Modeling and Verification of a Simple Real-Time Railroad Gate Controller

William D. Young

Technical Report 93 September, 1994

Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas 78703-4776

TEL: +1 512 322 9951
FAX: +1 512 322 0656

EMAIL: young@cli.com

This research was supported in part by ONR Contract N00014-91-C-0130 and by by the Defense Advanced Research Projects Agency, ARPA Order 7406. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Office of Naval Research, the Defense Advanced Research Projects Agency, or the U.S. Government.

Abstract

We address the formal specification and verification of a simple train crossing gate system using the Nqthm logic and automated proof system of Boyer and Moore[6]. This problem has been suggested[10] as a benchmark for evaluating the performance of specification tools and automated reasoning systems in the area of safety-critical systems. The system specification is presented and the proof of *safety* and *utility* properties is outlined. The performance of Nqthm on this problem is evaluated. The complete specification is provided in an appendix.

1 Introduction

Safety-critical systems with real-time aspects often involve complex and time-sensitive interactions between the environment and the system. Consequently, informal reasoning may be inadequate for providing assurance of the correct functioning of such systems (see e.g. [16]). The use of mechanically supported formal reasoning can often provide a much higher level of assurance than informal reasoning alone.

Given the diversity of automated reasoning systems, it is reasonable to ask which is the "right" system for a given problem. Often it is infeasible to experiment with various automated reasoning systems in the context of a realistic real-time application due to the considerable effort required to model a complex system in even one formal framework. There are seldom sufficient resources to model and compare the performances of several competing formalisms. An alternative approach is to compare the performance of various formal reasoning systems on *benchmark* problems of manageable complexity. Hopefully, the results of such comparisons can be extrapolated to gain insights into the applicability of these systems on problems of interest.

A potential benchmark in the area of safety-critical systems is a simple problem suggested by Leveson and Stolzy[15] involving modeling of the controller for a gate at a railroad crossing. This problem has been proposed[10] as a benchmark for investigating the expressiveness and proof power of various formal systems. Specifications of the gate controller have been performed in the EVES prover of Odyssey Research[1], with SRI's PVS prover[18], and with the Modechart graphical language[11]. Heitmeyer, *et al.*[10] report that solutions using CSP and Modecharts are underway as well as a solution using the FDR tool[17]. A related but more complex problem involving the control of multiple trains has been specified by Wood[21] using the Z specification language[20]. A version of the multiple train problems has also been specified using Nqthm[19].

We accepted the challenge of formalizing the Leveson/Stolzy benchmark with the Boyer-Moore logic Nqthm[4, 6, 13] and proving the relevant safety and utility properties with the Nqthm theorem prover. Because of limited time, the problem we solved is slightly more restricted than that proposed as a general benchmark by Heitmeyer. In particular, we do not consider multiple trains. However, we believe that our solution could be readily extended to handle this case and we hope to do so in the future.

In the following sections, we briefly describe the Nqthm logic and prover, outline the railroad-gate problem, and present our solution to it. The appendix contains the complete text of the railroad gate specification in an infix form derived from our Nqthm script.

2 Nqthm

The logic in which we performed our modeling work is the Nqthm (or Boyer-Moore) computational logic[6]. Nqthm is well known for the extensive body of verification work done with it. We briefly summarize it here.

The Nqthm logic is a first order, quantifier-free[1] logic resembling pure Lisp. The rules of inference are: propositional calculus with equality, together with instantiation and mathematical induction up to ϵ_0. Two extension principles, recursive definition and conservative axiomatization of functions, are also provided. These permit extending the logic with new functions symbols in a conservative and sound manner.

The mechanization of the logic is a system of Common Lisp programs allowing the user to define functions in the logic, execute them on concrete data, and state and prove theorems about such functions. The Nqthm theorem prover is a collection of hundreds of heuristics to orchestrate the application of rewrite rules, decision procedures, mathematical induction, and other proof techniques. An interactive proof checker for Nqthm, called "Pc-Nqthm," has also been developed [12, 14].

Important to Nqthm's success has been the fact that when a new user-supplied theorem is proved, rules are derived from it and stored in its data base; these rules change the way the system behaves. By stating an appropriate collection of lemmas the user can effectively program Nqthm to prove theorems in a given domain. A well chosen sequence of lemmas can lead Nqthm to the proofs of very deep theorems. Target theorems can often be changed and "re-proved" automatically because proof scripts tend to describe powerful and general-purpose rules for manipulating the concepts rather than particular proofs. That is, the user programs Nqthm to deal with the concepts and expects Nqthm to fill in the gaps between system-specific lemmas describing the proof at a high level. This makes it easy to "maintain" an evolving system of definitions and theorems if the system was initially verified with Nqthm. That, in turn, has allowed Nqthm to accumulate a vast quantity of benchmark theorems.

For example, some theorems from traditional mathematics that have been mechanically checked using the system include: Gauss' law of quadratic reciprocity; the Church-Rosser theorem for lambda calculus; the infinite Ramsey theorem for the exponent 2 case; and Gödel's incompleteness theorem. Outside the range of traditional mathematics, the theorem prover has been used to check: the recursive unsolvability of the halting problem for Pure Lisp; the proof of invertibility of the RSA public key encryption algorithm; the correctness of numerous metatheoretic simplifiers for the logic; the correctness of several control algorithms; and the correctness of an implementation of an algorithm for achieving agreement among concurrently executing processes in the presence of faults. In the area of "systems verification" the prover has been used in the verification of the "CLI short stack" comprising a chain of abstract machines starting with a microprocessor (the FM8502) described at the gate-level and ending with an operational semantics for a simple high-level programming language (Micro-Gypsy). Numerous other proofs have been done in the area of hardware and software verification.

[1] A treatment of quantified formulas has been added by Matt Kaufmann[13], and used in our work.

The latest version of Nqthm was released in 1994 along with a companion version of Pc-Nqthm. Nqthm is documented in two books [5, 6]. The release includes 1.3 megabytes of updated documentation consisting of new versions of the five most important chapters in [6]. In addition, the release includes more than 15 megabytes of example input for Nqthm and Pc-Nqthm, including many of the important benchmarks listed above. The system is available free of charge by anonymous ftp.

Boyer, Kaufmann, and Moore are currently developing the successor to Nqthm, called ACL2. ACL2 is an extended, reimplemented version of Nqthm that supports the applicative subset of Common Lisp as its logic. By formalizing a logic around applicative Common Lisp they take advantage of the exceptionally good optimizing compilers for Common Lisp to get, in many cases, execution speeds comparable to C. Two guiding tenets of the ACL2 project are to conform to all compliant Common Lisp implementations and to add nothing to the logic that violates the understanding that the user's input can be submitted directly to a Common Lisp compiler and then executed in an environment where suitable Acl2-specific macros and functions are defined. We anticipate redoing our railroad gate problem and many of the other Nqthm benchmarks in ACL2.

