Formalizing and Executing Message Sequence Charts via Timed Rewriting

P. Kosiuczenko, M.Wirsing

Institut für Informatik,
Ludwig-Maximilians-Universität,
München, Germany

Abstract

Message Sequence Charts (MSC) is a graphical trace language for describing and specifying the communication behaviour of distributed systems by means of message interchange. (Timed) Maude is a formal object-oriented specification language which combines algebraic specification techniques for describing complex data structures with (timed) term rewriting to deal with dynamic behaviour. In this paper we show first how to formalize MSC in Timed Maude. Then we give a translation of timed rewriting to untimed rewrite systems and use this translation to execute Message Sequence Charts with the Elan system, a powerful tool which combines Rewriting Logic with a language of rewriting strategies. We illustrate our approach with the benchmark example of a railroad crossing.

1 Introduction

Currently used methods in software engineering employ diagrammatic notations such as object diagrams, scenarios, Message Sequence Charts (MSC), or UML since their graphical representations facilitate reading, understanding, and the design of systems. Nevertheless these notations suffer from semantical problems, since their exact meaning is often unclear. Recently much effort has been made to overcome some of the semantical problems by applying formal techniques to methods used in practice of software engineering (e.g. [13]). This paper is another step in this direction. We show how to formalize Message Sequence Charts in the executable specification language Timed Maude and how to execute it with the Elan system.

1. This research has been partially sponsored by Bayerischer Forschungsverbund FORSOFT.
Maude [2] is a formal object-oriented specification language which combines algebraic specification techniques for describing complex data structures with term rewriting to deal with dynamic behaviour. It is based on so-called Rewriting Logic (RL) [9] and has a very efficient implementation. In Maude communication of system components is handled by interchanging messages. Timed Maude [7] is a real-time extension of Maude which adds a time view to the functional and the process view of Maude. It is based on Timed Rewriting Logic (TRL) in the same way as Maude is based on Rewriting Logic. TRL specifications with discrete time can be automatically translated into untimed RL specifications, as we show in this paper.

MSC [4] is a graphical trace language for describing and specifying the communication behaviour of a distributed system by means of message interchange. It extends interaction diagrams by several constructs allowing to compose different diagrams. The language has been recommended as a standard by the International Telecommunication Union. A Maude like model can be used to provide a semantics for MSC [6]. In this paper we show how a basic MSC can be translated into Timed Maude following [5, 6]. We illustrate our approach with the benchmark railroad crossing example and present its formal specification.

Elan [1] is a powerful tool which combines Rewriting Logic with a language of rewriting strategies. It allows to describe and automatically execute rule-based logical systems or processes. It provides also a modularisation mechanism for user-defined modules. We use Elan for executing MSC diagrams by translating their TRL semantics to Elan based on the interpretation of TRL in RL. We show how to execute, analyze and improve the MSC specification of the railroad crossing using Elan.

Our approach has several advantages: it provides a direct translation of MSC’s to timed rewrite specifications, it allows for compositional specifications by means of the composition operators [4, 5], and makes available efficient tool support via powerful term rewriting systems like Elan or Maude.

2 Formal Background.

Timed Rewriting Logic (TRL) [7] is an extension of Rewriting Logic (RL) [9] for describing real-time systems. It models time by archimedean monoids. Time evolves by executing rewrite rules. Each rule is labeled by a time stamp indicating the time needed for executing a rewrite step. The inference rules of TRL are similar to those of RL. The major difference to RL is the existence of time stamps, the synchronous replacement rule and restricted form of (i.e. 0-time) reflexivity.

