The Generalized Railroad Crossing Problem: An Evolving Algebra Based Solution

Yuri Gurevich, James K. Huggins, and Raghu Mani*

Abstract

We present an evolving algebra based solution for the Generalized Railroad Crossing problem – a specification and verification benchmark proposed by C. Heitmeyer at NRL. We specify the system as an evolving algebra and prove that the specification satisfies the desired safety and liveness properties.

1 Introduction

The Generalized Railway Crossing problem involves specifying a system that controls a railway crossing gate and proving the safety and liveness of the system. The problem statement (taken from [6]) is as follows.

> The system to be developed operates a gate at a railroad crossing. The railroad crossing I lies in a region of interest R, i.e., $I \subseteq R$. A set of trains travel through R on multiple tracks in both directions. A sensor system determines when each train enters and exits region R. To describe the system formally, we define a gate function $g(t) \in [0, 90]$, where $g(t) = 0$ means the gate is down and $g(t) = 90$ means the gate is up. We define a set $\{\lambda_i\}$ of *occupancy intervals*, where each occupancy interval is a time interval during which one or more trains are in I. The ith occupancy interval is represented as $\lambda_i = [\tau_i, \nu_i]$, where τ_i is the time of the ith entry of a train into the crossing when no other train is in the crossing and ν_i is the first time since τ_i that no train is in the crossing (i.e., the train that entered at τ_i has exited as have any trains that entered the crossing after τ_i).
>
> Given two constants ξ_1 and ξ_2, $\xi_1 > 0$, $\xi_2 > 0$, the problem is to develop a system to operate the crossing gate that satisfies the following two properties:
>
> **Safety Property:** $t \in \cup_i \lambda_i \Rightarrow g(t) = 0$ (The gate is down during all occupancy intervals.)
> **Utility Property:** $t \notin \cup_i [\tau_i - \xi_1, \nu_i + \xi_2] \Rightarrow g(t) = 90$ (The gate is up when no train is in the crossing.)

In order to make this paper self-contained, we describe the evolving algebra (often abbreviated *ealgebra* or *EA*) specification language in section 2. Section 3 contains the EA specification of the system. Section 4 contains some definitions and notations used in our proofs. Section 5 contains the proofs.

2 Evolving Algebras

In this section, we present a brief description of ealgebras. A popular exposition on sequential ealgebras can be found in [3]; a more complete description can be found in [4]. Examples of EA specifications and verifications of distributed and time-constrained systems can be found in [2], [5] and [7]; for other papers involving ealgebras, see [1].

*CSE Technical Report CSE-TR-230-95. EECS Department, University of Michigan–Ann Arbor, Ann Arbor, MI, 48109-2122, USA. {gurevich, huggins, raghu}@eecs.umich.edu. Partially supported by ONR grant N00014-91-J-1861 and NSF grant CCR-92-04742.

2.1 States

Every ealgebra has a *vocabulary* (or *signature*): that is, a finite collection of function names, each of a fixed arity. Every vocabulary contains the nullary function names *true*, *false*, *undef*, as well as the names of the usual Boolean operations and the equality sign.

A state (or *static algebra*) S of an ealgebra $\mathcal{E}$ of vocabulary Υ is a non-empty set X, called the *superuniverse* of S, along with interpretations of each function name in Υ over X. The interpretations of the nullary names *true*, *false*, and *undef* are always distinct. The interpretations of the Boolean function names behave in the usual way over $\{true, false\}$ and take the value *undef* otherwise.

Function names may be tagged as *external*; the idea is that external function names have their values determined outside of the control of the ealgebra.

The nullary name *undef* is used to represent partial functions: intuitively, $f(\bar{x}) = undef$ if $\bar{x}$ is a tuple of values outside the domain of f. Relations are represented as Boolean-valued functions. A boolean-valued unary function $U(x)$ can be seen to represent a set: namely, the set $\{x : U(x) = true\}$. In such a case we call U a *universe*. For example, the vocabulary of every ealgebra contains the universe name *Bool*; in every state, the universe *Bool* contains two elements, *true* and *false*.

2.2 Transition Rules

Transition rules describe how states of an ealgebra change over time. An *update instruction* is the simplest type of transition rule and has the form $f(t_1, t_2, \ldots, t_n) := t_0$ where f is a function name of arity n and each t_i is a term (as defined in propositional logic).[1] Executing such an instruction has the expected result: if $a_1, \ldots, a_n$ and b are the values of $t_1, \ldots, t_n$ and t_0 in the current state, $f(a_1, \ldots, a_n) = b$ in the next state.