Though our specification and proof work was done entirely in Nqthm, in this paper we have chosen not to use the Lisp-like Nqthm syntax, but an infix transliteration. For example, to define a function to sum a list of numbers, the Nqthm user would type:

```
(defn add-list (l)
  (if (nlistp l)
      0
      (plus (car l) (add-list (cdr l))))).
```

We will use the following syntax:

Definition 1 (Add-list) The function $\mathit{add\text{-}list} : \mathcal{N}^* \rightarrow \mathcal{N}$ is defined as follows:

$$\mathit{add\text{-}list}(\mathit{lst}) \equiv \begin{cases} 0 & \mathit{if}\ \mathit{train\text{-}tr} = \langle\rangle; \\ x + \mathit{add\text{-}list}(\mathit{lst}') & \mathit{if}\ \mathit{lst} = x \circ \mathit{lst}'. \end{cases}$$

Since Nqthm is an untyped logic, the "signatures" of functions should be read as merely suggestive. The primitive types appearing in our signatures are $\mathcal{L}$ for literal atoms, $\mathcal{N}$ for naturals, and $\mathcal{B}$ for Booleans. All Nqthm defined functions are total but may operate strangely on objects outside their intended domain. We use the syntax $x \circ y$ for the Nqthm `(cons x y)`, i.e., adding element x to the front of list y.

3 The Railroad Gate Problem

The problem we set out to solve with Nqthm may be stated very simply. Design and verify a system to operate safely a gate at a railroad crossing. The system should have the properties that:

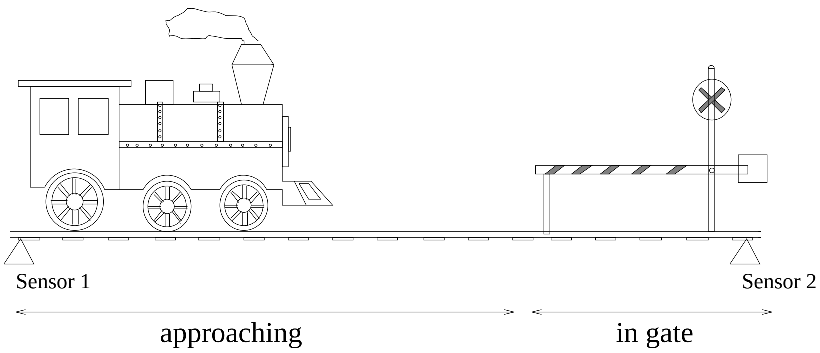

Figure 1: Train and Gate System

- the gate is down whenever the train is in the crossing, and
- the gate is up as often as possible.

These are called, respectively, the *safety property* and *utility property*[10]. Whether this is genuinely a "real-time" problem is a matter of debate; trains and crossing gates have very coarse timing requirements compared with the microsecond response time requirements of some application areas. However, we feel that this is a matter of degree and not of kind. Our approach, if not our exact solution, is adaptable to other domains.

Figure 1 illustrates the system and its environment. Our task in modeling this system is to devise a control algorithm for the gate that preserves the safety and utility properties. To do so we must make certain assumptions about the responses of the gate to the controller and about the general behavior of trains on this line. In particular, we assume that:

- trains passing the gate have a known maximum speed;
- the gate has known response parameters, including maximum and minimum times for opening and closing;
- sensors are available that can reliably determine when a train is within a given region of track;
- arrivals of trains at the gate are spaced sufficiently widely that we need only consider one train at a time.

The first assumption is required for any safe gate, since an arbitrarily fast train could always race into the crossing before the gate could close. Placement of sensors to signal an approaching train assumes that we know the response parameters of the gate. The requirement that we treat only one train at a time is a significant simplification, but one that we believe would not be difficult to relax.

Given these assumptions, we have devised a simple gate control system and proved that the resulting system satisfies the safety and utility requirements.

4 Our Formalization

Our model of system behavior is essentially a discrete-time simulation of the system. At each "tick" of the global clock, the controller determines the position of the train with respect to the gate and generates a control pulse for the actuator that raises and lowers the gate. The granularity of time is not specified, but can be as fine or as coarse as desired. The behavior of the controller, gate, and train are each specified independently and then the system specified in terms of their interactions. Finally, the system behavior is related to the behavior of the various subsystems by the correctness theorems.

4.1 The Sensors and Controller

Conceptually, the train can be in any of three "regions": approaching the gate, in the gate, or elsewhere. With reference to Figure 1, *approaching* means that the train has triggered Sensor_1; *elsewhere* means that the train has not yet triggered Sensor_1 or has already passed Sensor_2 indicating that the train has left the crossing. Being *in the gate* is a specification construct and is not really relevant to the controller behavior; hence no sensor is needed directly in front of the crossing.

According to its current reading of the train position, the controller generates a control pulse, which affects the position of the gate at the *next* clock tick. The control algorithm is trivial: if the train is *elsewhere* generate command `open`; otherwise, generate command `close`.

4.2 The Gate

As we noted in Section 3, we require that the gate have known operational parameters. In particular, we assume that the gate responds reliably to commands from the controller and that there are known minimum and maximum times required for the gate to open fully or to close fully.[2] We record the state of the gate at any time in a "ghost" variable with components $\langle$*position, duration*$\rangle$, recording the current position—`open`, `closed`, `going-up`, or `going-down`—and a duration indicating how long the gate will continue in that state. The duration is meaningful only when the gate is going up or going down.

The state of the gate is re-computed at each tick based on the command received from the controller (in response to the sensed position of the train in the previous tick). When the gate receives a command the new position is computed as a function of the previous position and the command. For example, given a command to `close` and a current gate state $\langle$*pos, dur*$\rangle$, the new gate position is computed as follows:

$$\begin{cases} \langle \texttt{going-down},\ k-1 \rangle & \textit{if } pos = \texttt{open}; \\ \langle \texttt{closed},\ dur \rangle & \textit{if } pos = \texttt{closed}; \\ \langle \texttt{going-down},\ k-1 \rangle & \textit{if } pos = \texttt{going-up}; \\ \langle \texttt{closed},\ 0 \rangle & \textit{if } pos = \texttt{going-down} \wedge\ dur = 0; \\ \langle \texttt{going-down},\ dur-1 \rangle & \textit{if } pos = \texttt{going-down} \wedge\ dur \neq 0. \end{cases}$$

[2] For technical reasons, we insist that the gate closing and gate opening maximum times be non-zero.

where k is some unspecified number in the range [*gate-closing-min-time* ... *gate-closing-max-time*]. A similar computation applies to the next state of the gate when commanded to `open`.

4.3 The Train

The train is that component of the system that is not under our control. However, to reason about the behavior of the controller and of the gate, we must make certain reasonable assumptions about the behavior of the train and model its behavior under those assumptions.

We model the behavior of the train with a *trace* of the "position" of the train at each tick of the simulation clock. Here "position" refers to our very coarse notion of whether the train is approaching, in the gate, or elsewhere. Therefore, a legal trace of the train behavior over time is a sequence of positions—`in-gate`, `approaching`, and `elsewhere`—subject to the following two constraints.

- The train may remain in a position for arbitrarily many ticks, but may only move from `elsewhere` to `approaching`, from `approaching` to `in-gate`, and from `in-gate` to `elsewhere`.
- Moreover, there is a system-dependent parameter *approaching-min-time* that specifies the minimum number of ticks the train must remain in the `approaching` region before entering the gate.