A TRL-specification is a pair of the form (Σ(Rₜ), Ax), where Σ(Rₜ) is a signature containing proper sorts and operation symbols and Ax is a set of equations and timed rewrite rules of the form: \( t₁ \rightarrow a \rightarrow t₂ \), where a is a label denoting an action or a system step, r
is a time stamp belonging to the underlying arithmetical monoid $R_+^2$, and $t_1, t_2$ are $\Sigma$-terms coding system states (configurations). TRL has the following deduction rules (plus rules for equational reasoning):

1. **Timed Transitivity (TT).** For each $t_1, t_2, t_3 \in T(\Sigma,X)$, $r_1, r_2 \in R_+$

   \[
   t_1 \rightarrow a \; r_1 \rightarrow t_2, \quad t_2 \rightarrow b \; r_2 \rightarrow t_3
   \]

   \[
   t_1 \rightarrow a; b \; r_1 + r_2 \rightarrow t_3
   \]

2. **Synchronous Replacement (SR).** Let $\{x_{i_1}, \ldots, x_{i_k}\} = \text{FV}(t_0) \cap \text{FV}(u_0)$ be the intersection of the free variables of $t_0$ and $u_0$. For each $t_0, \ldots, t_n, u_0, u_1, \ldots, u_n \in T(\Sigma,X)$, $r \in R_+$

   \[
   t_0 \rightarrow a \; r \rightarrow u_0, \quad t_{i_1} \rightarrow a_{i_1} \; r \rightarrow u_{i_1}, \ldots, t_{i_k} \rightarrow a_{i_k} \; r \rightarrow u_{i_k}
   \]

   \[
   t_0(t_1, \ldots, t_n) \rightarrow a(a_{i_1}, \ldots, a_{i_k}) \; r \rightarrow u_0(u_1, \ldots, u_n)
   \]

3. **Timed Compatibility with = (TCE).** For each $t_1, t_2, u_1, u_2 \in T(\Sigma,X)$, $r_1, r_2 \in R_+$

   \[
   t_1 = u_1, \quad r_1 = r_2, \quad u_1 \rightarrow a \; r_1 \rightarrow u_2, \quad u_2 = t_2
   \]

   \[
   t_1 \rightarrow a \; r_2 \rightarrow t_2
   \]

4. **0-time Reflexivity (0-R).** For each $t \in T(\Sigma,X)$

   \[
   t \rightarrow 0 \rightarrow t
   \]

A term $t$ is called **static** if it does not change over time, i.e. for all $r$, $t \rightarrow t \; r \rightarrow t$ holds, and if $t \rightarrow a \; r \rightarrow t'$ holds for some $r$, then $t = t'$. If a term $t$ is not static then we call it **dynamic** and denote it by: op dynamic $t$ (see the applications section). Similarly, a sort $s$ is **static** iff it contains static terms only, otherwise it is called **dynamic**.

Timed Maude is a formalism for specifying object-oriented distributed systems based on TRL. It borrows the object-oriented concepts of Maude. An **object class** is declared by an identifier and a list of attributes and their types. An **object** is represented by a term comprising a unique object name, an identifier for the class the object belongs to, and a set of attributes with their values. We will use capital letters for names and the corresponding small letters for objects being in certain states denoted by constants; e.g. the term $tg = <TG : TrainGate | state : up>$ represents an object $tg$ with name $TG$ belonging to the class $TrainGate$, and the attribute state has value up. In the following we often omit the class and the name of the attribute and write simply $<TG | up>$.

---

2. Examples of arithmetical monoids are discrete time models like natural numbers as well as continuous time models like reals.
A message $m$ is a term of the form $(X, dt, Y)$ that consists of object names $X, Y$ of sort names and data $dt$ of sort data carried by the message. $X$ and $Y$ can be understood as the sender and the receiver address, respectively. A configuration is a multiset (or bag) of messages and objects. Formally, configuration $c$ is denoted by a term of the form:

$$m_1 \otimes \ldots \otimes m_k \otimes o_1 \otimes \ldots \otimes o_l$$

(shortly written: $m_1 \ldots m_k o_1 \ldots o_l$) where $\otimes$ is a binary function symbol denoting multiset union. A Timed Maude program makes computational progress by rewriting its state. A rewrite step has the form

$$m_1 \ldots m_k o_1 \ldots o_l \rightarrow a r \rightarrow n_1 \ldots n_p o_1 \ldots o_w.$$ 

The rule says, that after receiving messages $m_1 \ldots m_k$ the objects which occur on both sides of the rule change their states, objects which do not appear on the right hand side are deleted, objects which do not appear on the left hand side are created, and messages $n_1 \ldots n_p$ are sent; all these happens in time $r$.

