Verification of Train Control Systems: Tools and Techniques

Andrew Lawrence

June, 2014

A thesis submitted to Swansea University
in candidature for the degree of Doctor of Philsophy

Department of Computer Science
Swansea University

Declaration

This work has not previously been accepted in substance for any degree and is not being currently submitted for any degree.

June , 2014

Signed:

Statement 1

This thesis is being submitted in partial fulfilment of the requirements for the degree of a MRes in Logic and Computation.

June , 2014

Signed:

Statement 2

This thesis is the result of my own independent work/investigation, except where otherwise stated. Other sources are specifically acknowledged by clear cross referencing to author, work, and pages using the bibliography/references. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure of this dissertation and the degree examination as a whole.

June, 2014

Signed:

Statement 3

I hereby give consent for my thesis to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

February 18, 2011

Signed:

Contents

Chapter 1

Introduction

1.1 Introduction

This thesis is concerned with the application of formal methods within the Railway Domain. Firstly we present a new approach to develop verified SAT solving algorithms and which have been applied the verification of a real world train control system: the solid state interlocking. Secondly we present an approach to formalise the European Rail Traffic Management System (ERTMS) and apply a model checker to verify the systems safety.

1.2 Aim

We have 2 sets of aims. The first set of aims is in regard to the development of verified SAT solving algorithms and their application to the verification of solid state railway interlocking programs:

- Formalize the DPLL proof system and a proof of its completeness.
- Extract a standard DPLL SAT algorithm from the formalisation into a functional programming language and test its performance
- Modify the formalisation and completeness proof of the DPLL proof system so that it captures the behaviour conflict driven clause learning SAT algorithms.
- Extract a clause learning SAT algorithm from the formalisation of the modified DPLL proof system and show that clause learning increases the efficiency of the solver.
- Apply verified SAT algorithms to the verification of solid state railway interlocking programs.

Our second set of aims is in regard to the formal specification and verification of the European Rail Traffic Management System (ERTMS) using Real Time Maude.

- Formalise ERTMS as a hybrid automata.
- Formalise ERTMS as a Real Time Maude specification.

- Verify ERTMS using Real Time Maude's linear temporal logic model checker.

1.3 Thesis Outline

Chapter 2

Background

From their birth in the 1800s to the present day, the railway and its control systems have seen many advances. Its control and safety has gone from being a completely manual human based system, to a mechanical system and finally to the electronic system we see today. We will now look at a brief history of the railway followed by information on our industrial partner Invensys Rail. We then look more closely at modern railways and the equipment which constitutes them. We also study Westrace interlocking which is produced by Invensys and the ladder logic programs which run on it. Finally, we look at some previous work in this field.

2.1 A History of Railway Signalling and Control Systems

Prior to the days of fixed signals, Policemen would be stationed at junctions and railway stations. They changed points manually and gave instructions to train drivers by using a system of either flags or oil lamps depending on the visibility. Since this was before the time of telecommunications and electricity there was no way of telling where a train was once it left a station and went out of sight. The only safety precaution that could be taken was to use an egg timer to delay the departure of the next train in order to give the previous train time to progress along the track. Train speeds were not very high during this period so this was an acceptable way of ensuring safety.

Modern railway signally makes use of **fixed signals**. These are permanently positioned by the side of the track and provide some visual information to the train driver. The original fixed signal consisted of a shaped wooden board that could be rotated on pole round a vertical axis. If the board was visible to the driver then he would have to stop the train. On the other hand if the driver couldn't see the board because it was side-on to him then he would be able to proceed.

One of the major developments in railway signalling was the introduction of the **Semaphore** fixed signal. These consisted of a board that could be moved into several preset positions. Typically these would have 3 different visible "aspects" which they could be set to: One aspect to indicate the driver can proceed, another that indicates the driver can proceed with caution and finally an aspect which indicates that the driver should stop.

Around about the same time as the introduction of the semaphore signal, the system for controlling the signals went under drastic change. The Policemen were replaced with professional **Signallers** whose job was specifically to manage the railways. A system of pulleys, wires and levers was also devised to allow multiple signals and points to be controlled from a central position. This central position became known as a **signal box** and was manned by one or more signallers. This centralisation allowed for further safety mechanisms to be installed. One in particular, namely the **interlocking**, is of interest to us. The interlocking physically locked levers if they were unsafe to move.

The next leap in railway technology came from the invention of the electronic **track circuit**. These would activate an indicator in the signal box if a segment of track was occupied by a train. As more and more track circuits became installed it was no longer necessary to have human intervention to control certain signals. **Automatic signals** were introduced which operated completely by track circuits without any intervention from human signallers. Around this time **electric point machines** were introduced removing a large amount of physical work performed by signallers allowing for a greater area of control for each signaller. Around this time electromechanical **relays** began to replace purely mechanical relays reducing the amount of space needed for a signal box.

In the 1920s **colour light signals** replaced mechanical semaphore signals these where much brighter than the oil lamps fitted to semaphores and greatly increased the safety of night time train travel. In the 1930s the mechanical levers were replaced with an electronic **control panel** containing switches and buttons. This allowed for the introduction of **route setting** where with the press of a button configurations of signals and points would be associated with a particular route could become activated. Prior to this time many levers would have had to have been pulled to set many different pieces of equipment. During the 1980s the most important advance from our point of view took place. The advent of electronic microprocessors enabled the replacement of the relay and mechanical interlockings with an electronic **solid state interlocking** system (SSI) [**?**]. The main focus of this project will be to investigate the safety of such solid state interlockings.

2.2 Invensys Rail

Invensys Rail [**?**] and its previous incarnation Westinghouse Rail Systems Ltd have been involved for over 140 years in producing equipment to increase safety in the railway industry. Originally they produced air brakes for trains, these had a failsafe state such that if the power was cut the brakes would automatically stop the train. Later on in the company's development they provided support to British Rail when the first solid state digital railway interlocking was installed in Leamington Spa. Today they supply railway control equipment to companies based around the globe, including companies based in Australia, Hong Kong, Germany, Spain and the UK. This project is mainly concerned with one of the solid state railway interlockings Invensys produces called the Westrace. The Westrace railway interlocking continuously runs a ladder logic program which prevents the railway control systems from entering a dangerous state. Ladder logic will be explained in a later chapter. David Kerr and Tony Rowbotham produced a book that explains the terminology and methodology used in the railway industry and by Invensys (See [**?**]).

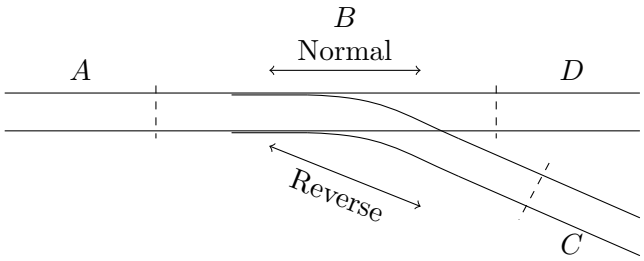


Figure 2.1: A Typical Junction

2.3 An Overview of the Railway Domain

In this section we present the features of the railway domain that are in the scope of this thesis. We hope to provide the reader with the background information and terminology necessary to understand the parts of this thesis.

2.3.1 The Railway Topology: Track and Points

We will now present an overview of the physical railway from a topological point of view. To do this we will present an example of a small track plan of a junction. If the reader is interested in learning more about the topology of the railway a more detailed description can be found in [**?**]

2.3.1.1 Track Segments

A section of track is typically broken down into track segments each containing one or more track circuits to detect the presence of a train. Typically track segments become larger on long straight stretches of track without any interesting topological features such as junctions or stations. Likewise track segments become smaller around junctions and stations where control over train movement is of greater importance.

2.3.1.2 Points

A point is a physical piece of equipment that is used to form a junction. Due to the nature of the rails and trains it is not possible to physically to just join two segments of track. Instead a point is needed to act as physical switch controlling the flow of trains through a junction. A point has two positions which are referred to as **normal** and **reverse**. This presents a safety hazard, for example see figure 2.1, if a train enters the junction b from c when the junction is locked in the position for normal then the train will be derailed.

2.3.2 Railway Signalling

Signals are the main means used to communicate information regarding the state of the track ahead of the train. Typically they are placed either on the track side or over hanging the railway. Visual indications known as aspects are used to convey information to the driver. A signal will have many such aspects which can be displayed, each with a particular meaning. The main type of signal considered in this project is the coloured light signal. Typically these have between one - four aspects each conveying a different indication about the state of the track ahead. Below is a description of the aspects used for a three light signal.