For example, if *approaching-min-time* were 2, the following would be a legal partial train trace:

$$\langle \texttt{elsewhere}, \texttt{approaching}, \texttt{approaching}, \texttt{in-gate}, \texttt{elsewhere} \rangle$$

But this would not

$$\langle \texttt{elsewhere}, \texttt{approaching}, \texttt{in-gate}, \texttt{approaching} \rangle$$

both because the approach is too short and because the train illegally jumps directly from `in-gate` to `approaching`.

These constraints on train traces are reasonable consequences of the assumptions we have made about our system. The progression from "elsewhere" to `approaching` to `in-gate` is merely the natural semantics of train behavior in our system, given our assumptions that we are considering only one train at a time and that the sensors reliably detect the progression of the train. The requirement that the approach be of a given minimum duration follows from our requirement that there is a known maximum speed of trains on the line. We always can adjust the placement of Sensor_1 to ensure that we have enough advance warning of the imminent arrival of the train. Without such an assumption, we could never be assured of having enough time to lower the gate in time to satisfy our safety requirement.

Given that we can specify a minimum approaching interval and can adjust the placement of Sensor_1 to implement it, we enforce the following requirement:

$$\textit{approaching-min-time} > \textit{gate-closing-max-time}.$$

That is, we position Sensor_1 in such a way that it will signal the imminent arrival of the train in time always to allow us to lower the gate.

Notice that the farther out we place Sensor_1, the "safer" the system but at a cost to "utility." Placing Sensor_1 in front of the gate at exactly the distance the fastest train could cover in *gate-closing-max-time* + 1 time units is adequate to ensure safety in our system and also guarantee that the gate is open "as much as possible," as required by the utility property. However, we do not require this in our proof. Instead, we only consider utility after the train has left the gate.

5 Specifying System Behavior

Given any legal trace of the train's position over time and an initial gate position, we can totally simulate the behavior of the system. This simulation involves computing successive gate states in response to the advancing train position and the resulting control action. The output of this simulation is a trace of the gate behavior, where each successive gate state in the trace is a function of its current state and the controller's command (based upon the train position at the previous tick). Reasoning about this simulation is what allows us to prove that the system satisfies the desired safety and utility properties.

Our safety property is simple: the gate is down whenever the train is in the crossing. The formal statement is merely a relationship between any legal train trace and the corresponding trace of the gate. The desired relationship is captured in the following Boolean-valued recursive function:[3]

Definition 21 (Safety) The function $\mathit{safety} : \mathcal{L}^* \times \langle \mathcal{L}, \mathcal{N} \rangle^* \to \mathcal{B}$ is defined as follows:

$$\mathit{safety}(\mathit{train\text{-}tr}, \mathit{gate\text{-}tr}) \equiv \begin{cases} \mathit{true} & \mathit{if}\ \mathit{train\text{-}tr} = \langle\rangle; \\ \quad g = \texttt{closed} & \mathit{if}\ \mathit{train\text{-}tr} = \texttt{in-gate} \circ \mathit{train\text{-}tr}' \\ \wedge\ \mathit{safety}(\mathit{train\text{-}tr}', \mathit{gate\text{-}tr}') & \quad \wedge\ \mathit{gate\text{-}tr} = g \circ \mathit{gate\text{-}tr}'; \\ \mathit{safety}(\mathit{train\text{-}tr}', \mathit{gate\text{-}tr}') & \mathit{otherwise}. \end{cases}$$

Since safety can be violated by beginning the simulation in a perverse state (as with the train `in-gate` and the gate `open`), we must specify a safe initial state. We choose to consider that the train is initially *elsewhere* and the gate is *open*. The safety theorem demonstrates that for a legal train trace (as described in Section 4.3) starting in this initial state the gate will always be `closed` whenever the train is *in-gate*. This appropriate theorem is shown below:

Theorem 27 (Controller Safety)
For all $\mathit{train\text{-}tr}' \in \mathcal{L}^*$, $x, g \in \mathcal{L}$, $a \in \mathcal{N}$,

$$\begin{aligned} & \mathit{initial\text{-}statep}(x, g) \wedge \mathit{legal\text{-}train\text{-}trace}(\mathit{train\text{-}tr}, a) \\ \to\ & \mathit{safety}(\mathit{train\text{-}tr}, \mathit{gate\text{-}behavior}(\mathit{train\text{-}tr}, g, n)), \end{aligned}$$

where $\mathit{train\text{-}tr} = x \circ \mathit{train\text{-}tr}'$.

[3] Definitions and theorems are numbered as in the appendix.

The variable a in this theorem records how long the train has been in the approaching region before the beginning of the trace. This is useful is specifying recursively when a final segment of a trace is legal.

The utility property is somewhat more difficult to state. Recall that utility means that the gate is open *as much as possible.* This is interpreted by Heitmeyer, et al.[10] to mean that there are some δ_1 and δ_2 such that the gate is open whenever the train is more than δ_1 before the crossing or more than δ_2 after the crossing. Our gate controller begins lowering the gate as soon as a train approaches. Therefore, δ_1 is essentially the interval between Sensor_1 and the gate. As we noted in Section 4.3 above, this can be as short as *gate-closing-max-time*+ 1.

The gate is raised as soon as the train leaves the crossing and may take as much as *gate-opening-max-time* to fully open. Our version of utility asserts that the gate is open whenever the train is `elsewhere` and has been `elsewhere` for at least *gate-opening-max-time* units. The amount of time that the train has been `elsewhere` is recorded in a separate variable n. Again, the property is expressed recursively as a relation between the train and gate traces.

Definition 23 (Utility) The function $\mathit{utility} : \mathcal{L}^* \times \langle \mathcal{L}, \mathcal{N} \rangle^* \times \mathcal{N} \rightarrow \mathcal{B}$ is defined as follows:

$$\mathit{utility}(\mathit{train\text{-}tr},\ \mathit{gate\text{-}tr},\ n) \quad \equiv$$

$$\begin{cases} \mathit{true} & \text{if } \mathit{train\text{-}tr} = \langle\rangle; \\ \begin{array}{l} (\quad \mathit{gate\text{-}opening\text{-}max\text{-}time} < n \\ \ \rightarrow g = \texttt{open}) \\ \wedge\ \mathit{utility}(\mathit{train\text{-}tr}',\ \mathit{gate\text{-}tr}') \end{array} & \begin{array}{r} \text{if } \mathit{train\text{-}tr} = \texttt{elsewhere} \circ \mathit{train\text{-}tr}' \\ \wedge\ \mathit{gate\text{-}tr} = g \circ \mathit{gate\text{-}tr}'; \\ \ \end{array} \\ \mathit{utility}(\mathit{train\text{-}tr}',\ \mathit{gate\text{-}tr}') & \text{otherwise.} \end{cases}$$

The theorem that establishes that our system satisfies this version of utility is given below:

Theorem 28 (Controller Utility)
For all $\mathit{train\text{-}tr}' \in \mathcal{L}^*$, $x, g \in \mathcal{L}$, $a, e \in \mathcal{N}$,

$$\begin{array}{rl} & \mathit{initial\text{-}statep}(x,\ g)\ \wedge\ \mathit{legal\text{-}train\text{-}trace}(\mathit{train\text{-}tr},\ a) \\ \rightarrow & \mathit{utility}(\mathit{train\text{-}tr},\ \mathit{gate\text{-}behavior}(\mathit{train\text{-}tr},\ g,\ n),\ e), \end{array}$$

where $\mathit{train\text{-}tr} = x \circ \mathit{train\text{-}tr}'$.

