImpl* and *OpSpec* as a corollary, using general results about composition of timed automata:

Corollary 6.5 *For any admissible timed execution* α *of SysImpl, there is an admissible timed execution* α' *of OpSpec such that* $\alpha'|Trains \times Gate = \alpha|Trains \times Gate$.

Putting this together with Lemma 4.1, we obtain the main theorem:

Theorem 6.6 *For any admissible timed execution* α *of SysImpl, there is an admissible timed execution* α' *of AxSpec such that* $\alpha'|Trains \times Gate = \alpha|Trains \times Gate$.

7 Realistic Models of the Real World

The models used above for the trains and gate are rather abstract. An applications expert might prefer more realistic models giving, for instance, exact or approximate positions for the trains and gate. However, a formal methods expert would probably not want to include such details, because they would complicate the proofs. Fortunately, we can satisfy everyone.

For any real world component, it is possible to define a *pair of models*, one abstract and one more realistic. The only constraint is that the realistic model should be an "implementation" of the abstract model, i.e., its set of admissible timed traces should be included in that of the abstract model. All the difficult proofs are carried out using the abstract models, as above. Then corollaries are given to extend the results to the realistic models. This extension is based on general results about composition of timed automata.

For example, we can define a new type of gate component, *Gate'*, similar to the *Gate* defined above, but having a more detailed model of gate position. *Gate'* is also a timed automaton. Fix any constant γ'_{down}, $0 \leq \gamma'_{down} \leq \gamma_{down}$. Define g_d to be a function mapping $[0, \gamma'_{down}]$ to $[0, 90]$. Function g_d is defined so that $g_d(0) = 90$, $g_d(\gamma'_{down}) = 0$, and g_d is monotone nonincreasing and continuous. $g_d(t)$ gives the position of the gate after it has been going down for time t. Similarly, fix a constant γ'_{up}, $0 \leq \gamma'_{up} \leq \gamma_{up}$, and define g_u to be a function mapping $[0, \gamma'_{up}]$ to $[0, 90]$. Function g_u is defined so that $g_u(0) = 0$, $g_u(\gamma'_{up}) = 90$, and g_u is monotone nondecreasing and continuous.

The actions of *Gate'* are the same as for *Gate*. The state is also the same, with the addition of one new component $pos \in [0, 90]$ to represent the gate position, initially 90. The *lower* and *raise* transitions are the same as for *Gate*, except that γ'_{down} and γ'_{up} are used in place of γ_{down} and γ_{up}; they are omitted below. The *up* and *down* transitions contains new preconditions stating that the correct position has been reached. The time-passage transitions adjust *pos*.

Transitions:

down
  Precondition:
    $s.status = going\text{-}down$
    $s.pos = 0$
  Effect:
    $s'.status = down$
    $s'.last(down) = \infty$

up
  Precondition:
    $s.status = going\text{-}up$
    $s.pos = 90$
  Effect:
    $s'.status = up$
    $s'.last(up) = \infty$

$\nu(\Delta t)$
  Precondition:
    $t = now + \Delta t$
    $t \leq s.last(down)$
    $t \leq s.last(up)$
  Effect:
    $s'.now = t$
    if $s.status = going\text{-}up$ then
      $s'.pos = \max\{s.pos, g_u(t - (s.last(up) - \gamma'_{up}))\}$
    elseif $s.status = going\text{-}down$ then
      $s'.pos = \min\{s.pos, g_d(t - (s.last(down) - \gamma'_{down}))\}$

Thus, unlike the more abstract automata considered so far, *Gate'* allows interesting state changes to occur in conjunction with time-passage actions. Note that *Gate'* contains a rather arbitrary decision about what happens if a *lower* event occurs when the gate is in an intermediate position. It says that the gate stays still for the initial time that it would take for the gate to move down to its current position if it had started from position 0. Alternative modeling choices would also be possible. A similar remark holds for *raise*.

We relate the new gate model to the old one. See the Appendix for the notation.

Lemma 7.1 *attraces*(*Gate'*) $\subseteq$ *attraces*(*Gate*).

