Modelling of Railway Control Systems in the Maude System

Initial Document

Lam Chak Yan - 667271

October 2014

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

Contents

Chapter 1

Introduction

1.1 Motivation

The number of computerized systems has been growing rapidly. There is high demand of computer systems. System failures may result in deaths and injuries. It is important to ensure the system meets all safety properties especially in a transport control system. Railway systems are computerized, they manage junctions and signals, avoiding collisions. However, there are lots of traffic accidents in the past which were caused by system failures. On June 22, 2009, two Red Line Metrorail trains crashed in the U.S. [15]. Nine people were killed. Failure of the signal system and operator error were the main reasons for the accident. The signal system is used to control the speed and braking of the trains, it cooperates with another system which detects the position of the trains and maintain the distance between them. The system should force the train to apply the brake if it is too close to another train. Ironically, the deaths and injuries are mainly caused by system failure. Another example is a train crash in Seoul [1]. On May 2, 2014, a subway train crashed into the rear of a stationary train. More than 200 people were injured. The use of two different warning systems in different lines seem to be the main reason for the accident. On 24 July 2013, the Santiago de Compostela derailment occurred [21]. A high-speed train in Spain derailed on a bend, results in 79 deaths. It was caused by the error of the train drivers. The train was travelling at 180km/h on an 80km/h curve, which is significantly over the speed limit. There was only only a speed reduction signal not far behind the bend. A safety-critical system can reduce the number of accidents occurred and save thousands of lives. Railway Control System should be automated. We cannot afford such a huge loss which is caused by a simple human error.

This thesis is concerned about modelling Railway Control Systems. In this thesis we will model and verify Railway Control Systems using Real-Time Maude. Then we can evaluate the feasibility of model checking on Railway using Real-Time Maude. We are interested in correct software therefore we will try to do it in different modelling approaches. This involves a lot of practical applications of formal methods. Formal method is a mathematically based technique that can be used for the development and verification of software and hardware system. We will mainly focus on formal verification in this dissertation. This dissertation first give an introduction about the Maude system with some background information. After that several case studies will be discussed. Then we will present with examples and explanations.

1.2 Project Aims and Approach

The aim of this project is to investigate how suitable the Maude system can be used for the verification within the railway domain. In order to archive this aim, we will do several case studies. In particular, we will demonstrate a simple example for the application of modelling in pelican crossing using the Maude system. We will apply model checking for the verification of safety conditions. Real-Time Maude can be used to simulate a concurrent pelican crossing control system with a given time. Then, we will try to formalize ladder logic in Maude. As ladder logic can be translated into propositional logic, it can also be modeled in Maude. Therefore every specification including pelican crossing can be run on this ladder logic program.

More concretely, the specific aims are as follows:

1. To study Maude and its extensions as well as its underlying theory.
2. To study techniques and case studies used in Railway domain.
3. To conduct case studies such as the pelican crossing in Maude. This will include formalism in Maude as well as proving safety properties.
4. To model ladder logic in Maude.
5. Possibly apply it to a large case study in the railway domain.

1.3 Related Work

Various work has been done in the past related to the formal methods in railway domain or other aspects. Andrew Lawrence has verified railway interlockings by using a software SCADE suite [13]. Verification of solid state interlocking programs has been explored by Swansea Railway Verification Group [10]. Model checking and verification in railway interlocking have been done in SCADE. Karim Kanso has done a formal verification on ladder logic [12]. The ladder logic is translated into propositional formulae based model in order to verify the safety conditions. Phillip James has also used various SAT-based model checking techniques to verify train control systems [11]. A wireless sensor network is simulated in Real-Time Maude by Peter Csaba Ölveczky and Stian Thorvaldsen [20]. Formal analysis on the specific OGDC algorithm has been performed.