Notice that more utility could be gained at the cost of complicating the control mechanism. We have taken a conservative approach by starting the gate down whenever a train reaches Sensor_1. This means that very slow trains may cause the gate to remain down for long periods before they reach the gate. However, modifying the controller to take advantage of the varying speeds of trains would require a much more complicated control algorithm.

Both our safety and utility theorems have been mechanically checked with the Nqthm prover.

6 An Alternative Version

We have specified our system model and safety and utility properties using recursive function definitions in a style natural to Nqthm. It has been argued that using recursive functions in this way makes Nqthm specifications somehow less intuitive or less readable than versions in some other specification languages, e.g. those making more extensive use of quantification. We would argue that this is simply false. Recursive functions are no less natural or intuitive than other formalisms and have a completely rigorous semantics. However, it is worth noting that we *can* give quantified versions of our safety and utility properties in Nqthm, using the first-order quantifiers provided by an extension to the prover by Matt Kaufmann[13].

For example, our definition of safety can be rephrased as follows:

Definition 29 (Safety2) The function $\mathit{safety2} : \mathcal{L}^* \times \langle \mathcal{L}, \mathcal{N} \rangle^* \rightarrow \mathcal{B}$ is defined as follows:

$$\mathit{safety2}(\mathit{train\text{-}tr},\ \mathit{gate\text{-}tr}) \quad \equiv$$

$$\forall i \in \mathcal{N}, i \leq \mathit{len}(\mathit{train\text{-}tr}) \ \wedge \ \mathit{train\text{-}tr}[i] = \texttt{in-gate} \rightarrow \mathit{gate\text{-}tr}[i] = \texttt{closed}.$$

We have proved, using the Nqthm prover, that this version of safety is implied by our earlier version. Thus, our train controller satisfies this "more intuitive" characterization. However, no additional assurance of correctness is gained by the exercise.

Similar remarks apply to our specification of the utility property. Definitions and lemmas involving these alternative versions are given in the appendix.

7 Proofs of the Properties

It is quite easy to see why safety and utility hold for our simple control system. The controller prompts the gate to begin lowering when it senses a train entering the approaching region at Sensor_1. Since the approach is guaranteed to be longer than the maximum time for the gate to close, the gate always will have closed fully by the time the train reaches the crossing. Similarly, the controller raises the gate when it senses that the train has left the crossing. The gate will therefore be open no later than the maximum opening interval[4] after that point.

However, neither of the theorems given in the previous section are provable directly. Proving each of them requires showing that a more complex invariant is maintained during the system simulation. This invariant (given in the following definition) establishes the necessary relationship among the current train position (p), current gate state ($\langle g, n \rangle$), distance (d) from the present position to the gate, time (e) already spent in the `elsewhere` state, and the opening and closing parameters of the gate. We prove that if this `good-statep` property is true initially and is maintained by the system behavior. The property is shown below:

[4] Actually, it may require one more tick to fully open since the control action trails the train position by one tick.

Definition 19 (Good-statep) The function *good-statep* : $\mathcal{L} \times \mathcal{L} \times \mathcal{N} \times \mathcal{N} \times \mathcal{N} \rightarrow \mathcal{B}$ is defined as follows:

$$
\begin{array}{ll}
\textit{good-statep}\,(p,\ g,\ n,\ d,\ e) \ \equiv & \\
& g = \texttt{going-down} \rightarrow n < \textit{gate-closing-max-time} \\
\wedge & g = \texttt{going-up} \rightarrow n < \textit{gate-opening-max-time} \\
\wedge & p = \texttt{in-gate} \rightarrow g = \texttt{closed} \\
\wedge & p = \texttt{approaching} \rightarrow \\
& \quad (g = \texttt{closed}) \ \vee \ (g = \texttt{going-down} \wedge \ n < d) \\
& \quad \vee \ (\textit{gate-closing-max-time} < d) \\
\wedge & p = \texttt{elsewhere} \rightarrow \\
& \quad (\textit{gate-opening-max-time} < e \rightarrow g = \texttt{open}) \\
& \quad \wedge \ [\quad (g = \texttt{closed} \ \wedge \ e = 0) \ \vee \ (g = \texttt{open}) \\
& \qquad \vee \ (g = \texttt{going-up} \ \wedge \ n \leq \textit{gate-opening-max-time} - e)\,]
\end{array}
$$

Proving that this property is maintained throughout the simulation is the key inductive component of the proof. We then prover that this in turn implies both the safety and utility properties. The requisite lemmas are given in the appendix.

8 Conclusions

Nqthm proved to be an excellent tool for modeling and reasoning about this problem. Our specification took a very "operational" form, a style quite familiar to seasoned Nqthm users (see e.g. [2]) and well-supported by the Nqthm prover. We would argue that an operational specification style such as we used is very natural for a control system such as this one where the behavior of the system is determined by a series of sensor "readings" taken at discrete time intervals and resulting in a series of actuator commands. We have previously used such an operational model to reason about some simple fuzzy controllers[8] and about a Byzantine-resilient control system[3].

The major proof effort involved identifying an appropriate system property and proving that our control algorithm maintained it invariantly. This is an inductive proof over the structure of legal train traces. Nqthm excels at just such tasks. Moreover, because of the number of possible combinations of train positions, gate states, and relations of the various parameters, a fairly large number of tedious subcases must be considered. Nqthm handles these expeditiously and entirely automatically.[5] We can easily imagine that this conceptually simple proof would be daunting for a more interactive proof system.

It is worth noting that we chose quite deliberately to implement an operational solution that we believe is in keeping with the "standard" uses of Nqthm. This contrasts with some other solutions such as the PVS solution[18] which involves embedding temporal operators in the PVS logic. Nqthm is certainly powerful enough to embed a rich specification language and this has been done several times, e.g. [7, 9]. However, we feel that while such an

[5] Some 740 cases were generated for the large inductive theorem; all were handled automatically in around 250 seconds on a Sun Sparc2-workstation.

approach may be a specification and proof *tour de force* it does not necessarily reflect well the strengths and weaknesses of the underlying logic and theorem prover. We believe that our solution reflects a natural style of using Nqthm. Still, faced with the prospect of attacking a large class of similar problems, we might feel the need to embed more specialized capabilities within the logic. One of the strengths of Nqthm is its proven versatility in the face of a wide variety of specification and proof problems.

Acknowledgements

We wish to acknowledge the assistance of our colleagues at Computational Logic, Inc., particularly Bob Boyer, Matt Kaufmann, J Moore, and Matt Wilding. This research was supported in part by ONR Contract N00014-91-C-0130 and by by the Defense Advanced Research Projects Agency, ARPA Order 7406. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Office of Naval Research, the Defense Advanced Research Projects Agency, or the U.S. Government.