Now, let *SysImpl'* be the composition of *Trains*, *Gate'*, and *CompImpl*, and let *AxSpec'* be the composition of *Trains*, *Gate'*, and *CompSpec*, with Safety and Utility Properties added as in *AxSpec*. Using Theorem 6.6 and general results about composition of timed automata, we obtain:

Theorem 7.2 *For any admissible timed execution α of SysImpl', there is an admissible timed execution α' of AxSpec' such that $\alpha'|Trains \times Gate' = \alpha|Trains \times Gate'$.*

8 Concluding Remarks

We have applied a formal method based on timed automata, invariants, and simulation mappings to model and verify the Generalized Railroad Crossing. Here, we extrapolate from this experience and attempt to evaluate the method for modeling and verifying other real-time systems. We also describe future work.

Generality. *Can the method be used to describe all acceptable implementations?* It seems so. Timed automata can have an infinite number of states and both discrete and continuous variables. Further, they can express the maximum allowable nondeterminism, use symbolic parameters to represent system constants, and represent asynchronous communication. Thus the method is significantly more general than model checking approaches, which typically require a finite number of states and constant timing parameters.

Readability. *Are the formal descriptions easy to understand?* The environment model and the system implementation model are easy to understand, since these are naturally modeled as automata. The requirements specifications do not look so natural when expressed as automata; an axiomatic form seems easier to understand. However, if one starts with an axiomatic specification, then one has to rewrite the specification as an automaton. It may be difficult to determine that the automaton specification is equivalent to (or implements) the axiomatic specification.

Information. *Does the proof yield information other than just the fact that the implementation is correct? Does it provide insight into the reasons that the implementation works?* Yes. The invariants and simulations that require considerable effort to produce yield payoffs by providing very useful documentation. They express key insights about the behavior of the implementation. In contrast, model checkers yield no such byproducts, only an assertion that the implementation satisfies the desired properties.

Power. *Can the method be used to verify all implementations?* Simulation methods (extended beyond what is described in this paper, to include "backward" as well as "forward" simulations) are theoretically complete for showing admissible timed trace inclusion. They also seem to be powerful in practice, although they might sometimes benefit from combination with other verification methods, such as model checking, process algebra, temporal logic or partial order techniques. Model checking alone is less powerful in practice, since it only checks whether a subfamily of solutions satisfy some specific properties.

Ease of Carrying out the Proof. *How hard is it to construct a proof using this method? Can typical engineers learn to do this?* Constructing these proofs, though not difficult, required significant work. The hardest parts were getting the details of the models right and finding the right invariants and simulation mapping. This is an art rather than an automatic procedure. The actual proofs of the invariants and the simulation were tedious but routine.

Carrying out such a modeling and verification effort requires the ability to do formal proofs, which most engineers are not trained to do. In contrast, using a model checker, an engineer can check automatically whether a given "model" satisfies the properties of interest. (Model checkers are already being used in practice by engineers to check the correctness of certain implementations, e.g., of circuits.) On the other hand, the proofs developed using the method of this paper are amenable to mechanical proof checking. So, automated support can be provided to engineers attempting to develop formal proofs.

Scalability. *Does the formalism scale up to handle larger problems?* We don't yet know. Just reasoning about this relatively simple problem was quite complex. A bigger system will mainly add complexity in the form of more system components and more actions, which leads in turn to more invariants, more components in the simulation mapping, and more cases in the proofs. But, in contrast to model checking, the blowup should not be exponential. Nonetheless, use of the method for larger problems should be coupled with various methods of decomposing a problem, so one need not reason about an entire complex system at once. Additional levels of abstraction and use of parallel composition should help.

Ease of Change. *How easy is it to modify the specifications and the proofs?* Separating the system model from the environment model and splitting the environment model into the individual gate model and train model makes it easy to change the descriptions. Should one want to use a more complex train model (for example, trains move backward as well as forward), one can easily substitute the revised model for the original. Expressing the required properties axiomatically and independently makes it easier to change the requirements.

Changes to the specifications and implementations require, of course, changes to the proofs. If the changes are fairly small, we expect most of the prior work to survive, and the stylized form of the proof provides useful structure for managing the modifications. Here is a place where mechanical aid would be most helpful – proofs could be rerun quickly to discover which parts need to be changed.

Future work. Our plans include:

1. Trying this method on larger examples from real-time process control and time-based communication. In real-time process control, transportation problems are especially interesting to us. New complications are expected to arise when the continuous quantities of interest include velocity and acceleration as well as time and position.

2. Developing computer assistance for carrying out and checking the proofs. We plan to use the proof systems PVS [18] and Larch [4] to check the proofs and to assess the utility of mechanical proof systems for such proofs.