A *block rule* is syntactically a sequence of transition rules. To execute a block rule, execute each of the rules in the sequence simultaneously. Conflicts between rules are not permitted. If a conflict is encountered in a run, execution stops.

A *conditional rule* has the form

if g_0 **then** R_0
elseif g_1 **then** R_1
$\vdots$
elseif g_n **then** R_n
endif

where the g_i are Boolean-valued first-order terms and the R_i are transition rules. To execute a transition rule of this form in state S, evaluate guards g_i in state S; if any of the g_i evaluate to *true*, execute transition rule R_k, where g_k is *true* but g_i is *false* for $i < k$. If none of the g_i evaluate to *true*, do nothing. (The phrase "**elseif** *true* **then** R_n" is usually abbreviated as "**else** R_n").

A *declaration rule* has the form

var x **ranges over** U
R

where x is a variable and U is a universe name. To execute such a declaration rule in a state in which U contains n elements, execute n copies of R simultaneously, with x taking a different value in U in each copy.

A program is simply a rule without any undeclared free variables.

Let Υ be the vocabulary of the ealgebra $\mathcal{E}$. Let Υ^- denote the set of all internal (*i.e.* non-external) function names of $\mathcal{E}$; we call this the *internal vocabulary*. We define an *internal state* to be a static Υ^--algebra. If S is a state of $\mathcal{E}$, then S^- denotes the corresponding internal state. An execution or *run* of a sequential ealgebra is a sequence of states. If S_i and S_{i+1} are consecutive states in a run, then S_{i+1}^- is the result of executing the program of $\mathcal{E}$ in S_i.

[1] Except that we treat predicates as functions whose range is always $\{true, false\}$. In particular, $t_1 = t_2$ is a term.

2.3 Distributed Evolving Algebras

We can visualize a sequential program as being executed by an *agent* that evaluates the rules of the program and then executes the enabled updates at each step. We can visualize that a distributed program is executed by a set of such agents, each independently executing a sequential program.

More formally, a *distributed ealgebra* is specified by a vocabulary Υ, a set M of sequential programs called *modules*, and a set I of initial states. Modules are executed by agents. In the simplest case, a run of a distributed ealgebra is a sequence of global states. If S_i and S_{i+1} are consecutive states in a sequential run then S_{i+1}^- is the result of executing the enabled updates of some non-conflicting subset of agents in S_i. This suffices for our purposes here; for a more general definition of runs, see [4].

The reader may wonder what an agent is. Included within Υ is a unary function *Mod* such that, in each global state S, the range of *Mod* consists of the elements of S representing modules. Agents are elements in the domain of *Mod*. For more on agents, see [4].

3 The Specification

We specify the system as two modules – one for the gate and one for the controller which directs the motion of the gate. Trains are not modeled explicitly; instead, we use external functions to model train movements within the region.

In our specification, we make various assumptions: trains only move in one direction through the crossing and do not break-down within the crossing, a track contains at most one train at any moment, and so on. We also treat time at a high level of abstraction. Similar assumptions are made in [6]. These assumptions can be strengthened or weakened; it is relatively straightforward to adjust the specification and proofs.

3.1 The Vocabulary

Real and Bool are the universes of reals and booleans respectively. (Here and elsewhere, we denote universe names by printing them in sans-serif.)

An external nullary function CT : Real (an allusion to the *current time*) gives the value of the time in a state according to some external clock. We restrict our attention to runs in which the value of CT increases monotonically.

Tracks is the universe of tracks that pass through the crossing. The external function *TrackStatus* : Tracks $\rightarrow$ {*coming*, *in_crossing*, *empty*} tells us where (if anywhere) a train is on a given track. We assert that *TrackStatus* takes the values *coming*, *in_crossing*, and *empty* in that order, possibly repeating this cycle many times.

The nullary function *GateStatus* takes values in {*up*, *down*, *going_up*, *going_down*} and gives us the current position of the gate. The nullary function *Dir* takes values in {*up*, *down*} and is used by the controller to tell the gate which direction to move.

Deadline : Tracks $\rightarrow$ Real records the time at which the controller must signal the gate to close because of an oncoming train.

3.2 Constants and Abbreviations

We use a few timing constants in our specification and proof.