Green - If this aspect is displayed it indicates that track ahead is clear for a sufficient distance and the train driver can proceed at full speed to the next signal.

Yellow - This aspect indicates the track immediately ahead in between this signal and the next is clear however the driver should proceed with caution as a train could be in the track after that.

Red - This aspect indicates that the track ahead is not clear, the driver should stop and wait at this signal.

The one aspect signal is typically a fixed red indicating that is not possible to proceed down the track at this current point in time. The two - four aspect signals are used on tracks with different speeds to convey different stopping distances. The two aspect signal for instance would be used on a low speed track segment where stopping distances are relatively short. Whereas the four aspect signal would be used on a high speed line where stopping distances are long and the driver needs information for a greater length of track. These signalling schemes are fixed in the UK however they are not fixed from country to country. On the continent, for example, they may use different conventions, colours and number of lights on each signal.

2.3.3 The Westrace Interlocking

The railway interlocking is a key component in ensuring the safety of the railway. Its job is to apply a set of rules to the requests and commands it receives from the control system and check whether or not the future state of the railway is safe. If the control signals it receives do not violate the safety of the railway then these signals are committed to the physical infrastructure. For example if the human controller requests for a route to be set the interlocking will process this request and ensure that it does not conflict with other routes before allowing the command to be passed to the physical railway.

The railway interlocking repeatedly executes a program or set of rules over some discrete time interval. Each time it uses the set of rules it contains to process a new set of inputs before committing them as outputs. The Westrace interlocking used by Invensys Rail executes a so-called ladder logic program to perform this process. The following are the three main stages of operation in the running of an Westrace interlocking.

Reading of Inputs - Read inputs from the control systems as well as the physical railway infrastructure.

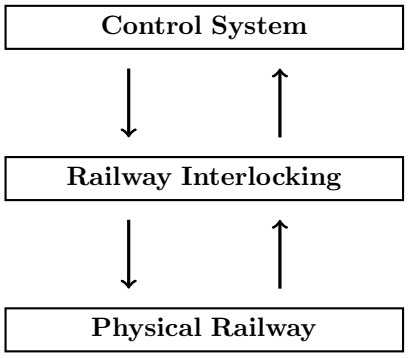


Figure 2.2: The Location of the Railway Interlocking

Internal Processing - Execute the ladder logic program with the above inputs and calculate outputs.

Committing of Outputs - The outputs calculated in the previous cycle are then passed on to various places including the physical railway.

2.4 Ladder Logic

The Westrace Interlocking performs calculations by executing a ladder logic program [**?**]. In the following we will look in more detail at these ladder logic programs. The main concepts behind their construction and behaviour will be presented. In later chapters we will provide a formal framework for the verification of these programs.

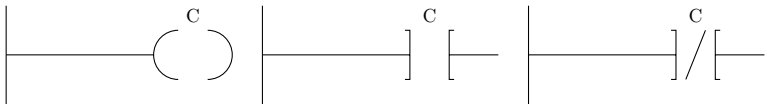
The international standard for programmable logic controllers IEC 61131 [**?**] describes the graphical language ladder logic. It gets its name from its graphical "ladder" like appearance which was chosen to suit the control engineers responsible for their design. Each rung of the ladder is used to compute an output variable from one or more input variables in the rung. In the railway industry these input variables are referred to as contacts and the output variables are referred to as coils. A description of the entities representing these variables is as follows:

Coils : These are used to represent values that are both stored for later use and output from the program. The value of a coil is calculated when a rung fires making use of the current set of inputs, the previous set of outputs and any outputs already computed for this cycle. The coil is always the right most entity of the rung and its value is computed by executing the rung from left to right.

Open Contacts : This entity represents the value of an un-negated variable

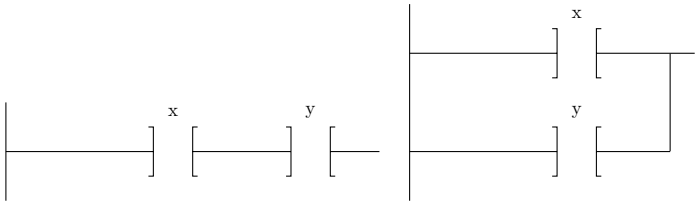
Closed Contacts : This entity represents the value of a negated variable.

A Ladder logic rung is built using these entities and connections between them. The shapes of the connections between the contacts determines how the value of the coil is computed from them. Using propositional logic for comparison, a horizontal connection between two contacts represents logical conjunction and a vertical connection between two contacts represents logical disjunction see Figure **??**.



(a) A coil (b) An open contact (c) A closed contact

Figure 2.3: The Entities Used In Ladder Logic



(a) $x \wedge y$ Conjunction (b) $x \vee y$ Disjunction

Figure 2.4: Logical Connectives In Ladder Logic

In section 4 an approach is presented to capture the semantics of ladder logic programs using propositional logic.

2.5 Previous Work in this Field

Previously James carried out work for Invensys, applying various SAT and model checking techniques to verify the correctness of a simple pelican crossing and two existing railway interlockings consisting of approximately 500 rungs (see James [**?**]).

James used Kanso's work (See [**?**]), in particular his translation from ladder logic into propositional logic, and applied several model checking techniques in order to try and reduce the complexity of the problems. Both the work by Kanso and James was based on an early feasibility study by Fokkink and Hollingshead [**?**]. The relationship between a ladder logic program and propositional logic was discussed in great detail. A method for formulating such a ladder logic program as a formula in propositional logic was presented. This laid the ground work for all successive projects involving ladder logic. The possible application of program slicing was discussed and this was later applied in the work by James [**?**].

Some of the techniques applied by James to the verification of ladder logic are discussed below.

Bounded model checking: This was the main topic of the work of Phil James. It had the advantage that it produced counter example traces which are highly valuable to the engineers at Invensys. It allowed for the verification of 2000 iterations of the ladder logic programs provided without programming slicing and up to 20000 iterations of ladder logic programs with program slicing.

Temporal Induction: This is another technique used in the verification of the ladder logic programs, it succeeded whenever the inductive verification method Kanso applied also succeeded. It should however be stronger than inductive verification but no example

was found to prove this. Temporal induction produced a counter example whenever the bounded model checking produced a counter example.

Program Slicing: This technique was combined with application of bounded model checking to reduce the state space requiring verification. This reduced the number of rungs in a ladder logic program by up to a factor of 10.

Chapter 3

Extracting Verified Decision Procedures

Chapter 4

Extracting a Clause Learning SAT Algorithm

Chapter 5

The Application of Real Time Maude to Model and Verify the European Rail Traffic Management System

In the following we present the Maude [**?**, **?**] and Real Time Maude [**?**, **?**, **?**] tools and describe an approach using these tools to model and verify ERTMS **??**.The property we verify is that no two trains on our example railway share the same movement authority which

5.1 The European Rail Traffic Management System

5.2 Formalising the European Rail Traffic Management System Using Hybrid Automata

One formalism which we can use to real about systems such as ERTMS is hybrid automata [**?**].
Definition 1 (Hybrid Automaton)**.**

The following is a simple example railway we shall use for the purposes of demonstrating our verification approach. It contains 5 track segments connected to form a pentagon with two trains. This captures most of the behaviours found in a larger more complicated railway as the trains and control systems view any piece of track as a set of track segments joined together to form a route.

We have modelled the ETMS system controlling the pentagon example (see fig **??**) using several hybrid automata. In this example the value $D(t)$ represents the distance from the start point at time t. It contains 2 trains A and B and 5 track circuits $l_0, \ldots, l_4$. The track is uni-directional allowing trains to travel from $0 - 2499$.

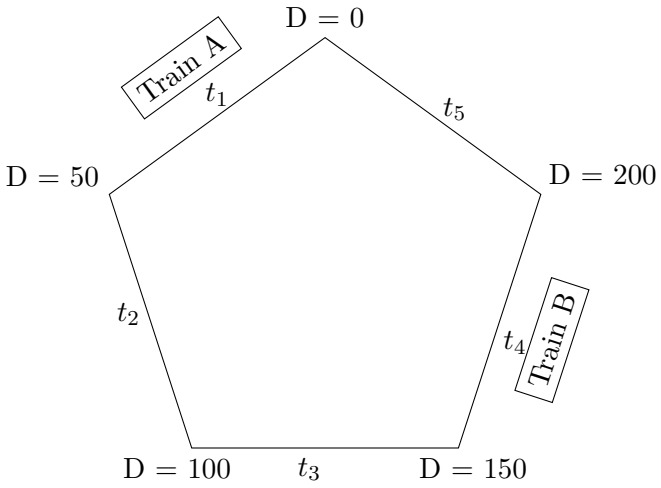


Figure 5.1: Pentagon Example

Definition 2 (Interlocking Hybrid Automaton)**.** *We define a hybrid automaton H_{IL} as follows:*

Variables *The state of the interlocking automaton consists of five boolean variables* $\underbrace{l_0, \ldots, l_4}_{Occupied/Free}$ *and a variable ReqID ranging over* $\{0, \ldots, 4\}$.