Appendix
Train Gate System Specification

This appendix contains the complete specification and major lemmas relating to the train crossing gate problem. We use an infix notation derived (by hand) from the Nqthm script. Seasoned Nqthm users will note that we have taken some liberties. In particular, we have suggested the intended "signatures" of functions despite the fact that Nqthm is an untyped logic. In doing so we use the types: $\mathcal{L}$ for literal atoms, $\mathcal{N}$ for naturals, and $\mathcal{B}$ for Booleans. We use $x \circ y$ for the Nqthm `(cons x y)`, i.e., adding element x to the front of list y. *Embedded comments are in this italic font.* The "official" and replayable Nqthm event list is available by request from the author.

The following constrained function selects an arbitrary member of the designated range.

Conservative Axiom 1 (Choose Time Introduction)
We introduce a new function symbol *choose-time* : $\mathcal{N} \times \mathcal{N} \rightarrow \mathcal{N}$ via the following conservative axiom:

$$mn \leq mx \;\rightarrow\; mn \leq \textit{choose-time}\,(mn,\, mx) \leq mx.$$

The Gate

The gate is our controlled system; we assume that it responds reliably to a series of commands that are generated by the control algorithm. These commands are either `open` *or* `close`*. The simulation of the gate tells us its state at each moment of time. The following gives the response parameters of the gate.*

Conservative Axiom 2 (Gate Parameters Introduction)
We introduce the new constant function symbols *gate-closing-min-time* $\in \mathcal{N}$, *gate-closing-max-time* $\in \mathcal{N}$, *gate-opening-min-time* $\in \mathcal{N}$, and *gate-opening-max-time* $\in \mathcal{N}$ via the following conservative axiom:

$$\begin{array}{ll} & 0 < \textit{gate-closing-max-time} \\ \wedge & \textit{gate-closing-min-time} \leq \textit{gate-closing-max-time} \\ \wedge & 0 < \textit{gate-opening-max-time} \\ \wedge & \textit{gate-opening-min-time} \leq \textit{gate-opening-max-time} \end{array}$$

The gate state is a pair $\langle l, n \rangle$ *where* l *is one of* `open`*,* `closed`*,* `going-up`*, or* `going-down`*.* n *is the time that the gate can be expected to remain in that state. This is only meaningful when the gate is* `going-up` *and* `going-down`*.*

Definition 3 (Gate-positionp) The function *gate-positionp* : $\mathcal{L} \rightarrow \mathcal{B}$ is defined as follows:

$$\begin{array}{rcl} \textit{gate-positionp}\,(x) & \equiv & (\quad x = \texttt{open} \vee x = \texttt{closed} \\ & & \;\vee\; x = \texttt{going-up} \vee\; x = \texttt{going-down}). \end{array}$$

Definition 4 (Gate-current-state) The function *gate-current-state* : $\langle \mathcal{L}, \mathcal{N} \rangle \rightarrow \mathcal{L}$ is defined as follows:

$$\textit{gate-current-state}(\langle s, n \rangle) \quad \equiv \quad s.$$

Definition 5 (Gate-duration) The function *gate-duration* : $\langle \mathcal{L}, \mathcal{N} \rangle \rightarrow \mathcal{N}$ is defined as follows:

$$\textit{gate-duration}(\langle s, n \rangle) \quad \equiv \quad n.$$

Definition 6 (Compute-going-down-state) The function *compute-going-down-state*: $\langle \mathcal{L}, \mathcal{N} \rangle \rightarrow \langle \mathcal{L}, \mathcal{N} \rangle$ is defined as follows:

$$\textit{compute-going-down-state}(\langle l, n \rangle) \quad \equiv \quad \langle l', n' \rangle \text{ where}$$

$$\begin{cases} l' = \texttt{closed} \wedge \ n' = 0 & \textit{if } n = 0; \\ l' = \texttt{going-down} \wedge n' = n - 1 & \textit{otherwise.} \end{cases}$$

Definition 7 (Compute-going-up-state) The function *compute-going-up-state* : $\langle \mathcal{L}, \mathcal{N} \rangle \rightarrow \langle \mathcal{L}, \mathcal{N} \rangle$ is defined as follows:

$$\textit{compute-going-up-state}(\langle l, n \rangle) \quad \equiv \quad \langle l', n' \rangle \text{ where}$$

$$\begin{cases} l' = \texttt{open} \wedge \ n' = 0 & \textit{if } n = 0; \\ l' = \texttt{going-up} \wedge n' = n - 1 & \textit{otherwise.} \end{cases}$$

The following function gives the next state for any potential current state of the gate.

Definition 8 (Gate-next-state) The function *gate-next-state* : $\mathcal{L} \times \langle \mathcal{L}, \mathcal{N} \rangle \rightarrow \langle \mathcal{L}, \mathcal{N} \rangle$ is defined as follows:

$$\textit{gate-next-state}(c, \langle l, n \rangle) \quad = \quad \langle l', n' \rangle \text{ where}$$

if $c = \texttt{close}$,

$$\begin{cases} l' = \texttt{going-down} \wedge \ n' = k - 1 & \textit{if } l = \texttt{open}; \\ l' = l \ \wedge \ n' = n & \textit{if } l = \texttt{closed}; \\ l' = \texttt{going-down} \wedge \ n' = k - 1 & \textit{if } l = \texttt{going-up}; \\ \langle l', n' \rangle = \textit{compute-going-down-state}(\langle l, n \rangle) & \textit{if } l = \texttt{going-down}; \end{cases}$$

if $c = \texttt{open}$,

$$\begin{cases} l' = l \ \wedge \ n' = n & \textit{if } l = \texttt{open}; \\ l' = \textit{going-up} \ \wedge \ n' = k - 1 & \textit{if } l = \texttt{closed}; \\ \langle l', n' \rangle = \textit{compute-going-up-state}(\langle l, n \rangle) & \textit{if } l = \texttt{going-up}; \\ l' = \texttt{going-up} \ \wedge \ n' = k - 1 & \textit{if } l = \texttt{going-down}; \end{cases}$$

where $k = \textit{choose-time}(\textit{gate-closing-min-time}, \textit{gate-closing-max-time})$.

Modeling the Train

The train provides the input to our control system; the gate must act in response to the train. It can do so only if the behavior of the train meets certain reasonable criteria. Therefore, the specification of the train consists of a set of constraints on the possible traces of the train behavior as sensed by input sensors along the track. The train can be in one of 3 possible states: approaching the gate, in the crossing, or elsewhere.

Definition 9 (Train-positionp) The function *train-positionp* : $\mathcal{L} \rightarrow \mathcal{B}$ is defined as follows:

$$\textit{train-positionp}\,(x) \quad \equiv \quad (\quad x = \texttt{approaching} \lor \ x = \texttt{in-gate} \\ \lor\ x = \texttt{elsewhere})$$

A train can only make certain legal transitions from one position to another. For example, the train must pass through the `approaching` *region before entering the gate.*

Definition 10 (Legal-transitionp) The function *legal-transitionp* : $\mathcal{L} \times \mathcal{L} \rightarrow \mathcal{B}$ is defined as follows:

$$\textit{legal-transitionp}\,(x,\, y) \quad \equiv \\ \begin{cases} y = \texttt{elsewhere} \lor \ y = \texttt{approaching} & \textit{if } x = \texttt{elsewhere}; \\ y = \texttt{approaching} \lor \ y = \texttt{in-gate} & \textit{if } x = \texttt{approaching}; \\ y = \texttt{in-gate} \lor \ y = \texttt{elsewhere} & \textit{if } x = \texttt{in-gate}. \end{cases}$$

The following function partially defines what it means for a train trace to be legal. Notice that this definition assures that we cannot skip a region and go directly, say, from `elsewhere` *to* `in-gate`.