- d_{min} and d_{max} are lower and upper bounds on the time between the controller detecting the entry of a train into the region and the entry of that train into the crossing. Naturally, we assume $d_{min} \leq d_{max}$.
- d_{down} and d_{up} are upper bounds on the time taken to lower and raise the gate. We assume $d_{down} < d_{min}$; otherwise, a train could arrive at the intersection before the controller has a chance to close the gate.

SafeToOpen is an abbreviation for "$\forall t \in \mathsf{Tracks}(TrackStatus(t) = empty \vee CT + d_{up} < Deadline(t))$"; the intended meaning is that it is safe to attempt to open the gate, since any oncoming train is more than d_{up} time away from its deadline for closing the gate. If d_{min} is small, this attempt might be aborted.

3.3 The Program

MODULE Gate

Rule: GateUp
if $(Dir = up)$ **then**
  if $(GateStatus = down) \vee (GateStatus = going\_down)$ **then** $GateStatus := going\_up$
  elseif $(GateStatus = going\_up)$ **then** $GateStatus := up$
  endif
endif

Rule: GateDown
if $(Dir = down)$ **then**
  if $(GateStatus = up) \vee (GateStatus = going\_up)$ **then** $GateStatus := going\_down$
  elseif $(GateStatus = going\_down)$ **then** $GateStatus := down$
  endif
endif

MODULE Controller

Rule: SignalDown
var t **ranges over** $Tracks$
  if $(Deadline(t) = \infty) \wedge (TrackStatus(t) = coming)$ **then** $Deadline(t) := CT + d_{min} - d_{down}$
  elseif $CT = Deadline(t)$ **then** $Dir := down$
  endif

Rule: SignalUp
var t **ranges over** $Tracks$
  if $(TrackStatus(t) = empty) \wedge Deadline(t) < \infty$ **then**
    $Deadline(t) := \infty$
    if $SafeToOpen \wedge Dir = down$ **then** $Dir := up$ **endif**
  endif
endif

For simplicity, we suppose that in the initial state of any run, $GateStatus = up$, $Dir = up$, and for all tracks t, $TrackStatus(t) = empty$ and $Deadline(t) = \infty$.

Note that CT is evaluated within a particular state; time increases from one state to the next. Note also that our controller attempts to raise the gate more often than strictly necessary; if a train is coming but there is enough time to raise and lower the gate before the train reaches the intersection, the controller attempts to raise the gate. Strictly speaking, this is not required by the problem specification.

4 Definitions and Notation

Consider a run ρ and let a and b range over states in ρ. The states of ρ form a sequence. Accordingly, we write $a < b$ if a precedes state b in that sequence. We denote by $a - 1$ and $a + 1$ the immediate predecessor and successor states to a, if they exist. We denote by $[a, b)$ the interval of states between a and b, including a but not b. $(a, b]$ and $[a, b]$ are similarly defined.

e *becomes true* in $(a, a + 1)$ if e is false in a and true in $a + 1$. e *becomes false* is defined similarly. CT_a denotes the value of function CT in state a.

If an expression e_1 is true in a and there exists $b > a$ such that e_1 holds in $[a, b)$ and an expression e_2 holds in b, then we denote by $Succ_{e_1,e_2}(a)$ the least such b. If $Succ_{e_1,e_2}(a)$ is defined for every a such that

e_1 becomes true in $(a-1, a)$, we say e_1 *holds until* e_2. For example, $(TrackStatus(t) = coming)$ holds until $(TrackStatus(t) = in\_crossing)$, which holds until $(TrackStatus(t) = empty)$. Similarly, $Pred_{e_1,e_2}(a)$ is the latest state $b < a$ such that e_2 is true in b and e_1 is true in $(b, a]$ (if it exists).

If e_1 holds until e_2, a ranges over states in which e_1 becomes true in $(a-1, a)$, and $b = Succ_{e_1,e_2}(a)$, then $MaxTime(e_1, e_2) = \sup_a(CT_b - CT_a)$ and $MinTime(e_1, e_2) = \inf_a(CT_b - CT_a)$. If e_1 does not precede e_2, $MaxTime$ and $MinTime$ are undefined.

