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

During the last decade, a large collection of formal methods have been invented for specifying, designing, and analyzing real-time systems. To compare these methods and to better understand their use in developing practical real-time systems, one of us (Heitmeyer) has defined a benchmark problem, called the Generalized Railroad (GRC) Crossing [7]. The problem 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 also 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

[*]Code 5546, Naval Research Laboratory, Washington, D.C. 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.

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 originated 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 quickly to 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.

The rest of the paper is organized as follows. Section 2 describes our approach: general principles for applying formal methods to specify and verify real-time systems, our formal model and proof techniques, and an outline of how we applied the formal methods to the GRC problem. Section 3 presents our highest-level problem specification, intended to be understood by applications experts; it improves over the original problem statement given above by resolving some ambiguities. Section 4 contains a secondary operational specification, intended to be useful in formal verification. Section 5 contains our system implementation. Section 6 contains the main correctness proof. Section 7 describes extensions to more realistic, continuous models of the real world components. Section 8 evaluates our solution and method. Several appendices provide background on the formal methods we use, plus two proofs about the high-level specification. A concise version of this report, which omits the details of the proofs, appears in [8].

2 Our Approach

In this section, we describe our approach to solving the GRC problem. Section 2.1 contains some general principles for applying formal methods to real-time systems. Section 2.2 contains a description of the timed automaton model and of invariant and simulation mapping proof methods. Section 2.3 contains an overview of how we apply these formal methods to the GRC problem.

2.1 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 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 the formal methods expert and the 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 [6]. 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 as being 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. A trivial modification of the original problem statement to include δ appears in Appendix D.

4. *Specify the system requirements axiomatically 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 is supposed to 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 reformulated the requirements specification operationally, as a timed automaton. This reformulation incorporated both the Safety and Utility Properties into a single automaton description, thus losing the advantage of independence. Also, our reformulation was stronger than the original, specifying 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 reformulation 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 an operational secondary 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 secondary 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 specification 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. It might be done 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 specification 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.

2.2 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, so that we may obtain the greatest flexibility in selecting specification and proof methods. The formal definition of a timed automaton appears in Appendix A.

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. Formal definitions of the MMT model are given in Appendix B. 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. Instead of thinking of the MMT model as a different model, we often find it useful to regard it as simply a way of describing a large subclass of timed automata.

2.3 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 specifications 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

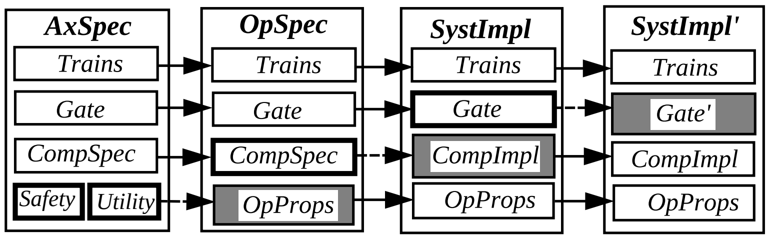

Figure 1: The four system descriptions and how they are related. In *OpSpec*, *OpProps* incorporates the Safety and Utility properties into the automaton that results from composing *Trains*, *Gate*, and *CompSpec*.

Properties. The Safety Property states that any time there is a train in the crossing, the gate must be down. The Utility Property states that the gate is up unless there is a train 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 interface. Figure 2 illustrates *AxSpec*.

Next, because it is easier to use in proving correctness, we produce a secondary, 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 *SystImpl*, the composition of *CompImpl*, *Trains* and *Gate*, provides the same view to the environment components, *Trains* and *Gates*, 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 of these proofs can be verified using current mechanical proof technology.

In both specification automata, *AxSpec* and *OpSpec*, and also in the implementation automaton *SystImpl*, time information is built into the state. Timing information consists of the current time plus some deadline information, 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 *SystImpl*. 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 *SystImpl* and *OpSpec*, again by induction; this is enough to prove the Utility Property. Appendix C contains formal definitions for simulation mappings and the correctness properties they guarantee. Like the invariants, the simulations 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 an abstract, discrete model of the trains and gate rather than with a realistic model that allows 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

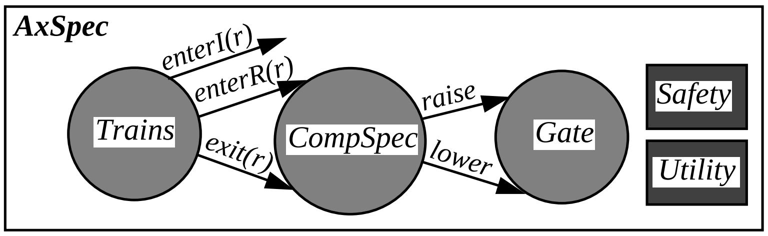

Figure 2: *AxSpec* is the composition of *Trains*, *Gate*, and *CompSpec*, constrained by the Safety and Utility properties.

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 begin with a high level *axiomatic specification*, *AxSpec*, describing the problem in terms most easily understood by application experts. We express the axioms solely in terms of the environment.

We first define two timed automata, *Trains* and *Gate*, which are abstract representations of the trains and gate, respectively. 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 interface that the computer system must have 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.

3.1 Parameters and Other Notation

We use the notation r, r', etc. to denote (railroad) trains. We use 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 for raising the gate is sufficient.)

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

3.2 *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.

Actions:

Input:
  none
Output:
  enterR(r), r a train
  enterI(r), r a train
  exit(r), r a train
Internal:

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-here*

The state transitions are described by specifying the "preconditions" under which each action can occur and the "effect" of each action. We use s to denote the state before the event occurs and s' the state afterwards. We use the convention that if a state component is not mentioned, it is unchanged (although sometimes, to resolve ambiguities or for emphasis, we say explicitly that a component is unchanged).

Transitions:

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

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 the 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 Appendix B 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 each task can occur, once the task is enabled. The transition relation is augmented with conditions to enforce the bound assumptions, that is, that an event cannot happen before its *first* time, and that 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. (In this automaton description, as well as elsewhere in the paper, we sometimes omit mention of the state where there is no ambiguity.)

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 $\infty$, initially $\infty$.
```

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:
      $s'.now = s.now + \Delta t$
```

Lemma 3.1 *In any reachable state of Trains:*
For any r such that $r.status = P$, $first(enterI(r)) + \epsilon_2 - \epsilon_1 = last(enterI(r))$.

Proof: By induction on the length, i.e., the total number of non-time-passage and time-passage steps, of an execution. Because $\epsilon_2 - \epsilon_1$ is a constant, we need only consider actions that change $first(enterI(r))$, $last(enterI(r))$ or make $r.status = P$, namely, $enterR(r)$ and $enterI(r)$. The actions $exit(r)$ and $\nu(\Delta t)$ do not affect the statement.

After 0 steps, the claim is vacuously satisfied. Assume the claim is true after m steps. We must prove it is true after $m+1$ steps. For $enterI(r)$, the claim is vacuously satisfied. For $enterR(r)$, the effect is $s'.r.status = P$. Then, $s'.first(enterI(r)) = now + \epsilon_1$ and $s'.last(enterI(r)) = now + \epsilon_2$, which implies that $s'.first(enterI(r)) + \epsilon_2 - \epsilon_1 = s'.last(enterI(r))$ as required. ■

3.3 *Gate*

We model the gate as another MMT automaton, this one with inputs *lower* and *raise* and outputs *down* and *up*.