Control Graph *The control graph of the interlocking automaton consists of two control modes* $\{Response, Idle\}$ *with four control switches connecting them;* $Response \to Idle$, $Idle \to Response$, $Response \to Response$, $Idle \to Idle$.

Initial, invariant and flow conditions

- $init(Idle) := [l_0 = Free, l_1 = Free, l_2 = Free, l_3 = Free, l_4 = Free]$.

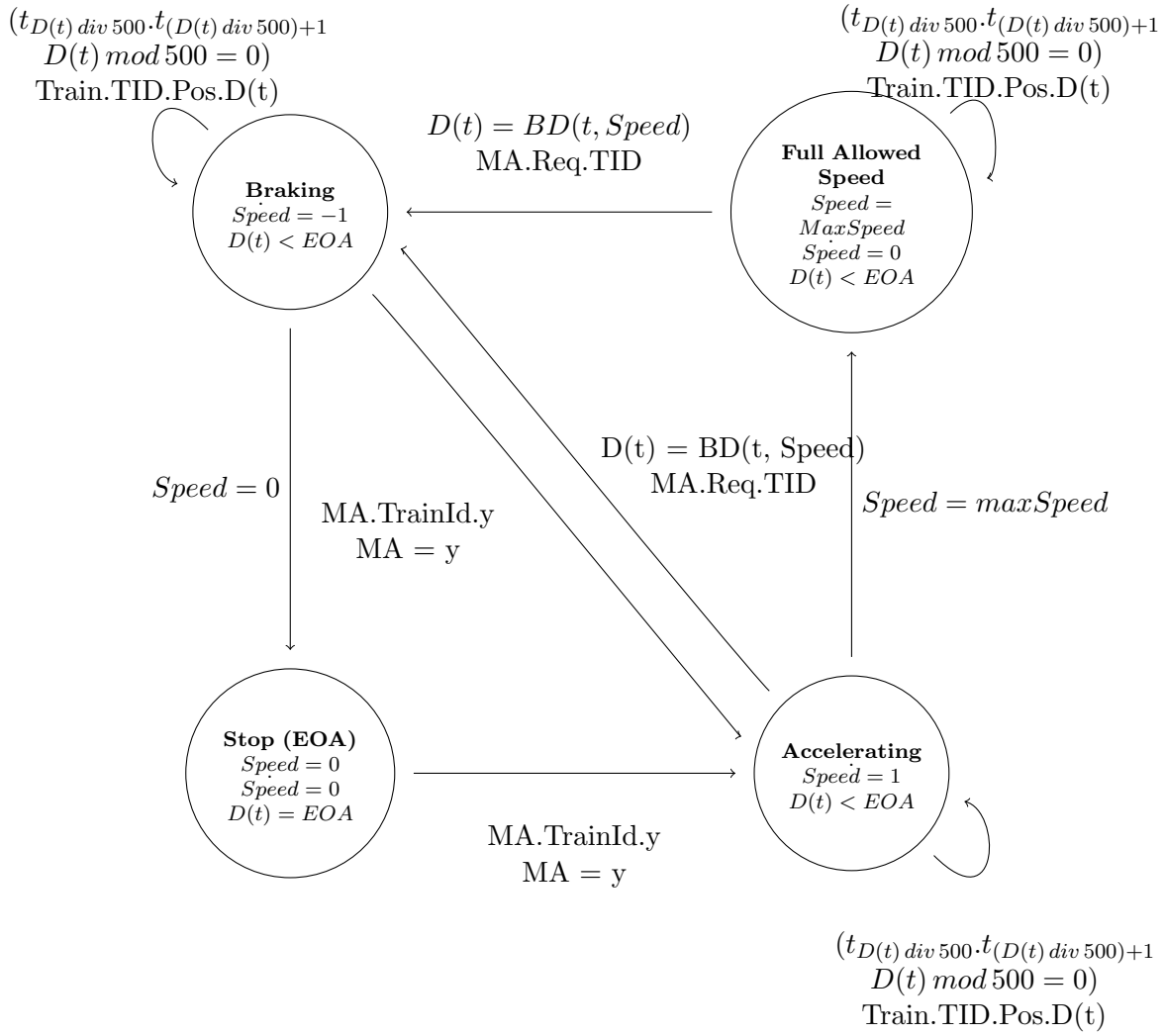
Jump Conditions

- $jump(Idle \to Response) := Req.z, ReqId' = z$
- $jump(Response \to Idle) := if\ (l_{ReqId} = Free \wedge l_{ReqId+1\ mod\ 5} = Free)\ then\ Grant.ReqId\ else\ Den$
- $jump(Idle \to Idle) := l_x.l_{x+1}, [l'_x = Free, l'_{x+1} = Occupied]$
- $jump(Response \to Response) := l_x.l_{x+1}, [l'_x = Free, l'_{x+1} = Occupied]$

Events

- $event(Idle \to Response) := Req.z$
- $event(Response \to Idle) := Grant.z, Deny.z$
- $event(Idle \to Idle) := l_x.l_{x+1}$
- $event(Response \to Response) := l_x.l_{x+1}$

Secondly we define a hybrid automaton H_T which models an individual train.
Definition 3 (Train Automaton)**.** *We define a hybrid automaton H_T as follows:*



Variables *The state of the interlocking automaton consists of* $\underbrace{D(t), EoA}_{0,\,\ldots,\,2499}$,

$\underbrace{Speed, \dot{Speed}, MaxSpeed, TID}_{\mathbb{N}}$.

Control Graph *The control graph of the interlocking automaton consists of four control modes* $\{Stop(EoA)$, $Braking$, $Accelerating$, $Full\,Allowed\,Speed\}$.

Initial, invariant and flow conditions

- $init(Stopped) := D(t) < EoA.$
- $inv(FullAllowedSpeed) := \dot{Speed} = 0 \wedge D(t) < EoA$
- $inv(Accelerating) := D(t) < EoA$
- $inv(Braking) := D(t) < EoA$

- $inv(Stop(EoA)) := Speed = 0$
- $flow(Accelerating) := \dot{Speed} = 1$
- $flow(Braking) := \dot{Speed} = -1$

Jump Conditions

- $jump(FullAllowedSpeed \rightarrow FullAllowedSpeed) := l_{D(t)\,div\,500}.l_{(D(t)\,div\,500)+1} \wedge D(t)\,mod\,500 = 0$
- $jump(Braking \rightarrow Braking) = l_{D(t)\,div\,500}.l_{(D(t)\,div\,500)+1} \wedge D(t)\,mod\,500 = 0$
- $jump(Accelerating \rightarrow Accelerating) = l_{D(t)\,div\,500}.l_{(D(t)\,div\,500)+1} \wedge D(t)\,mod\,500 = 0$
- $jump(Stop(EoA) \rightarrow Accelerating) := MA.TID.y \wedge EoA' = y$
- $jump(Braking \rightarrow Accelerating) := MA.TID.y \wedge EoA' = y$
- $jump(Accelerating \rightarrow Braking) := D(t) = BD(t, Speed) \wedge MA.Req.TID$
- $jump(FullAllowedSpeed \rightarrow Braking) := D(t) = BD(t, Speed) \wedge MA.Req.TID$
- $jump(Accelerating \rightarrow FullAllowedSpeed) := Speed = MaxSpeed$
- $jump(Braking \rightarrow Stop(EoA)) := Speed = 0$