Regular Runs ρ is called a *regular run* if it satisfies the following constraints:

1. *[Activity]* A state a_{i+1} is obtained from its predecessor a_i by one or more of the following: executing the gate module, executing the controller module, or changing $TrackStatus$.
2. *[Liveness]* An agent may not be enabled forever without making a move. Further, if a rule with a guard $(CT = x)$ is enabled in a state a, it fires in a.
3. *[Train Movement]* $(TrackStatus(t) = coming)$ holds until $(TrackStatus(t) = in\_crossing)$, which holds until $(TrackStatus(t) = empty)$. That is, once a train arrives on a track, it eventually reaches the crossing and leaves.
4. *[Train Detection]* $(TrackStatus(t) = empty)$ holds until $(Deadline(t) = \infty)$. This clarifies our requirement that only one train be present on a track at a time. If a train leaves the intersection from track t, the controller must detect its departure and reset $Deadline(t)$ before another train arrives.
5. *[Deadline Timing]* $(CT < Deadline(t))$ holds until $(CT = Deadline(t))$. That is, if the controller sets a future deadline time in a particular state, there will be a later state in which that deadline is reached.
6. *[Train Timing]*

$$MinTime(Deadline(t) < \infty, TrackStatus(t) = in\_crossing) \geq d_{min}$$
$$MaxTime(Deadline(t) < \infty, TrackStatus(t) = in\_crossing) \leq d_{max}$$

   This clarifies the requirements on d_{min} and d_{max}. Suppose the controller detects an arrival on track t and sets $Deadline(t)$ in $(a-1, a)$. Let $b > a$ be the first state such that the train has arrived in the crossing. Then $(CT_b - CT_a)$ is at least d_{min} and at most d_{max}.
7. *[Gate Timing]*

$$MaxTime(SafeToOpen, (\neg SafeToOpen \vee GateStatus = up)) \leq d_{up}$$
$$MaxTime(CT \geq Deadline(t), (SafeToOpen \vee GateStatus = down)) \leq d_{down}$$

   This clarifies the requirements on d_{down} and d_{up}. (The reader may expect simpler conditions, e.g. $MaxTime(SafeToOpen, GateStatus = up) \leq d_{up}$. However, this is not necessarily true, because the act of opening the gate might be aborted.) The condition asserts that the period of time between the controller detecting that a change in the gate is desired and the change taking effect, if uninterrupted, is at most d_{down} or d_{up}, depending on the change being made.

5 Proving Safety and Utility

All proofs are performed over regular runs.

5.1 Preliminaries

Lemma 1 $(Dir = down)$ *holds until* $((GateStatus = down) \vee (Dir = up))$.

Proof: Suppose $(Dir = down)$ becomes true in $(a, a+1)$. Rule *SignalUp* might execute $(Dir := up)$ in some $b > a$, which would satisfy the lemma. Suppose this does not happen; since *SignalUp* is the only rule which can execute $(Dir := up)$, $(Dir = down)$ for every $b > a$.

What is the value of $GateStatus$ in state $a+1$?

- If $GateStatus = down$, we're done.
- If $GateStatus = going\_down$, rule *GateDown* is enabled, and will remain so until executing ($GateStatus := down$), which must happen (by our liveness assertion). This yields the desired condition.
- Otherwise, rule *GateDown* is enabled. It will remain enabled until executing ($GateStatus := going\_down$). By the previous argument, this eventually leads to a state satisfying the desired condition. □.

Lemma 2 *($Dir = up$) holds until ($(GateStatus = up) \vee (Dir = down)$).*

Proof: Parallel to that of the last lemma. □

Lemma 3 *($SafeToOpen$) holds until ($\neg SafeToOpen \vee GateStatus = up$).*

Proof: Suppose ($SafeToOpen$), *i.e.* $\forall t \in \mathsf{Tracks}(TrackStatus(t) = empty \vee CT + d_{up} < Deadline(t))$, becomes true in $(a, a+1)$ and holds for every $b > a$. We need to show that there exists some $c > a$ such that ($GateStatus = up$) holds in c.

What occurred between a and $a+1$ to make ($SafeToOpen$) true? Since CT monotonically increases from state to state, we must have had either $Deadline(t)$ set to a sufficiently large value or $TrackStatus(t)$ set to *empty* for some $t_0, \ldots, t_k$.

$Deadline(t)$ can be changed in two ways. Rule *SignalDown* can change $Deadline(t)$ from ∞ to a finite value for a given t; this is obviously a decrease in $Deadline(t)$. Rule *SignalUp* can change $Deadline(t)$ from a finite value to ∞. The guard of *SignalUp* ensures that ($TrackStatus(t) = empty$) when this change is made; consequently, this change in $Deadline(t)$ could not have contributed to $SafeToOpen$ becoming true.