Actions:

Input:
    lower
    raise
Output:

down
up
Internal:

The state consists of a single *status* component:

State:
$status \in \{up, down, going\text{-}up, going\text{-}down\}$, initially up

Transitions:

```
lower
  Effect:
    if $s.status \in \{up, going\text{-}up\}$ then
      $s'.status = going\text{-}down$
    else unchanged $status$

raise
  Effect:
    if $s.status \in \{down, going\text{-}down\}$ then
      $s'.status = going\text{-}up$
    else unchanged $status$

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

up
  Precondition:
    $s.status = going\text{-}up$
  Effect:
    $s'.status = up$
```

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
      $s'.status = going\text{-}down$
      $s'.last(down) = now + \gamma_{down}$
      $s'.last(up) = \infty$
    else unchanged $status$, $last(down)$, $last(up)$

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

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:
    $s'.now = s.now + \Delta t$
```

3.4 *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*.

Actions:

Input:
    enterR(r), r a train
    exit(r), r a train
Output:
    lower
    raise
Internal:

CompSpec receives sensor information when a train arrives in the region R and when it leaves the crossing I. Note that *CompSpec* does not have an input action *enterI*(r); this expresses the assumption that there is no sensor that 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
```

3.5 *AxSpec*

To get the full specification, we compose the three MMT automata given above, *Trains*, *Gate* and *CompSpec*, yielding a new MMT automaton. But this is not enough: 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 the states reached in α, saying that they all satisfy 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. In Appendix D, we show that the above statement of the Safety and Utility Properties is equivalent to a trivial modification of the original problem statement.

3.6 Implementation Requirements

An implementation of *AxSpec* uses a new timed automaton, called *CompImpl*, with the same interface as *CompSpec*. *CompImpl* will be composed with the same *Trains* and *Gate* automata given above, yielding a new system *SystImpl*. The system *SystImpl* should produce executions that, when projected on the environment (*Trains* composed with *Gate*), yields behavior that is also produced by the system specification *AxSpec*. More precisely, for every admissible timed execution α of *SystImpl*, there should be a corresponding admissible timed execution α' of *AxSpec* such that $\alpha' | \mathit{Trains} \times \mathit{Gate} = \alpha | \mathit{Trains} \times \mathit{Gate}$. That is, the two executions project identically on the *Trains* and *Gate* automata.

4 Operational Spec

In contrast to *AxSpec*, which consists of a timed automaton together with some axioms that describe restrictions on the automaton's executions, the secondary 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 of interest satisfy *OpSpec*.

The two types of 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.

4.1 The Specification

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 $\mathit{Trains.r.status} = I$ for any r, then $\mathit{Gate.status} = \mathit{down}$."

Second, the time-bound restrictions expressed by the Utility Property are encoded as restrictions on the steps. The strategy is similar to that used in Appendix B 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
          $s'.last_1 = now + \xi_1$
        else unchanged $last_1$

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

   There is also a new precondition added: 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 there are two alternative bounds 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 or not 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 add these new components to *OpSpec*, we include the following additional effects in three of the actions:

   Transitions:

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

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

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 no such limitation is imposed for time passing beyond $last_2(up)$, because this is just a milestone to be recorded, not a time-blockage.

4.2 Properties

We make some simple claims about *OpSpec*:

Lemma 4.1 *In all reachable states of OpSpec:*

1. *If* $Trains.r.status = I$ *for any* r, *then* $Gate.status = down$.
2. $last_2(up) + \delta + \xi_1 = last_2(I)$.

Proof: The first property is by definition of *OpSpec*. The second property is proved by induction. We need only consider actions that affect $last_2(up)$ and $last_2(I)$, namely, *up*, *enterI*(r), and *exit*(r) for some r.

For the action *up*, the only case in which the $last_2$ components are affected is where $now \leq s.last_2(up)$. In this case, the effect of *up* is $s'.last_2(up) = s'.last_2(I) = \infty$, and the claim is satisfied. The effect of *enterI*(r) is $s'.last_2(up) = s'.last_2(I) = \infty$, and thus the claim is satisfied. For *exit*(r), the only case in which the $last_2$ components are affected is where r's exit leaves I empty. Then the effect is $s'.last_2(up) = now + \xi_2$ and $s'.last_2(I) = now + \xi_2 + \delta + \xi_1$, which implies that $s'.last_2(up) + \delta + \xi_1 = s'.last_2(I)$, as needed. ■

Lemma 4.2 *In all reachable states of OpSpec:*

1. $now \leq last_1$.
2. $now \leq last_2(I)$.
3. *If* $last_1 \neq \infty$ *then* $last_1 \leq now + \xi_1$.
4. *If* $last_2(I) \neq \infty$ *then* $last_2(I) \leq now + \xi_2 + \delta + \xi_1$.
5. *If* $last_2(up) \neq \infty$ *then* $last_2(up) \leq now + \xi_2$.

Proof: By induction. Note that the only actions that can affect the truth of 1 or 3 are time passage, *lower* and *enterI* actions, and only time passage and *lower* actions could falsify 1 or 3. Also, the only actions that can affect the truth of 2, 4, or 5 are time passage, *exit*, *up* and *enterI* actions, and only time passage actions and *exit* actions that leave I empty could falsify 2, 4 or 5.

1. The precondition for time passage prevents $s'.now$ from exceeding $s.last_1 = s'.last_1$. The effect of *lower* is $s'.last_1 = now + \xi_1$. By definition of the constants, $\xi_1 > 0$, which implies that $s'.last_1 \geq now$.

2. The precondition for time passage prevents $s'.now$ from exceeding $s.last_2(I) = s'.last_2(I)$. If r's exit leaves I empty, then an effect of *exit*(r) is $s'.last_2(I) = now + \xi_2 + \delta + \xi_1$. By definition of the constants, $\xi_2 + \delta + \xi_1 > 0$, which implies that $s'.last_2(I) \geq now$.

3. If *lower* causes a change, then its effect is $s'.last_1 = now + \xi_1$, which implies $s'.last_1 \leq now + \xi_1$ as needed. For the time passage action, suppose that $s'.last_1 \neq \infty$. Since $s.last_1 = s'.last_1$, we have $s.last_1 \neq \infty$. Then by inductive hypothesis, $s.last_1 \leq s.now + \xi_1$. But $s.now < s'.now$, so $s.now + \xi_1 < s'.now + \xi_1$. Therefore, $s'.last_1 \leq s'.now + \xi_1$ as needed.

4. Time passage causes time to increase, so it cannot cause the claim to be violated. For an *exit*(r) action that leaves I empty, an effect is $s'.last_2(I) = now + \xi_2 + \delta + \xi_1$, which suffices.

5. Similar to 4.

■

4.3 Relationship Between *OpSpec* and *AxSpec*

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

Lemma 4.3 *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.)*

We leave the proof of Lemma 4.3 to Appendix E.

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*.

4.4 Proof Strategy

The relationship between *OpSpec* and *AxSpec* immediately suggests a strategy for showing that an implementation *SystImpl*, based on a particular computer system implementation *CompImpl*, satisfies *AxSpec*. The strategy is to show that every admissible timed execution of *SystImpl* has a corresponding admissible timed execution of *OpSpec* that projects identically on the *Trains* and *Gate* automata. Then use Lemma 4.3.

5 Implementation

To describe our implementation *SystImpl*, 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.

5.1 *CompImpl*

CompImpl is not an MMT automaton but 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 nonnegative real number or $\infty$, initially $\infty$
$gate\text{-}status \in \{up, down\}$, 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:
        $s.gate\text{-}status = down$
        $\not\exists r : s.r.sched\text{-}time \leq now + \gamma_{up} + \delta + \gamma_{down}$
    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-status* = *up* does not mean that *Gate.status* = *up* but just that *Gate.status* $\in$ {*up*, *going-up*}. A similar remark holds for *CompImpl.gate-status* = *down*.

Note that *r.sched-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).

5.2 The Full System Implementation, *SystImpl*

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

Here, we give some useful basic invariants about *SystImpl*; the next two lemmas say that *CompImpl* has accurate information about the trains and gate, respectively.

Lemma 5.1 *The following are true in any reachable state of SystImpl:*

1. *CompImpl.r.status* $= R$ *iff Trains.r.status* $\in \{P, I\}$.

2. *If Trains.r.status* $= P$, *then CompImpl.r.sched-time* = *Trains.first*(*enterI*(r)).

3. *If CompImpl.r.status* $= R$ *and CompImpl.r.sched-time* $>$ *now, then Trains.r.status* $= P$.

4. *If Trains.r.status* $= I$, *then CompImpl.r.sched-time* $\leq$ *now.*

5. *If CompImpl.r.sched-time* $\neq \infty$, *then Trains.r.status* $\in \{P, I\}$.

Proof: By induction.