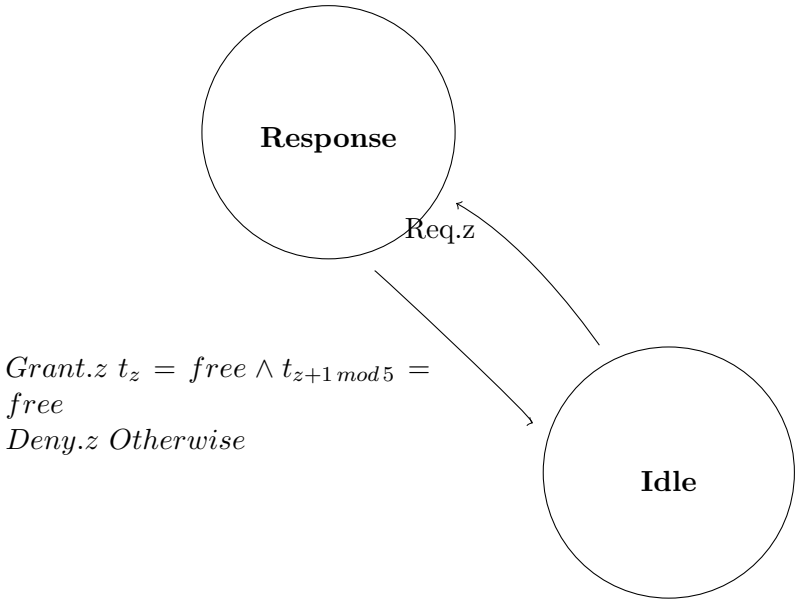
Events

- $event(Stop(EoA) \rightarrow Accelerating) := MA.TID.y$
- $event(Braking \rightarrow Accelerating) := MA.TID.y$
- $event(FullAllowedSpeed \rightarrow FullAllowedSpeed) := l_{D(t)\,div\,500}.l_{(D(t)\,div\,500)+1} \wedge D(t)\,mod\,500 = 0$
- $event(Braking \rightarrow Braking) = l_{D(t)\,div\,500}.l_{(D(t)\,div\,500)+1} \wedge D(t)\,mod\,500 = 0$
- $event(Accelerating \rightarrow Accelerating) = l_{D(t)\,div\,500}.l_{(D(t)\,div\,500)+1} \wedge D(t)\,mod\,500 = 0$
- $event(Accelerating \rightarrow Braking) = MA.Req.TID$
- $event(FullAllowedSpeed \rightarrow Braking) = MA.Req.TID$

In both H_{IL} and H_T we make use of an event $l_x.l_{x+1}$ to capture the movement of the train from one track segment to the next. In the above hybrid automaton modelling the trains we make use of a function BD that calculates location required to stop at the EOA based on the trains speed. The point at which the train should break would then be modelled as $BD(EOA, speed) = EOA - \frac{speed^2}{2} \bmod 2500$.

This hybrid automaton has 4 control modes namely Braking (Auth), Full Allowed Speed, Accelerating and Stopped and has 5 variables namely $Maxspeed$,$D(t)$ (Position), $Speed$, $\dot{Speed}$ and EoA. We have that $D(t) \leq EOA$ is an invariant of the stopped mode and $D(t) \leq EOA$ is an invariant of all other control modes. We define a transition with the event $l_x.l_{x+1}$ which is triggered by the condition $D\,mod\,500 = 0$ in the modes $FullSpeed$,

Accelerating and *Brake* and causes TC = x +1. The simplest way to model deceleration would be to assume that the trains speed decreases at −1 unit of distance per unit of time. We compose both the automata for both the interlocking and the hybrid $H_{IL}||H_T$. It is possible for a new MA EOA' with $EOA < EOA'$ to be received in the *Braking* and *Stopped* states.



Definition 4 (Radio Block Controller Hybrid Automaton)**.** *We define a hybrid automaton H_{RBC} as follows:*

Variables *The state of the radio block controller automaton consists of* $\underbrace{MA_1, Pos_1, MA_2, Pos_2}_{0,\,\ldots,\,2499}$, $\underbrace{LastTrain}_{\mathbb{N}}$.

Control Graph *The control graph of the radio block controller automaton consists of four control modes* $\{Idle, Ready\,to\,Request, Wait, Granted\}$.

Initial, invariant and flow conditions

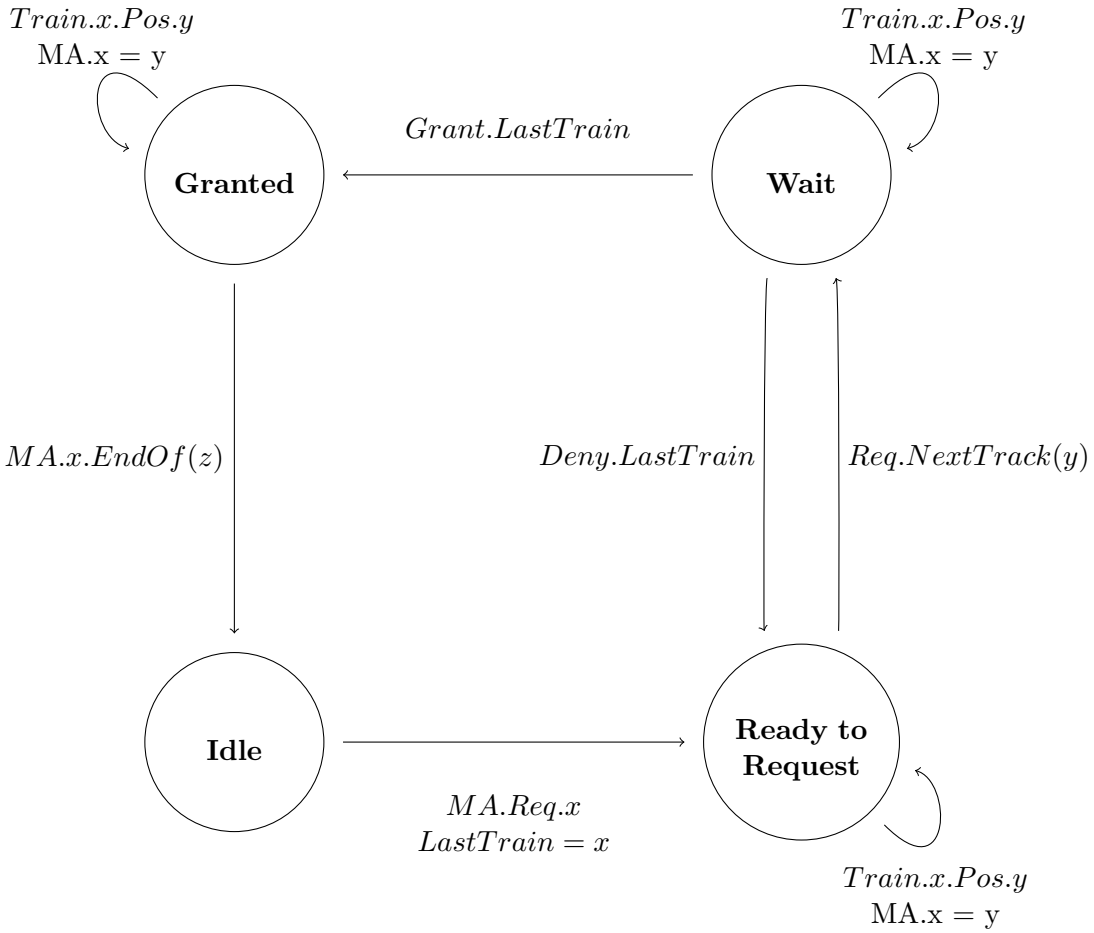
- $init(Idle) := \ldots$

Jump Conditions

- $jump(ReadytoRequest \rightarrow ReadytoRequest) := Train.x.Pos.y \wedge MA.x = y$
- $jump(Wait \rightarrow Wait) := Train.x.Pos.y \wedge MA.x = y$
- $jump(Granted \rightarrow Granted) := Train.x.Pos.y \wedge MA.x = y$
- $jump(Idle \rightarrow ReadytoRequest) := MA.Req.x \wedge LastTrain = x$
- $jump(ReadytoRequest \rightarrow Wait) := Req.NextTrack(y)$
- $jump(Wait \rightarrow ReadytoRequest) := Deny.LastTrain$
- $jump(Wait \rightarrow Granted) := Grant.LastTrain$
- $jump(Granted \rightarrow Idle) := MA.x.EndOf(z)$

Events

- $jump(ReadytoRequest \rightarrow ReadytoRequest) := Train.x.Pos.y$
- $jump(Wait \rightarrow Wait) := Train.x.Pos.y$
- $jump(Granted \rightarrow Granted) := Train.x.Pos.y$
- $jump(Idle \rightarrow ReadytoRequest) := MA.Req.x$
- $jump(ReadytoRequest \rightarrow Wait) := Req.NextTrack(y)$
- $jump(Wait \rightarrow ReadytoRequest) := Deny.LastTrain$
- $jump(Wait \rightarrow Granted) := Grant.LastTrain$
- $jump(Granted \rightarrow Idle) := MA.x.EndOf(z)$



Safety conditions for the combined system can be separated into discrete and continuous parts. If we want to specify a safety condition which states that it is not possible for two trains to collide in our current system this could have the following components.

the continuous part would state that any movement authority issued by the radio block processor would respect the interlockings separation policy. The discrete part would basically

state that there is at least two free track circuits in-between each train.