Definition 11 (Train-tracep) The function *train-tracep* : $\mathcal{L}^* \rightarrow \mathcal{B}$ is defined as follows:

$$\textit{train-tracep}\,(tr) \quad \equiv \\ \begin{cases} \textit{true} & \textit{if } tr = \langle\rangle; \\ \textit{train-positionp}\,(x) & \textit{if } tr = \langle x \rangle; \\ \quad \textit{train-positionp}\,(x) \ \land \ \textit{train-positionp}\,(y) & \textit{if } tr = x \circ y \circ tr'. \\ \land\ \textit{legal-transitionp}\,(x,\, y) \ \land \ \textit{train-tracep}(y \circ tr') & \end{cases}$$

We introduce a new constant that characterizes the amount of time the train must spend in the `approaching` *region before it can enter the gate. We enforce this by placing $Sensor_1$ far enough in front of the gate to assure time for the gate to close.*

Conservative Axiom 12 (Train Constraint Introduction)
We introduce the new constant function symbol *approaching-min-time* $\in \mathcal{N}$ via the following conservative axiom:

$$\textit{gate-closing-max-time} < \textit{approaching-min-time}.$$

Definition 13 (Seq-long-enough) The function $\mathit{seq\text{-}long\text{-}enough} : \mathcal{L} \times \mathcal{L}^* \times \mathcal{N} \times \mathcal{N} \to \mathcal{B}$ is defined as follows:

$$\mathit{seq\text{-}long\text{-}enough}(l,\, tr,\, n,\, mn) \;\equiv$$
$$\begin{cases} \mathit{true} & \mathit{if}\ n = 0 \wedge tr = \langle\rangle; \\ \mathit{seq\text{-}long\text{-}enough}(x,\, rst,\, 1,\, mn) & \mathit{if}\ n = 0 \wedge tr = x \circ rst; \\ \mathit{seq\text{-}long\text{-}enough}(x,\, rst,\, 0,\, mn) & \mathit{if}\ n = 0 \wedge tr = y \circ rst \\ & \wedge\ x \neq y; \\ mn \leq n & \mathit{if}\ n \neq 0 \wedge tr = \langle\rangle; \\ \mathit{seq\text{-}long\text{-}enough}(x,\, rst,\, n+1,\, mn) & \mathit{if}\ n \neq 0 \wedge tr = x \circ rst; \\ \quad mn \leq n & \mathit{otherwise}. \\ \wedge\ \mathit{seq\text{-}long\text{-}enough}(x,\, rst,\, 0,\, mn) & \end{cases}$$

This predicate checks a train trace to see that each time spent in the `approaching` *region is at least* approaching-min-time *units, where we assume we have already spent n units approaching at the start of the trace.*

Definition 14 (Approaches-long-enough) The function *approaches-long-enough*: $\mathcal{L}^* \times \mathcal{N} \to \mathcal{B}$ is defined as follows:

$$\mathit{approaches\text{-}long\text{-}enough}(tr,\, n) \;\equiv\; \mathit{seq\text{-}long\text{-}enough}(\texttt{approaching},\, tr,\, n,\, \mathit{approaching\text{-}min\text{-}time})$$

This function computes the distance from the beginning of a train trace until the train enters the gate. If the trace is empty, it assumes a minimal safe distance.

Definition 15 (Distance-to-gate) The function $\mathit{distance\text{-}to\text{-}gate} : \mathcal{L}^* \to \mathcal{N}$ is defined as follows:

$$\mathit{distance\text{-}to\text{-}gate}(tr) \;\equiv$$
$$\begin{cases} \mathit{approaching\text{-}min\text{-}time} + 2 & \mathit{if}\ tr = \langle\rangle; \\ 0 & \mathit{if}\ tr = x \circ tr' \wedge x = \texttt{in-gate}; \\ \mathit{distance\text{-}to\text{-}gate}(tr' + 1 & \mathit{otherwise}. \end{cases}$$

Definition 16 (Legal-train-trace) The function $\mathit{legal\text{-}train\text{-}trace} : \mathcal{L}^* \times \mathcal{N} \to \mathcal{B}$ is defined as follows:

$$\mathit{legal\text{-}train\text{-}trace}(tr,\, n) \;\equiv\; \mathit{train\text{-}tracep}(tr) \wedge \mathit{approaches\text{-}long\text{-}enough}(tr,\, n)$$

The Controller

Our control algorithm is very simple: generate a control pulse of `open` *or* `close` *depending on the current position of the train.*

Definition 17 (Control-output) The function *control-output* : $\mathcal{L} \rightarrow \mathcal{L}$ is defined as follows:

$$\textit{control-output}(i) \equiv \begin{cases} \texttt{open} & \textit{if } i = \texttt{elsewhere}; \\ \texttt{close} & \textit{otherwise.} \end{cases}$$

Modeling the System

Given specifications of the train, gate, and controller, we can now combine them to analyze the behavior of the system. The following function returns a trace of successive states of the gate, given a trace of the train and an initial gate state.

Definition 18 (Gate-behavior) The function *gate-behavior* : $\mathcal{L}^* \times \langle \mathcal{L}, \mathcal{N} \rangle \rightarrow \langle \mathcal{L}, \mathcal{N} \rangle^*$ is defined as follows:

$$\textit{gate-behavior}(tr, \langle l, n \rangle) \equiv \begin{cases} \langle \rangle & \textit{if } tr = \langle \rangle; \\ \langle l', n' \rangle \circ \textit{gate-behavior}(tr', \langle l', n' \rangle) & \textit{if } tr = x \circ tr', \end{cases}$$

where $\langle l', n' \rangle = \textit{gate-next-state}(\textit{control-output}(x, l, tr), \langle l, n \rangle, x)$.

This function defines the key system property in terms of the relation between the train's current position and the state of the gate at that step. It assures that the gate has time to close before the train reaches it, and that the gate starts up as soon as possible after the train has left the gate.