1. The only actions that can cause a violation are actions that change the truth values of either *CompImpl.r.status* $= R$ or *Trains.r.status* $\in \{P, I\}$, namely, *enterR*(r) and *exit*(r). But, *enterR*(r) makes both sides of the equivalence true, and *exit*(r) makes both sides false. Hence, the property is true.

2. The only actions that can cause a violation are actions that make *Trains.r.status* $= P$, or else change *CompImpl.r.sched-time* or *Trains.first*(*enterI*(r)), namely, *enterR*(r), *exit*(r), and *enterI*(r). But *exit*(r) and *enterI*(r) cause *Trains.r.status* $\neq P$, so the claim is vacuously satisfied. Further, the effect of *enterR*(r) ensures that s'.*Trains.first*(*enterI*(r)) $=$ *now* $+ \epsilon_1$, and s'.*CompImpl.r.sched-time* $=$ *now* $+ \epsilon_1$. Hence, s'.*Trains.first*(*enterI*(r)) $=$
s' *CompImpl.r.sched-time*, as required.

3. The only actions that can cause a violation are those that can make *CompImpl.r.status* $= R$, increase *CompImpl.r.sched-time*, or make *Trains.r.status* $\neq P$, namely, *enterR*(r), *exit*(r), and *enterI*(r). The effect of *exit*(r) is *CompImpl.r.status* $\neq R$, and thus the claim is vacuously satisfied. The effect of *enterR*(r) is *Trains.r.status* $= P$, and thus the claim is satisfied.

   Now consider *enterI*(r). By the precondition of *enterI*(r), s.*Trains.r.status* $= P$ and *now* $\geq$ s.*Trains.first*(*enterI*(r)). Part 2 of this lemma implies that *now* $\geq$ s.*CompImpl.r.sched-time*. Since s.*CompImpl.r.sched-time* $=$ s'.*CompImpl.r.sched-time*, we have
   now $\geq$ s'.*CompImpl.r.sched-time*, and the claim is vacuously satisfied.

4. Suppose s'.*Trains.r.status* $= I$. Then Part 1 implies that s'.*CompImpl.r.status* $= R$. If s'.*CompImpl.r.sched-time* $>$ s'.*now*, then Part 3 implies that s'.*Trains.r.status* $= P$, a contradiction. So, s'.*CompImpl.r.sched-time* $\leq$ s'.*now* as needed.

5. The only action that could cause a violation, *enterR*(r), sets s'.*CompImpl.r.sched-time* $\neq \infty$. In this case, we have s'.*CompImpl.r.status* $= R$. Then Part 1 implies s'.*Trains.r.status* $\in \{P, I\}$ as needed.

■

Lemma 5.2 *The following are true in any reachable state of SystImpl:*

1. *CompImpl.gate-status* $=$ *up if and only if Gate.status* $\in$ $\{$*up*, *going-up*$\}$.

2. *CompImpl.gate-status* $=$ *down if and only if Gate.status* $\in$ $\{$*down*, *going-down*$\}$.

Proof: By induction.

1. We need only consider actions that change the truth values of $\mathit{CompImpl.gate\text{-}status} = \mathit{up}$ and $\mathit{Gate.status} \in \{\mathit{up}, \mathit{going\text{-}up}\}$, namely, *lower* and *raise*. For a *lower* action, $s'.\mathit{CompImpl.gate\text{-}status} = \mathit{down}$ and $s'.\mathit{Gate.status} \in \{\mathit{down}, \mathit{going\text{-}down}\}$, which suffices. For a *raise* action, $s'.\mathit{CompImpl.gate\text{-}status} = \mathit{up}$ and $s'.\mathit{Gate.status} \in \{\mathit{up}, \mathit{going\text{-}up}\}$, which again suffices.

2. Follows from Part 1.

■

6 Correctness Proof

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

We first prove a collection of invariants, leading to a proof of the safety property. All of the invariants are proved by induction on the length of an execution. Then we give a simulation mapping to show the Utility Property. Technically speaking, the simulation mapping only preserves timed traces, not the complete view of the environment components. However, standard composition techniques for timed automata show that the view is also preserved.

6.1 Invariants

In this section, we prove the main safety invariant, namely: "If $\mathit{Trains.r.status} = I$ for any r, then $\mathit{Gate.status} = \mathit{down}$." We do this with the help of two preliminary invariants. 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 the reachable states of SystImpl, if* $\mathit{Trains.r.status} = P$ *and* $\mathit{Gate.status} \in \{\mathit{up}, \mathit{going\text{-}up}\}$*, then* $\mathit{Trains.first}(\mathit{enterI}(r)) > \mathit{now} + \gamma_{\mathit{down}}$.

Proof: By induction. Fix any particular train r. We need only consider actions that cause $\mathit{Trains.r.status}$ to become equal to P, cause $\mathit{Gate.status}$ to change to be in $\{\mathit{up}, \mathit{going\text{-}up}\}$, decrease $\mathit{Trains.first}(\mathit{enterI}(r))$, or increase *now*, namely $\mathit{enterR}(r)$, *raise*, and ν.

1. $\mathit{enterR}(r)$

   An effect is $s'.\mathit{Trains.first}(\mathit{enterI}(r)) = \mathit{now} + \epsilon_1$. Since $\epsilon_1 > \gamma_{\mathit{down}}$ by an assumption on the constants, we have $s'.\mathit{Trains.first}(\mathit{enterI}(r)) > \mathit{now} + \gamma_{\mathit{down}}$, as needed.

2. *raise*

   Assume that $s'.\mathit{Trains.r.status} = P$. The precondition implies that $s.\mathit{CompImpl.r.sched\text{-}time} > \mathit{now} + \gamma_{\mathit{up}} + \delta + \gamma_{\mathit{down}}$, so that $s.\mathit{CompImpl.r.sched\text{-}time} > \mathit{now} + \gamma_{\mathit{down}}$ and therefore $s'.\mathit{CompImpl.r.sched\text{-}time} > \mathit{now} + \gamma_{\mathit{down}}$. By Lemma 5.1, Part 2, $s'.\mathit{CompImpl.r.sched\text{-}time} = s'.\mathit{Trains.first}(\mathit{enterI}(r))$. So $s'.\mathit{Trains.first}(\mathit{enterI}(r)) > \mathit{now} + \gamma_{\mathit{down}}$, as needed.

3. $\nu(\Delta t)$

   Assume $s'.Trains.r.status = P$ and $s'.Gate.status \in \{up, going\text{-}up\}$. Then, Lemma 5.2 implies that $s'.CompImpl.gate\text{-}status = up$, and the precondition for time passage implies that $s'.now < s.CompImpl.r.sched\text{-}time - \gamma_{down}$. By Lemma 5.1, Part 2, $s.CompImpl.r.sched\text{-}time = s.Trains.first(enterI(r))$. So, $s.Trains.first(enterI(r)) > s'.now + \gamma_{down}$, which implies $s'.Trains.first(enterI(r)) > s'.now + \gamma_{down}$, as needed.

■

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 SystImpl, if $Trains.r.status = P$ and $Gate.status = going\text{-}down$, then $Trains.first(enterI(r)) > Gate.last(down)$.*

Proof: Another inductive argument. Fix any particular train r. We need only consider actions that make $Trains.r.status = P$ or $Gate.status = going\text{-}down$, decrease $Trains.first(enterI(r))$ or increase $Gate.last(down)$, namely, *enterR*(r), *lower*, *enterI*(r), *raise* and *down*.

1. *enterI*(r), *raise*, or *down*.

   In each case, the claim is vacuously satisfied.

2. *enterR*(r)

   Assume $s'.Trains.r.status = P$ and $s'.Gate.status = going\text{-}down$. Then, Part 2 of Lemma B.1 implies that $s'.Gate.last(down) \leq now + \gamma_{down}$. The effect of *enterR*$(r)$ is $s'.Trains.first(enterI(r)) = now + \epsilon_1$. By an assumption about the constants, $\epsilon_1 > \gamma_{down}$, and so $now + \epsilon_1 > now + \gamma_{down}$. So $s'.Trains.first(enterI(r)) > s'.Gate.last(down)$ as needed.

