The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions[*]

Yuri Gurevich[†]and James K. Huggins[†]
EECS Department, University of Michigan, Ann Arbor, MI, 48109-2122, USA.

May 21, 2002

Abstract

We give an evolving algebra solution for the well-known railroad crossing problem and use the occasion to experiment with agents that perform instantaneous actions in continuous time and in particular with agents that fire at the moment they are enabled.

1 Introduction

The well-known railroad crossing problem has been used as an example for comparing various specification and validation methodologies; see for example [6, 7] and the relevant references there. The evolving algebras (EA) methodology has been used extensively for specification and validation for real-world software and hardware systems; see the EA guide [3] and the EA bibliography [1]. The merits of using "toy" problems as benchmarks are debatable; not every methodology scales well to real-world problems. Still, toy problems are appropriate for experimentation. Here we present an evolving algebra solution for the railway crossing problem and use the opportunity for experimentation with instantaneous actions and reactions in real time.

In Sect. 2, we describe a version of the railroad crossing problem. It is not difficult to generalize the problem (e.g. by relaxing our assumptions on trains) and generalize the solution respectively. An interested reader may view that as an exercise.

In Sect. 3, we give a brief introduction to evolving algebras (in short, ealgebras), in order to make this paper self-contained. We omit many important aspects of ealgebras and refer the interested reader to a fuller definition in the EA guide [3]. In Sect. 4, experimenting with instantaneous actions in real time, we define special distributed real-time ealgebras appropriate to situations like that of the railroad crossing problem.

In Sect. 5 and Sect. 6, we give a solution for the railroad crossing problem which is formalized as an ealgebra. The program for the ealgebra is given in Sect. 5. The reader may wish to look at Sect. 5 right away; the notation is self-explanatory to a large extent. In Sect. 6, we define regular runs (the only relevant runs) of our ealgebra and analyze those runs. Formally speaking, we have to prove the existence of regular runs for every possible pattern of trains; for technical reasons, we delay the existence theorem until later.

In Sect. 7, we prove the safety and liveness properties of our solution. In Sect. 8 we prove a couple of additional properties of our ealgebra. In Sect. 9, we take advantage of the additional properties and prove the existence theorem for regular runs and analyze the variety of regular runs.

The ealgebra formalization is natural and this allows us to use intuitive terms in our proofs. One may have an impression that no formalization is really needed. However, a formalization is needed if one wants a

[*]In *Computer Science Logic, Selected papers from CSL'95*, ed. H. K. Büning, Springer Lecture Notes in Computer Science 1092, 1996, 266–290.

[†]Partially supported by NSF grant CCR-95-04375 and ONR grant N00014-94-1-1182.

mathematical verification of an algorithm: mathematical proofs are about mathematical objects. Of course, we could avoid intuitive terms and make the proofs more formal and pedantic, but this paper is addressed to humans and it is so much harder to read pedantic proofs. It is a long standing tradition of applied mathematics to use intuitive terms in proofs. Let us notice though that more formal and pedantic proofs have their own merits; if one wants to check the details of our proofs by machine, it is useful to rewrite the proofs in a pedantic way. In any case, we see a great value in the naturality of formalization. No semantical approach makes inherent difficulties of a given problem go away. At best, the approach does not introduce more complications and allows one to deal with the inherent complexity of the given problem.

Acknowledgments. Raghu Mani participated in an initial stage of the work [5]. During the final stage of the work, the first author was a CNRS[1] visitor in the Laboratoire Informatique Theoretique et Programmation, Paris, France [4].

2 The Railroad Crossing Problem

Imagine a railroad crossing with several train tracks and a common gate, such as the one depicted in Fig. 1. Sensors along every track detect oncoming and departing trains. Let us consider one of the tracks, shown in Fig. 2. It has four sensors at points L1, L2, R1 and R2. Sensor L1 detects trains coming from the left, and sensor L2 detects when those trains leave the crossing. Similarly sensor R1 detects trains coming from the right, and sensor R2 detects when those trains leave the crossing. Based on signals from these sensors, an automatic controller signals the gate to open or close.

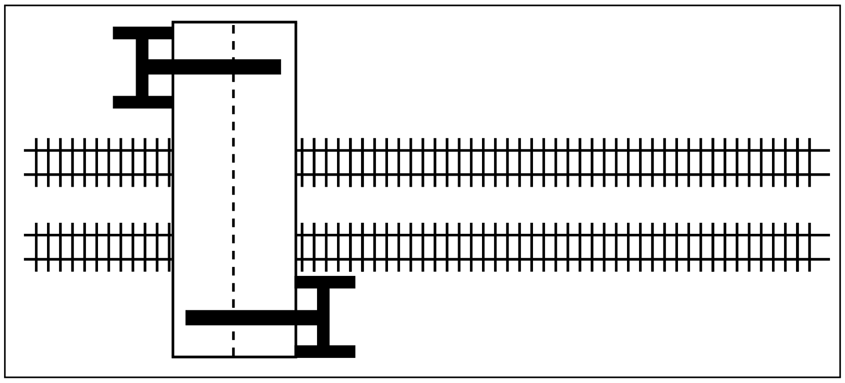

Figure 1: A railroad crossing.

The problem is to design a controller that guarantees the following requirements.

Safety If a train is in the crossing, the gate is closed.

Liveness The gate is open as much as possible.

Several assumptions are made about the pattern of train movement. For example, if a train appears from the left, it leaves the crossing to the right. It is easiest to express those assumptions as a restriction on possible histories of train motion on any given track.

[1]Centre National de la Recherche Scientifique

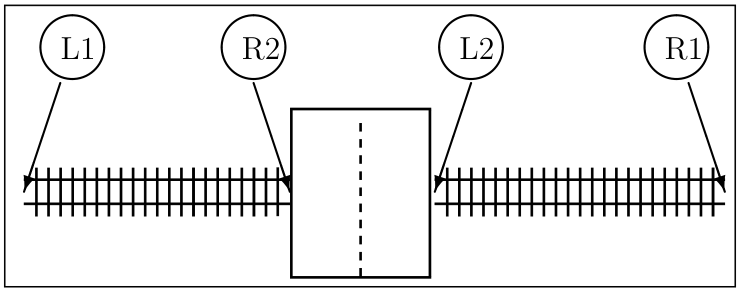

Figure 2: Placement of sensors along a railroad track.

Assumptions Regarding Train Motion. For any given track, there is a finite or infinite sequence of moments

$$t_0 < t_1 < t_2 < t_3 < \dots$$

satisfying the following conditions.

Initial State The moment t_0 is the initial moment. The observed part $[L1, R1]$ of the track is empty at t_0.

Train Pattern If t_{3i+1} appears in the sequence then t_{3i+3} appears in the sequence and we have that

- at t_{3i+1}, one oncoming train is detected at L1 or R1,
- at t_{3i+2} the train reaches the crossing, and
- at t_{3i+3} the train is detected to have left the crossing at L2 or R2 respectively.

Completeness There are no other trains.

Additional Assumptions. From the moment that an oncoming train is detected, it takes time between $d_{\min}$ and $d_{\max}$ for the train to reach the crossing. In terms of the sequence $\langle t_0 < t_1 < t_2 < t_3 < \dots\rangle$ above, this assumption can be stated as follows:

1 Every difference $t_{3i+2} - t_{3i+1}$ belongs to the interval $[d_{\min}, d_{\max}]$.

Further, the gate closes within time d_{close} and opens within time d_{open}. This does not necessarily mean that if the controller signals the gate to close (respectively open) at moment t then the gate closes (respectively opens) by time $t + d_{\text{close}}$ (respectively $t + d_{\text{open}}$). Let us state the assumption more precisely as a restriction on possible histories.

2 There is no interval $I = (t, t + d_{\text{close}})$ (respectively $I = (t, t + d_{\text{open}})$) during which the signal to close (respectively to open) is in force but the gate is not closed (respectively opened) at any moment in I.

It is easy to see that the controller cannot guarantee the safety requirement is satisfied if $d_{\min} < d_{\text{close}}$. We ignore the case $d_{\min} = d_{\text{close}}$ and assume that

3 $d_{\text{close}} < d_{\min}$.

Finally, we will assume that actions are performed instantaneously. Of course, real actions take time and the use of instantaneous actions is an abstraction. But this may be a useful abstraction. For example, in our case, it is natural to ignore the time taken by the controller's actions. It is not natural at all to view closing and opening of the gate as instantaneous actions, and we will not do that. Let us stress that the evolving algebra methodology does not require that actions are necessarily instantaneous. See for example [2] where an instantaneous action ealgebra is refined to a prolonged-action ealgebra.