We will now attempt to formalise the safety condition "The train will always break on time". What this means formally is that starting from the stop mode whenever the train is in the stop start $D(t) \leq EOA$ Assuming we are in the stop mode and the we receive a movement authority with $EOA > D(t)$. Then we move to the accelerating state in this case we shall perform a case distinct on whether we reach the braking point i.e. $D(t) = BD(EoA, speed)$ or we reach maxspeed.

Theorem 1. *Given a live transition system* $(S^t_{H_T}, L^t_{H_T})$

$$\forall \langle a_i, q_i \rangle_{i \geq 1} \in L^t_{H_T}. \forall (a_n, (v, [D(t), EoA, Speed, \dot{Speed}, TID])) \in \langle a_i, q_i \rangle_{i \geq 1}$$
$$\wedge D(t) \leq BD(EoA, Speed) \leq EoA$$

Proof. The proof is performed by fixing a trace $\langle a_i, q_i \rangle_{i \geq 1}$ and a label/state pair (a_n, q_n) in the timed trace in which the property holds and then proving that for all possible successor label/state pairs (a_{n+1}, q_{n+1}). Where the state $q_n = (v, [D(t), EoA, Speed, \dot{Speed}, TID])$ and $q_{n+1} = (v', [D(t)', EoA', Speed', \dot{Speed}', TID'])$

There are 4 cases for v in which we must argue that the transition $q_n \xrightarrow{a_{n+1}} q_{n+1}$ maintains the property $D(t) \leq EOA$. We further divide each other these cases into two sub cases in which the duration of the transition $\delta = 0$ or $\delta \in \mathbb{R}_{>0}$

v = Stop All possible transitions from the stop mode have a duration $\delta = 0$. There is one possible event that $MA.TID.y$ which will grant a new movement authority y such that $EoA < y$ and cause a jump to the *Accelerating* mode with $D(t)' \leq BD(y, Speed') < y$

v = Accelerating In the case that the duration of the transition $\delta = 0$ and the successor state is $q_{n+1} = (Braking, [D(t)' = D(t), EoA' = EoA, Speed' = Speed, \dot{Speed}' = \dot{Speed}, TID' = TID]$ and either $D(t) = BD(EoA, Speed)$ or $Speed = MaxSpeed$ has occurred. If $D(t) = BD(EoA, Speed)$ then $D(t)' = BD(EoA', Speed') \leq EoA'$. The other case that $Speed = MaxSpeed$, $v' = Full\ Allowed\ Speed$ and $D(t)' \leq BD(EoA', Speed') \leq EoA'$. If the successor state and the current state are the same then an event has occurred and $D(t) \leq BD(EoA, Speed) <$.

In the case that $\delta \in \mathbb{R}_{<0}$

In the *Accelerating* mode we have $\dot{Speed} = 1$ with $D(t) \leq BD(Speed, t_1) \leq EoA$. There are two cases either $D(t) = BD(Speed, t_1)$ or $D(t) < BD(Speed, t_1)$. In the case that $D(t) = BD(Speed, t_1)$ a jump occurs taking the system into the Braking mode with $D(t_1) = BD(Speed, t_1) < EoA$. In the case that $D(t_1) < BD(Speed, t_1)$ time will progress and at some point in the future t_2 the train will with reach the braking point $D(t_2) = BD(Speed', t_2)$ or $Speed' = MaxSpeed \wedge (D(t_2) < BD(Speed', t_2)$. In the case that $D(t_2) = BD(Speed', t_2)$ a jump will occur taking the train into the braking mode with $D(t_2) = BD(Speed', t_2) < EoA$. Otherwise $Speed = MaxSpeed$ and a jump is performed to the $FullAllowedSpeed$ mode with $D(t_2) < BD(Speed', t_2) < EoA$.

v = Full Allowed Speed In the $FullAllowedSpeed$ mode have two cases $D(t) = BD(Speed, t)$ with $t = t_1$ or t_2, $t_1 < t_2$. In the first case a jump occurs instantaneously to the $Braking$ mode with $D(t_1) = BD(Speed, t_1) < EoA$ In the second case time elapses to a point

in the future t_2 and $D(t)$ increases until $D(t_2) = BD(Speed, t_2)$ then the system will perform a jump to the *Braking* mode with $D(t_2) = BD(Speed, t_2) < EoA$.

v = Braking In the Braking mode there are two cases either $MA.TID.y \wedge Speed > 0$ or $Speed = 0$. Since the train is braking by the definition of BD, $D(t_2) \leq BD(Speed, t_2)$ will continue to hold for any amount of time in this state. In the case that $MA.TID.y \wedge Speed > 0$ we receive a new movement authority y with $y < EoA$ and a jump is performed to the *Accelerating* mode with $D(t_2) \leq BD(Speed', t_2) < y$. In the case that $Speed = 0$ a jump occurs to the *Stopped* state.

□

Another interesting property of our model is that alone the model of the interlocking allows for "jumping trains" i.e. it allows for track circuits to become free and occupied in a way that does not model the normal movement of trains. However when the interlocking automaton is placed in parallel with a train automaton it behaves in a

Theorem 2. *In the composite automaton $H_{IL}||H_T$ the event $t_x.t_{x+1}$ will only occur if and only ift_x is currently occupied in which case t'_x = free and t'_{x+1} = occupied.*

Proof. We have to prove the two directions of the statement.

If the

Assume a train is in t_x and $((x-1)mod5) * 500 \leq D(t) < x * 500$.

There are two cases

1. If the train is in the Stopped state then a $t_x.t_{x+1}$ event will never occur as it is not possible in this state.
2. The train is moving i.e. it is in either of the *Accelerating*, *Braking*, *Full Allowed Speed* states. In this case the position of the train will be increasing and eventually it will happen that $D(t)mod500 = 0$. When since $D(t)mod500 = 0$ satisfies the jump condition the event $t_x.t_{x+1}$ will occur.

□

5.3 Maude

In the following section we shall describe the Maude tool and specifications.

5.3.1 Maude Specifications

A Maude specification consists of *functional modules* declared using `fmod` and `endfm` which contain the following:

sorts	`sort` s or `sorts` s s'.
subsorts	`subsort` $s < s'$.
function symbols	`op` f : $s_1 \ldots s_n$ `->` s .
variables	`vars` v v' : s'.
uncondition equations	`eq` $t = t'$.
condition equations	`ceq` t = t' `if` *cond*
membership axioms	`mb` t : s . or `cmb` t : s `if` *cond* .

Formally a Maude specification is a *rewrite theory* of the form $\mathfrak{R} = (\Sigma, E, L, R)$, where

$$[l] : t \to t' \textbf{if} \bigwedge_{i=1}^{n} u_i \to v_i \bigwedge_{j=1}^{m} w_j = w'_j$$

where l is a label $l \in L$ and t, t', u_i, v_i, w_j and w'_j are implicitly universally quantified variables representing Σ-terms. Maude theories are *order-sorted* allowing for the specification of subsorts which is achieved by including into the specification a partial order relation over sorts. Given two sorts s and s' this partial order relation $s \leq s'$ is interpretted as subset inclusion $A_s \subseteq A_{s'}$ in a model A.
Definition 5 (Rewrite Theory)**.**

5.3.1.1 Example Maude Specification

The following is a specification of the natural numbers in Maude:

```
fmod BASIC-NAT is
        sort Nat .

        op 0 : -> Nat .
        op s : Nat -> Nat .
        op _+_ : Nat Nat -> Nat .

        vars N M : Nat .

        eq 0 + N = N .
        eq s(M) + N = s(M + N) .
endfm
```

5.4 Real Time Maude

One extension of Maude that has been used to model and verify hybrid systems is Real Time Maude. There is an example of a distributed sensor network that has been modelled and verified using this tool.

5.4.1 Real Time Maude Specifications

A Real Timed Maude specification consists of *timed modules* that start with `tmod` and end with `endtm`. Formally a Real Time Maude specification is a real-time rewrite theory which can be thought of as a rewrite theory with an interpretation for the abstract time domain together with rewrite rules for terms of type `System` that have a time duration [`?`]. These rewrite rules can be seperated into two categories, the *tick* rules have a non-zero time elapse and the *instantaneous* rules have a time elapse of zero.