3. *lower*

   Suppose $s'.Trains.r.status = P$. We only need to consider the case where $s.Gate.status \in \{up, going\text{-}up\}$, since otherwise the *lower* doesn't change anything. By Lemma 6.1, this implies that $s.Trains.first(enterI(r)) > now + \gamma_{down}$, which implies that $s'.Trains.first(enterI(r)) > now + \gamma_{down}$. Since $now + \gamma_{down} = s'.Gate.last(down)$, the needed inequality follows.

■

These invariants yield the main safety result:

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

Proof: By induction again. This time, the interesting cases are *enterI* and *raise*. Fix r.

1. $enterI(r)$

   By the precondition, $s.Trains.r.status = P$.

   If $s.Gate.status \in \{up, going\text{-}up\}$, then Lemma 6.1 implies that $s.Trains.first(enterI(r)) > now + \gamma_{down}$, so $s.Trains.first(enterI(r)) > now$. But, the precondition for $enterI(r)$ is $s.Trains.first(enterI(r)) \leq now$. This means that it is impossible for this action to occur, a contradiction.

   If $s.Gate.status = going\text{-}down$, then Lemma 6.2 implies that $s.Trains.first(enterI(r)) > s.Gate.last(down)$. By Lemma B.1, $s.Gate.status = going\text{-}down$ implies $s.Gate.last(down) \geq now$. This implies that $s.Trains.first(enterI(r)) > now$, which again means that it is impossible for this action to occur.

   The only remaining case is $s.Gate.status = down$. This implies $s'.Gate.status = down$, which suffices.

2. *raise*

   We need to show that the gate doesn't get raised when a train is in I. So suppose that $s.Trains.r.status = I$. The precondition of *raise* states that $\not\exists r$: $s.CompImpl.r.sched\text{-}time \leq now + \gamma_{up} + \delta + \gamma_{down}$, which implies that, for all r, $s.CompImpl.r.sched\text{-}time > now$. But Parts 1 and 3 of Lemma 5.1 imply that in this case, $s.Trains.r.status = P$, a contradiction.

■

6.2 Simulation Mapping

Now, in order to show the Utility Property, we present the simulation mapping from *SystImpl* to *OpSpec*. Specifically, if s and u are states of *SystImpl* 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 $s.Gate.status = going\text{-}up$.

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

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

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 SystImpl to OpSpec.*

Proof: We must show the three conditions in the definition of a simulation mapping. The three conditions are defined in Appendix C. The first condition, preservation of the *now* value, is immediate from the definition of f. The second condition is also immediate, because the unique start states of the two automata satisfy all the relationships in the definition of f. The interesting condition is the step condition.

Suppose that $s \xrightarrow{\pi}_{SystImpl} s'$, s and s' satisfy the invariants of *SystImpl*, and $u \in f[s]$ satisfies the invariants of *OpSpec*. We must produce $u' \in f[s']$ such that there is a timed execution fragment from u to u' having the same timed visible actions as the given step. We do this using a case analysis on π.

For each non-time-passage action π, we first argue that π is enabled in u and then define u' to be the unique state that results from applying the indicated action from state u. For the time-passage action, we first argue that the same amount of time can pass from u, and then define u' to be the unique state that results from allowing that amount of time to pass.

Then in each case, we must check that $u' \in f[s']$; in each case, Conditions 1-3 are easy to check, so we need only consider Conditions 4 and 5. Condition 4 is also easy for all cases except the *lower* action, since that is the only action that can decrease $last_1$. (The only action that can raise the minimum on the right-hand side of the inequality is *enterI*, but that sets $last_1$ to ∞.) So we omit mention of Condition 4 in all other cases.

1. $\pi = enterR(r)$.

   Enabling: Since π is enabled in s, we have $s.Trains.r.status = not\text{-}here$. Since $u \in f[s]$, we have $u.Trains.r.status = not\text{-}here$. This implies that π is enabled in u.

   Condition 5: The only alternative that might be falsified by π is the second, and only if $enterR(r)$ falsifies the *raise* precondition. So suppose that $u.last_2(up) \geq now + \gamma_{up}$ and the *raise* precondition holds in s but gets falsified in s'. Then, there exists r such that $s'.CompImpl.r.sched\text{-}time \leq now + \gamma_{up} + \delta + \gamma_{down}$. Since an effect of the action is $s'.CompImpl.r.sched\text{-}time = now + \epsilon_1$, we have $\epsilon_1 \leq \gamma_{up} + \delta + \gamma_{down}$.

   It suffices to show that $u'.last_2(I) \geq s'.Trains.last(enterI(r))$, since that would show that the action makes the first alternative of Condition 5 true. We have that $u'.last_2(I) = u.last_2(I)$ and $s'.Trains.last(enterI(r)) = now + \epsilon_2$. So it suffices to show that $u.last_2(I) \geq now + \epsilon_2$.

   Since $u.last_2(up) \geq now + \gamma_{up}$, and $u.last_2(I) = u.last_2(up) + \delta + \xi_1$ (this by Part 2 of Lemma 4.1), it is enough to show that $now + \gamma_{up} + \delta + \xi_1 \geq now + \epsilon_2$, or, more simply, that $\gamma_{up} + \delta + \xi_1 \geq \epsilon_2$.

   But $\gamma_{up} + \delta \geq \epsilon_1 - \gamma_{down}$ as noted above. And we have that $\xi_1 \geq \gamma_{down} + \epsilon_2 - \epsilon_1$, by an assumption about the constants. So, $\gamma_{up} + \delta + \xi_1 \geq \epsilon_2$ as needed.

2. $\pi = enterI(r)$.

   Enabling: Similar to $enterR(r)$.

   Condition 5: π makes $last_2(I) = \infty$ and $last_2(up) = \infty$, which make the condition trivially true.

3. $\pi = exit(r)$.

   Enabling: Similar to $enterR(r)$.

   Condition 5:

   There are three cases, based on which of the three alternatives becomes falsified.

   (a) The first alternative is falsified.

   Then, it must be that π decreases $u.last_2(I)$, and that no train is in I after the step.

   We have that $u'.last_2(I) = now+\xi_2+\delta+\xi_1$ and $u'.last_2(up) = now+\xi_2$. If the precondition for *raise* holds after the step, then it suffices to show that $u'.last_2(up) \geq now + \gamma_{up}$. That is, it suffices to show that $\xi_2 \geq \gamma_{up}$. But this follows from an assumption about the constants.

   Suppose that the precondition for *raise* does not hold after the step. Then there is some r' such that $s'.CompImpl.r'.sched\text{-}time \leq now + \gamma_{up} + \delta + \gamma_{down}$. (The first precondition of *raise*, $gate\text{-}status = down$, cannot fail after the step because of Lemma 6.3 applied to s.) The fact that $s'.CompImpl.r'.sched\text{-}time \neq \infty$ and Part 5 of Lemma 7.1 together imply that $s'.Trains.r'.status \in \{P, I\}$. However, by assumption, no trains are in I after the step, so it must be that $s'.Trains.r'.status = P$. Then Lemma 5.1, Part 2, implies that $s'.Trains.first(enterI(r')) \leq now + \gamma_{up} + \delta + \gamma_{down}$, and then Lemma 3.1 implies that $s.Trains.last(enterI(r')) \leq now + \gamma_{up} + \delta + \gamma_{down} + \epsilon_2 - \epsilon_1$. It suffices to show that $u'.last_2(I) \geq s'.Trains.last(enterI(r'))$. To show this, it suffices to show that $now+\xi_2+\delta+\xi_1 \geq now+\gamma_{up}+\delta+\gamma_{down}+\epsilon_2-\epsilon_1$, or, more simply, that $\xi_2+\xi_1 \geq \gamma_{up}+\gamma_{down}+\epsilon_2-\epsilon_1$. But this follows from the inequalities $\xi_2 \geq \gamma_{up}$ and $\xi_1 \geq \gamma_{down} + \beta + \epsilon_2 - \epsilon_1 > \gamma_{down} + \epsilon_2 - \epsilon_1$.

   (b) The second alternative is falsified.

   Then the *raise* precondition must be true in s. By the precondition, $s.Trains.r.status = I$. Then Lemma 5.1, Part 4, implies that $s.CompImpl.r.sched\text{-}time \leq now$. But this violates the *raise* precondition in s, which is a contradiction.