The design part of the railway crossing problem is not difficult, especially because the problem has been addressed in a number of papers. What remains is to formalize the design in a specification language, in our case as an evolving algebra, and prove the safety and liveness requirements are satisfied.

3 Evolving Algebras Reminder

We give a brief reminder on evolving algebras based on the EA guide [3]. We present only what is necessary here and ignore many important features.

3.1 Static Algebras

Static algebras are essentially logicians' structures except that a tiny bit of meta-mathematics is built into it. They are indeed algebras in the sense of the science of universal algebra.

A *vocabulary* is a collection of function symbols; each symbol has a fixed arity. Some function symbols are tagged as relation symbols (or predicates). It is supposed that every vocabulary contains the following *logic symbols*: nullary symbols *true*, *false*, *undef*, a binary symbol =, and the symbols of the standard propositional connectives.

A *static algebra* (or a *state*) *A of vocabulary* Υ is a nonempty set X (the *basic set* or *superuniverse* of A), together with interpretations of all function symbols in Υ over X (the *basic functions* of A). A function symbol f of arity r is interpreted as an r-ary operation over X (if $r = 0$, it is interpreted as an element of X). The interpretations of predicates (*basic relations*) and the logic symbols satisfy some obvious requirements stated below.

Remark on notations and denotations. A symbol in Υ is a name or notation for the operation that interprets it in A, and the operation is the meaning or denotation of the symbol in A. In English, a word "spoon" is a name of a familiar table utensil, and one says "I like that spoon" rather than a more cumbersome "I like that utensil named 'spoon'". Similarly, when a state is fixed, we may say that f maps a tuple $\bar{a}$ to an element b rather than that the interpretation of f maps a tuple $\bar{a}$ to an element b.

On the interpretations of logic symbols and predicates. Intuitively, (the interpretations of) *true* and *false* represent truth and falsity respectively. Accordingly, the symbols *true* and *false* are interpreted by different elements. These two elements are the only possible values of any basic relation. The Boolean connectives behave in the expected way over these two elements, and the equality function behaves in the expected way over all elements.

Universes and typing. Formally speaking, a static algebra is one-sorted. However, it may be convenient to view it as many-sorted; here we describe a standard way to do this. Some unary basic relations are designated as universes (or sorts) and their names may be called universe symbols. One thinks about a universe U as a set $\{x : U(x) = \mathit{true}\}$. Basic functions are assigned universes as domains. For example, the domain of a binary function f may be given as $U_1 \times U_2$ where U_1 and U_2 are universes. If f is a relation, this means that $f(a_1, a_2) = \mathit{false}$ whenever $a_1 \notin U_1$ or $a_2 \notin U_2$. Otherwise this means that $f(a_1, a_2) = \mathit{undef}$ whenever $a_1 \notin U_1$ or $a_2 \notin U_2$, so that f is intuitively a partial function.

Remark on the built-in piece of meta-mathematics. In first-order logic, an assertion about a given structure does not evaluate to any element of the structure. For technical convenience, in evolving algebras truth and falsity are represented internally and many assertions can be treated as terms. This technical modification does not prevent us from dealing with assertions directly. For example, let f, g be nullary function symbols and P a binary function symbol. Instead of saying that $P(f, g)$ evaluates to *true* (respectively *false*) at a state A, we may say $P(f, g)$ holds (respectively fails) at A. In some cases, we may even omit "holds"; for example, we may assert simply that $f \neq g$. Admittedly, this is not very pedantic, but we write for humans, not machines.

3.2 Updates

Alternatively, a state can be viewed as a kind of memory. A *location* ℓ of a state A of vocabulary Υ is a pair $\ell = (f, \bar{a})$ where f is a symbol in Υ of some arity r and $\bar{a}$ is an r-tuple of elements of A (that is, of the superuniverse of A). The element $f(\bar{a})$ is the *content* of location ℓ in A.

An *update* of state A is a pair (ℓ, b), where ℓ is some location $(f, \bar{a})$ of A and b is an element of A; it is supposed that b is (the interpretation of) *true* or *false* if f is a predicate. This update is *trivial* if b is the content of ℓ in A. An update can be performed: just replace the value at location ℓ with b. The vocabulary, the superuniverse and the contents of other locations remain unchanged. The state changes only if the update is nontrivial.

Call a set $S = \{(\ell_1, b_1), \ldots, (\ell_n, b_n)\}$ of updates of a state A *consistent* if the locations are distinct. In other words, S is *inconsistent* if there are i, j such that $\ell_i = \ell_j$ but $b_i \neq b_j$. In the case that S is consistent it is performed as follows: replace the content of ℓ_1 with b_1, the content of ℓ_2 with b_2 and so on. To perform an inconsistent update set, do nothing.

A pedantic remark. The equality used in the previous paragraph is not the built-in equality of A but rather the equality of the meta language. One could use another symbol for the built-in equality, but this is not necessary.

A remark to theoreticians. At the point that updates are introduced, some people, in particular Robin Milner [8], raise an objection that an update may destroy algebraic properties. For example, an operation may lose associativity. That is true. So, in what sense are static algebras algebraic? They are algebraic in the sense that the nature of elements does not matter and one does not distinguish between isomorphic algebras. A standard way to access a particular element is to write a term that evaluates to that element. Coming back to algebraic properties like associativity (and going beyond the scope of this paper), let us note that, when necessary, one can guarantee that such a property survives updating by declaring some functions static or by imposing appropriate integrity constraints or just by careful programming.

3.3 Basic Rules

In this subsection we present the syntax and semantics of basic rules. Each rule R has a vocabulary, namely the collection of function symbols that occur in R. A rule R is applicable to a state A only if the vocabulary of A includes that of R. At each state A of sufficiently rich vocabulary, R gives rise to a set of updates. To execute R at such a state A, perform the update set at A.

A *basic update rule* R has the form

$$f(e_1, \ldots, e_r) := e_0$$

where f is an r-ary function symbol (the *head* of R) and each e_i is a ground term, that is, a term without any variables. (In programming languages, terms are usually called expressions; that motivates the use of letter e for terms.) To execute R at a state A of sufficiently rich vocabulary, evaluate all terms e_i at A and then change f accordingly. In other words, the update set generated by R at A consists of one update (ℓ, a_0) where $\ell = (f, (a_1, \ldots, a_r))$ and each a_i is the value of e_i at A.

For example, consider an update rule $f(c_1 + c_2) := c_0$ and a state A where $+$ is interpreted as the standard addition function on natural numbers and where c_1, c_2, c_0 have values $3, 5, 7$ respectively. To execute the rule at A, set $f(8)$ to 7.

There are only two basic rule constructors. One is the *conditional constructor* which produces rules of the form:

$$\textbf{if } g \textbf{ then } R_1 \textbf{ else } R_2 \textbf{ endif}$$

where g is a ground term (the *guard* of the new rule) and R_1, R_2 are rules. To execute the new rule in a state A of sufficiently rich vocabulary, evaluate the guard. If it is true, then execute R_1; otherwise execute R_2. (The "**else**" clause may be omitted if desired.)

The other constructor is the *block constructor* which produces rules of the form:

```
block
        $R_1$
        $\vdots$
        $R_k$
endblock
```

where $R_1, \dots, R_k$ are rules. (We often omit the keywords "**block**" and "**endblock**" for brevity and use indentation to eliminate ambiguity.) To execute the new rule in a state A of sufficiently rich vocabulary, execute rules $R_1, \dots, R_k$ simultaneously. More precisely, the update set generated by the new rule at A is the union of the update sets generated by the rules R_i at A.

A *basic program* is simply a basic rule.

In this paper we say that a rule R is *enabled* at a state A of sufficiently rich vocabulary if the update set generated by R at A is consistent and contains a non-trivial update; otherwise R is *disabled* at A. (The notion of being enabled has not been formalized in the EA guide.) Rules will be executed only if they are enabled, so that the execution changes a given state. This seems to be a very pedantic point. What harm is done by executing a rule that does not change a given state? It turns out that the stricter notion of being enabled is convenient in real-time computational theory; see Lemma 4.4 in this connection.

3.4 Parallel Synchronous Rules

Generalize the previous framework in two directions. First, permit terms with variables and generalize the notion of state: in addition to interpreting some function names, a generalized state may assign values to some variables. (Notice that a variable cannot be the head of an update rule.)

Second, generalize the notion of guards by allowing bounded quantification. More formally, we define *guards* as a new syntactical category. Every term $P(e_1, \dots, e_r)$, where P is a predicate, is a guard. A Boolean combination of guards is a guard. If $g(x)$ is a guard with a variable x and U is a universe symbol then the expression $(\forall x \in U)g(x)$ is also a guard.