The interpretation of the abstract time domain is mapped to a concrete specification of time during the specification process. Real time Maude comes with two of these specifications as standard the most simple is discrete time based on the natural numbers and the more complicated of the two is dense time based on the rational numbers. We shall use discrete time during the specification as it allows for the best results during the verification process.

Definition 6 (Equational theory morphism)**.** *We define an* equational theory morphism $H : (\Sigma, E) \to (\Sigma', E')$ *to consist of the following*

- *a monotone map* $H : sorts(\Sigma) \to sorts(\Sigma')$ *which maps sorts in* Σ *to sorts in* Σ'
- *a mapping of function symbols* $f : s_1 \dots s_n \to s$ *in* Σ

1. *blah*

Definition 7 (Real-Time Rewrite Theory)**.** *We define a* real-time rewrite theory $\mathfrak{R}_{\phi,\tau}$ *to be a tuple* $(\mathfrak{R}, \phi, \tau)$ *containing a rewrite theory* $\mathfrak{R} = (\Sigma, E, L, R)$ *where:*

- ϕ *is an equational theory morphism* $\phi : Time \to (\Sigma, E)$ *that maps objects in the theory Time to objects in the equational theory* (Σ, E)*.*
- *The time domain is functional i.e. if a term* r *of sort* $\phi(Time)$ *has a rewrite proof* $\alpha : r \to r'$ *then* $r = r'$ *and the identity proof of* r *is equivalent to* α*.*
- *There is a designated sort typically called State and a sort System, contained within* (Σ, E)*, which has no supersorts or subsorts and only one operator that does not satisfy any non trivial equations:*

$$\{\_\} : State \to System$$

  Further to this condition, it is all required that the sort System does not appear in $s_1, \dots s_n$ *the domain of any operator* $f : s_1, \dots s_n$*.*
- *For each rewrite rule with* u *and* u' *of sort System in* $\mathfrak{R}$ *of the following form*[1]*:*

  there is an assigned term $\tau_l(x_1 \dots x_n)$ *of sort* $\phi(Time)$ *within* τ*.*

Example Real Time Maude Specification

The following a very simple model of a train Real Time Maude that moves one unit of distance in one time unit along a circular track of length 500. It defines a sort `TrainState` and a single

[1] All other rules which are not of this form are called *local* rules and have an instantaneous zero time elapse. These do not act on the system as a whole but rather on one or more of its components.

state `move`. We have a constructor train of type `System` which consists of a train state and a natural number.

```
(tmod DISCRETE-SINGLE-TRAIN is protecting NAT-TIME-DOMAIN .
  sort TrainState .
  ops  move :  -> TrainState [ctor] .
  op train : TrainState Nat -> System [ctor] .

  vars N : Time .
  crl [travel] : {train(move,N)} => {train(move,N + 1)} in time 1 if N < 500 .
  rl [reset] : {train(move,500)} => {train(move,0)} .

endtm)
```

Executing a Real Time Maude Specification

Real Time Maude allows one to execute or simulate a real time system by applying rewriting rules to a term of type `System`. The command `(trew {System} in time <= t)` will attempt to rewrite the system to a state t time units in the future. This isnt always possible though as the system may deadlock. The following command attempts to rewrite a train, which initially has distince 0, to its state in 100 units of time in the future:

```
(trew train(move,0) in time <= 100 .)
```

The result from this timed rewrite is as follows:

```
rewrites: 4027 in 4ms cpu (3ms real) (1006750 rewrites/second)

Timed rewrite  {train(move,0)} in DISCRETE-SINGLE-TRAIN
with mode deterministic time increase in time <= 100

Result ClockedSystem :
  {train(move,100)} in time 100
```

5.4.2 Object Orientated Specification in Real Time Maude

Real Time Maude is based on Full Maude which contains message and object constructs for object orientated specification. Using these constructs its possible to model ERTMS as several synchronously communicating processes. A Real Time Maude specification consists of *timed object orientated modules* which start with `tomod` and end with `endtom`. We can define the following:

classes : `class C | a1 :  Sort-1, ...  , an :  Sort-n .`
objects : `< O : C | a1 :  v1, ...  , an :  vn > .`

where C is the class identifier O is the object name $a_1 \dots a_n$ are attribute names and $v_1 \dots v_n$ are values.
To model communicating processes Real Time Maude uses a messages construct.

```
msgs M1 ... Mn : Sort-1 ... Sort-n -> Msg .
```

Configurations in Real Time Maude

Objects and messages in a specification are given a sort of `Configuration` which is a subsort of type `System`. The composition operation for these configuration allows for a combination of objects and messages to form new configurations. This composition operation is also commutative meaning the order of messages and objects in any configuration is ignored. This allows for more generalised writing rules in an object orientated specification that speak about specific objects and messages that are part of some bigger configuration that does not affect the firing of the rule.

```
mod CONFIGURATION is
    *** basic object system sorts
    sorts Object Msg Configuration .

    *** construction of configurations
    subsort Object Msg < Configuration .
    op none : -> Configuration [ctor] .
    op __ : Configuration Configuration -> Configuration
         [ctor config assoc comm id: none] .
```

Example Object Orientated Specification

The following is an example consisting of two classes of object which serves to demonstrate some of the advantages of object orientated specification as well as some of the pit falls. Firstly there is the class defining D objects which count modulo 500 and transmit a message containing the current count. Secondly there is class defining a object B that reads messages from a D object.

```
(tomod EXAMPLE1 is protecting NAT-TIME-DOMAIN .
  msgs msgD  : Nat -> Msg .
  class  D | d : Nat .
  class  B | b : Nat .
  vars N: Nat .
  var O : Oid .
  var REST : Configuration .

  rl [makeD] : {< O : D | d : N > REST} =>
  {msgD(N + 1 rem 500)
  < O : D | d : (N + 1) rem 500 > REST} in time 1 .

  rl [readB] : {msgD(N) < O : B | > REST} =>
  {< O : B | b : N > REST}  .
endtom)
```

We can execute the specification as follows:

```
(trew < myD : D | d :  0 > < myB : B | b :  0 > in time <= 2 .)
```

Resulting in the following output:

```
rewrites: 6246 in 7ms cpu (7ms real)
(805000 rewrites/second)

Timed rewrite  {< myD : D | d : 0 > < myB : B | b : 0 >}
in EXAMPLE1 with mode deterministic time
increase in time <= 2

Result ClockedSystem :
  {< myB : B | b : 2 > < myD : D | d : 2 >} in time 2
```

The following are some states reachable in time 2.

```
{< myB : B | b : 1 > < myD : D | d : 2 >}
{< myB : B | b : 2 > < myD : D | d : 2 >}
{msgD(1)< myB : B | b : 2 > < myD : D | d : 2 >}
{msgD(1)msgD(2)< myB : B | b : 0 > < myD : D | d : 2 >}
{msgD(2)< myB : B | b : 1 > < myD : D | d : 2 >}
```

Non-determinism should be avoided unless it is really is a property you want in a model. One way to remove some of these reachable states is to make the variable `REST` of sort `ObjectConfiguration`. This forces objects of the B class to read messages after each incrementation of time. If we make this modification to the specification then the following states are reachable in time 2.

```
{< myB : B | b : 2 > < myD : D | d : 2 >}
{msgD(2)< myB : B | b : 1 > < myD : D | d : 2 >}
```

5.5 Modelling the European Rail Traffic Management System

In the following we shall give an overview of the specifications used to model the ERTMS system. Following the hybrid automata describe in **??** we have defined one class for each of the 3 hybrid automata.