3. Trying to systematize the reasoning about the correspondence between the axiomatic and operational specifications.

References

[1] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Trans. Prog. Lang. and Sys.*, 15(1):36–72, Jan. 1993.

[2] Oxford Formal Systems (Europe) Ltd. Failure Divergence Refinement, user manual and tutorial, 1992.

[3] R. Gerber and I. Lee. A proof system for communicating shared resources. In *Proc. 11th IEEE Real-Time Systems Symp.*, pages 288–299, 1990.

[4] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[5] C. Heitmeyer and R. Jeffords. Formal specification and verification of real-time systems: A comparison study. Technical report, NRL, Wash., DC, 1994. In preparation.

[6] C. Heitmeyer and N. Lynch. The Generalized Railroad Crossing: A case study in formal verification of real-time systems. Technical Report MIT/LCS/TM-51, Lab. for Comp. Sci., MIT, Cambridge, MA, 1994. Also Technical Report 7619, NRL, Wash., DC 1994.

[7] C. Heitmeyer and J. McLean. Abstract requirements specifications: A new approach and its application. *IEEE Trans. Softw. Eng.*, SE-9(5), September 1983.

[8] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. A benchmark for comparing different approaches for specifying and verifying real-time systems. In *Proc., 10th Intern. Workshop on Real-Time Operating Systems and Software*, May, 1993.

[9] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, NJ, 1985.

[10] F. Jahanian and A. K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Trans. Softw. Eng.*, SE-12(9), September 1986.

[11] S. Kromodimoeljo, W. Pase, M. Saaltink, D. Craigen, and I. Meisels. A tutorial on EVES. Technical report, Odyssey Research Associates, Ottawa, Canada, 1993.

[12] N. Lynch and H. Attiya. Using mappings to prove timing properties. *Distrib. Comput.*, 6:121–139, 1992.

[13] N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

[14] Nancy Lynch. Simulation techniques for proving properties of real-time systems. In *REX Workshop '93*, Lecture Notes in Computer Science, Mook, the Netherlands, 1994. Springer-Verlag. To appear.

[15] Nancy Lynch and Frits Vaandrager. Forward and backward simulations – Part II: Timing-based systems. Submitted for publication.

[16] Nancy Lynch and Frits Vaandrager. Forward and backward simulations for timing-based systems. In *Proceedings of REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *Lecture Notes in Computer Science*, pages 397–446, Mook, The Netherlands, June 1991. Springer-Verlag.

[17] Michael Merritt, Francesmary Modugno, and Mark R. Tuttle. Time constrained automata. In J. C. M. Baeten and J. F. Goote, editors, *CONCUR'91: 2nd International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*, pages 408–423, Amsterdam, The Netherlands, August 1991. Springer-Verlag.

[18] S. Owre, N. Shankar, and J. Rushby. User guide for the PVS specification and verification system (Draft). Technical report, Computer Science Lab, SRI Intl., Menlo Park, CA, 1993.

[19] N. Shankar. Verification of real-time systems using PVS. In *Proc. Computer Aided Verification (CAV '93)*, pages 280–291. Springer-Verlag, 1993.

A The Timed Automaton Model

This section contains the formal definitions for the timed automaton model, taken from [14].

Timed Automata. A *timed automaton* A consists of a set $states(A)$ of states, a nonempty set $start(A) \subseteq states(A)$ of start states, a set $acts(A)$ of actions, including a special *time-passage* action ν, a set $steps(A)$ of steps (transitions), and a mapping $now_A : states(A) \rightarrow \mathsf{R}^{\geq 0}$. ($\mathsf{R}^{\geq 0}$ denotes the nonnegative reals.) The actions are partitioned into *external* and *internal* actions, where ν is considered external; the *visible* actions are the non-ν external actions; the visible actions are partitioned into *input* and *output* actions. The set $steps(A)$ is a subset of $states(A) \times acts(A) \times states(A)$. We write $s \xrightarrow{\pi}_A s'$ as shorthand for $(s, \pi, s') \in steps(A)$ and usually write $s.now_A$ in place of $now_A(s)$. We sometimes suppress the subscript or argument A.

A timed automaton satisfies five axioms: [A1] If $s \in start$ then $s.now = 0$. [A2] If $s \xrightarrow{\pi} s'$ and $\pi \neq \nu$ then $s.now = s'.now$. [A3] If $s \xrightarrow{\nu} s'$ then $s.now < s'.now$. [A4] If $s \xrightarrow{\nu} s''$ and $s'' \xrightarrow{\nu} s'$, then $s \xrightarrow{\nu} s'$.