The semantics of guards is quite obvious. A guard $g(\bar{y})$ with free variables $\bar{y}$ holds or fails at a (generalized) state A that assigns values to all free variables of g. The least trivial case is that of a guard $g(\bar{y}) = (\forall x \in U)g'(x, \bar{y})$. For every element b of U in A, let A_b be the expansion of A obtained by assigning the value b to x. Then $g(\bar{y})$ holds at A if $g'(x, \bar{y})$ holds at every A_b; otherwise it fails at A.

Now consider a generalized basic rule $R(x)$ with a variable x and let U be a universe symbol. Form the following rule R^*:

```
var $x$ ranges over $U$
        $R(x)$
endvar
```

Intuitively, to execute R^*, one executes $R(x)$ for every $x \in U$. To make this more precise, let A be a (generalized) state that interprets all function names in the vocabulary of $R(x)$ and assigns values to all free variables of $R(x)$ except for x. For each element b of the universe U in A, let A_b be the expansion of A obtained by assigning the value b to x, and let E_b be the update set generated by $R(x)$ at A_b. Since x does not appear as the head of any update instruction in $R(x)$, each E_b is also a set of updates of A. The update set generated by R^* at A is the union of the update sets E_b.

Call the new rule a *parallel synchronous rule* (or a *declaration rule*, as in the EA guide). A *parallel synchronous program* is simply a parallel synchronous rule without free variables. Every occurrence of a variable should be bound by a declaration or a quantifier.

3.5 Special Distributed Programs

For our purposes here, a *distributed program* Π is given by a vocabulary and a finite set of basic or parallel synchronous programs with function symbols from the vocabulary of Π. The constitutent programs are the *modules* of $\mathcal{A}$. A *state* of Π is a state of the vocabulary of Π. Intuitively, each module is executed by a separate agent.

This is a very restricted definition. For example, the EA guide allows the creation of new agents during the evolution.

Intuitively, it is convenient though to distinguish between a module (a piece of syntax) and its executor, and even think about agents in anthropomorphic terms. But since in this case agents are uniquely defined by their programs, there is no real need to have agents at all, and we may identify an agent by the name of its program.

4 Special Distributed Real-Time Ealgebras

A program does not specify a (distributed) ealgebra completely. We need to define what constitutes a computation (or a run) and then to indicate initial states and maybe a relevant class of runs. In this section, we define a restricted class of distributed real-time evolving algebras by restricting attention to static algebras of a particular kind and defining a particular notion of run.

We are interested in computations in real time that satisfiy the following assumptions.

I1 Agents execute instantaneously.

I2 Enviromental changes take place instantaneously.

I3 The global state of the given distributed ealgebra is well defined at every moment.

Let us stress again that the three assumptions above are not a part of the evolving algebra definition. The prolonged-action ealgebra [2], mentioned in Sect. 2, satisfies none of these three assumptions.

Vocabularies and Static Structures. Fix some vocabulary Υ with a universe symbol Reals and let Υ^+ be the extension of Υ with a nullary function symbol CT; it is supposed of course that Υ does not contain CT. Restrict attention to Υ^+-states where the universe Reals is the set of real numbers and CT evaluates to a real number. Intuitively, CT gives the current time.

4.1 Pre-runs

Definition 4.1 A *pre-run* R of vocabulary Υ^+ is a mapping from the interval $[0, \infty)$ or the real line to states of vocabulary Υ^+ satisfying the following requirements where $\rho(t)$ is the reduct of $R(t)$ to Υ.

Superuniverse Invariability The superuniverse does not change during the evolution; that is, the superuniverse of every $R(t)$ is that of $R(0)$.

Current Time At every $R(t)$, CT evaluates to t.

Discreteness For every $\tau > 0$, there is a finite sequence $0 = t_0 < t_1 < \ldots < t_n = \tau$ such that if $t_i < \alpha < \beta < t_{i+1}$ then $\rho(\alpha) = \rho(\beta)$. $\Box$

Remarks. Of course, we could start with an initial moment different from 0, but without loss of generality we can assume that the initial moment is 0. Our discreteness requirement is rather simplistic (but sufficient for our purposes in this paper). One may have continuous time-dependent basic functions around (in addition to CT); in such cases, the discreteness requirement becomes more subtle.

In the rest of this section, R is a pre-run of vocabulary Υ^+ and $\rho(t)$ is the reduct of $R(t)$ to Υ.

The notation $\rho(t+)$ and $\rho(t-)$ is self-explanatory; still, let us define it precisely. $\rho(t+)$ is any state $\rho(t+\varepsilon)$ such that $\varepsilon > 0$ and $\rho(t+\delta) = \rho(t+\varepsilon)$ for all positive $\delta < \varepsilon$. Similarly, if $t > 0$ then $\rho(t-)$ is any state $\rho(t-\varepsilon)$ such that $0 < \varepsilon \leq t$ and $\rho(t-\delta) = \rho(t-\varepsilon)$ for all positive $\delta < \varepsilon$.

Call a moment t *significant* for R if (i) $t = 0$ or (ii) $t > 0$ and either $\rho(t) \neq \rho(t-)$ or $\rho(t) \neq \rho(t+)$.

Lemma 4.1 *For any moment t, $\rho(t+)$ is well defined. For any moment $t > 0$, $\rho(t-)$ is well defined. If there are infinitely many significant moments then their supremum equals ∞.*

Proof Obvious. $\Box$

Recall that a set S of nonnegative reals is *discrete* if it has no limit points. In other words, S is discrete if and only if, for every nonnegative real τ, the set $\{t \in S : t < \tau\}$ is finite. The discreteness requirement in the definition of pre-runs means exactly that the collection of the significant points of R is discrete.

We finish this subsection with a number of essentially self-evident definitions related to a given pre-run R. Let e be a term of vocabulary Υ^+. If e has free variables then fix the values of those variables, so that e evaluates to a definite value in every state of vocabulary Υ^+. (Formally speaking e is a pair of the form (e', ξ) where e' is a term and ξ assigns elements of $R(0)$ to free variables of e'.)

The value e_t of e at moment t is the value of e in $R(t)$. Accordingly, e *holds* (respectively *fails*) at t if it does so in $R(t)$. Likewise, a module is *enabled* (respectively *disabled*) at t if it is so in $R(t)$. In a similar vein, we speak about a time interval I. For example, *e holds over I* if it holds at every $t \in I$.

If e has the same value over some nonempty interval $(t, t+\varepsilon)$, then this value is *the value e_{t+} of e at $t+$* (respectively *at $t-$*). Similarly, if $t > 0$ and e has the same value over some nonempty interval $(t-\varepsilon, t)$, then this value is *the value e_{t-} of e at $t-$*. Define accordingly when e holds, fails at $t+, t-$ and when an agent is enabled, disabled at $t+, t-$.

Further, *e is set to* a value a (or simply *becomes a*) at t if either (i) $e_{t-} \neq a$ and $e_t = a$, or else (ii) $e_t \neq a$ and $e_{t+} = a$. Define accordingly when an agent becomes enabled, disabled at t.

4.2 Runs

Now consider a distributed program Π with function symbols from vocabulary Υ^+. Runs of Π are pre-runs with some restrictions on how the basic functions evolve. Depending upon their use, the basic functions of Π fall into the following three disjoint categories.

Static These functions do not change during any run. The names of these functions do not appear as the heads of update rules in Π.

Internal Dynamic These functions may be changed only by agents. The names of these functions appear as the heads of update rules and the functions are changed by executing the modules of Π. For brevity, we abbreviate "internal dynamic" to "internal".

External Dynamic These functions may be changed only by the environment. The names of these functions do not appear as the heads of update rules; nevertheless the functions can change from one state to another. Who changes them? The environment. Some restrictions may be imposed on how these functions can change. For brevity, we abbreviate "external dynamic" to "external".

Remark. It may be convenient to have functions that can by changed both by agents and the environment. The EA guide allows that, but we do not need that generality here.

Before we give the definition of runs, let us explain informally that one should be cautious with instantaneous actions. In particular, it may not be possible to assume that agents always fire at the moment they become enabled. Consider the following two interactive scenarios.

Scenario 1 The environment changes a nullary external function f at moment t. This new value of f enables an agent X. The agent fires immediately and changes another nullary function g.