Definition 19 (Good-statep) The function *good-statep* : $\mathcal{L} \times \mathcal{L} \times \mathcal{N} \times \mathcal{N} \times \mathcal{N} \rightarrow \mathcal{B}$ is defined as follows:

$$\textit{good-statep}(p, g, n, d, e) \equiv$$

$$\begin{array}{ll} & g = \texttt{going-down} \rightarrow n < \textit{gate-closing-max-time} \\ \wedge & g = \texttt{going-up} \rightarrow n < \textit{gate-opening-max-time} \\ \wedge & p = \texttt{in-gate} \rightarrow g = \texttt{closed} \\ \wedge & p = \texttt{approaching} \rightarrow \\ & \quad (g = \texttt{closed}) \vee (g = \texttt{going-down} \wedge n < d) \\ & \quad \vee (\textit{gate-closing-max-time} < d) \\ \wedge & p = \texttt{elsewhere} \rightarrow \\ & \quad (\textit{gate-opening-max-time} < e \rightarrow g = \texttt{open}) \\ & \quad \wedge [\ (g = \texttt{closed} \wedge e = 0) \vee (g = \texttt{open}) \\ & \qquad \vee (g = \texttt{going-up} \wedge n \leq \textit{gate-opening-max-time} - e)\ ] \end{array}$$

SU-invariant *checks that* good-statep *is maintained invariantly over a train trace and corresponding gate-trace.*

Definition 20 (SU-invariant) The function *su-invariant* : $\mathcal{L}^* \times \langle \mathcal{L}, \mathcal{N} \rangle^* \times \mathcal{N} \to \mathcal{B}$ is defined as follows:

$$
\textit{su-invariant}\,(tr,\ \textit{gate-tr},\ n) \quad \equiv
$$
$$
\begin{cases}
\textit{true} & \textit{if } tr = \langle\rangle; \\
\textit{false} & \textit{if gate-tr} = \langle\rangle; \\
\quad \textit{good-statep}\,(\texttt{elsewhere},\ z,\ d,\ n) & \textit{if } tr = \texttt{elsewhere} \circ y \\
\wedge\ \textit{su-invariant}\,(y,\ w,\ 0) & \wedge\ \textit{gate-tr} = z \circ w; \\
\quad \textit{good-statep}\,(x,\ z,\ d,\ n) & \textit{if } tr = x \circ y\ \wedge\ \textit{gate-tr} = z \circ w; \\
\wedge\ \textit{su-invariant}\,(y,\ w,\ n+1) &
\end{cases}
$$

Now we define the desired safety and utility properties and prove that they follow from the su-invariant *property.*

Definition 21 (Safety) The function *safety* : $\mathcal{L}^* \times \langle \mathcal{L}, \mathcal{N} \rangle^* \to \mathcal{B}$ is defined as follows:

$$
\textit{safety}\,(\textit{train-tr},\ \textit{gate-tr}) \quad \equiv
$$
$$
\begin{cases}
\textit{true} & \textit{if train-tr} = \langle\rangle; \\
\quad g = \texttt{closed} & \textit{if train-tr} = \texttt{in-gate} \circ \textit{train-tr}' \\
\wedge\ \textit{safety}\,(\textit{train-tr}',\ \textit{gate-tr}') & \wedge\ \textit{gate-tr} = g \circ \textit{gate-tr}'; \\
\textit{safety}\,(\textit{train-tr}',\ \textit{gate-tr}') & \textit{otherwise.}
\end{cases}
$$

Theorem 22 (SU-Invariant implies Safety)
For all $\textit{train-tr} \in \mathcal{L}^*,\ \textit{gate-tr} \in \langle \mathcal{L}, \mathcal{N} \rangle^*,\ n \in \mathcal{N}$,

$$
\textit{su-invariant}\,(\textit{train-tr},\ \textit{gate-tr},\ n) \to \textit{safety}\,(\textit{train-tr},\ \textit{gate-tr}).
$$

Definition 23 (Utility) The function *utility* : $\mathcal{L}^* \times \langle \mathcal{L}, \mathcal{N} \rangle^* \times \mathcal{N} \to \mathcal{B}$ is defined as follows:

$$
\textit{utility}\,(\textit{train-tr},\ \textit{gate-tr},\ n) \quad \equiv
$$
$$
\begin{cases}
\textit{true} & \textit{if train-tr} = \langle\rangle; \\
\quad (\textit{gate-opening-max-time} < n & \textit{if train-tr} = \texttt{elsewhere} \circ \textit{train-tr}' \\
\qquad \to g = \texttt{open}) & \wedge\ \textit{gate-tr} = g \circ \textit{gate-tr}'; \\
\wedge\ \textit{utility}\,(\textit{train-tr}',\ \textit{gate-tr}') & \\
\textit{utility}\,(\textit{train-tr}',\ \textit{gate-tr}') & \textit{otherwise.}
\end{cases}
$$

Theorem 24 (SU-Invariant implies Utility)
For all $\textit{train-tr} \in \mathcal{L}^*,\ \textit{gate-tr} \in \langle \mathcal{L}, \mathcal{N} \rangle^*,\ n \in \mathcal{N}$,

$$
\textit{su-invariant}\,(\textit{train-tr},\ \textit{gate-tr},\ n) \to \textit{utility}\,(\textit{train-tr},\ \textit{gate-tr},\ n).
$$

Theorem 25 (Controller maintains Safety and Utility)
For all $\textit{train-tr} \in \mathcal{L}^*,\ x, g \in \mathcal{L},\ n, e, a \in \mathcal{N}$,

$$\begin{array}{rl} & \textit{legal-train-trace}\,(\textit{train-tr},\ a) \\ \wedge & \textit{good-statep}\,(x,\ g,\ n,\ \textit{distance-to-gate}\,(\textit{train-tr}), e) \\ \rightarrow & \\ & \textit{safety}(\textit{train-tr},\ \textit{gate-behavior}\,(\textit{train-tr},\ g,\ n)) \\ \wedge & \textit{utility}(\textit{train-tr},\ \textit{gate-behavior}\,(\textit{train-tr},\ g,\ n),\ e) \end{array}$$

where $\textit{train-tr} = x \circ \textit{train-tr}'$.

We now characterize an initial state and show that our invariant is true in that initial state.

Definition 26 (Initial-statep) The function *initial-statep* : $\mathcal{L} \times \mathcal{L} \rightarrow \mathcal{B}$ is defined as follows:

$$\textit{initial-statep}\,(\textit{train},\ \textit{gate}) \quad \equiv \quad \textit{train} = \texttt{elsewhere} \wedge\ \textit{gate} = \texttt{open}.$$

The following two theorems establish that our system maintains the safety and utility properties if started in a safe initial state.

Theorem 27 (Controller Safety)
For all $\textit{train-tr} \in \mathcal{L}^*,\ x, g \in \mathcal{L},\ a \in \mathcal{N}$,

$$\begin{array}{rl} & \textit{initial-statep}\,(x,\ g)\ \wedge \textit{legal-train-trace}\,(\textit{train-tr},\ a) \\ \rightarrow & \textit{safety}\,(\textit{train-tr},\ \textit{gate-behavior}\,(\textit{train-tr},\ g,\ n)), \end{array}$$

where $\textit{train-tr} = x \circ \textit{train-tr}'$.

Theorem 28 (Controller Utility)
For all $\textit{train-tr} \in \mathcal{L}^*,\ x, g \in \mathcal{L}\ a, e \in \mathcal{N}$,

$$\begin{array}{rl} & \textit{initial-statep}\,(x,\ g)\ \wedge \textit{legal-train-trace}\,(\textit{train-tr},\ a) \\ \rightarrow & \textit{utility}\,(\textit{train-tr},\ \textit{gate-behavior}\,(\textit{train-tr},\ g,\ n),\ e), \end{array}$$

where $\textit{train-tr} = x \circ \textit{train-tr}'$.

An Alternative Version

Here we offer alternative versions of the safety and utility properties using quantifiers, and prove that they are consequences of our earlier versions.