Modelling the Transition of Time

In our previous Real Time Maude specifications we have had a rewrite rules that work on a specific object or state and increment time for the whole system. This presents a problem as we could have two objects that can both increment the time of the global system independently. We need a way of incrementing the time of the whole global system and then distributing this increment of time amongst the objects of the system. To solve this problem we define operator δ which describes how distributed system evolves with time.

```
op delta : Configuration -> Configuration [frozen] .
```

```
var OCREST : ObjectConfiguration .

op delta : Configuration -> Configuration [frozen] .

rl [timetrans] : {OCREST} => {delta(OCREST)} in time 1 .
rl [delta1] : delta(CON1 CON2) => delta(CON1)delta(CON2) .
```

Modelling Trains

We shall now describe the specification of trains. We define the Train class to have a state which like our hybrid automata can be in one of 4 possibilities, constant speed, accelerating, stopped and breaking. As well as the state it also has 6 fields of sort Nat representing the distance, speed, acceleration, movement authority, current track segment and max speed.

```
sort TrainState .
ops  cons acc stop break :  -> TrainState [ctor] .

class Train | state : TrainState, dist : Nat, speed : Nat, ac : Nat,
                                ma : Nat, tseg : Nat , maxspeed : Nat .
```

We capture the behaviour of trains using a number of instantaneous and tick rewriting rules. The instantaneous rules are used to capture a change of state within a train. For example if the a train is in the cons state distance to the movement authority becomes less than the distance needed to break to zero in the next moment of time one of these rules will fire and cause a state transition.

Modelling the Radio Block Processor

We define the RBC state as follows:

```
sort RBCState .
ops rbcidle ready wait :  -> RBCState [ctor] .
```

We define the RBC class as follows:

```
class RBC | state :  RBCState, lasttrain :  Oid, ma :  MapON, pos :  MapON ,
                              curreq :  SetO .
```

We use maps to store the current movement authorities and positions of the various trains. In order to prevent an infinite amount of requests being made to an interlocking we limit the rbc to make one request on behalf of each train to the interlocking. To do this we use `curreq`, a set of oids, to store which trains have had a request made for them at a given moment of time.

An example RBC transition is as follows:

```
 rl [rbcgrant] :  {grant(N) < O : RBC | state :  wait, lasttrain :  T1, ma :
           MAP1 > REST} => { < O : RBC | state :  rbcidle, ma :
```

```
insert(T1,endoftrack(N),MAP1) > grantma(T1, endoftrack(N)) REST} .
```

Modelling the Interlocking

An interlocking has two states, either it is idle or it has received a request for a movement authority and must respond. We define an Interlockings state in Real Time Maude as follows:

```
sort InterState .
ops idle res :  -> InterState [ctor] .
```

The interlocking class consists of a interlocking state, a request id which stores the current track segment requestions and 5 Booleans which indicate whether or not a track segment is occupied. We define an Interlocking class in Real Time Maude as follows:

```
class Inter | state :  InterState, reqid :  Nat, t0 :  Bool , t1 :  Bool , t2
                    :  Bool , t3 :  Bool , t4 :  Bool .
```

All of the rewrite rules on Interlocking objects are instantaneous and do not effect the transition of time. We can split these rules into two types. Firstly there are interlocking rules that deal with a request

Secondly there are several rewriting rules that deal with granting or denying a received request. There is currently a rule for both denying and granting a request for each track segment depending on the value of the boolean variable.

```
rl [resreq1g] :  < O : Inter | state :  res, t1 :  false, reqid :  1 > REST
             => < O : Inter | state :  idle > grant(1) REST .
```

5.5.1 Simulating the European Rail Traffic Management System Using Real Time Maude

We will now see how this executable specification can be used to simulate and obtain a better understanding of the modelled system. For this purpose we have written a small Haskell program that takes the output from multiple timed rewrites and plots them as a graph.

In the following we use an example initial state containing two trains one of which is slower (train1) than the other (train2), an interlocking and a radio block processor. The faster train will eventually catch up with the slow train and it waits for that train to clear all successive track segments before proceeding. The trains start at positions on opposing sides of the track with train1 starting at 0 and train2 starting at 50. The RBC and the interlocking are both initialised with the trains in these positions.

The model of ERTMS in the is executed using the following command:

(trew initialstate2 in time ¡= 10 .)

In the resulting output at time 10 both of the trains have moved a considerable distance and have requested and received new movement authorities.

In fig**??** we see that train2 never enters the same track segment as train1.

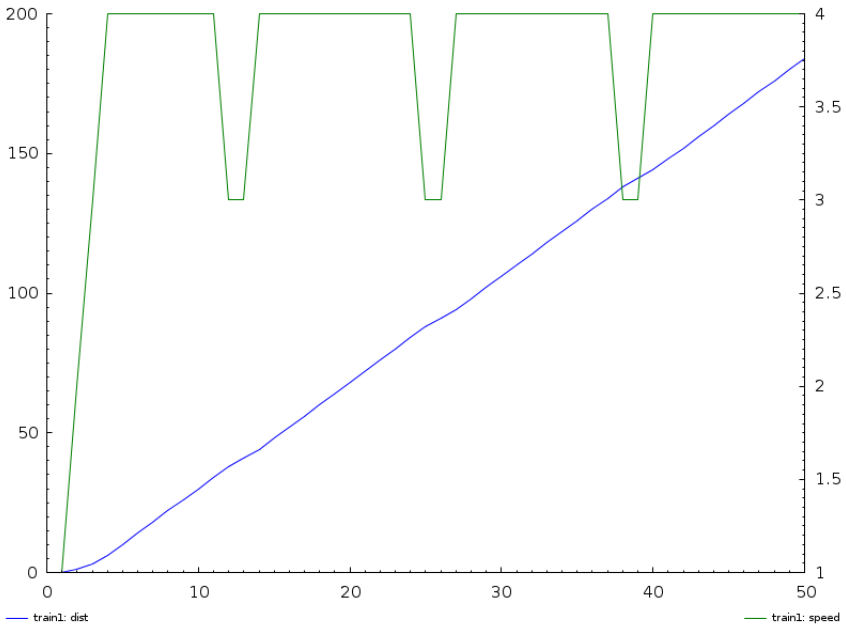


Figure 5.2: A graph comparing the distance and speed of train1

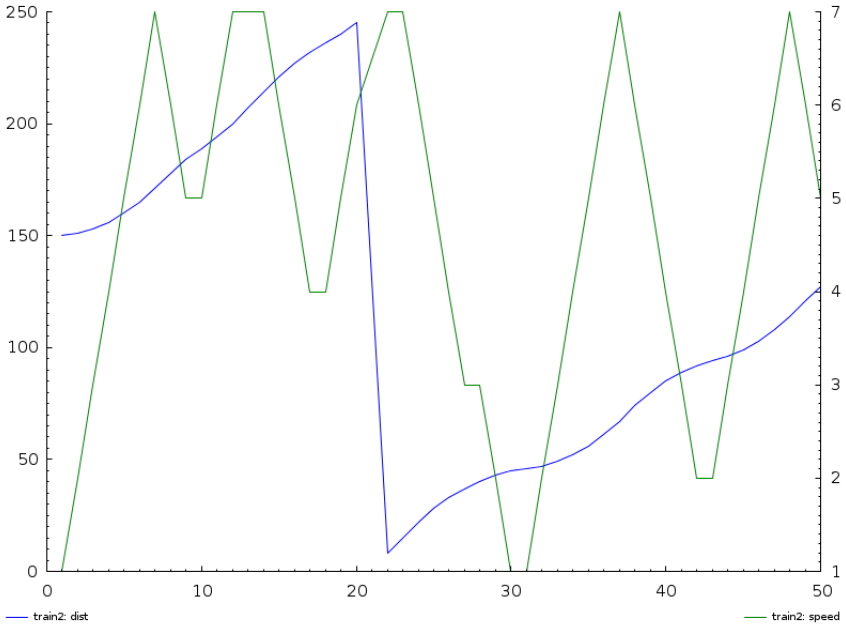


Figure 5.3: A graph comparing the distance and speed of train2

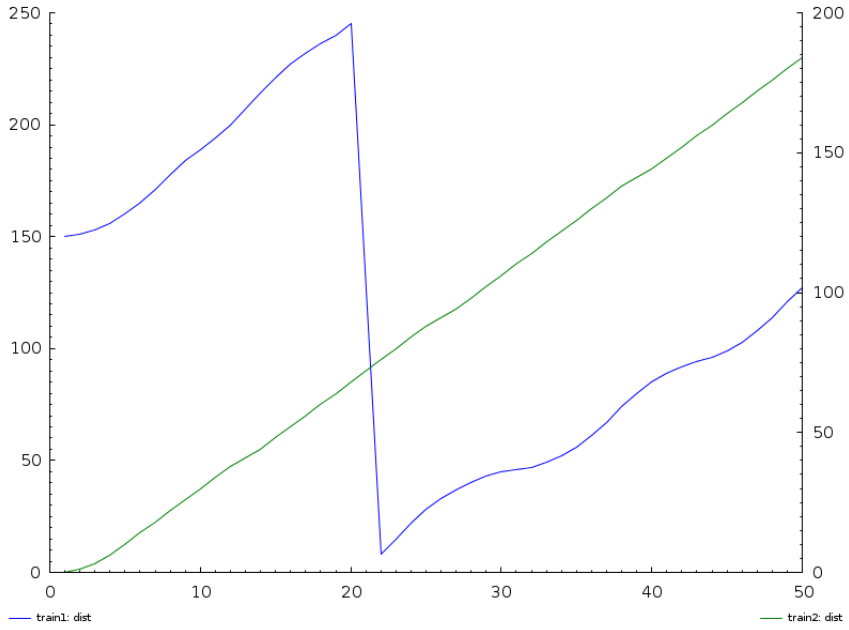


Figure 5.4: A graph comparing the distances of train1 and train2

5.6 The Maude Linear Temporal Logic Model Checker

The Real Time Maude system includes a model checker for linear temporal logic [**?**]. In the following we shall present formal definitions for linear temporal logic and the model checking problem for formulae in this logic.