What are the values of f and g at time t, and at what time does X fire? If f has its old value at t then X is disabled at t and fires at some time after t; thus X does not fire immediately. If g has its new value already at t then X had to fire at some time before t; that firing could not be triggered by the change of f. We arrive at the following conclusions: f has its new value at t (and thus f_t differs from f_{t-}), X fires at t, and g has its old value at t (and thus g_t differs from g_{t+}).

Scenario 2 At time t, an agent X changes a function g and in so doing enables another agent Y while disabling himself.

When does Y fire? Since X fires at t, it is enabled at t and thus g has its old value at t. Hence Y is disabled at t and fires at some time after t. Thus Y cannot react immediately.

The following definition is designed to allow immediate agents.

Definition 4.2 A pre-run R of vocabulary Υ^+ is a *run* of Π if it satisfies the following conditions where $\rho(t)$ is the reduct of R to Υ.

1. If $\rho(t+)$ differs from $\rho(t)$ then $\rho(t+)$ is the Υ-reduct of the state resulting from executing some modules $M_1, \ldots, M_k$ at $R(t)$. In such a case we say t is *internally significant* and the executors of $M_1, \ldots, M_k$ *fire* at t. All external functions with names in Υ have the same values in $\rho(t)$ and $\rho(t+)$.

2. If $i > 0$ and $\rho(\tau)$ differs from $\rho(\tau-)$ then they differ only in the values of external functions. In such a case we say τ is *externally significant*. All internal functions have the same values in $\rho(t-)$ and $\rho(t)$. □

Remark. Notice the global character of the definition of firing. An agent fires at a moment t if $\rho(t+) \neq \rho(t)$. This somewhat simplified definition of firing is sufficient for our purposes in this paper.

In the rest of this section, R is a run of Π and $\rho(t)$ the reduct of $R(t)$ to Υ. Let e be a term e with fixed values of all its free variables. A moment t is *significant for* e if, for every $\varepsilon > 0$, there exists a moment α such that $|\alpha - t| < \varepsilon$ and $e_a \neq e_t$. Call e *discrete* (in the given run R) if the collection of significant moments of e is discrete. In other words, e is discrete if and only, for every $t > 0$, there is a finite sequence

$$0 = t_0 < t_1 < \ldots < t_n = t$$

such that if $t_i < \alpha < \beta < t_{i+1}$ then $e_\alpha = e_\beta$.

Lemma 4.2 ((Discrete Term Lemma)) *If a term e is discrete then*

1. *For every t, e has a value at $t+$.*

2. *For every $t > 0$, e has a value at $t-$.*

Proof Obvious. □

Lemma 4.3 ((Preservation Lemma)) *Suppose that a term e with fixed values of its free variables does not contain CT. Then e is discrete. Furthermore,*

1. *If e contains no external functions and $t > 0$ then $e_t = e_{t-}$.*

2. *If e contains no internal functions then $e_{t+} = e_t$.*

Proof This is an obvious consequence of the definition of runs. □

It may be natural to have agents that fire the instant they are enabled.

Definition 4.3 An agent is *immediate* if it fires at every state where it is enabled. □

Lemma 4.4 ((Immediate Agent Lemma))

1. *The set of moments when an immediate agent is enabled is discrete.*

2. *If the agent is enabled at some moment t then it is disabled at $t+$ and, if $t > 0$, at $t-$.*

Proof

1. If the agent is enabled at a moment t, it fires at t and therefore (according to our notion of being enabled) changes the state; it follows that t is a significant moment of the run. By the discreteness condition on pre-runs, the collection of significant moments of a run is discrete. It remains to notice that every subset of a discrete set is discrete.

2. Follows from 1. □

Recall the scenario S2. There agent Y cannot be immediate. Nevertheless, it may make sense to require that some agents cannot delay firing forever.

Definition 4.4 An agent X is *bounded* if it is immediate or there exists a bound $b > 0$ such that there is no interval $(t, t + b)$ during which X is continuously enabled but does not fire. □

Notice that it is not required that if a bounded agent X becomes enabled at some moment α, then it fires at some moment $\beta < \alpha + b$. It is possible a priori that X becomes disabled and does not fire in that interval.

5 The Ealgebra for Railroad Crossing Problem

We present our solution for the railroad crossing problem formalized as an evolving algebra $\mathcal{A}$ of a vocabulary $\Upsilon^+ = \Upsilon \cup \{\mathrm{CT}\}$. In this section, we describe the program and initial states of $\mathcal{A}$; this will describe the vocabulary as well. The relevant runs of $\mathcal{A}$ will be described in the next section.

The program of $\mathcal{A}$ has two modules GATE and CONTROLLER, shown in Fig. 3.

```
GATE
    if Dir = open then GateStatus := opened endif
    if Dir = close then GateStatus := closed endif

CONTROLLER
    var $x$ ranges over Tracks
        if TrackStatus($x$) = coming and Deadline($x$) = $\infty$ then
            Deadline($x$) := CT+ WaitTime
        endif
        if CT =Deadline($x$) then Dir := close endif
        if TrackStatus($x$) = empty and Deadline($x$) < $\infty$ then
            Deadline($x$) := $\infty$
        endif
    endvar
    if Dir=close and SafeToOpen then Dir := open endif
```

Figure 3: Rules for GATE and CONTROLLER.

Here WaitTime abbreviates the term $d_{\min} - d_{\mathrm{close}}$, and SafeToOpen abbreviates the term

$$(\forall x \in \text{Tracks})[\text{TrackStatus}(x) = \text{empty} \;\text{ or }\; \text{CT} + d_{\mathrm{open}} < \text{Deadline}(x)].$$

We will refer to the two constituent rules of GATE as OpenGate, CloseGate respectively. We will refer to the three constituent rules of CONTROLLER's parallel synchronous rule as SetDeadline(x), SignalClose(x), ClearDeadline(x), respectively, and the remaining conditional rule as SignalOpen.

Our GateStatus has only two values: opened and closed. This is of course a simplification. The position of a real gate could be anywhere between fully closed and fully opened. (In [6], the position of the gate ranges between 0^o and 90^o.) But this simplification is meaningful. The problem is posed on a level of abstraction where it does not matter whether the gate swings, slides, snaps or does something else; it is even possible that there is no physical gate, just traffic lights. Furthermore, suppose that the gate is opening and consider its position as it swings from 0^o to 90^o. Is it still closed or already open at 75^o? One may say that it is neither, that it is opening. But for the waiting cars, it is still closed. Accordingly GateStatus is intended to be equal to closed at this moment. It may change to opened when the gate reaches 90^o. Alternatively, in the case when the crossing is equipped with traffic lights, it may change to opened when the light becomes green. Similarly, it may change from opened to closed when the light becomes red. If one is interested in specifying the gate in greater detail, our ealgebra can be refined by means of another ealgebra.

The program does not define our evolving algebra $\mathcal{A}$ completely. In addition, we need to specify a collection of *initial states* and relevant runs.

Initial states of $\mathcal{A}$ satisfy the following conditions:

1. The universe Tracks is finite. The universe ExtendedReals is an extension of the universe Reals with an additional element ∞. The binary relation $<$ and the binary operation $+$ are standard; in particular ∞ is the largest element of ExtendedReals.

2. The nullary functions close and open are interpreted by different elements of the universe Directions. The nullary functions closed and opened are interpreted by different elements of the universe GatеStatuses. The nullary functions empty, coming, in_crossing are different elements of the universe TrackStatuses.

3. The nullary functions $d_{\text{close}}, d_{\text{open}}, d_{\text{max}}, d_{\text{min}}$ are positive reals such that

$$d_{\text{close}} < d_{\text{min}} \leq d_{\text{max}}.$$

   One may assume for simplicity of understanding that these four reals are predefined: that is, they have the same value in all initial state. This assumption is not necessary.

4. The unary function TrackStatus assigns (the element called) empty to every track (that is, to every element of the universe Tracks). The unary function Deadline assigns ∞ to every track.

It is easy to see that, in any run, every value of the internal function Deadline belongs to ExtendedReals.

6 Regular Runs

The following definition takes into account the assumptions of Sect. 2.

6.1 Definitions

Definition 6.1 A run R of our evolving algebra is *regular* if it satisfies the following three conditions.

Train Motion For any track x, there is a finite or infinite sequence

$$0 = t_0 < t_1 < t_2 < t_3 < \dots$$

of so-called *significant moments of track x* such that

- TrackStatus(x) = empty holds over every interval $[t_{3i}, t_{3i+1})$;
- TrackStatus(x) = coming holds over every interval $[t_{3i+1}, t_{3i+2})$, and $d_{\text{min}} \leq (t_{3i+2} - t_{3i+1}) \leq d_{\text{max}}$;

- TrackStatus(x) = in_crossing holds over every interval $[t_{3i+2}, t_{3i+3})$; and
- if t_k is the final significant moment in the sequence, then k is divisible by 3 and TrackStatus(x) = empty over $[t_k, \infty)$.