   (c) The third alternative is falsified.

   Then $s.Gate.status = going\text{-}up$. By the precondition, we have $s.Trains.r.status = I$, so by Lemma 6.3, we have that $s.Gate.status = down$, a contradiction.

4. $\pi = raise$.

   Enabling: Clearly π is enabled in u, because *OpSpec* imposes no preconditions on its performance.

   Condition 5: The only alternative that can be falsified is the second. Suppose that $u.last_2(up) \geq now + \gamma_{up}$. We have $u'.last_2(up) = u.last_2(up)$, so $u'.last_2(up) \geq now + \gamma_{up}$. But, $s'.Gate.last(up) = now + \gamma_{up}$ and $s'.Gate.status = going\text{-}up$, which yields the third alternative.

5. $\pi = lower$.

   Enabling: As for *raise*.

   The precondition for *lower* in *SystImpl* implies that $s.CompImpl.gate\text{-}status = up$ and that there is a particular r such that $s.CompImpl.r.sched\text{-}time \leq now + \gamma_{down} + \beta$. Then Lemma 5.2 implies that $s.Gate.status \in \{up, going\text{-}up\}$. If $s.Trains.r.status = I$, then Lemma 6.3 is violated in s. Because $s.\mathit{CompImpl.r.sched\text{-}time} \neq \infty$, it cannot be that $s.\mathit{Trains.r.status} = \mathit{not\text{-}here}$. So it must be that $s.\mathit{Trains.r.status} = P$.

Then Lemma 5.1, Part 2, implies that $s.\mathit{CompImpl.r.sched\text{-}time} = s.\mathit{Trains.first}(\mathit{enterI}(r))$, and Lemma 3.1 implies that $s.\mathit{CompImpl.r.sched\text{-}time} + \epsilon_2 - \epsilon_1 = s.\mathit{Trains.last}(\mathit{enterI}(r))$. Thus, we have $s.\mathit{Trains.last}(\mathit{enterI}(r)) = s.\mathit{CompImpl.r.sched\text{-}time} + \epsilon_2 - \epsilon_1$, $\leq \mathit{now} + \gamma_{\mathit{down}} + \beta + \epsilon_2 - \epsilon_1$, $\leq \mathit{now} + \xi_1$ by an assumption about the constants. That is, $\mathit{now} + \xi_1 \geq s.\mathit{Trains.last}(\mathit{enterI}(r))$.

Condition 4: By definition of *lower* in *OpSpec*, $u'.\mathit{last}_1 = \mathit{now} + \xi_1$. But as we showed above, this is at least as great as $s.\mathit{Trains.last}(\mathit{enterI}(r)) = s'.\mathit{Trains.last}(\mathit{enterI}(r))$. That is, $u'.\mathit{last}_1 \geq s'.\mathit{Trains.last}(\mathit{enterI}(r))$, as needed.

Condition 5: The only alternative that π can falsify is the third. So suppose that $u.\mathit{last}_2(\mathit{up}) \geq s.\mathit{Gate.last}(\mathit{up})$ and $s.\mathit{Gate.status} = \mathit{going\text{-}up}$. Lemma B.1 implies that $s.\mathit{Gate.last}(\mathit{up}) \geq \mathit{now}$, so $u.\mathit{last}_2(\mathit{up}) \geq \mathit{now}$. Since $u.\mathit{last}_2(\mathit{up}) = u'.\mathit{last}_2(\mathit{up})$, we have $u'.\mathit{last}_2(\mathit{up}) \geq \mathit{now}$.

Then Lemma 4.1 implies that $u'.\mathit{last}_2(I) = u'.\mathit{last}_2(\mathit{up}) + \xi_1 + \delta$, which is in turn $\geq \mathit{now} + \xi_1$. Since $\mathit{now} + \xi_1 \geq s.\mathit{Trains.last}(\mathit{enterI}(r))$, we have $u'.\mathit{last}_2(I) \geq s.\mathit{Trains.last}(\mathit{enterI}(r))$, so $u'.\mathit{last}_2(I) \geq s'.\mathit{Trains.last}(\mathit{enterI}(r))$. This suffices for alternative 1.

6. $\pi = \mathit{up}$.

   Enabling: Similar to *enterR*.

   Condition 5: Because π doesn't decrease any of the left sides of the inequalities. it cannot falsify alternative 1. Alternative 2 can't hold before the step, because the *raise* precondition and the *up* precondition are exclusive. Suppose that π falsifies alternative 3. Then $u.\mathit{last}_2(\mathit{up}) \geq s.\mathit{Gate.last}(\mathit{up})$ and $s.\mathit{Gate.status} = \mathit{going\text{-}up}$. Lemma B.1 implies that $s.\mathit{Gate.last}(\mathit{up}) \geq \mathit{now}$, so $u.\mathit{last}_2(\mathit{up}) \geq \mathit{now}$. But then the effect of the action implies that $u'.\mathit{last}_2(I) = \infty$, which suffices to satisfy alternative 1.

7. $\pi = \mathit{down}$.

   Enabling: Similar to *enterR*.

   Condition 5: Straightforward.

8. $\pi = \nu(\Delta t)$.

   Enabling: We must show that time Δt is allowed to pass in *OpSpec*. This amounts to showing that $s'.\mathit{now} \leq u.\mathit{last}_1$ and $s'.\mathit{now} \leq u.\mathit{last}_2(I)$.

   To show $s'.\mathit{now} \leq u.\mathit{last}_1$, we only need to consider the case where $u.\mathit{last}_1 \neq \infty$. In this case, Condition 4 implies that $u.\mathit{last}_1 \geq s.\mathit{Trains.last}(\mathit{enterI}(r))$ for some r. The precondition on time-passage in *CompImpl* implies that
   $s'.\mathit{now} \leq s.\mathit{Trains.last}(\mathit{enterI}(r))$. So $s'.\mathit{now} \leq u.\mathit{last}_1$, as needed.

   To show that $s'.\mathit{now} \leq u.\mathit{last}_2(I)$, we only need to consider the case where $u.\mathit{last}_2(I) \neq \infty$. In this case, we consider the three alternatives, for s and u. If the first alternative holds, then the argument is as for last_1. If the second alternative holds, then the *raise* precondition holds in s. But this implies that ν cannot be enabled in s, a contradiction. If the third alternative holds, then $s'.\mathit{now} \leq s.\mathit{Gate.last}(\mathit{up}) \leq u.\mathit{last}_2(\mathit{up}) \leq u.\mathit{last}_2(I)$, which suffices.

Condition 5: The only alternative that the time-passage action might falsify is the second. But this means that the *raise* precondition holds in s, which is impossible since then ν could not be enabled in s.

■

6.3 Putting the Pieces Together

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

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

Putting this together with Lemma 4.3, we obtain the main theorem:

Theorem 6.6 *For any admissible timed execution α of SystImpl, 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.

It is possible to define a *pair of models* for any real world component, 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:

State:

status in {*up*, *down*, *going-up*, *going-down*}, initially *up*
$pos \in [0, 90]$, initially 90
now, a nonnegative real, initially 0
last(*down*), a nonnegative real or ∞, initially ∞

last(*up*), a nonnegative real or ∞, initially ∞

The transitions are as follows. 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}. The *up* and *down* transitions contains new preconditions stating that the correct position has been reached. The time-passage transitions adjust *pos*.