Chapter 2

Maude

2.1 Full Maude: Basic Information

In this chapter we will give background information on Maude and its basic syntax. Maude is a high-level programming language and high-performance system that can be used for implementing rewriting logic. Rewriting logic can be interpreted as a logic for concurrent computation. It can be applied to various concurrent models. The Maude system is associated with mathematical based concept, it supports equational specification. Equations and rules are the basic statements in Maude. They both obey the rewriting semantics i.e. things on the left can be replaced by those on the right. More examples of equational logic will be shown below. Various works related to the specification and verification in Maude have been done in the past [2]. Discrete event models such as traffic light has been simulated in Real-Time Maude.

Here we have several definitions regarding the Maude system. *Module* is the basic unit of a Maude program which contains syntax declarations. This functional module begins with **fmod** and ends with **endfm**. The simplest component for specification is the data type. It is declared as *sort*(*s*) in Maude. Its subtype is more specific, declared as *subsort*(*s*). If we define real-number as a sort then *natural-number* < *real-number* can be the subsort. Here we use a functional module STACK as an example. At the beginning, the module can be written in this way.

```
fmod STACK is
   sorts Stack Entity NEStack  .
   subsort NEStack < Stack .
endfm
```

If we run the program, nothing will be displayed as the module does not know what to be computed. It is not completed yet. Therefore we need operations that can handle the sorts in order to produce the type of result that we are looking for. Each operator should have a name and argument(s). Equations are used to specify how the operation works in different scenarios. We need to declare variables for the declaration of equation. The following is a complete version of the stack example.

```
fmod STACK is
   sorts Stack Entity NEStack  .
   subsort NEStack < Stack .

   op push : Stack Entity -> NEStack .
   op pop : NEStack -> Stack .
   op top : NEStack -> Entity .

   vars E F : Entity .
   var S : Stack .

   eq pop(push(S,E)) = S .
   eq top(push(S,E)) = E .
endfm
```

We can test the program by running the following command :

```
red top(pop(push(push(push(S,E),F),E))) .
```

red is the abbreviation of reduction. We try to add E to the top of the stack S, then we add F follows by the addition of E, after that we remove the top entity. We are looking for the entity at the top. The result obtained is F. Here is the response from the program.

```
reduce in STACK : top(pop(push(push(push(S, E), F), E))) .
rewrites: 2 in 0ms cpu (0ms real) (2000000 rewrites/second)
result Entity: F
```

There are other types of declaration. Below is a list of possible declarations in a functional module.

- `sort s .`
- `subsort s < s' .`
- `op f : s -> s .`
- `vars a b : s .`
- `eq a = b .`
- `ceq a = b if` *cond* `.`
- `mb a : s .`
- `cmb a : s if` *cond* `.`

2.2 Real-Time Maude

The Maude system has its extensions such as HI-Maude and Real-Time Maude. Real-Time Maude is a tool that supports the formal specification for real-time and hybrid systems based on Maude. It is suitable for specifying object-oriented real-time systems as the concurrent behaviours can be simulated. In the simulation process, we can do searching with or without a given time limit to look for a state, which can be reached from an initial state. This involves term rewriting. We can discover whether a state in the system is reachable from its initial state with this function. Real-Time Maude also supports temporal logic model checking. We are able to specify and simulate larger systems with concurrent behaviours.
There is slightly difference between Maude and Real-Time Maude in terms of the syntax. For the concurrent object-oriented systems, we use object-oriented timed modules which is written as **tomod** at the beginning of the module and **endtom** at the end of the module. The module should be surrounded by parentheses. There are instantaneous rewrite rules and tick rules which can be used in Real-Time Maude. Instantaneous rewrite rule is assumed to be taken zero time in the process as it models instantaneous transition. Here we have the syntax of an instantaneous rewrite rule as an example:

```
rl [l] : s => s' .
```

This represents a one-step transition from state s to s'. Both s and s' have the sort system. *l* is the label for the rewriting logic. For the tick rewriting rule, we have the following syntax:

```
rl [l] : {s} => {s'} in time t if cond .
```