Controller Timing Agent CONTROLLER is immediate.

Gate Timing Agent GATE is bounded. Moreover, there is no time interval $I = (t, t + d_{\text{close}})$ such that [Dir=close and GateStatus = opened] holds over I. Similarly there is no interval $I = (t, t + d_{\text{open}})$ such that [Dir=open and GateStatus = closed] holds over I. □

In the rest of this paper, we restrict attention to regular runs of $\mathcal{A}$. Let R be a regular run and ρ be the reduct of R to Υ.

6.2 Single Track Analysis

Fix a track x and let $0 = t_0 < t_1 < t_2 < \ldots$ be the significant moments of x.

Lemma 6.1 ((Deadline Lemma))

1. *Deadline*(x) $= \infty$ *over* $(t_{3i}, t_{3i+1}]$, *and Deadline*(x) $= t_{3i+1} +$ *WaitTime over* $(t_{3i+1}, t_{3i+3}]$.
2. *Let* $D_{\text{close}} = d_{\text{close}} + (d_{\max} - d_{\min}) = d_{\max} -$ *WaitTime. If TrackStatus*(x) $\neq$ *in_crossing over an interval* (α, β), *then Deadline*(x) $\geq \beta - D_{\text{close}}$ *over* (α, β).

Proof

1. A quite obvious induction along the sequence

$$(t_0, t_1], (t_1, t_3], (t_3, t_4], (t_4, t_6], \ldots.$$

The basis of induction. We prove that Deadline(x) $= \infty$ over $I = (t_0, t_1)$; it will follow by Preservation Lemma that Deadline(x) $= \infty$ at t_1. Initially, Deadline(x) $= \infty$. Only SetDeadline(x) can alter that value of Deadline(x), but SetDeadline(x) is disabled over (t_0, t_1). The induction step splits into two cases.

Case 1. Given that Deadline(x) $= \infty$ at t_{3i+1}, we prove that Deadline(x) $= t_{3i+1} +$ WaitTime over $I = (t_{3i+1}, t_{3i+3})$; it will follow by Preservation Lemma that Deadline(x) $= t_{3i+1} +$ WaitTime at t_{3i+3}. SetDeadline(x) is enabled and therefore fires at t_{3i+1} setting Deadline(x) to $t_{3i+1} +$ WaitTime. ClearDeadline(x) is the only rule that can alter that value of Deadline(x) but it is disabled over I because TrackStatus(x) $\neq$ empty over I.

Case 2. Given that Deadline(x) $< \infty$ at t_{3i} where $i > 0$, we prove that Deadline(x) $= \infty$ over $I = (t_{3i}, t_{3i+1})$; it will follow by Preservation Lemma that Deadline(x) $= \infty$ at t_{3i+1}. ClearDeadline(x) is enabled and therefore fires at t_{3i} setting Deadline(x) to ∞. Only SetDeadline(x) can alter that value of Deadline(x) but it is disabled over I because TrackStatus(x) = empty $\neq$ coming over I.

2. By contradiction suppose that Deadline(x) $< \beta - D_{\text{close}}$ at some $t \in (\alpha, \beta)$. By 1, there is an i such that $t_{3i+1} < t \leq t_{3i+3}$ and Deadline(x) $= t_{3i+1} +$ WaitTime at t. Since (α, β) and the in_crossing interval $[t_{3i+2}, t_{3i+3})$ are disjoint, we have that $t_{3i+1} < t < \beta \leq t_{3i+2}$. By the definition of regular runs, $d_{\max} \geq t_{3i+2} - t_{3i+1} \geq \beta - t_{3i+1}$, so that $t_{3i+1} \geq \beta - d_{\max}$. We have

$$\begin{array}{rcl} \beta - D_{\text{close}} & > & \text{Deadline}(x) \text{ at t} \quad = \quad t_{3i+1} + \text{WaitTime} \\ & \geq & \beta - d_{\max} + \text{WaitTime} \quad = \quad \beta - D_{\text{close}} \end{array}$$

which is impossible. □

Corollary 6.1 ((Three Rules Corollary))

1. *SetDeadline(x) fires exactly at moments t_{3i+1}, that is exactly when TrackStatus(x) becomes coming.*
2. *SignalClose(x) fires exactly at moments t_{3i+1} + WaitTime.*
3. *ClearDeadline(x) fires exactly at moments t_{3i} with $i > 0$, that is exactly when TrackStatus(x) becomes empty.*

Proof Obvious. □

Let $s(x)$ be the quantifier-free part

$$\text{TrackStatus}(x) = \text{empty} \ \text{ or } \ \text{CT} + d_{\text{open}} < \text{Deadline}(x).$$

of the term SafeToOpen with the fixed value of x.

Lemma 6.2 ((Local SafeToOpen Lemma))

1. *Suppose that WaitTime $> d_{\text{open}}$. Then $s(x)$ holds over intervals $[t_{3i}, t_{3i+1} + \textit{WaitTime} - d_{\text{open}})$ (the* maximal positive intervals of $s(x)$*) and fails over intervals $[t_{3i+1} + \textit{WaitTime} - d_{\text{open}}, t_{3i+3})$.*
2. *Suppose that WaitTime $\le d_{\text{open}}$. Then $s(x)$ holds over intervals $[t_{3i}, t_{3i+1}]$ (the* maximal positive intervals of $s(x)$*) and fails over intervals (t_{3i+1}, t_{3i+3}).*
3. *The term $s(v)$ is discrete.*
4. *$s(x)$ becomes true exactly at moments t_{3i} with $i > 0$, that is exactly when TrackStatus(x) becomes empty.*
5. *If $[\alpha, \beta)$ or $[\alpha, \beta]$ is a maximal positive interval of $s(x)$, then SignalClose(x) is disabled over $[\alpha, \beta]$ and at $\beta+$.*

Proof

1. Over $[t_{3i}, t_{3i+1})$, TrackStatus(x) = empty and therefore $s(x)$ holds. At t_{3i+1}, Deadline$(x) = \infty$ and therefore $s(x)$ holds. SetDeadline(x) fires at t_{3i+1} and sets Deadline(x) to t_{3i+1} + WaitTime. Over $(t_{3i}, t_{3i+1} + \text{WaitTime} - d_{\text{open}})$,

$$\begin{aligned} \text{CT} + d_{\text{open}} &< (t_{3i+1} + \text{WaitTime} - d_{\text{open}}) + d_{\text{open}} \\ &= t_{3i+1} + \text{WaitTime} = \text{Deadline}(x) \end{aligned}$$

and therefore $s(x)$ holds. Over the interval $[t_{3i+1} + \text{WaitTime} - d_{\text{open}}, t_{3i+3})$, TrackStatus$(x) \neq$ empty and CT $+ d_{\text{open}} \ge t_{3i+1}$ + WaitTime = Deadline(x) and therefore $s(x)$ fails.

2. The proof is similar to that of 1.
3. This follows from 1 and 2.
4. This follows from 1 and 2.
5. We consider the case when WaitTime $> d_{\text{open}}$; the case when WaitTime $\le d_{\text{open}}$ is similar. By 1, the maximal open interval of $s(x)$ has the form $[\alpha, \beta) = [t_{3i}, t_{3i+1} + \text{WaitTime} - d_{\text{open}})$ for some i. By Three Rules Corollary, SignalClose(x) fires at moments t_{3j+1} + WaitTime. Now the claim is obvious. □

6.3 Multiple Track Analysis

Lemma 6.3 ((Global SafeToOpen Lemma))

1. *The term SafeToOpen is discrete.*
2. *If SafeToOpen holds at $t+$ then it holds at t.*
3. *If SafeToOpen becomes true at t then some TrackStatus(x) becomes empty at t.*
4. *If SafeToOpen holds at t then t belongs to an interval $[\alpha, \beta)$ (a* maximal positive interval of *SafeToOpen) such that SafeToOpen fails at $\alpha-$, holds over $[\alpha, \beta)$ and fails at β.*

Proof

1. Use part 3 of Local SafeToOpen Lemma and the fact that there are only finitely many tracks.
2. Use parts 1 and 2 of Local SafeToOpen Lemma.
3. Use parts 1 and 2 of Local SafeToOpen Lemma.
4. Suppose that SafeToOpen holds at t. By parts 1 and 2 of Local SafeToOpen Lemma, for every track x, t belongs to an interval $[\alpha_x < \beta_x)$ such that $s(x)$ fails at α_x-, holds over $[\alpha_x, \beta_x)$ and fails at β_x. The desired $\alpha = \max_x \alpha_x$, and the desired $\beta = \min_x \beta_x$.□