Thus, ($SafeToOpen$) became true in $(a, a+1)$ through ($TrackStatus(t_i) = empty$) becoming true for some tracks t_i. By our train detection assertion, there exists a state $b \geq a+1$ such that ($Deadline(t_i) = \infty$) becomes true in $(b, b+1)$. This is the result of rule *SignalUp* firing in state b. If ($Dir \neq up$) in state b, *SignalUp* will also execute ($Dir := up$). Consequently, ($Dir = up$) is true in $(b+1)$.

Lemma 2 implies that there exists $c > b$ such that ($(GateStatus = up) \vee (Dir = down)$) becomes true in $(c, c+1)$. Since ($SafeToOpen$) is true in $[b, c]$, rule *SignalDown* cannot execute ($Dir := down$) within that interval, as its guard contradicts ($SafeToOpen$). Thus ($GateStatus = up$) becomes true in $(c, c+1)$. □.

Lemma 4 *($CT \geq Deadline(t)$) holds until ($Deadline(t) = \infty \vee GateStatus = down$).*

Proof: We are specifically interested in proving that a state in which ($CT = Deadline(t)$) is followed closely by a state in which ($Deadline = \infty \vee GateStatus = down$). But the nature of the "holds until" relation requires us to use this form.

Suppose ($CT \geq Deadline(t)$) becomes true in $(a-1, a)$. By our deadline timing assertion, ($CT = Deadline(t)$) becomes true in $(a-1, a)$. By our liveness assertion, rule *SignalDown* executes ($Dir := down$), and ($Dir = down$) becomes true in $(a, a+1)$.

By Lemma 1, there exists $b > a$ such that ($(GateStatus = down) \vee (Dir = up)$) becomes true in $(b, b+1)$. If ($GateStatus = down$) becomes true, we're done. Otherwise, ($Dir = up$) becomes true in $(b, b+1)$, by the execution of rule *SignalUp*. Rule *SignalUp* also executes ($Deadline(t) := \infty$) at the same time, yielding the desired condition. □.

Lemma 5 *($Deadline(t) < \infty$) holds until ($TrackStatus(t) = in\_crossing$).*

Proof: Suppose ($Deadline(t) < \infty$) becomes true in $(a, a+1)$. This is caused by rule *SignalDown* executing, which requires ($TrackStatus(t) = coming$), in state a. $Deadline(t)$ can only be set to ∞ by rule *SignalUp*, which requires ($TrackStatus(t) = empty$). Before ($TrackStatus(t) = empty$) can become true ($TrackStatus(t) = in\_crossing$) must become true. □

Lemma 6 *In any state in which ($TrackStatus(t) = in\_crossing$), ($Deadline(t) < \infty$).*

Proof: From Lemma 5, we have that $(Deadline(t) < \infty)$ at the moment that $(TrackStatus(t))$ changes from *coming* to *in_crossing*. Only rule *SignalUp* can set $Deadline(t)$ to ∞; it must wait until some state b in which $TrackStatus(t) = empty$. Consequently, $(Deadline(t))$ will remain unaltered in all intermediate states in which $(TrackStatus(t) = in\_crossing)$. □

Lemma 7 *If* $(Deadline(t) < \infty \wedge TrackStatus(t) = coming)$ *is true for some track* t *in state* a*, then*

$$b = Succ_{TrackStatus(t)=coming, TrackStatus(t)=in\_crossing}(a)$$

is defined and $(Deadline(t) + d_{down} \leq CT_b \leq Deadline(t) + d_{down} + d_{max} - d_{min})$.

Proof: Suppose $(Deadline(t) < \infty \wedge TrackStatus(t) = coming)$ is true in some state a'. Let $a = Pred_{Deadline(t)<\infty, Deadline(t)=\infty}(a')$; that is, $(Deadline(t) < \infty)$ becomes true in $(a, a+1)$. By our train timing assertion and Lemma 5, the desired state $b > a' > a$ exists and $d_{min} \leq CT_b - CT_a \leq d_{max}$. Rule *SignalDown* sets $(Deadline(t))$ to $(CT_a + d_{min} - d_{down})$ in $(a, a+1)$; substitution for CT_a yields the desired result. □.