Transitions:

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

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

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
      $s'.pos = \min\{s.pos, g_d(t - (s.last(down) - \gamma'_{down}))\}$
    else unchanged $pos$
```

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*.

One detail needs mentioning: we need to verify that when we apply the functions g_u and g_d to arguments in the time-passage transitions, the arguments are in fact within the specified interval. For example, consider the application $g_u(t-(s.last(up)-\gamma'_{up}))\}$, which occurs when $s.status = going\text{-}up$. We must be sure that $0 \leq t - (s.last(up) - \gamma'_{up}) \leq \gamma'_{up}$. The first inequality follows from the facts that $s.last(up) \leq s.now + \gamma'_{up}$ and $s.now \leq t$, while the second inequality follows from the fact that $t \leq s.last(up)$.

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

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

Proof: By Theorem C.1, it suffices to show the existence of a simulation mapping from *Gate*$'$ to *Gate*. If s and u are states of *Gate*$'$ and *Gate*, respectively, then we define s and u to be related by relation f provided that:

1. $u.status = s.status$.
2. $u.now = s.now$.
3. $u.last(down) \geq s.last(down)$.
4. $u.last(up) \geq s.last(up)$.

It is straightforward to show that f is a simulation mapping. ■

Now, let $SystImpl'$ 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 $SystImpl'$, there is an admissible timed execution β' of $AxSpec'$ such that $\alpha'|Trains \times Gate' = \alpha|Trains \times Gate'$.*

Proof: We define two environment automata: let E be the composition of *Gate* and *Trains*, and E' the composition of $Gate'$ and *Trains*. Lemma 7.1 says that $attraces(Gate') \subseteq attraces(Gate)$. This and a basic substitutivity property of composition, Lemma A.1, imply that $attraces(E') \subseteq attraces(E)$.

Let α' be any admissible timed execution of $SystImpl'$, and let $\delta = ttrace(\alpha')$.

Lemma A.2 implies that $\alpha'|E' \in atexecs(E')$ and $\alpha'|CompImpl \in atexecs(CompImpl)$. Since $attraces(E') \subseteq attraces(E)$, we may obtain $\gamma \in atexecs(E)$ with $ttrace(\gamma) = ttrace(\alpha'|E') = ttrace(\alpha')|E' = \delta|E' = \delta$.

Now we construct an admissible timed execution α of *SystImpl*. Since $\delta|E = ttrace(\gamma)$ and $\delta|CompImpl = ttrace(\alpha'|CompImpl)$, Lemma A.3 implies that there is an admissible timed execution α of *SystImpl* such that $\alpha|E = \gamma$ and $\alpha|CompImpl = \alpha'|CompImpl$. We have that $ttrace(\alpha) = ttrace(\gamma) = \delta$.

Next, by Theorem 6.6 applied to α, we obtain an admissible timed execution β of *AxSpec* such that $\beta|E = \alpha|E$. It follows that $ttrace(\beta) = ttrace(\alpha) = \delta$ and that β satisfies the Safety and Utility properties.

Now we define an admissible timed execution β' of the system composed of E' and *CompSpec*. Since $\delta|E' = ttrace(\alpha'|E')$ and $\delta|CompSpec = ttrace(\beta|CompSpec)$, Lemma A.3 implies that there is an admissible timed execution β' of the system composed of E' and *CompSpec* such that $\beta'|E' = \alpha'|E'$ and $\beta'|CompSpec = \beta|CompSpec$. Note that $ttrace(\beta') = \delta$.

We claim that β' satisfies the required properties. First, β' is defined to be an admissible timed execution of the system composed of E' and *CompSpec*; to see that it is an admissible timed execution of $AxSpec'$ it suffices to show that it satisfies the Safety and Utility properties. But this follows from the facts that $ttrace(\beta') = \delta = ttrace(\beta)$ and that β satisfies the Safety and Utility properties. (These properties can be inferred from the timed traces.) Second, $\beta'|E' = \alpha'|E'$ by construction. This is as needed. ■

8 Discussion

We have applied a formal method based on timed automata, invariants, and simulation mappings to model and verify the Generalized Railroad Crossing example [7]. Here, we extrapolate from this experience and attempt to evaluate the method for use in 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. For instance, 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 approaches based on model-checking, 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 it is natural to model these 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.

- **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 model-checking, 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.

- **Information.** *Does the proof yield information other than just the fact that the implementation is correct? Does it give any 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-checking methods yield no such byproducts, only an assertion that the implementation satisfies the desired properties.

- **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, however, 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 includes:

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

2. Developing the appropriate computer assistance for carrying out and checking the proofs. We plan to try 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.

Acknowledgments

Discussions with R. Jeffords of NRL helped identify and clarify software engineering principles useful in developing the GRC solution and led to improvements in the proofs in Appendix D.

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

[7] 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.

[8] Constance Heitmeyer and Nancy Lynch. The Generalized Railroad Crossing: A case study in formal verification of real-time systems. In *Proceedings, Real-Time Systems Symposium*, San Juan, Puerto Rico, December 1994. To appear.

[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].

A.1 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 $now_A : states \rightarrow \mathsf{R}^{\geq 0}$. ($\mathsf{R}^{\geq 0}$ 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 $s \xrightarrow{a}_A s'$ 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 must satisfy 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 $s \xrightarrow{\nu} s'$. Axiom [A1] says that the current time is always 0 in a start state. Axiom [A2] says that non-time-passage steps do not change the time; that is, they occur "instantaneously", at a single point in time. Axiom [A3] says that time-passage steps must cause the time to increase; this is a convenient technical restriction. Axiom [A4] allows repeated time-passage steps to be combined into one step.

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 define $w.ftime = min(I)$ and $w.fstate = w(w.ftime)$, while if I is right-closed, then define $w.ltime = max(I)$ and $w.lstate = w(w.ltime)$. 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'. Axiom [A5] is a kind of converse to [A4]; it says that any time-passage step can be "filled in" with states for each intervening time, in a "consistent" way.

A.2 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 then $w_j.lstate \xrightarrow{\pi_{j+1}} w_{j+1}.fstate$.

The trajectories describe the changes of state during the time-passage steps. The last item says that the actions in α span between successive trajectories. 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 $\mathit{atexecs}(A)$ 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.

In order 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 $\mathit{attraces}(A)$ for the set of admissible timed traces of timed automaton A. Often, we express requirements to be satisfied by a timed automaton A as the set of admissible timed traces of another timed automaton B. Then we say that A *implements* B if $\mathit{attraces}(A) \subseteq \mathit{attraces}(B)$. If α is any timed execution, we use the notation $\mathit{ttrace}(\alpha)$ 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 $\mathit{time}(\pi) = s.\mathit{now}$.

A.3 Composition

We define a simple binary parallel composition operator for timed automata. 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.

- $\mathit{states}(A \times B) = \{(s_A, s_B) \in \mathit{states}(A) \times \mathit{states}(B) : s_A.\mathit{now}_A = s_B.\mathit{now}_B\}$;
- $\mathit{start}(A \times B) = \mathit{start}(A) \times \mathit{start}(B)$;