There is a sort GlobalSystem which is denoted by a constructor {_} in Real-Time Maude. {s} means the entire system in state s. The transition above takes *t* number of time-steps. Real-Time Maude is also suitable for object-oriented system. For declaring a object in an object-oriented module, we first need to declare a class. We use a class *C* as an example:

```
class C | a0 : s0, ..., an : sn
```

Class *C* contains several attributes a_0 to a_n with respective sorts s_0 to s_n. Object is an instance of a class. **Configuration** is the sort that represents an object's state. An object has the following format:

```
< O : C | att0 : val0, ..., attn : valn >
```

In order to get familiar with the Real-Time Maude system, we will also model a simple clock in an object-oriented timed module.

2.2.1 Example: Modelling a clock

```
(tomod CLOCK is protecting NAT-TIME-DOMAIN .
   class Clock | hour : Time , minute : Time , second : Time .
   op c1 : -> Oid [ctor] .
   op initstate : -> Configuration .
   eq initstate = < c1 : Clock | hour : 0 , minute : 0 , second : 0 > .
   var O : Oid .
   vars S M H : Time .

   crl [second] : {< O : Clock | second : S >}
      => {< O : Clock | second : S + 1 >} in time 1 if S < 59 .
```

Above is the first ten lines of the simple clock model program. The whole implementation is given in Appendix A. This module is including another module called NAT-TIME-DOMAIN. That is one of the predefined modules for the time domain. We want the clock to show the time correctly and that the hours, minutes and seconds can be counted and shown at any elapsed time since the initial state. We have a constructor *c*1 as an object identifier, we assume *c*1 is a clock. The initial state of this clock has been set. It has the sort **Configuration**. All attributes are set to zero. We have several conditional tick rules. Each rule is simulating one time in each step, which will be one second in this example. We can check the state of the clock system with given elapsed time. We can try the following command :

```
(trew {initstate} in time <= 13880 .)
```

This timed rewrite command performs the simulation by showing the state of the system after the given number of time units. We want to see what time will be displayed by clock *c*1 after 13880 time units (seconds). The result is shown below:

```
rewrites: 252679 in 391ms cpu (391ms real)
   (645765 rewrites/second)

Timed rewrite {initstate} in CLOCK
   with mode deterministic time increase in time <= 13880

Result ClockedSystem :
  {< c1 : Clock | hour : 3,minute : 51,second : 20 >} in time 13880
```

As shown by the result, 3 hours 51 minutes and 20 seconds has elapsed after 13880 time units as seconds. We can verified this by the formula $(3 \times 60 \times 60) + (51 \times 60) + 20 = 13880$. However, can we conclude that the clock is always correct? In order to ensure the system is correct and safe, we should perform verification, it can be done by applying the model checking. Model checking will be discussed in chapter 4.

Chapter 3

Introduction to the Railway Domain

3.1 History

The railway system was established in the 1800s. Trains were controlled by train drivers. They receive signals from Policemen who stand at stations and junctions. However this method is not reliable. The position of the trains could not be tracked therefore accidents often happened. Then fixed signal was introduced. It replaced the hand signal. However one signal was still controlled by one person. Therefore signal box was introduced, it is a point which connects signals. Message could be sent by a signalman. The introduction of signal box still caused some accidents. In order to improve the reliability, interlocking and block signal were developed. It became mandatory in the late 1800s.

3.1.1 Interlocking

The interlocking is a critical part in railway. It prevents conflicting movement of the trains by locking the levers physically. A reliable interlocking provides safety-critical functions and avoids collision between trains. A control system cannot directly control the railway without the interlocking. The interlocking is act as a medium between the railway control system and the physical side of the railway. It operates in discrete cycles. The procedure can be explained in few steps.