Definition 29 (Safety2) The function *safety2* : $\mathcal{L}^* \times \langle \mathcal{L}, \mathcal{N} \rangle^* \rightarrow \mathcal{B}$ is defined as follows:

$$\textit{safety2}\,(\textit{train-tr},\ \textit{gate-tr}) \quad \equiv$$
$$\forall i \in \mathcal{N}, i < \textit{len}\,(\textit{train-tr})\ \wedge\ \textit{train-tr}[i] = \texttt{in-gate} \rightarrow \textit{gate-tr}[i] = \texttt{closed}.$$

Theorem 30 (Safety implies Safety2)
For all $train\text{-}tr \in \mathcal{L}^*$ *and* $gate\text{-}tr \in \langle \mathcal{L}, \mathcal{N} \rangle^*$,

$$safety(train\text{-}tr,\ gate\text{-}tr) \rightarrow safety2(train\text{-}tr,\ gate\text{-}tr).$$

Theorem 31 (Controller Safety2)
For all $train\text{-}tr' \in \mathcal{L}^*$, $x, g \in \mathcal{L}$ $a \in \mathcal{N}$,

$$\begin{array}{ll} & initial\text{-}statep(x,\ g)\ \wedge\ legal\text{-}train\text{-}trace(train\text{-}tr,\ a) \\ \rightarrow & safety2(train\text{-}tr,\ gate\text{-}behavior(train\text{-}tr,\ g,\ n)), \end{array}$$

where $train\text{-}tr = x \circ train\text{-}tr'$.

Like distance-to-gate *this is purely a specification function. It computes the time we have been in the* `elsewhere` *region at location* i *in the trace. It assumes that we have already been* `elsewhere` n *units at the beginning of the trace.*

Definition 32 (Time-elsewhere) The function $time\text{-}elsewhere : \mathcal{N} \times \mathcal{N} \times \mathcal{L}^* \rightarrow \mathcal{N}$ is defined as follows:

$$time\text{-}elsewhere(i,\ n,\ train\text{-}tr) \quad \equiv$$
$$\begin{cases} n & if\ i = 0; \\ time\text{-}elsewhere(i-1,\ n+1,\ train\text{-}tr') & train\text{-}tr = \texttt{elsewhere} \circ train\text{-}tr'; \\ time\text{-}elsewhere(i-1,\ 0,\ train\text{-}tr') & train\text{-}tr = y \circ train\text{-}tr', \end{cases}$$

where $y \neq \texttt{elsewhere}$.

Definition 33 (Utility2) The function $utility2 : \mathcal{L}^* \times \langle \mathcal{L}, \mathcal{N} \rangle^* \times \mathcal{N} \rightarrow \mathcal{B}$ is defined as follows:

$$utility2(train\text{-}tr,\ gate\text{-}tr,\ n) \quad \equiv$$

$$\forall i \in \mathcal{N},$$
$$\begin{array}{rl} & i \leq len(train\text{-}tr) \\ \wedge & train\text{-}tr[i] = \texttt{elsewhere} \\ \wedge & gate\text{-}opening\text{-}max\text{-}time < time\text{-}elsewhere(i,\ n,\ train\text{-}tr) \\ \rightarrow & gate\text{-}tr[i] = \texttt{open}. \end{array}$$

Theorem 34 (Utility implies Utility2)
For all $train\text{-}tr \in \mathcal{L}^*$, $gate\text{-}tr \in \langle \mathcal{L}, \mathcal{N} \rangle^*$, *and* $n \in \mathcal{N}$,

$$utility(train\text{-}tr,\ gate\text{-}tr,\ n) \rightarrow utility2(train\text{-}tr,\ gate\text{-}tr,\ n).$$

Theorem 35 (Controller Utility2)
For all $train\text{-}tr' \in \mathcal{L}^*$, $x, g \in \mathcal{L}$, $n, a, e \in \mathcal{N}$,

$$\begin{array}{rl} & initial\text{-}statep(x,\ g) \\ \wedge & legal\text{-}train\text{-}trace(train\text{-}tr,\ a) \\ \rightarrow & \\ & utility2(train\text{-}tr,\ gate\text{-}behavior(train\text{-}tr,\ g,\ n),\ e), \end{array}$$

where $train\text{-}tr = x \circ train\text{-}tr'$.

References

[1] Odyssey Research Associates. *Introduction to EVES: Exercises and Notes.* ORA, Ottawa, Canada, 1992.

[2] W.R. Bevier, W.A. Hunt, Jr., J S. Moore, and W.D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[3] W.R. Bevier and W.D. Young. Machine checked proofs of the design of a fault-tolerant circuit. *Formal Aspects of Computing*, 4:755–775, 1992.

[4] R.S. Boyer, D. Goldschlag, M. Kaufmann, and J S. Moore. Functional instantiation in first order logic. In *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 7–26. Academic Press, 1991.

[5] R.S. Boyer and J S. Moore. *A Computational Logic.* ACM Monograph Series. Academic Press, Inc., 1979.

[6] R.S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, Boston, 1988.

[7] B.C. Brock, W.A. Hunt, Jr., and W.D. Young. Introduction to a formally defined hardware description language. In *Proceedings of the IFIP Conference on Theorem Provers in Circuit Design*, pages 3–36. North-Holland, 1992.

[8] M. Carranza and W.D. Young. Verifying a fuzzy controller. In *Proceedings of the Second International Workshop on Industrial Fuzzy Control and Intelligent Systems*, pages 194–203. IFIP, December 1992.

[9] D.M. Goldschlag. *Mechanically Verifying Concurrent Programs.* PhD thesis, University of Texas at Austin, 1991.

[10] C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw. Comparing different approaches for specifying and verifying real-time systems. in preparation, 1993.

[11] F. Jahanian and D.A. Stuart. A method for verifying properties of Modechart specifications. In *Proceedings of the Real-Time Systems Symposium*, December 1988.

[12] M. Kaufmann. A user's manual for an interactive enhancement to the Boyer-Moore theorem prover. Technical Report CLI 19, Computational Logic, Inc., May 1988.

[13] M. Kaufmann. An extension of the Boyer-Moore theorem prover to support first-order quantification. *Journal of Automated Reasoning*, 9(3):355–372, December 1992.

[14] Matt Kaufmann. A mechanically-checked correctness proof for generalization in the presence of free variables. *Journal of Automated Reasoning*, 7, 1991.

[15] N.G. Leveson and J.L. Stolzy. Safety analysis using Petri nets. *IEEE Transactions on Software Engineering*, 13(3):386–97, March 1987.

[16] P. Lincoln and J. Rushby. The formal verification of an algorithm for interactive consistency under a hybrid fault model. In *Proceedings of CAV '93*, pages 292–304. Springer-Verlag Lecture Notes in Computer Science 697, June 1993. Matt Wilding has the book.

[17] Formal Systems (Europe) Ltd. *Failure Divergence Refinement, User Manual and Tutorial.* Formal Systems Ltd., Oxford, UK, 1992.

[18] N. Shankar. Mechanized verification of real-time systems using PVS. to appear, March 1993.

[19] M.K. Smith. Track-segment safety in Nqthm. Internal note (unnumbered), Computational Logic, Inc., September 1993.

[20] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, New York, N.Y., 1989.

[21] W.G. Wood. Specification of operation and controller design constraints for a real-time system. Unpublished, September 1992.