Lemma 6.4 ((Dir Lemma)) *Suppose that $[\alpha, b)$ is a maximal positive interval of SafeToOpen.*

1. *Dir = close at α.*
2. *Dir = open over $(\alpha, \beta]$ and at $\beta+$.*

Proof

1. By Global SafeToOpen Lemma, some TrackStatus(x) becomes empty at t. Fix such an x and let $0 = t_0 < t_1 < t_2 < \ldots$ be the significant moments of TrackStatus(x). Then $\alpha = t_{3i+3}$ for some i. By Three Rules Corollary, SetDeadline(x) fires at t_{3i+1} + WaitTime setting Dir to close. By Local SafeToOpen Lemma, $s(x)$ fails over $I = (t_{3i+1} + \text{WaitTime}, t_{3i+3}]$. Hence SafeToOpen fails over I and therefore every SignalClose(y) is disabled over I. Thus Dir remains close over I.
2. By 1, SignalOpen fires at α setting Dir to open. By part 5 of Local SafeToOpen Lemma, every SignalClose(x) is disabled over $[\alpha, \beta]$ and at $\beta+$. Hence Dir remains open over $(\alpha, \beta]$ and at $\beta+$. □

Corollary 6.2 ((SignalOpen Corollary)) *SignalOpen fires exactly when SafeToOpen becomes true. SignalOpen fires only when some TrackStatus(x) becomes true.*

Proof Obvious. □

We have proved some properties of regular runs of our ealgebra $\mathcal{A}$, but the question arises if there any regular runs. Moreover, are there any regular runs consistent with a given pattern of trains? The answer is positive. In Sect. 8, we will prove that every pattern of trains gives rise to a regular run and will describe all regular runs consistent with a given pattern of trains.

7 Safety and Liveness

Recall that we restrict attention to regular runs of our ealgebra $\mathcal{A}$.

Theorem 7.1 ((Safety Theorem)) *The gate is closed whenever a train is in the crossing. More formally, GateStatus = closed whenever TrackStatus(x) = in_crossing for any x.*

Proof Let $t_0 < t_1 < \ldots$ be the significant moments of some track x. Thus, during periods $[t_{3i+2}, t_{3i+3})$, TrackStatus(x) = in_crossing. We show that GateStatus = closed over $[t_{3i+2}, t_{3i+3}]$ and even over $[t_{3i+1} + d_{\text{min}}, t_{3i+3}]$. (Recall that $d_{\text{min}} \le t_{3i+2} - t_{3i+1} \le d_{\text{max}}$ and therefore $t_{3i+1} + d_{\text{min}} \le t_{3i+2}$.)

By Three Rules Corollary, SetDeadline(x) fires at t_{3i+1} setting Deadline(x) to $\alpha = t_{3i+1}$ + WaitTime. If Dir$_\alpha$ = open then SignalClose(x) fires at α setting Dir to close; regardless, Dir$_{\alpha+}$ = close. By Local SafeToOpen Lemma, $s(x)$ fails over $I = (\alpha, t_{3i+3})$. Hence, over I, SafeToOpen fails, SignalOpen is disabled, Dir = close, and OpenGate is disabled.

By the definition of regular runs, GateStatus = closed at some moment t such that $\alpha < t < \alpha + d_{\text{close}} = t_{3i+1} + \text{WaitTime} + d_{\text{close}} = t_{3i+1} + d_{\text{min}}$. Since OpenGate is disabled over I, GateStatus remains closed over I and therefore over the interval $[t_{3i+1} + d_{\text{min}}, t_{3i+3})$. By Preservation Lemma, GateStatus = closed at t_{3i+3}. □

Let $D_{\text{close}} = d_{\text{close}} + (d_{\text{max}} - d_{\text{min}}) = d_{\text{max}} - \text{WaitTime}$.

Theorem 7.2 ((Liveness Theorem)) *Assume $\alpha + d_{\text{open}} < \beta - D_{\text{close}}$. If the crossing is empty in the open time interval (α, β), then the gate is open in $[\alpha + d_{\text{open}}, \beta - D_{\text{close}}]$. More formally, if every TrackStatus$(x) \neq$ in_crossing over (α, β), then GateStatus = opened over $[\alpha + d_{\text{open}}, \beta - D_{\text{close}}]$.*

Proof By Deadline Lemma, every Deadline$(x) \ge \beta - D_{\text{close}} > \alpha + d_{\text{open}}$ over (α, β). By the definition of SafeToOpen, it holds at α. If Dir$_\alpha$ = close then SignalOpen fires at α; in any case Dir$_{\alpha+}$ = open.

By Deadline Lemma, every Deadline$(x) \ge \beta - D_{\text{close}} > CT$ over $(\alpha, \beta - D_{\text{close}})$. Hence, over $(\alpha, \beta - D_{\text{close}})$, every SignalClose$(x)$ is disabled, Dir remains open, and StartClose is disabled.

By the definition of regular runs, GateStatus = opened at some moment $t \in (\alpha, \alpha + d_{\text{open}})$. Since StartClose is disabled over $(\alpha, \beta - D_{\text{close}})$, GateStatus remains opened over $(t, \beta - D_{\text{close}})$ and therefore is opened over $[\alpha + d_{\text{open}}, \beta - D_{\text{close}})$. By Preservation Lemma, GateStatus = opened at $b - D_{\text{close}}$. □

The next claim shows that, in a sense, Liveness Theorem cannot be improved.

Claim 7.1

1. *Liveness Theorem fails if d_{open} is replaced with a smaller constant.*
2. *Liveness Theorem fails if D_{close} is replaced with a smaller constant.*

Proof The first statement holds because the gate can take time arbitrarily close to d_{open} to open. The second statement holds for two reasons. Recall that $D_{\text{close}} = d_{\text{close}} + (d_{\text{max}} - d_{\text{min}})$. The term $(d_{\text{max}} - d_{\text{min}})$ cannot be reduced; to be on the safe side, the controller must act as if every oncoming train is moving as fast as possible, even if it is moving as slow as possible. The term d_{close} cannot be reduced either; the gate can take arbitrarily short periods of time to close. Now we give a more detailed proof.

Part 1. Given some constant $c_{\text{open}} < d_{\text{open}}$, we construct a regular run of our ealgebra $\mathcal{A}$ and exhibit an open interval $I = (\alpha, \beta)$ such that the crossing is empty during I but the gate is not opened during a part of interval $(\alpha + c_{\text{open}}, \beta - D_{\text{close}})$.

We assume that $d_{\text{open}}, D_{\text{close}} < 1$ (just choose the unit of time appropriately) and that there is only one track.

The traffic. Only one train goes through the crossing. It appears at time 100, reaches the crossing at time $100 + d_{\max}$ and leaves the crossing at time $110 + d_{\max}$, so that Dir should be changed only twice: set to close at $100 + \text{WaitTime}$ and set to open at $110 + d_{\max}$.

The run. We don't care how quickly the gate closes, but we stipulate that the time Δ that the gate takes to open belongs to $(c_{\text{open}}, d_{\text{open}})$.

The interval I: $(110 + d_{\max}, 110 + d_{\max} + d_{\text{open}})$.

Since the only train leaves the crossing at $110 + d_{\max}$, the crossing is empty during I. However the gate takes time $\Delta > c_{\text{open}}$ to open and thus is not opened during the part $(110 + d_{\max} + c_{\text{open}}, 110 + d_{\max} + \Delta)$ of I.

Part 2. Given some constant $C_{\text{close}} < D_{\text{close}}$, we construct a regular run of our ealgebra $\mathcal{A}$ and exhibit an open interval $I = (\alpha, \beta)$ such that the crossing is empty during I but the gate is not opened (even closed) during a part of interval $(\alpha + d_{\text{open}}, \beta - C_{\text{close}})$.

We assume that $d_{\text{open}}, C_{\text{close}} < 1$, and that there is only one track with the same traffic pattern as in part 1.

The run. This time we don't care how quickly the gate opens, but we stipulate that the time Δ that the gate takes to close satisfies the following condition:

$$0 < \Delta < \min\{d_{\text{close}}, D_{\text{close}} - C_{\text{close}}\}.$$

The interval I is $(0, 100 + d_{\max})$, so that $\alpha = 0$ and $\beta = 100 + d_{\max}$.