1. There are new requests in every cycle, the control signal will be passed to the interlocking by the control system.
2. The interlocking receives signal and executes the program, it only accepts requests that will not cause any conflict in the route configuration. Output will be passed to the railway.
3. Actions performed on the railway will be sent back to the interlocking as signal.
4. The interlocking updates its route configuration and informs the control system.
5. Control system receives the response for the request. Accepted request will be committed in the next cycle.

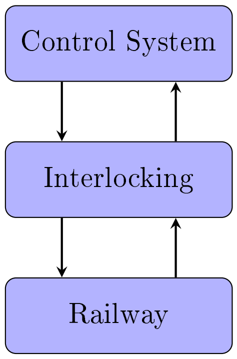


Figure 3.1: The Control Flow in Each Cycle

3.2 Ladder Logic

Ladder diagrams are often applied in control logic systems. This graphical ladder like diagram is implemented by engineers for representing sets of boolean equations. It is widely used in railway interlockings. A ladder consists of one or more horizontal rungs. Each rung typically consists of a coil at the right, contact(s) may exists on the left. A ladder logic diagram looks like **Figure 3.2**.

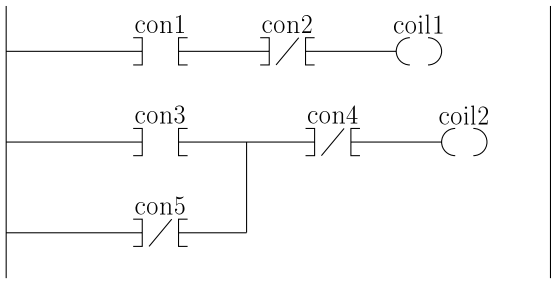


Figure 3.2: An Example of Ladder Logic Diagram

The entities will be described in the following table.

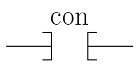
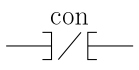
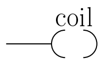
Entity	**Symbol**	**Function**	**Type**
Open Contact		An open contact is a boolean variable. It contains one of the inputs in a rung. Its value is un-negated.	Input: Boolean
Closed Contact		A closed contact is a boolean variable. It contains one of the inputs in a rung. It is read as the negation of its value.	Input: Boolean
Coil		A coil represents the output of its rung. Each rung is computed from left to right with connectives and contacts as inputs. The coils are computed from the top to bottom. The value of the coil could be used by the rungs below. The value of the coils can be stored.	Output: Boolean

Table 3.1: Entities in Ladder Logic

Apart from the basic entities, logical connectives are used to connect different contacts.

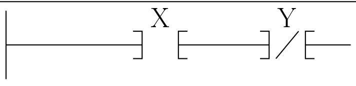
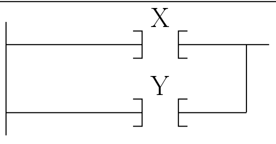
Connective	**Representation**	**Propositional logic**
Conjunction		$X \wedge \neg Y$
Disjunction		$X \vee Y$

Table 3.2: Connectives

Chapter 4

Case Study: Pelican Crossing in Maude

4.1 Introduction

In this chapter we will have our first case study which is conducted in Maude. It is a pelican crossing example which simulates the behaviours of a pelican crossing control system. It has been modeled in SCADE by Lawrence [13]. A pelican crossing control system controls two sets of lights for pedestrians and traffic respectively. Each of the lights shows either green or red signal. Two traffic lights will normally show the same signal, red signal means the traffic flow should stop and green means to proceed. The same thing applies to pedestrian lights, red signal means pedestrians should wait and green means they are safe to cross the road. Pedestrians can press the button on the control panel when they want to cross the road, a request will be made. We would like to model how these buttons interact with the traffic lights. The pelican crossing control system has been formalised by James [9]. It can can be expressed in ladder logic. As every set of ladder logic corresponds to some propositional logic, we first present these rules by a set of propositional logic:

$$\begin{aligned}
crossing' &\Leftrightarrow (req \wedge \neg\, crossing) \\
req' &\Leftrightarrow (pressed \wedge \neg\, req) \\
tlag &\Leftrightarrow ((\neg\, crossing') \wedge (\neg\, pressed \vee req')) \\
tlbg &\Leftrightarrow ((\neg\, crossing') \wedge (\neg\, pressed \vee req')) \\
tlar &\Leftrightarrow crossing' \\
tlbr &\Leftrightarrow crossing' \\
plag &\Leftrightarrow crossing' \\
plbg &\Leftrightarrow crossing' \\
plar &\Leftrightarrow (\neg\, crossing') \\
plbr &\Leftrightarrow (\neg\, crossing')
\end{aligned}$$