3 Interpreting TRL in RL.

In this section we show how to translate (timed) TRL specifications to (untimed) RL specifications. In principle one could choose also any other term rewriting formalism, but RL is specially convenient due to its conceptual similarity to TRL. The translation [11] works for all linear TRL specifications with discrete time and therefore can be applied to more general TRL specifications than the first translation of TRL to RL by Olveczky and Meseguer [10].

3.1 Definition.

Let $\mathcal{Sp} = (\Sigma(R_+), Ax)$ be an arbitrary TRL specification such that $Ax$ contains rewrite rules only and each rewrite rule has time stamp 0 or 1. We define the RL interpretation $\text{Int}(\mathcal{Sp}) = (\text{Int}(\Sigma(R_+)), \text{Int}(Ax))$ as follows:

The signature $\text{Int}(\Sigma(R_+))$ extends $\Sigma(R_+)$ with the following new (polymorphic) function symbols for every dynamic sort $s$:

$$\text{Go} : s \rightarrow s, \quad \text{D} : s \rightarrow s,$$

$$\text{clean} : s \rightarrow \text{Bool}, \quad \text{[]}(-) : \text{Time}, s \rightarrow s.$$ 

The function $\text{Go}$ (Go stands for go) is used to mark dynamic arguments where time must progress. The function $\text{D}$ is used to mark where time has already progressed (D stands for done). These functions allow to synchronize progression of time in a term. The function clean ensures that a term is ground and does not contain Go or D. The first argument of the function $\text{[]}(-)$ indicates the time available for executing the specification.

3. X, Y may be also gate names, see [8].
The rewrite theory $\text{Int}(\text{Ax})$ contains the following rules: The rules (i) allow us to simulate the 0-time rules on terms which do not contain $\text{Go}$ or $D$; this is assured by the clean function. The cleanness condition is needed to ensure that the RL rules modelling 0-time rewrites are not applied to a term part way through a time progression as this could lead potentially to a deadlock. The rules (ii) allow us to propagate time down through a term, in a sense. The meta-level term mapping $\Gamma$ is used to describe how time progresses when the term is rewritten; it leaves static terms unchanged whereas dynamic terms receive new markings: $\text{Go}$ if the time has to propagate down, and $D$ if we are ready with a subterm. Rule (iii) serves to pull up the $D$ function using another term mapping called $\Delta$.

(i) For each 0-time rule $t \rightarrow t' \in \text{Ax}$ we have the corresponding rule:

$$t \rightarrow t' \text{ if } \text{clean}(t),$$

(ii) For each rule $t \rightarrow t' \in \text{Ax}$ we have the corresponding conditional rule:

$$\text{Go}(t) \rightarrow \Gamma(t'),$$

(iii) For each function symbol $f : s_1, \ldots, s_n \rightarrow s \in \Sigma(\text{R}_+)$ we have a rule to pull up the $D$ function:

$$f(\Delta(x_1), \ldots, \Delta(x_n)) \rightarrow D(f(x_1, \ldots, x_n))$$

(iv) Finally, for each dynamic sort $s$ and a variable $x$ of sort $s$ we have a rule to initiate the next time step:

$$[n+1](D(x)) \rightarrow [n](\text{Go}(x)).$$

The clean function is axiomatized by the following equations:

$$\text{clean}(\text{Go}(x)) = \text{False}, \quad \text{clean}(D(x)) = \text{False},$$

$$\text{clean}(c) = \text{True}, \quad \text{clean}(f(x_1, \ldots, x_n)) = \text{clean}(x_1) & \ldots & \text{clean}(x_n),$$

for each constant symbol $c \in \Sigma(\text{R}_+$), and each n-ary function symbol $f \in \Sigma(\text{R}_+)$. The term mapping $\Gamma : T(\Sigma(\text{R}_+), X) \rightarrow T(\text{Int}(\Sigma(\text{R}_+)), X)$ is defined as follows:

1. $\Gamma(t) = t$ if $t$ is a member of a static sort;
2. $\Gamma(t) = D(t)$ if the term $t$ is a member of a dynamic sort but contains no variables of dynamic sorts;
3. $\Gamma(t) = \text{Go}(x)$ if $t \equiv x$ and $x$ is a dynamic variable ($\equiv$ is the syntactic identity);
4. $\Gamma(t) = f(\Gamma(t_1), \ldots, \Gamma(t_n))$ if $t \equiv f(t_1, \ldots, t_n)$, and $t$ is of dynamic sort and contains variables of dynamic sorts.

The mapping $\Delta : X \rightarrow T(\text{Int}(\Sigma(\text{R}_+)), X)$ is defined by

$$\Delta(x) = x$$

if $x$ is of a static sort, else $\Delta(x) = D(x)$.

In [11] it has been shown that the interpretation of TRL in RL is correct in the sense that a timed rewrite formula without variables is deducible from a TRL specification, which is linear with respect to dynamic variables, if and only if its interpretation is deducible in RL from the untimed interpretation of the timed specification.
4 Formalization of MSC.

MSC is a trace language for description and specification of communication behaviour of system components and their environment by means of message interchange [4]. There is an interesting relation between Timed Maude and MSC, in particular, there exists a formal, Maude like semantics of MSC-96 [5, 6, 8].

Basically, a MSC-diagram describes a system behaviour in the following way [4]: the behaviour of an object (or instance) is represented by a vertical line defining a total ordering of its actions. Such a vertical line starts with an empty box and ends with a black box. If a message is being sent from one object to another, then this is indicated by a horizontal arrow directed from the sender to the receiver. Message passing is asynchronous; therefore it corresponds to two events: sending and receiving the message. One may specify the time a message deliverance will take by giving a real number under the horizontal arrow. The initial (final) states of an MSC are the states which occur directly after (before, resp.) an empty (black, resp.) box. There are only three kinds of atomic steps an MSC instance can execute: sending or receiving a message, creating or stopping another object, and a special action which changes only the state of an object.

Basic MSC can be formalized in Timed Maude as follows: For any instance O of a MSC diagram we introduce an object of appropriate class which contains an attribute called state, i.e. \(<O : \text{Class} | \text{state} : s \>\). For simplicity, states will always be denoted by constants in this paper. Different object actions are separated by intermediate states. Object states may be described using (local) conditions denoted by hexagons (see below). Any message sent is split into a send and a receive part (indicated by “!” and “?” resp.). In general the underlying set L of atomic steps is of the form \{?, m, ! n, start, stop, τ, ac_1, ..., ac_n\} where “start” stands for creating an object, “stop” for deleting an object, “τ” for time elapse, and “ac_1”, ..., “ac_n” stand for special actions. Finally, we introduce a rewrite rule for any action as follows:
The object named O sends message m and changes its state from s to s' in 0-time
\[
\text{rl} \quad \langle O | \text{state} : s \rangle \xrightarrow{! m} 0 \rightarrow \langle O | \text{state} : s' \rangle m.
\]

**the object named O sends message m and changes its state from s to s' in 0-time

The object named O receives message m and changes its state from s to s' in 0-time
\[
\text{rl} \quad m \langle O | \text{state} : s \rangle \xrightarrow{? m} 0 \rightarrow \langle O | \text{state} : s' \rangle.
\]