Since the only train reaches the crossing at $100 + d_{\max}$, the crossing is empty during I. The gate is closed by $100 + \text{WaitTime} + \Delta$ and is closed during the part $(100 + \text{WaitTime} + \Delta, 100 + \text{WaitTime} + (D_{\text{close}} - C_{\text{close}}))$ of interval $(\alpha + d_{\text{open}}, \beta - C_{\text{close}})$. Let us check that $(100 + \text{WaitTime} + \Delta, 100 + \text{WaitTime} + (D_{\text{close}} - C_{\text{close}})$ is indeed a part of $(\alpha + d_{\text{open}}, \beta - C_{\text{close}})$. Clearly, $\alpha + d_{\text{open}} < 0 + 1 < 100 + \text{WaitTime} + \Delta$. Further:

$$\begin{aligned} & 100 + \text{WaitTime} + \Delta \\ < \; & 100 + \text{WaitTime} + (D_{\text{close}} - C_{\text{close}}) \\ = \; & 100 + (d_{\min} - d_{\text{close}}) + [(d_{\text{close}} + d_{\max} - d_{\min}) - C_{\text{close}}] = \beta - C_{\text{close}}. \end{aligned}$$

□

8 Some Additional Properties

Theorem 8.1 ((Uninterrupted Closing Theorem)) *The closing of the gate is never interrupted. More formally, if Dir is set to close at some moment α, then Dir = close over the interval $I = (\alpha, \alpha + d_{\text{close}})$.*

Recall that, by the definition of regular runs, GateStatus = closed somewhere in I if Dir = close over I.

Proof Since Dir is set to close at α, some SignalClose(x) fires at α. Fix such an x and let $t_0 < t_1 < \ldots$ be the significant moments of track x. By Three Rules Corollary, there is an i such that $\alpha = t_{3i+1} + \text{WaitTime} = t_{3i+1} + d_{\min} - d_{\text{close}}$. Then $\alpha + d_{\text{close}} = t_{3i+1} + d_{\min} \leq t_{3i+2}$. By the definition of regular runs, TrackStatus(x) = coming over I. By Deadline Theorem, Deadline(x) = α over I, so that $\text{CT} + d_{\text{open}} > \text{CT} > \text{Deadline}(x)$ over I. Because of this x, SafeToOpen fails over I and therefore SignalOpen is disabled over I. Thus Dir = close over I.

Theorem 8.2 ((Uninterrupted Opening Theorem)) *Suppose WaitTime $\geq d_{\text{open}}$; that is, $d_{\min} \geq d_{\text{close}} + d_{\text{open}}$. Then the opening of the gate is not interrupted; in other words, if Dir is set to open at some moment α, then Dir = open over the interval $I = (\alpha, \alpha + d_{\text{open}})$.*

Recall that, by the definition of regular runs, GateStatus = opened somewhere in I if Dir = open over I.

Proof It suffices to prove that every SignalClose(x) is disabled over I. Pick any x and let $t_0 < t_1 < \ldots$ be the significant moments of track x. Since Dir is set to open at α, SignalOpen fires at α, SafeToOpen holds at α, and $s(x)$ holds at α. We have two cases.

Case 1. $\alpha + d_{\text{open}} < \text{Deadline}(x)_\alpha < \infty$. Since $\text{Deadline}(x)_\alpha < \infty$, $\tau_{3i+1} < \alpha \leq t_{3i+3}$ and $\text{Deadline}(x)_\alpha = t_{3i+1} + \text{WaitTime}$ for some i (by Deadline Lemma). We have

$$\alpha + d_{\text{open}} < \text{Deadline}(x)_\alpha = t_{3i+1} + \text{WaitTime} < t_{3i+1} + d_{\min} \leq t_{3i+2} < t_{3i+3}.$$

By Deadline Lemma, $\text{Deadline}(x)$ does not change in I, so that CT remains $< \text{Deadline}(x)$ in I and therefore $\text{SignalClose}(x)$ is disabled over I.

Case 2. $\alpha + d_{\text{open}} \geq \text{Deadline}_\alpha(x)$ or $\text{Deadline}_\alpha(x) = \infty$.

We check that $t_{3i} \leq \alpha \leq t_{3i+1}$ for some i. Indeed, if $\text{TrackStatus}(x)_\alpha = \text{empty}$ then $t_{3i} \leq \alpha < t_{3i+1}$ for some i. Suppose that $\text{TrackStatus}(x)_\alpha \neq \text{empty}$. Since $s(x)$ holds at a, $\alpha + d_{\text{open}} < \text{Deadline}_\alpha(x)$. By the condition of Case 2, $\text{Deadline}(x)_\alpha = \infty$. Recall that $\text{TrackStatus}(x) \neq \text{empty}$ exactly in intervals $[t_{3i+1}, t_{3i+3}$ and $\text{Deadline}(x) = \infty$ exactly in periods $(t_{3i}, t_{3i+1}]$. Thus $\alpha = t_{3i+1}$ for some i.

The first moment after α that $\text{SignalClose}(x)$ is enabled is $t_{3i+1} + \text{WaitTime}$. Thus it suffices to check that $\alpha + d_{\text{open}} \leq t_{3i+1} + \text{WaitTime}$. Since $d_{\min} \geq d_{\text{close}} + d_{\text{open}}$, we have

$$\alpha + d_{\text{open}} \leq t_{3i+1} + d_{\text{open}} \leq t_{3i+1} + (d_{\min} - d_{\text{close}}) = t_{3i+1} + \text{WaitTime}. \square$$

Corollary 8.1 ((Dir and GateStatus Corollary)) *Assume $d_{\min} \geq d_{\text{close}} + d_{\text{open}}$.*

1. *If the sequence $\gamma_1 < \gamma_2 < \gamma_3 < \ldots$ of positive significant moments of Dir is infinite, then the sequence $\delta_1 < \delta_2 < \delta_3 < \ldots$ of positive significant moments of GateStatus is infinite and each $\delta_i \in (\gamma_i, \gamma_{i+1})$.*
2. *If the positive significant moments of Dir form a finite sequence $\gamma_1 < \gamma_2 < \ldots < \gamma_n$, then the positive significant moments of GateStatus form a sequence $\delta_1 < \delta_2 < \ldots < \delta_n$ such that $\delta_i \in (\gamma_i, \gamma_{i+1})$ for all $i < n$ and $\delta_n > \gamma_n$.*

Proof We prove only the first claim; the second claim is proved similarly.

Since Dir = open and GateStatus = opened initially, GateStatus does not change in $(0, \gamma_1)$. Suppose that we have proved that if $\gamma_1 < \ldots < \gamma_j$ are the first j positive significant moments of Dir, then there are exactly $j-1$ significant moments $\delta_1 < \ldots < \delta_{j-1}$ of GateStatus in $(0, g_j]$ and each $\delta_i \in (\gamma_i, \gamma_{i+1})$. We restrict attention to the case when j is even; the case of odd j is similar. Since j is even, Dir is set to open at γ_j. If γ_j is the last significant moment of Dir, then the gate will open at some time in $(\gamma_j, \gamma_j + d_{\text{open}})$ and will stay open forever after that. Otherwise, let $k = j + 1$. By Uninterrupted Opening Theorem, the gate opens at some moment $\delta_j \in (\gamma_j, \gamma_k)$. Since Dir remains open in (δ_j, γ_k), GateStatus = opened holds over (δ_j, γ_k). By Preservation Lemma, GateStatus = opened at γ_k. $\square$

9 Existence of Regular Runs

We delayed the existence issue in order to take advantage of Sect. 8. For simplicity, we restrict attention to an easier but seemingly more important case when $d_{\min} \geq d_{\text{close}} + d_{\text{open}}$. The Existence Theorem and the two Claims proved in this section remain true in the case $d_{\min} < d_{\text{close}} + d_{\text{open}}$; we provide remarks explaining the necessary changes.

Let $\Upsilon_1 = \Upsilon - \{\text{GateStatus}\}$, and $\Upsilon_0 = \Upsilon_1 - \{\text{Deadline}, \text{Dir}\}$. For $i = 0, 1$, let $\Upsilon_i^+ = \Upsilon_i \cup \{\text{CT}\}$.

Theorem 9.1 ((Existence Theorem)) *Let P be a pre-run of vocabulary Υ_0 satisfying the train motion requirement in the definition of regular runs, and let A be an initial state of $\mathcal{A}$ consistent with $P(0)$. There is a regular run R of $\mathcal{A}$ which starts with A and agrees with P.*

Proof Let the significant moments of P be $0 = \alpha_0 < \alpha_1 < \ldots$. For simplicity, we consider only the case where this sequence is infinite. The case when the sequence is finite is similar. Our construction proceeds in two phases. In the first phase, we construct a run Q of module CONTROLLER (that is of the corresponding one-module evolving algebra of vocabulary Υ_1^+) consistent with A and P. In the second phase, we construct the desired R by extending Q to include the execution of module GATE.