There are several variables. *req* means required, it represents a request to cross the road as a button has been pressed in the previous iteration. *crossing* means pedestrian is crossing the road. *pressed* means a button is pressed in current iteration. Other variables represent light signals.

4.2 Modelling in Maude

We will model the system by translating above formulae into code and run it in Full Maude. The following is an object-oriented module.

```
(omod PELICAN is
   class System | crossing : Bool , req : Bool , tlag : Bool ,
   tlbg : Bool , tlar : Bool , tlbr : Bool , plag : Bool ,
   plbg : Bool , plar : Bool , plbr : Bool .

   vars S D : Oid .
   vars A B C R : Bool .
   vars tlar' tlbr' plar' plbr' tlag' tlbg' plag' plbg' : Bool .
   op system1 : -> Oid [ctor] .
   ops crossing' req' : Bool Bool -> Bool .
   eq crossing'(A,B) = A and not B .
   eq req'(A,B) = A and not B .

   op press : Configuration -> Configuration .
   eq press(< S : System | crossing : C , req : R >) =
    < S : System | crossing : crossing'(R,C) , req : req'(true,R) ,
   tlag : (req'(true,R)) and crossing'(R,C) == false ,
   tlar : crossing'(R,C) ,
   tlbg : (req'(true,R)) and crossing'(R,C) == false ,
   tlbr : crossing'(R,C) ,
   plag : crossing'(R,C) , plar : crossing'(R,C) == false ,
   plbg : crossing'(R,C) , plbr : crossing'(R,C) == false > .

   op notpress : Configuration -> Configuration .
   eq notpress(< S : System | crossing : C , req : R >) =
    < S : System | crossing : crossing'(R,C) , req : req'(false,R) ,
   tlag : crossing'(R,C) == false ,
   tlar : crossing'(R,C) ,
   tlbg : crossing'(R,C) == false ,
   tlbr : crossing'(R,C) ,
   plag : crossing'(R,C) , plar : crossing'(R,C) == false ,
   plbg : crossing'(R,C) , plbr : crossing'(R,C) == false > .

   op initstate : -> Configuration .
   eq initstate = < system1 : System | crossing : false
   , req : false , tlag : false , tlbg : false , tlar : true ,
   tlbr : true , plag : false , plbg : false , plar : true ,
   plbr : true > .
endom)
```

This is the module that contains the operations and the equations for modelling the pelican crossing. We do not model it as timed system at this stage but this module will be modified and timed check will be conducted at a

later stage. We started with several variables *tlar′*, *tlbr′*, *plar′*, *plbr′*, *tlag′*, *tlbg′*, *plag′*, *plbg′*, which represent different states of the traffic lights. When *tlar′* is set to true, that means the traffic light is showing red. Lights for pedestrians begin with pla and plb. *e.g. plar′ and plbg′*.
The two operations *press* and *notpress* are used to simulate the action of the pedestrians (*i.e.* whether the button has been pressed by them or not). We have a configuration called initstate, it is the initial state of the control program. it is set as:

```
eq initstate = < system1 : System | crossing : false , req : false ,
tlag : false , tlbg : false , tlar : true , tlbr : true ,
plag : false , plbg : false , plar : true , plbr : true > .
```

All the attributes are set to false except the red lights which are set to true.