- $\mathit{acts}(A \times B) = \mathit{acts}(A) \cup \mathit{acts}(B)$; an action is *external* in $A \times B$ exactly if it is external in either A or B, and likewise for *internal* actions; 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 $\pi \in \mathit{acts}(A)$, else $s_A = s'_A$, and
  2. $s_B \xrightarrow{\pi}_B s'_B$ if $\pi \in \mathit{acts}(B)$, else $s_B = s'_B$;
- $(s_A, s_B).\mathit{now}_{A\times B} = s_A.\mathit{now}_A$.

Then $A \times B$ is a timed automaton. If α is a timed execution of $A \times B$, we write $\alpha|A$ and $\alpha|B$ for the projections of α on A and B, respectively. 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, writing, e.g., $\beta|A$ for the subsequence of β consisting of actions of A.

Lemma A.1 *(Substitutivity) Let A and B be timed automata with the same input and output actions, and let C be a timed automaton compatible with both. If attraces$(A) \subseteq$ attraces(B) then attraces$(A \times C) \subseteq$ attraces$(B \times C)$.*

Lemma A.2 *If $\alpha \in$ atexecs$(A \times B)$ then $\alpha|A \in$ atexecs(A) and $\alpha|B \in$ atexecs(B).*

Lemma A.3 *Suppose that $\alpha_A \in$ atexecs(A) and $\alpha_B \in$ atexecs(B). Suppose β is a sequence of timed visible actions of $A \times B$ such that $\beta|A =$ ttrace(α_A) and $\beta|B =$ ttrace(α_B). Then there exists $\alpha \in$ atexecs$(A \times B)$ such that $\alpha|A = \alpha_A$ and $\alpha|B = \alpha_B$.*

Since the composition operation is associative, up to isomorphism, we may extend it to an arbitrary finite number of argument timed automata.

B MMT Automata

B.1 Automaton Definition

MMT automata were originally defined by Merritt, Modugno and Tuttle [17]; we use a special case of their definition from [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 $\mathit{states}(A)$ of states, a nonempty set $\mathit{start}(A) \subseteq \mathit{states}(A)$ of start states, a set $\mathit{acts}(A)$ of actions, (partitioned into *external* and *internal* actions; the external actions are further partitioned into *input* and *output* actions), a set $\mathit{steps}(A)$ of steps, and a partition $\mathit{part}(A)$ of the locally controlled (i.e., output and internal) actions into at most countably many equivalence classes. The set $\mathit{steps}(A)$ is a subset of $\mathit{states}(A) \times \mathit{acts}(A) \times \mathit{states}(A)$; An action π is said to be *enabled* in a state s provided that there exists a state s' such that $(s, \pi, s') \in \mathit{steps}(A)$, 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. It is required that the automaton be *input-enabled*, by which is meant 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, $\mathit{lower}(C)$ and $\mathit{upper}(C)$, where $0 \leq \mathit{lower}(C) < \infty$ and $0 < \mathit{upper}(C) \leq \infty$; that is, the lower bounds cannot be infinite and the upper bounds cannot be 0.

B.2 Timed Executions and Timed Traces

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. More specifically, define j to be an *initial index* for a class C provided that C is enabled in s_j, and either $j = 0$, or else C is not enabled in s_{j-1}, or else $\pi_j \in C$; initial indices are the points at which the bounds for C begin to be measured. Then for every initial index j for a class C, the following conditions must hold:

1. (Upper bound)
   If $\mathit{upper} \neq \infty$, then there exists $k > j$ with $t_k \leq t_j + \mathit{upper}(C)$ such that either $\pi_k \in C$ or C is not enabled in s_k.

2. (Lower bound)
   There does not exist $k > j$ with $t_k < t_j + \mathit{lower}(C)$ and $\pi_k \in C$.

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.

MMT automata can be composed in much the same way as ordinary I/O automata, using synchronization on common actions. More specifically, define two MMT automata A and B to be *compatible* according to the same definition of compatibility for timed automata. Then the *composition* of the two automata is the MMT automaton consisting of the I/O automaton that is the composition of the two component I/O automata (according to the definition of composition in [13]), together with the bounds arising from the components. This composition operator is substitutive for the admissible timed trace inclusion ordering on MMT automata.

B.3 MMT Automata and Timed Automata

MMT automata are not exactly a special case of timed automata. This is because the MMT model uses an "external" way of specifying the time bound restrictions, via the added lower and upper bounds. Timed automata, in contrast, build the time-bound restrictions explicitly into the time-passage steps. However, 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, respectively, the earliest and latest time in the future that an action in class C is allowed to occur. The *now*, *first* and *last* components all take on values that represent *absolute* times, not incremental times. 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 be at least equal to *first*(C).

In more detail, each state of A' is a record consisting of a component *basic*, which is a state of A, a component $now \in \mathsf{R}^{\geq 0}$, and, for each class C of A, components *first*(C) and *last*(C), each in $\mathsf{R}^{\geq 0} \cup \{\infty\}$. Each start state s of A' has $s.basic \in start(A)$, and $s.now = 0$. Also, if C is enabled in $s.basic$, then $s.first(C) = lower(C)$ and $s.last(C) = upper(C)$; otherwise $s.first(C) = 0$ and $s.last(C) = \infty$. The actions of A' are the same as those of A, with the addition of the time-passage action ν. Each non-time-passage action is classified as an input, output or internal action according to its classification in A.

The steps are defined as follows. If $\pi \in acts(A)$, then $s \xrightarrow{\pi}_{A'} s'$ exactly if all the following conditions hold:

1. $s.now = s'.now$.

2. $s.basic \xrightarrow{\pi}_{A} s'.basic$.

3. For each $C \in part(A)$:

   (a) If $\pi \in C$ then $s.first(C) \leq s.now$.

   (b) If C is enabled in both s and s', and $\pi \notin C$, then $s'.first(C) = s.first(C)$ and $s'.last(C) = s.last(C)$.

   (c) If C is enabled in s' and either C is not enabled in s or $\pi \in C$ then $s'.first(C) = s.now + lower(C)$ and $s'.last(C) = s.now + upper(C)$.

   (d) If C is not enabled in s' then $s'.first(C) = 0$ and $s'.last(C) = \infty$.

On the other hand, if $\pi = \nu$, then $s' \xrightarrow{\pi}_{A'} s$ exactly if all the following conditions hold:

1. $s'.now < s.now$.

2. $s.basic = s'.basic$.

3. For each $C \in part(A)$:

   (a) $s.now \leq s'.last(C)$.

   (b) $s.first(C) = s'.first(C)$ and $s.last(C) = s'.last(C)$.

The resulting timed automaton A' has exactly the same admissible timed traces as the MMT automaton A.

Moreover, this transformation commutes with the operation of composition, up to isomorphism. We refer to an MMT automaton and to its transformed version interchangeably. Another way of looking at the preceding construction is as showing exactly how the MMT notation is used to denote timed automata.

The following is a technical lemma that is useful in some of the invariant and simulation proofs.

Lemma B.1 *In all reachable states of a (transformed) MMT automaton, and for all classes C, the following hold.*

1. *$now \leq last(C)$*

2. *If $last(C) \neq \infty$ then $last(C) \leq now + upper(C)$.*

C 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, respectively. 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 C.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).*

D Correspondence Between Original Specification and *AxSpec*

We show how the Utility Property in the original formulation of the GRC [7] relates to the Utility Property in *AxSpec*. In the original formulation, the Utility Property is expressed as "If $t \notin \cup_i[\tau_i - \xi_1, \nu_i + \xi_2]$, then $g(t) = 90$," which can be rewritten as "If $g(t) \neq 90$, then $t \in \cup_i[\tau_i - \xi_1, \nu_i + \xi_2]$." In the Lynch-Vaandrager model, the Utility Property can be stated as an axiom of any admissible timed execution α: "If s is a state in α with $s.Gate.status \neq up$, then $s.now \in [\tau_i - \xi_1, \nu_i + \xi_2]$ for some i."

In the description of *AxSpec* in Section 3.5, the Utility Property is expressed as

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 $s''.now - s'.now \leq \xi_1 + \xi_2 + \delta$.

Lemmas D.1 and D.2 show that the Lynch-Vaandrager form of the original Utility Property and the statement of utility in *AxSpec* (parts a and b only) are equivalent.