5.6.1 Linear Temporal Logic

In order to perform model checking over a system we typically need a formal language that allows one to speak about time. We need to be able to formalise sentences such as "the next moment of time" and "all moments in time in the future". One such logic that allows us to formalise these statements is linear temporal logic (LTL)[**?**].

Definition 8 (Atomic Propositions)**.** *Given a set of symbols S we inductively define the set of atomic propositions AP as follows:*

- *If s is in the set of symbols $s \in S$ then $s \in AP$.*
- *if p_1 is an atomic proposition $p_1 \in AP$ then $\neg p_1 \in AP$.*
- *given two atomic propositions p_1 and p_2 then $p_1 \circ p_2 \in AP$ where $\circ$ is a propositional connective $\circ \in \{\wedge, \vee, \rightarrow\}$.*

Using these atomic propositions combined with the temporal operators it is possible to define a syntax for temporal logic formulae.

Definition 9 (Syntax of Linear Temporal Logic)**.** *Let AP be the set of atomic proposition names then:*

- $\top$ *and* $\bot$ *are well formed formulas.*
- *if* $p \in AP$ *then* p *is well formed formula (wff).*
- *if* f *and* g *are wff then* $\star f$ *and* $f \circ g$ *are wffs where* $\star \in \{\neg, \mathbf{X}, \mathbf{G}, \mathbf{F}\}$ *and* $\circ \in \{\wedge, \vee, \boldsymbol{R}, \boldsymbol{U}\}$.

LTL operations can be used to speak about paths through a system specified as a Kripke structure. a *path* is a sequence of states $s_1, \ldots s_n$ and a *path formula* is one that holds in each given state of a path.

We shall now look at the semantics of LTL firstly using an informal description of the LTL operations and secondly by giving a formal semantics for LTL. The following is a description of the 5 LTL operations over paths of a Kripke Structure.

- **X** f : The property f holds in the *next* moment of time.
- **G** f : The property f is *globally* true. i.e. it holds for all times on all paths.
- **F** f : The property f is *finally* true. i.e. there exists a time such that the property f holds on a path.
- f **U** g : For all paths the property globally f holds *until* property g holds.
- f **R** g : f holds up to and including the point when g holds.

Definition 10 (Semantics of Linear Temporal Logic)**.** *Linear Temporal Logic has the following operations:*

5.7 Model Checking the European Rail Traffic Management System

In the following section we will demonstrate an approach to apply the Real Time Maude LTL model checker to verify the Real Time Maude specification described previously. Firstly we shall define the property we want to check in the Real Time Maude system. To do this we have to define a satisfaction relation which describes what it means for the property to hold in a state of the system we are checking. The property we shall consider in this case is that the moment authorities of two trains in system do not overlap. Logical property we define identifies the individual trains using their object identifiers and then computes whether or not an over lap has occured using a boolean formula. We shall show that it is possible to verify this property over a time period large enough for all normal behaviours of the system to occur.

Defining a Satisfaction Relation

We shall now describe how to define the behaviour of the satisfaction relation for a given system and property as a Real Time Maude specification. The satisfaction relation itself is predefined in a specification as an operation which takes a State and a property which is of type Prop and computes a boolean. Maude defines the satisfaction relation $\models$ in the following module:

```
fmod SATISFACTION is
  protecting BOOL .
  sorts State Prop .
  op _|=_ : State Prop -> Bool [frozen] .
endfm
```

The sorts `State` and `Prop` are undefined as is the behaviour of $\models$. It is left to the user of the model checker to define the behaviour for their own purposes.

We can check that a train is in a specific state as follows: `ops train-cons : Oid -> Prop [ctor] .`
`eq REST < O1 : Train | state : cons > |= train-cons(O1') = (O1 == O1') .`

We can check that a train has a certain speed as follows:

`op train-s : Oid Nat -> Prop [ctor] .`
`eq REST < O1 : Train | speed : N1 > |= train-s(O1',N1') = (N1 == N1') and (O1 == O1') .`

The main movement authority that we want to check is that the movement authorities of two trains does not over lap. To do this we define a Property nomaoverlap(O1,O2) of type `Oid Oid -> Prop [ctor]` as follows:

```
eq REST < O1 : Train | dist : D1 , ma : M1 > < O2 : Train | dist : D2 ,
    ma : M2 > |= nomaoverlap(O1',O2') = (O1 == O1') and (O2 == O2') and
                          noolap(D1,M1,D2,M2) .
```

This definition features an external operation which computes a boolean depending on whether a set of inequalities are satisfied.

```
 eq noolap(D1,M1,D2,M2) = ((D1 < M1) and (M1 < D2) and (D2 < M2)) or ((D2 <
M2) and (M2 < D1) and (D1 < M1)) or ((D1 < M1) and (M2 < D1) and (M1 < D2))
               or ((D2 < M2) and (M1 < D2) and (M2 < D1) ) .
```

5.7.1 Execution of The LTL Model Checker

The Real Time MaudeLTL model checker is then called using the following command:

```
(mc initialstate2 |=t [] nomaoverlap(train1,train2) in time <= 30 .)
```

The command states that we are applying timed model checking to check that it is globally true that the movement authorities of the trains do not over lap at all moments in time less than or equal to 30. This results in the following output from real time maude which indicates that the property was succesfully verified in 3 minutes and 22 seconds.

```
rewrites: 138889387 in 179663ms cpu (202024ms real) (773054 rewrites/second)

Model check initialstate2 |=t[]nomaoverlap(train1,train2)in
MODEL-CHECK-ERTMS8 in time <= 30 with mode deterministic time increase
```

```
Result Bool :
  true
```

Unfortunately the Real Time MaudeLTL model checker can not detect the congruence between difference configurations of control system. Therefore the state space is infinite and it is impossible to do untimed model checking on the above model checking problem. It is however possible to do timed model checking within a reasonable time limit that allows for all behaviours of the system to be verified.

Bibliography

[CDE+03] M. Clavel, F. Durn, S. Eker, P. Lincoln, N. Mart-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 system. In R. Nieuwenhuis, editor, *International Conference on Rewriting Techniques and Applications (RTA)*, volume 2706 of *Lecture Notes in Computer Science*, pages 76–87. Springer Berlin Heidelberg, 2003.

[EMS02] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In F. Gadducci and U. Montanari, editors, *Fourth Workshop on Rewriting Logic and its Applications, WRLA '02*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[FH98] W. Fokkink and P. Hollingshead. Verification of interlockings: from control tables to ladder logic diagrams. In J. F. Groote, S. P. Luttik, and J. J. van Wamel, editors, *Proceedings of the 3rd Workshop on Formal Methods for Industrial Critical Systems - FMICS'98*, pages 171–185, 1998.

[Hen96] T. A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, pages 278–292. IEEE Computer Society, 1996.

[KR01] D. Kerr and T. Rowbotham. Introduction to Railway Signalling. *Institution of Railway Signal Engineers*, 2001.

[Mau] Maude. `http://maude.cs.uiuc.edu/`. Accessed: 2014-01-22.

[ÖM02] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285:359–405, 2002.

[ÖM04] P. C. Ölveczky and J. Meseguer. Specification and analysis of real-time systems using Real-Time Maude. In T. Margaria and M. Wermelinger, editors, *Fundamental Approaches to Software Engineering (FASE 2004)*, volume 2984 of *Lecture Notes in Computer Science*, pages 354–358. Springer, 2004.

[Pnu77] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, 1977.

[RTM] Real Time Maude. `http://heim.ifi.uio.no/peterol/RealTimeMaude/`. Accessed: 2014-01-22.