5.2 The Main Results

Theorem 1 $t \in \lambda_i \Rightarrow g(t) = 0$ *(The gate is down during all occupancy intervals.)*

First, we restate the claim in our terminology. Fix an i. Let x be the state such that $(\exists t \in \mathsf{Tracks})$ $(TrackStatus(t) = in\_crossing)$ becomes *true* for the ith time in $(x-1, x)$, and let $y > x$ be the earliest state such that $(\exists t \in \mathsf{Tracks})(TrackStatus(t) = in\_crossing)$ is *false*. Thus $CT_x = \tau_i$ and $CT_y = \nu_i$. Then for every state $a \in [x, y]$, $(GateStatus = down)$.

We first show that our translation of the desired property is accurate. Let $t \in \lambda_i$. There exists a state $a \in [x, y]$ such that either $CT_a = t$ or $CT_a < t < CT_{a+1}$. In the former case, it suffices to prove $(GateStatus = down)$ at a. In the latter case, it suffices to prove $(GateStatus = down)$ at a and $a+1$. Suppose by contradiction that the above holds but $g(t) \neq 0$. Since $g(CT_a) = 0$, rule *GateUp* must fire at a to execute $(GateStatus := going\_up)$. But then $(GateStatus = going_up)$ would hold at $a+1$, the first state after a. This contradicts our assertion that $(GateStatus = down)$ holds at $a+1$.

Proof of the Theorem. We have that $(TrackStatus(t) = in\_crossing)$ becomes true in $(x-1, x)$. By Lemma 6, $e = (Deadline(t) < \infty)$ is true in state x. Let $b = Pred_{e, \neg e}(x)$; thus, $(Deadline(t) < \infty)$ is true in $(b, x]$. Since $(Deadline(t) = \infty)$ in the initial state, b is well defined. By our train timing assertion, $(CT_x - CT_b) > d_{min}$.

In state b, rule *SignalDown* sets $Deadline(t)$ to $CT_b + d_{min} - d_{down}$; since $d_{min} > d_{down}$, we have $Deadline(t) > CT_b$. Thus, by our deadline timing assertion, there exists a $c > b$ such that $CT_c = Deadline(t)$. By our liveness assersion, *SignalDown* fires in c, performing $(Dir := down)$. Thus, $(Dir = down)$ in $c+1$ and $(CT_x - CT_c) > d_{down}$.

By Lemma 4, $(CT \geq Deadline(t))$ (which becomes true in $(c-1, c)$) holds until $(Deadline(t) = \infty \vee GateStatus = down)$. By our gate time assertion, the earliest state a in which $(Deadline(t) = \infty \vee GateStatus = down)$ holds satisfies $(CT_a - CT_c) \leq d_{down}$. Consequently, $a < x$.

$(Deadline(t) = \infty)$ can only become true if rule *SignalUp* fires, which requires $(TrackStatus(t) = empty)$. Since $(TrackStatus(t) = coming)$ over $[c, a)$ (and therefore over $[c, x]$), this cannot occur. Thus, rule *GateDown* must execute $(GateStatus := down)$ in $(a-1, a)$.

Within $(a, y]$, *GateStatus* can only be changed if rule $(Dir = up)$ becomes true somewhere in that interval. This can happen only if rule *SignalUp* executes $(Dir := up)$, which requires $(TrackStatus(t) = empty)$. This condition will not be true until after state a, when $(TrackStatus(t) = in\_crossing)$ becomes true. Consequently, $(GateStatus = down)$ in state a and every succeeding state until $(TrackStatus(t))$ becomes *empty*. □

Two constants Ξ_1, Ξ_2 are used in the original problem description. In our terms, $\Xi_1 = d_{max} - d_{min} + d_{down}$ and $\Xi_2 = d_{up}$. The appropriateness of these definitions should become clear in the proof of the following theorem.

Theorem 2 $t \notin \cup_i[\tau_i - \xi_1, \nu_i + \xi_2] \Rightarrow g(t) = 90$ *(The gate is up when no train is in the crossing.)*

The statement above is equivalent to the following three statements:

1. Let t_0 be the value of CT in the initial state of the run. Then $(t_0 \leq t < \tau_1)$ implies $g(t) = 90$.
2. For any i, $(\nu_i + \Xi_2 < t < tau_{i+1} - \Xi_1)$ implies $g(t) = 90$.
3. If there is a final occupancy interval λ_k, $t > \nu_k + \Xi_2$ implies $g(t) = 90$.