4.3 Running in the Specification in Maude

We can simulate a set of actions by running this command as an example:

```
 (rew notpress(press(initstate)) .)
```

rew is the abbreviation of rewrite which performs the operations or rewrite rules and produces the result. This is the result obtained:

```
rewrite in MODEL-CHECK-PELICANCROSS :
  notpress(press(initstate))

result Object :
  < system1 : System | crossing : true, plag : true, plar : false,
    plbg : true, plbr : false, req : false, tlag : false,
    tlar : true, tlbg : false, tlbr : true >
```

We are simulating that the button on the control panel is pressed once but not in the next iteration. We can see that some values in the object are changed. *crossing* is set to true, that means pedestrians are allowed to cross the road. Therefore we can see that *plag* and *plbg* are set to true, *plar* and *plbr* are set to false , meanwhile *tlar* and *tlbr* are set to true, *tlag* and *tlbg* are set to false. Since the button is not pressed in the last iteration. *req* is set to false.

4.4 Model Checking

We have to verify safety conditions and analyse the behaviours in different real-time operating systems. It can be done in Maude which is equipped with a linear temporal logic explicit-state model checker [6]. Temporal logic consists of atom propositions and temporal operators. An untimed and time-bounded model checking can be done in Real-Time Maude. We are interested in real-time system therefore timed module will be used for model checking in pelican crossing. Conversion and changes are made from the previous module. It is turned into object-oriented timed module (**tomod**). We used two rewrite rules **[pressed]** and **[notpressed]** to replace the operations and equations. Therefore the system can be simulated in any given time limit. We also have an additional module which includes the predefined module *TIMED-MODEL-CHECKER*. We can define the condition in which the properties hold. Here is one of the properties in pelican crossing:

```
eq {< system1 : System | tlar : tlar':Bool , tlag : tlag':Bool ,
tlbr : tlbr':Bool , tlbg : tlbg':Bool , crossing : C:Bool ,
plar : plar':Bool , plag : plag':Bool , plbr : plbr':Bool ,
plbg : plbg':Bool >} |=
safecross(system1) = ((C and plag' and plbg' and tlar' and tlbr') or
(C == false)) .
```

We are looking for a property that no car should be passing when the pedestrians are allowed to cross the road. C represents crossing. This $safecross$ property will return true if one of the following holds:

$$C \;==\; false$$
$$C \;\wedge\; plag' \;\wedge\; plbg' \;\wedge\; tlar' \;\wedge\; tlbr' \;==\; true$$

We can test the safety properties by running these commands:

```
(mc {initstate} |=t [] safecross(system1) in time <= 100 .)
(mc {initstate} |=t [] safelight(system1) in time <= 100 .)
(mc {initstate} |=t [] counter(system1) in time <= 3 .)
```

This is the result obtained for the $safecross$ property:

```
rewrites: 23378 in 25ms cpu (25ms real) (935120 rewrites/second)

Model check {initstate} |=[]safecross(system1) in
   MODEL-CHECK-PELICANCROSS in time
    <= 100 with mode deterministic time increase

Result Bool :
  true
```

It shows that the control program for the pelican crossing satisfies the $safecross$ properties as the result is true in 100 time steps while it covers all the states in the system. Another property *safelight* is verified, we ensure no pedestrian light and traffic light can be green at the same time. In addition, we want to make sure not every cases are true in model checking, an incorrect property *counter* is tested as a counter example. Violations to the property can be found. The full implementation is given in Appendix B.

Chapter 5

Planning

5.1 Milestones

In this chapter, we will explain about our project plan. The main steps of whole project will be as follows:

1. Learning Maude: Study lecture material [8] and resource on the internet.
2. Overview on Railway domain: Study thesis of Andrew Lawrence [13].
3. Pelican crossing example running in Maude.
4. Writing initial document.
5. Specification of modelling ladder logic.
6. Study model checking in detail: apply to pelican crossing.
7. Ladder logic modelled in Maude.
8. Running case study with ladder logic implemented in 6.
9. Building larger examples.
10. Study Hi-Maude.
11. Possible work related to Hi-Maude.