**the object named O receives message m and changes its state from s to s' in 0-time

The object named O is created in 0-time
\[
\text{rl} \quad \text{start}(o) \rightarrow 0 \rightarrow o.
\]

**the object named O is created in 0-time

The object o is deleted in 0-time
\[
\text{rl} \quad \text{stop}(o) \rightarrow 0 \rightarrow \varepsilon.
\]

**the object o is deleted in 0-time

The object named O performs the special action aci changing its state in 0-time
\[
\text{rl} \quad \langle O | \text{state} : s \rangle \xrightarrow{\text{aci}} 0 \rightarrow \langle O | \text{state} : s' \rangle.
\]

**the object named O performs the special action aci changing its state in 0-time

The value of timer t_timer is decreased by r1 in time r1
\[
\text{rl} \quad \text{t} \_\text{timer}(r_1 + r_2) \xrightarrow{\tau} r_1 \rightarrow \text{t} \_\text{timer}(r_2).
\]

**the value of timer t_timer is decreased by r1 in time r1

In all but the last case we say that the object named O performs the action a \in L (e.g. TG performs the action ?m_con). Unless stated otherwise, we will consider actions of the form \( t_1 \otimes \ldots \otimes t_n \otimes a \), \( a \in L \), and of the form \( t_1(\tau) \otimes \ldots \otimes t_n(\tau) \) only for \( t_i \in T(\Sigma, X) \), i.e. actions performed by one object only or rules where all components perform time step \( \tau \). For simplicity, we will write “a” for the former and “\( \tau \)” for the latter.

The semantics of MSC’s time aspects is based on the following assumptions (cf [5]): All terms are static per default, unless stated otherwise. All actions \( a \in L \), except of \( \tau \), take 0-time. Time constraints imposed on an object behaviour or a message deliverance are modeled by timers attached to object states or to messages, respectively. An object can spawn multiple timers running in parallel. An atomic state (i.e. state without timers) can be declared either static or dynamic. A static state can last arbitrarily long, whereas a dynamic state is supposed to change immediately (i.e. its duration is 0-time units). In this paper we assume that the equational axioms concern only data carried by messages and the multisum operator \( \otimes \). Formally, we define:

4.1 Definition.
A MSC-specification \( S^{MP} \) is a tuple of the form \( (\Sigma(R_+), A x, I n, F i n) \), where

• \( (\Sigma(R_+), A x) \) is a TRL-specification\(^5\) such that the underlying set of atomic labels as well as rewrite rules and axioms forming \( A x \) have the form described above;

• \( I n \) is a set of objects in \textit{initial} states and \( F i n \) is a set of objects in \textit{terminal} states, both kinds of states are atomic (i.e. denoted by constants).

---

4. \( \varepsilon \) denotes the empty configuration.
5. For the sake of simplicity we treat here (flat) TRL-specifications instead of Timed Maude specifications in general.
• Objects having different names must have different states.

A configuration \( c = o_{i1} \ldots o_{ik} \) is called initial (final, resp.), if \( \text{In} = \{o_{i1}, \ldots, o_{ik}\} \) (\( \text{Fin} = \{o_{i1}, \ldots, o_{ik}\}, \) resp.).

4.2 Definition.

Let \( SP \) be a MSC specification. A partial trajectory is a finite sequence of configurations, actions, and time values \( c_0 a_1 r_1 c_1 \ldots c_{n-1} a_n r_n c_n \) such that \( c_0 \) is the initial configuration and

\[
\begin{align*}
    c_i &\rightarrow a_{i+1} r_{i+1} \rightarrow c_{i+1}, \text{ for } i = 0, \ldots, n - 1 
\end{align*}
\]

A partial trajectory is a trajectory if in addition \( c_n \) is the final configuration. A sequence \( a_1 r_1 a_2 r_2 \ldots a_n r_n \) is a (partial) trace of \( SP \) iff there exists a (partial) trajectory \( tr \) of the form \( c_0 a_1 r_1 c_1 \ldots c_{n-1} a_n r_n c_n \).