We consider only the most interesting case, case 2. The proof for cases 1 and 3 is similar.

Again, we translate the statement to be proven into our terminology. Fix an i. Let a, b be states such that $CT_a = \nu_i$ and $CT_b = \tau_{i+1}$. Thus $(\forall t \in \mathsf{Tracks})(TrackStatus(t) \neq in\_crossing))$ becomes true in $(a-1, a)$ and false in $(b-1, b)$. Then at every state $y \in [a, b)$ $(CT_a + \xi_2 < CT_y < CT_b - \xi_1)$ implies $(GateStatus = up)$.

Again, we must show that this is a faithful translation of the desired property; the proof is similar to that for Theorem 1. Fix a t in the range of interest. If there exists an x such that $CT_x = t$, it suffices to prove the property for x. Otherwise, there exists an x such that $CT_x < t < CT_{x+1}$ and it suffices to prove the property for x and $x + 1$, since if $g(CT_x) = 90$ but $g(t) \neq 90$, then $g(CT_{x+1}) \neq 90$, a contradiction.

Proof of the Theorem. We assume that $(CT_b - CT_a > \xi_1 + \xi_2)$; otherwise, there is nothing to prove.

We claim that $(SafeToOpen)$ becomes true in $(a-1, a)$. By assertion above, for every train t, either $(TrackStatus(t) = empty)$ (which is compatible with $SafeToOpen$) or $(TrackStatus(t) = coming)$ is true in a. If $(TrackStatus(t) = coming)$ in a, either $(Deadline(t) = \infty)$ (which is compatible with $SafeToOpen$) or $(Deadline(t) < \infty)$. If $(Deadline(t) < \infty)$, Lemma 7 implies $(CT_b \leq Deadline(t) + d_{down} + d_{max} - d_{min})$. Since $(CT_b - CT_a > \xi_1 + \xi_2)$, we have $(CT_a < Deadline(t) - d_{up})$, satisfying $(SafeToOpen)$. So in any event, $SafeToOpen$ becomes true in $(a-1, a)$.

By Lemma 3, there exists $c > a$ such that $(\neg SafeToOpen \vee GateStatus = up)$ becomes true in $(c-1, c)$, and by gate timing, $CT_{c-1} - CT_{a-1} \leq d_{up}$. $(\neg SafeToOpen)$ could only occur if $(CT_{c-1} = Deadline(t))$; yielding $(CT_{c-1} - CT_{a-1}) > d_{up}$, a contradiction. So $(GateStatus = up)$ becomes true in $(c-1, c)$ and $(CT_{c-1} < CT_a + d_{up})$.

$(GateStatus)$ cannot be altered again until $(Dir = down)$. The earliest this can occur is when *SignalDown* fires before the arrival of the train in state b; by Lemma 7, the state c' in which this occurs is such that $CT_{c'} \geq CT_b - (d_{max} - d_{min} + d_{down})$. Thus, $(GateStatus = up)$ must remain true in $[c, c']$. c and c' satisfy the desired constraints. □

References

[1] E. Börger. "Annotated Bibliography on Evolving Algebras." In E. Börger, editor, *Specification and Validation Methods*, Oxford University Press, 1995.

[2] E. Börger, Y.Gurevich, and D.Rosenzweig. The bakery algorithm: Yet another specification and verification. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[3] Y. Gurevich, "Evolving Algebras: An Introductory Tutorial", Bulletin of European Assocation for Theoretical Computer Science, February 1991. Slightly revised and reprinted in "Current Trends in Theoretical Computer Science", Eds. G. Rozenberg and A. Salomaa, World Scientific, 1993, 266–292.)

[4] Y. Gurevich, "Evolving Algebras 1993: Lipari Guide", in *Specification and Validation Methods*, ed. E. Börger, Oxford University Press, 1995.

[5] Y. Gurevich and R. Mani. Group Membership Protocol: Specification and Verification. In *Specification and Validation Methods*, Ed. E. Börger, Oxford University Press, 1995.

[6] C. Heitmeyer and N. Lynch. The Generalized Railroad Crossing: A Case Study in Formal Verification of Real-Time Systems. Technical Report 7619, Naval Research Laboratory, Washington DC, 1994.

[7] J. Huggins, "Kermit: Specification and Verification", in *Specification and Validation Methods*, ed. E. Börger, Oxford University Press, 1995.