Lemma D.1 *If s is a state in α and $s.now \in [\tau_i - \xi_1, \nu_i + \xi_2]$ for some i, then (a) or (b) holds.*

Proof: There are three cases.

1. For $s.now \in [\tau_i - \xi_1, \tau_i]$, choose s' to be the last state in α such that $s'.now = \tau_i$. Then, $s'.Trains.r.status = I$ for some r, and s' follows (or is equal to) s in α. Further, $\tau_i - \xi_1 \leq s.now$, which implies $\tau_i \leq s.now + \xi_1$. This implies $s'.now \leq s.now + \xi_1$.

2. For $s.now \in [\tau_i, \nu_i]$, choose $s' = s$. Then, $s'.Trains.r.status = I$ for some r, and $s.now \geq s.now - \xi_2$ by assumptions on the constants. This implies $s'.now \geq s.now - \xi_2$.

3. For $s.now \in [\nu_i, \nu_i + \xi_2]$, choose s' to be the first state in α such that $s'.now = \nu_i$. Then, $s'.Trains.r.status = I$ for some r, and s' precedes (or is equal to) s in α. Further, $s.now \leq \nu_i + \xi_2$, which implies $s.now - \xi_2 \leq \nu_i$. This implies $s.now - \xi_2 \leq s'.now$.

■

Lemma D.2 *If s is a state in α and (a) or (b) holds, then $s.now \in [\tau_i - \xi_1, \nu_i + \xi_2]$ for some i.*

Proof: Assume (a). Then, $s.now - \xi_2 \leq s'.now$, which implies $s.now \leq s'.now + \xi_2$. Further, $s'.Trains.r.status = I$ for some r implies $\tau_i \leq s'.now \leq \nu_i$ for some i, which implies $s'.now + \xi_2 \leq \nu_i + \xi_2$. This implies $s.now \leq \nu_i + \xi_2$ as needed. By assumptions on the constants, $\tau_i - \xi_1 \leq \tau_i$. This implies $\tau_i - \xi_1 \leq s.now$, since s' precedes or is equal to s.

Assume (b). Then, $s'.Trains.r.status = I$ for some r implies $\tau_i \leq s'.now \leq \nu_i$. Further, s' follows (or equals) s in α implies $s.now \leq s'.now$. By assumptions on the constants, $\nu_i \leq \nu_i + \xi_2$. Hence, $s.now \leq \nu_i + \xi_2$ as needed. Also, $\tau_i - \xi_1 \leq s'.now - \xi_1$. Then, $s'.now \leq s.now + \xi_1$ implies $s'.now - \xi_1 \leq s.now$. This implies $\tau_i - \xi_1 \leq s.now$ as needed.

■

To prevent the gate from being raised and lowered uselessly, we revised the Utility Property so that the gate is only raised if there is sufficient time δ, $\delta > 0$, for at least one car to pass through the crossing. To express this added constraint, the original statement can be rewritten as

If $g(t) \neq 90$, then at least one of the following holds:

1. $t \in \cup_i [\tau_i - \xi_1, \nu_i + \xi_2]$ or
2. $t \in [\nu_i + \xi_2, \tau_{i+1} - \xi_1]$ with $\tau_{i+1} - \nu_i \leq \xi_2 + \delta + \xi_1$ for some i.

In the Lynch-Vaandrager model, this is expressed as

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

1. $s.now \in [\tau_i - \xi_1, \nu_i + \xi_2]$ or
2. $s.now \in [\nu_i + \xi_2, \tau_{i+1} - \xi_1]$ with $\tau_{i+1} - \nu_i \leq \xi_2 + \delta + \xi_1$.

We show the equivalence between the latter statement of the Utility Property and the statement of utility (parts a, b, and c) in *AxSpec*.

Lemma D.3 *If there exists a state s in α such that for some i either condition (1) or condition (2) holds, then at least one of conditions (a), (b), or (c) holds.*

Proof: Lemma D.1 shows that if condition (1) holds, either (a) or (b) holds. We show that condition (2) implies condition (c). Let s' be some state such that $s'.now = \nu_i$ and s'' be some state such that $s''.now = \tau_{i+1}$. Then, by definition, $s'.Trains.r.status = I$ for some r and $s''.Trains.r.status = I$ for some r. Further, $s.now \in [\tau_i - \xi_1, \nu_i + \xi_2]$ implies $s'.now < s.now < s''.now$, so s' precedes s and s'' follows s in α. Finally, $\tau_{i+1} - \nu_i \leq \xi_2 + \delta + \xi_1$ implies $s''.now - s'.now \leq \xi_1 + \xi_2 + \delta$.

■

Lemma D.4 *If there exists a state s in α such that at least one of conditions (a), (b), or (c) holds, then for some i condition (1) or condition (2) holds.*

Proof: Lemma D.2 shows that if either (a) or (b) holds, then (1) holds. We show that if condition (c) holds, then either (1) or (2) holds. There are two cases.

1. Suppose s occurs during the interval $[\tau_i - \xi_1, \nu_i + \xi_2]$ for some i. Clearly, (1) holds.
2. Suppose s occurs during the interval $[\nu_i + \xi_2, \tau_{i+1} - \xi_1]$ for some i. Then, $s'.Trains.r.status = I$ for some r and s' precedes or is equal to s implies $s'.now \leq \nu_i$. Similarly, $s''.Trains.r.status = I$ for some r and s'' follows or is equal to s implies $s''.now \geq \tau_{i+1}$. Hence, $s''.now - s'.now \geq \tau_{i+1} - \nu_i$. This together with $s''.now - s'.now \leq \xi_1 + \xi_2 + \delta$ implies $\tau_{i+1} - \nu_i \leq \xi_1 + \xi_2 + \delta$. Therefore, (2) holds.

■

E Proof of Relationship Between *OpSpec* and *AxSpec*

We prove Lemma 4.3. We first prove an easy property of *OpSpec*:

Lemma E.1 *Let α be any admissible timed execution of OpSpec. Let π be any lower event occurring in α from a state in which Gate.status $\in \{going\text{-}up, up\}$. Then there is an enterI event ϕ occurring after π in α, with time$(\phi) \leq$ time$(\pi) + \xi_1$.*

Proof: Suppose that π occurs from state s, leading to state s'; then $time(\pi) = s.now = s'.now$. If $s.last_1 = \infty$, then the effect of *lower* implies that $s'.last_1 = s'.now + \xi_1$. Otherwise, part 3 of Lemma 6.2 implies that $s'.last_1 \leq s'.now + \xi_1$. Thus, in either case, $s'.last_1 \leq s'.now + \xi_1$.

By the precondition for the time-passage action and the fact that only *enterI* actions can increase $last_1$, real time cannot pass beyond $s'.last_1$ unless a train enters I at a time $\leq s'.last_1 \leq s'.now + \xi_1$. But α is an admissible execution, so time passes to ∞. It follows that there must be an *enterI* event ϕ occurring after π in α such that $time(\phi) \leq time(\pi) + \xi_1$. ■

Now for the proof of Lemma 4.3:

Proof: Let α be any admissible timed execution of *OpSpec*. By assumption, α satisfies the Safety Property, i.e., the safety invariant is true in all states of α. We show that it also satisfies the Utility Property.

Let s be any state occurring in α with $s.Gate.status \neq up$. If $s.Trains.r.status = I$ for any r, then the claim is immediate. So assume that $s.Trains.r.status \neq I$ for all r.

Since $s.Gate.status \neq up$, it must be that there was a previous *lower* event occurring from a state in which $Gate.status \in \{going\text{-}up, up\}$. (Consider the possible transitions in the automaton *Gate*.) Let π be the last *lower* event preceding s that occurs from a state in which $Gate.status \in \{going\text{-}up, up\}$. Then $Gate.status \neq up$ in the entire interval from just after π to s. Also, $time(\pi) \leq s.now$.

By Lemma E.1, an *enterI* event occurs within time ξ_1 after π. Let ϕ be the first such *enterI* event. Thus, $time(\phi) \leq time(\pi) + \xi_1 \leq s.now + \xi_1$. By the Safety Property, $Gate.status = down$ when ϕ occurs. We consider two cases.

1. ϕ is after s.

   Then take s' to be the state just after ϕ. Then $s'.now = time(\phi) \leq s.now + \xi_1$, and so s' satisfies the axiomatic Utility Property, part (b).

2. ϕ is before s.

   Since (by assumption) there is no train in I in state s, we can find a latest *exit* event ψ following ϕ and preceding s, which must leave I empty. When ψ occurs, the $last_2$ variables are set, which ensures that either the gate reaches the *up* position within time ξ_2 after ψ, or some train reaches I within time $\xi_2 + \delta + \xi_1$ after ψ. We consider two subcases.

   (a) $s.now \leq time(\psi) + \xi_2$.
   Then take s' to be the state just prior to ψ. Then $s'.now = time(\psi) \geq s.now - \xi_2$, and so s' satisfies the axiomatic Utility Property, part (a).

(b) $s.now > time(\psi) + \xi_2$.

Then the gate cannot reach the *up* position within time ξ_2 after ψ, because $Gate.status \neq up$ throughout the interval from π to s. So it must be that some train reaches I within time $\xi_2 + \delta + \xi_1$ after ψ. That is, there must be some *enterI* event ψ' following ψ, with $time(\psi') \leq time(\psi) + \xi_2 + \delta + \xi_1$.

We claim that ψ' must follow s. For if it did not, the fact that I is empty in s would imply that there must be an *exit* event following ψ' and preceding s, which contradicts our choice of ψ. Then take s' to be the state just before ψ and s'' to be the state just after ψ'. Then $s''.now - s'.now = time(\psi') - time(\psi) \leq \xi_2 + \delta + \xi_1$. Thus, s' and s'' satisfy the axiomatic Utility Property, part (c).

■