The statement of [A5] requires the preliminary definition of a *trajectory*, which describes restrictions on the state changes that can occur during time-passage. Namely, if I is any interval of $\mathsf{R}^{\geq 0}$, then an *I-trajectory* is a function $w : I \rightarrow states$, such that $w(t).now = t$ for all $t \in I$, and $w(t_1) \xrightarrow{\nu} w(t_2)$ for all $t_1, t_2 \in I$ with $t_1 < t_2$. That is, w assigns, to each time t in interval I, a state having the given time t as its *now* component. This assignment is done in such a way that time-passage steps can span between any pair of states in the range of w. If w is an I-trajectory and I is left-closed, then let $w.fstate$ be the first state of w, while if I is right-closed, then let $w.lstate$ denote the last state of w. If I is a closed interval, then an I-trajectory w is said to *span* from state s to state s' if $w.fstate = s$ and $w.lstate = s'$. The final axiom is: [A5] If $s \xrightarrow{\nu} s'$ then there exists a trajectory that spans from s to s'.

Timed Executions and Timed Traces A *timed execution fragment* is a finite or infinite alternating sequence $\alpha = w_0 \pi_1 w_1 \pi_2 w_2 \cdots$, where

1. Each w_j is a trajectory and each π_j is a non-time-passage action.
2. If α is a finite sequence, then it ends with a trajectory.
3. If w_j is not the last trajectory in α, then its domain is a closed interval. If w_j is the last trajectory, then its domain is left-closed (and either right-open or right-closed).
4. If w_j is not the last trajectory in α, then $w_j.lstate \xrightarrow{\pi_{j+1}} w_{j+1}.fstate$.

A *timed execution* is a timed execution fragment for which the first state of the first trajectory, w_0, is a start state. In this paper, we restrict attention to the *admissible* timed executions, i.e., those in which the *now* values occurring in the states approach ∞. We use the notation $atexecs(A)$ for the set of admissible timed executions of timed automaton A. A state of a timed automaton is defined to be *reachable* if it is the final state of the final trajectory in some finite timed execution of the automaton.

To describe the problems to be solved by timed automata, we require a definition for their visible behavior. We use the notion of *timed traces*, where the *timed trace* of any timed execution is just the sequence of visible events that occur in the timed execution, paired with their times of occurrence. The *admissible timed traces* of the timed automaton are just the timed traces that arise from all the admissible timed executions. We use the notation $attraces(A)$ for the set of admissible timed traces of timed automaton A. If α is any timed execution, we use the notation $ttrace(\alpha)$ to denote the timed trace of α.

We define a function *time* that maps any non-time-passage event in an execution to the real time at which it occurs. Namely, let π be any non-time-passage event. If π occurs in state s, then define $time(\pi) = s.now$.

Composition. Let A and B be timed automata satisfying the following *compatibility* conditions: A and B have no output actions in common, and no internal action of A is an action of B, and vice versa. Then the *composition* of A and B, written as $A \times B$, is the timed automaton defined as follows:

- $states(A \times B) = \{(s_A, s_B) \in states(A) \times states(B) : s_A.now_A = s_B.now_B\}$;
- $start(A \times B) = start(A) \times start(B)$;
- $acts(A \times B) = acts(A) \cup acts(B)$; an action is *external* in $A \times B$ exactly if it is external in either A or B; a visible action of $A \times B$ is an *output* in $A \times B$ exactly if it is an output in either A or B and is an *input* otherwise;
- $(s_A, s_B) \xrightarrow{\pi}_{A \times B} (s'_A, s'_B)$ exactly if
  1. $s_A \xrightarrow{\pi}_A s'_A$ if $\pi \in acts(A)$, else $s_A = s'_A$, and
  2. $s_B \xrightarrow{\pi}_B s'_B$ if $\pi \in acts(B)$, else $s_B = s'_B$;
- $(s_A, s_B).now_{A \times B} = s_A.now_A$.

Then $A \times B$ is a timed automaton. If α is a timed execution of $A \times B$, $\alpha|A$ and $\alpha|B$ denote the projections of α on A and B. For instance, $\alpha|A$ is defined by projecting all states in α on the state of A, removing actions that do not belong to A and collapsing consecutive trajectories. We also use the projection notation for sequences of actions; e.g., $\beta|A$ denotes the subsequence of β consisting of actions of A.