Let us mention that the semantics allows for compositional specification using composition operations like vertical and horizontal composition [4]. The first corresponds to the so called weak sequencing; the second is a kind of parallel composition allowing to specify system composed of components working in parallel [8].

5 Application: Railroad Crossing.

We illustrate our approach with the benchmark example of railroad crossing (e.g. [3]). First we present the informal (textual) specification, then its graphical and formal counterparts, and finally its translation to Elan [1].

5.1 Informal Description.

A railroad crossing consists of train gate TG and a train track T which has sensors attached to it. (see the figure). Trains are moving along a track. When a sensor detects an incoming train on the track it sends immediately the message \( m_{\text{com}} \) to the train gate. When a sensor detects that a train has passed the crossing it sends immediately the message \( m_{\text{passed}} \) to the train gate. The train gate can be in two positions: up, down. It is initially in position up, it moves down after receiving a message \( m_{\text{com}} \), and moves up after receiving a message \( m_{\text{passed}} \). We impose the following time constraints on the system behaviour:

• A far away train needs at least 8 minutes to approach the gate.

6. Let us point out that in the case when gates are allowed, the definition is a bit more complicated, see [8].
• A sensor detects the arrival of a train before the train reaches the train gate. It sends the message $m_{\text{com}}$ to the train gate as soon as it detects a train arrival and the message $m_{\text{passed}}$ as soon as the train has passed the crossing.

• The train gate moves immediately down (up, resp.) from position up (down, resp.) after receiving message $m_{\text{com}}$ ($m_{\text{passed}}$, resp.).

• Every message is delivered immediately (in 0-time).

5.2 Formal Design.

An abstract design of this system can be given by a Message Sequence Chart which shows the message flow and the real-time constraints. A Message Sequence Chart showing the railroad crossing is given in the figure. The timings of a message deliverance are indicated by a time value below the message arrow. Setting a timer in an object (e.g. $t_{\text{far}}$) is indicated by a double triangle. Timers are set with certain values ($t_{\text{far}}$ is set with value 8). Time-outs are indicated by arrows starting in the double triangles.

The behaviour of train is described by object T (with class name Train). We do not model the sensor explicitly but as a part of the train specification. The attribute of a train stores information about the next train. A train is initially in state far. A timer is set to 8 time units ensuring the 8 minutes distance between two trains. Sometime after the time-out of the timer the message $m_{\text{com}}$ is sent changing the state of T to at indicating that a train has arrived at the crossing. Later the message $m_{\text{passed}}$ is sent and the state of T is set back to far (indicating that the train has left the crossing). The attribute of the train gate object TG (with class name TGate) stores information about the gate position. It is initially in position up and moves down after receiving message $m_{\text{com}}$, and then moves up after receiving message $m_{\text{com}}$. Let the messages $m_{\text{com}}$ and $m_{\text{passed}}$ be of the form $(T, \text{com}, g)$, $(T, \text{passed}, g)$, respectively.

\begin{align*}
\text{op} & \quad \text{far}^\text{i}, \text{far}^\text{r}, \text{at}, \text{far}^\text{f}, \text{up}^\text{i}, \text{down}, \text{up}^\text{f} : \rightarrow \text{state} , \\
& \quad \text{far}^\text{i}, \text{up}^\text{i} : \text{initial} , \text{far}^\text{f}, \text{up}^\text{f} : \text{final} .
\end{align*}

** the states are static and can last arbitrarily long

\begin{align*}
\text{op dynamic} & \quad \text{com}, \text{passed} : \rightarrow \text{data} , \\
& \quad t_{\text{far}} : \text{Time} \rightarrow \text{Timer} .
\end{align*}

** the messages $m_{\text{com}}, m_{\text{passed}}$ will be delivered in 0-time because com, passed are declared dynamic

\begin{align*}
\text{rl} & \quad <T | \text{far}^\text{i}> \rightarrow \text{set} t_{\text{far}} 0 \rightarrow <T | t_{\text{far}}(8) \circ \text{far}^\text{i}> .
\end{align*}