Phase 1: Constructing Q from P. Let $\beta_0 < \beta_1 < \ldots$ be the sequence that comprises the moments α_i and the moments of the form $t+\text{WaitTime}$ where t is a moment when some TrackStatus(x) becomes coming. By Three Rule and SignalOpen Corollaries, these are exactly the significant moments of the desired Q. We define the desired Q by induction on β_i. It is easy to see that $Q(T)$ is uniquely defined by its reduct $q(t)$ to Υ_1.

$Q(0)$ is the appropriate reduct of A. Suppose that Q is defined over $[0, \beta_j]$ and $k = j+1$. Let γ range over (β_j, β_k). If CONTROLLER does not execute at β_j, define $q(\gamma) = q(\beta_j)$; otherwise let $q(\gamma)$ e the state resulting from executing CONTROLLER at $q(\beta_j)$. Define $q(\beta_k)$ to agree with $q(\gamma)$ at all functions except TrackStatus, where it agrees with $P(\beta_k)$.

Clearly Q is a pre-run. It is easy to check that Q is a run of CONTROLLER and that CONTROLLER is immediate in Q.

Phase 2: Constructing R from Q. We construct R by expanding Q to include GateStatus. Let $\gamma_1 < \gamma_2 < \ldots$ be the sequence of significant moments of Q at which Dir changes. Thus Dir becomes close at moments γ_i where i is odd, and becomes open at moments γ_i where i is even.

There are many possible ways of extending Q depending on how long it takes to perform a given change in GateStatus. Chose a sequence $a_1, a_2, \ldots$ of reals such that (i) $a_i < \gamma_{i+1} - \gamma_i$ and (ii) $a_i < d_{\text{close}}$ if i is odd and $a_i < d_{\text{open}}$ if i is even. The idea is that GATE will delay executing OpenGate or CloseGate for time a_i.

The construction proceeds by induction on γ_i. After i steps, GateStatus will be defined over $[0, g_i]$, and GateStatus$_{g_i}$ will equal opened if i is odd and will equal closed otherwise.

Set GateStatus = opened over $[0, \gamma_1]$. Suppose that GateStatus is defined over $[0, \gamma_i]$ and let $j = i + 1$. We consider only the case when i is even. The case of odd i is similar.

By the induction hypothesis, GateStatus = closed at γ_i. Since i is even, Dir is set to open at γ_i. Define GateStatus = closed over $(\gamma_i, \gamma_i + a_i]$ and opened over $(\gamma_i + a_i, \gamma_j]$.

It is easy to see that R is a regular run of $\mathcal{A}$. □

Remark. If the assumption $d_{\min} \geq d_{\text{close}} + d_{\text{open}}$ is removed, Phase 1 of the construction does not change but Phase 2 becomes more complicated. After i steps, GateStatus is defined over $[0, g_i]$, and GateStatus$_{g_i}$ = closed if i is even; it cannot be guaranteed that GateStatus$_{g_i}$ = opened if i is odd. The first step is as above. For an even i, we have three cases.

Case 1: $a_i < \gamma_j - \gamma_i$. Define GateStatus over $(g_i, g_j]$ as in the Existence Theorem Proof.

Case 2: $a_i > \gamma_j - \gamma_i$. Define GateStatus = closed over $(g_i, g_j]$.

Case 3: $a_i = \gamma_j - \gamma_i$. Define GateStatus = closed over $(g_i, g_j]$ as in sub-case 2 but also mark g_j (to indicate that OpenGate should fire at γ_j).

For an odd i, we have two cases.

Case 1: Either GateStatus = opened at γ_i or else GateStatus = closed at g_i but g_i is marked. Define GateStatus over $(g_i, g_j]$ as in the Existence Theorem Proof.

Case 2: GateStatus = closed at γ_i and γ_i is not marked. Ignore a_i and define GateStatus = closed over $(g_i, g_j]$.

Claim 9.1 ((Uniqueness of Control)) *There is only one run of* CONTROLLER *consistent with A and P.*

Proof Intuitively, the claim is true because the construction of Q was deterministic: we had no choice in determining the significant moments of Q. More formally, assume by reductio ad absurdum that Q_1, Q_2 are runs of CONTROLLER consistent with A and P and the set $D = \{t : Q_1(t) \neq Q_2(t)\}$ is non-empty. Let $\tau = \inf(D)$. Since both Q_1 and Q_2 agree with A, $\tau > 0$. By the choice of τ, Q_1 and Q_2 agree over $[0, \tau)$. Since both Q_1 and Q_2 agree with A and P, they can differ only at internal functions; let q_1, q_2 be reductions of Q_1, Q_2 respectively to the internal part of the vocabulary. By Preservation Lemma, q_1 and q_2 coincide at τ. But the values of internal functions at $\tau+$ are completely defined by the state at t. Thus q_1 and q_2 coincide at $\tau+$ and therefore Q_1, Q_2 coincide over some nonempty interval $[\tau, \tau + \varepsilon)$. This contradicts the definition of τ. □

Claim 9.2 ((Universality of Construction)) *Let R' be any regular run of the ealgebra consistent with A and P. In the proof of Existence Theorem, the sequence $a_1, a_2, \ldots$ can be chosen in such a way that the regular run R constructed there coincides with R'.*

Proof By Uniqueness of Control Claim, the reducts of R and R' to Υ_1^+ coincide. The moments $\gamma_1 < \gamma_2 < \ldots$ when Dir changes in R are exactly the same moments when Dir changes in R'. We have only to construct appropriate constants a_i.

Let $\delta_1 < \delta_2 < \ldots$ be the significant moments of GateStatus in R'. With respect to Dir and GateStatus Corollary, define $a_i = \delta_i - \gamma_i$. It is easy to check that $R = R'$. □

Remark. If the assumption $d_{\text{min}} \geq \text{close} + d_{\text{open}}$ is removed, the proof of Uniqueness of Control Claim does not change but the proof of Universality of Construction Claim becomes slightly complicated. Let $j = i + 1$. For an even i, we have two cases.

Case 1: $\delta_i \leq \gamma_j$. Define $a_i = \delta_i - \gamma_i$.

Case 2: $\delta_i > \gamma_j$. In this case $\gamma_j - \gamma_i < d_{\text{open}}$. The exact value of a_i is irrelevant; it is only important that $a_i \in (\gamma_j - \gamma_i, d_{\text{open}})$. Choose such an a_i arbitrarily.

For an odd i, we also have two cases.

Case 1: In R', either GateStatus = opened at γ_i or else GateStatus = closed at γ_i but OpenGate fires at γ_i. Define $a_i = \delta_i - \gamma_i$.

Case 2: In R', GateStatus = closed at γ_i. The exact value of a_i is irrelevant; it is only important that $a_i < d_{\text{close}}$. Choose such an a_i arbitrarily.

References

[1] Egon Börger, Annotated Bibliography on Evolving Algebras, in "Specification and Validation Methods", ed. E. Börger, Oxford University Press, 1995, 37–51.

[2] Egon Börger, Yuri Gurevich and Dean Rosenzweig: The Bakery Algorithm: Yet Another Specification and Verification, in "Specification and Validation Methods", ed. E. Börger, Oxford University Press, 1995.

[3] Yuri Gurevich, "Evolving Algebra 1993: Lipari Guide", in "Specification and Validation Methods", Ed. E. Börger, Oxford University Press, 1995, 9–36.

[4] Yuri Gurevich and James K. Huggins, "The Railroad Crossing Problem: An Evolving Algebra Solution," LITP 95/63, Janvier 1996, Centre National de la Recherche Scientifique Paris, France.

[5] Yuri Gurevich, James K. Huggins, and Raghu Mani, "The Generalized Railroad Crossing Problem: An Evolving Algebra Based Solution," University of Michigan EECS Department Technical Report CSE-TR-230-95.

[6] Constance Heitmeyer and Nancy Lynch: The Generalized Railroad Crossing: A Case Study in Formal Verification of Real-Time Systems, Proc., Real-Time Systems Symp., San Juan, Puerto Rico, Dec., 1994, IEEE.

[7] Ernst-Rüdiger Olderog, Anders P. Ravn and Jens Ulrik Skakkebaek, "Refining System Requirements to Program Specifications", to appear.

[8] Robin Milner. A private discussion, Aug. 1994.