MMT Automata. We use the special case of MMT automata defined in [12, 14]. An MMT automaton is an I/O automaton [13] together with upper and lower bounds on time. An I/O automaton A consists of a set $states(A)$ of states, a nonempty set $start(A) \subseteq states(A)$ of start states, a set $acts(A)$ of actions (partitioned into *external* and *internal* actions; the external actions are further partitioned into *input* and *output* actions), a set $steps(A)$ of steps, and a partition $part(A)$ of the locally controlled (i.e., output and internal) actions into at most countably many equivalence classes. The set $steps(A)$ is a subset of $states(A) \times acts(A) \times states(A)$; an action π is said to be *enabled* in a state s provided that there exists a state s' such that $(s, \pi, s') \in steps(A)$, i.e., such that $s \xrightarrow{\pi}_A s'$. A set of actions is said to be *enabled* in s provided that at least one action in that set is enabled in s. The automaton must be *input-enabled*, which means that π is enabled in s for every state s and input action π. The final component, *part*, is sometimes called the *task partition*. Each class in this partition groups together actions that are supposed to be part of the same "task".

An MMT automaton is obtained by augmenting an I/O automaton with certain upper and lower time bound information. Let A be an I/O automaton with only finitely many partition classes. For each class C, define lower and upper time bounds, $lower(C)$ and $upper(C)$, where $0 \leq lower(C) < \infty$ and $0 < upper(C) \leq \infty$; that is, the lower bounds cannot be infinite and the upper bounds cannot be 0.

A timed execution of an MMT automaton A is defined to be an alternating sequence of the form $s_0, (\pi_1, t_1), s_1, \cdots$ where the π's are input, output or internal actions (but not time-passage actions). For each j, it must be that $s_j \xrightarrow{\pi_{j+1}} s_{j+1}$. The successive times are nondecreasing, and are required to satisfy the given *lower* and *upper* bound requirements. Finally, *admissibility* is required: if the sequence is infinite, then the times of actions approach ∞.

Each timed execution of an MMT automaton A gives rise to a *timed trace*, which is just the subsequence of external actions and their associated times. The *admissible timed traces* of the MMT automaton A are just the timed traces that arise from all the timed executions of A.

It is not hard to transform any MMT automaton A into a naturally-corresponding timed automaton A'. First, the state of the MMT automaton A is augmented with a *now* component, plus $first(C)$ and $last(C)$ components for each class of the task partition. The $first(C)$ and $last(C)$ components represent the earliest and latest time in the future that an action in class C is allowed to occur. The time-passage action ν is also added. The *first* and *last* components get updated in the natural way by the various steps, according to the *lower* and *upper* bounds specified in the MMT automaton A. The time-passage action has explicit preconditions saying that time cannot pass beyond any of the $last(C)$ values, since these represent deadlines for the various tasks. Restrictions are also added on actions in any class C, saying that the current time *now* must at least equal $first(C)$. The resulting timed automaton A' has exactly the same admissible timed traces as the MMT automaton A.

Invariants and Simulation Mappings. We define an *invariant* of a timed automaton to be any property that is true of all reachable states.

The definition of a simulation mapping is paraphrased from [16, 15, 14]. We use the notation $f[s]$, where f is a binary relation, to denote $\{u : (s, u) \in f\}$. Suppose A and B are timed automata and I_A and I_B are invariants of A and B. Then a *simulation mapping* from A to B with respect to I_A and I_B is a relation f over $states(A)$ and $states(B)$ that satisfies:

1. If $u \in f[s]$ then $u.now = s.now$.
2. If $s \in start(A)$ then $f[s] \cap start(B) \neq \emptyset$.
3. If $s \xrightarrow{\pi}_A s'$, $s, s' \in I_A$, and $u \in f[s] \cap I_B$, then there exists $u' \in f[s']$ such that there is a timed execution fragment from u to u' having the same timed visible actions as the given step.

Note that π is allowed to be the time-passage action in the third item of this definition. The most important fact about these simulations is that they imply admissible timed trace inclusion:

Theorem A.1 *If there is a simulation mapping from timed automaton A to timed automaton B, with respect to any invariants, then $attraces(A) \subseteq attraces(B)$.*