** timer $t_{\text{far}}$ is set and in 8 minutes a train may come

\begin{align*}
\text{rl} & \quad <T | t_{\text{far}}(0) \circ \text{far}^\text{i}> \rightarrow \text{t_out} t_{\text{far}} 0 \rightarrow <T | \text{far}^\text{r}> .
\end{align*}

** 8 minutes elapsed, the timer times out, and the train may appear any time
\textbf{5.3 Implementation.}

In this subsection we implement the above specification in Elan. The specification is composed of three modules: a module template containing the declaration of basic sorts and operations (we skip this module because of lack of space), the module \texttt{tGate\_spec} specifying the train gate, and the module \texttt{Trains\_spec} specifying the whole railroad crossing. The module \texttt{Trains\_spec} imports the module \texttt{tGate\_spec} which in turn imports the template module. The module \texttt{tGate\_spec} concerns the train gate, it is a literal translation of the corresponding part of \texttt{TGate\_spec} according to definition 3.1.

\begin{verbatim}
module tGate_spec
  import global       template ;
end
operators          global
  upi : state ;  upf : state; down : state ;
  com : data ;  passed : data ;
  TG : obn ;  T : obn ;
end
rules for conf  global
  [] m(T, com, TG) * <TG | upi> => <TG | down>   end
  [] m(T, passed, TG) * <TG | down> => <TG | upf>   end
end
\end{verbatim}

For the specification of T we follow the interpretation schema except in case (ii) where the distributivity of Go is restricted to configurations without pending messages and where time progress in a timer is modelled as time progress in a whole object. The first change ensures that all messages will be read before the time progresses, the second makes the specification more compact. Let us observe that every 0-time rule applies only to terms having no variables, therefore we do not need the cleanness function.
module Trains_spec
import global tGate_spec ; end
operators global
  fari : state ; far´ : state ; farf : state ; at : state ;
t_far( @ ) : ( int ) timer ; end
rules for conf
  s, x, y : state ; c1, c2 : conf;
  n : int ;
global
[] <T | fari> => <T | t_far(8) & fari> end
[] <T | t_far(0) & fari> => <T | far´> end
[] <T | far´> => <T | at > * m(T, com, TG) end
[] <T | at> => <T | farf> * m(T, passed, TG) end
[] D(c1) * D(c2) => D(c1 * c2) end
[] Go(<TG | x> * <T | y>) => Go(<TG | x>) * Go(<T | y>) end
[] Go(<TG | x>) => D(<TG | x>) end
[] Go(<T | farf>) => D(<T | farf>) end
[] Go(<T | t_far(n) & s>) => D(<T | t_far(n-1) & s>) if n > 0 end
[] [n] D(cf) => [n-1] Go(cf) if n > 0 end
end end

The Elan implementation allows us to execute the specification. We have tested the railroad crossing specification systematically for inputs ranging from 0 to 15 minutes (of railroad crossing system time) starting from the initial configuration. For example, we have rewritten the term: [15] D(<T | fari> * <TG | up i>). The execution time was 0.380 seconds. Elan provides a possibility to trace the rewriting. By analyzing the execution trace we have observed that the specification, as it stands, does not prohibit racing between messages m_com and m_passed, since the second message can be read before the first one. This is due to the fact that there are no time constraints on the train speed. The train may reach the gate in 0-time before m_com will be read. This will not happen if we ensure that the train moves with a bounded speed and therefore reaches the train gate after a specified time elapse; a requirement like this may be specified by another timer.

The Elan implementation provides also more sophisticated facilities for executing specifications using the built-in strategy language one may specify complex search algorithms such as depth first search, or constrain the rewriting. In the future work
we are going to use it for automatically checking simple temporal properties of the specified systems.

References