The milestones will be as follows:

1. Complete first own case study in Maude: pelican crossing including model checking.
2. Complete formalisation of ladder logic in Maude/Real-Time Maude.
3. Complete larger case study in Real-Time Maude.
4. Complete own case study in Hi-Maude.

Study on Hi-Maude may be introduced only if everything is following the schedule. The timeline can be presented by a Gantt chart in the next section. M_n represents the milestone number n.

5.2 Timeline

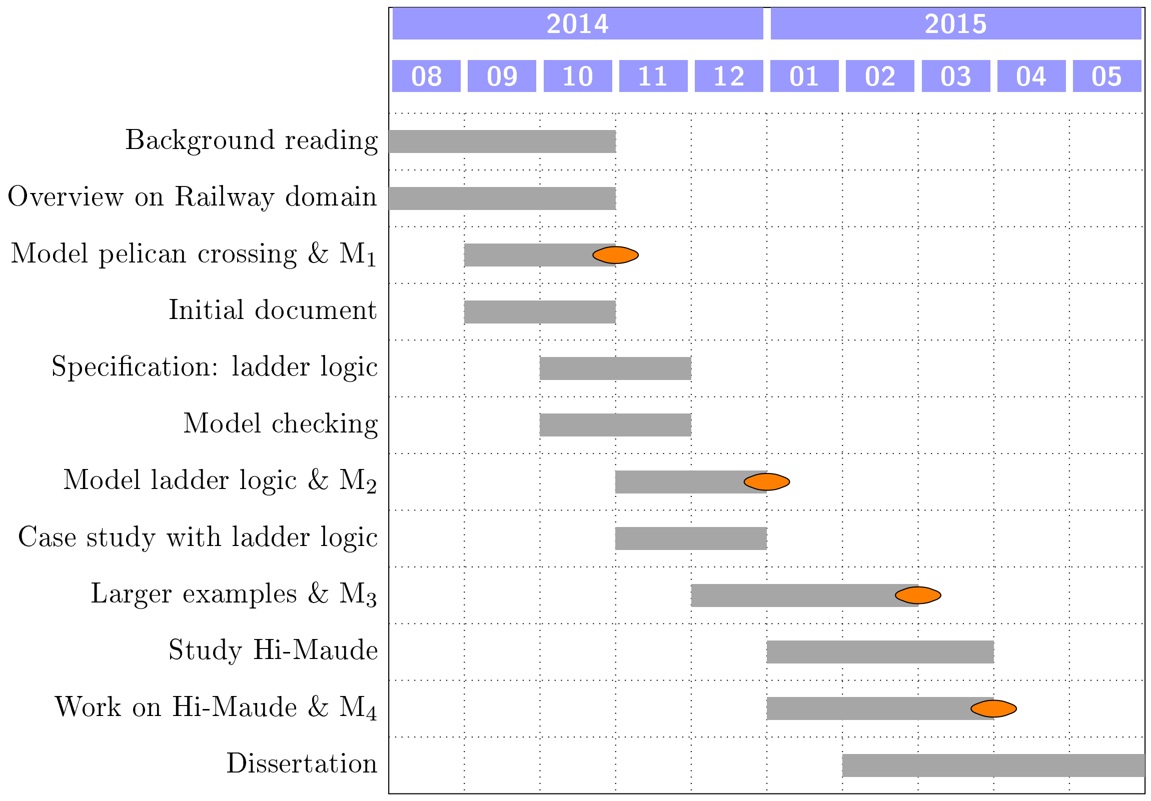


Figure 5.1: Gantt Chart

This project has been started since the beginning of August, some background reading on Maude has been done. It includes basic concept of this language, information about rewriting logic, syntax and semantics of Full Maude and Real-Time Maude [16]. Some articles and theses written in the area of Railway domain have been studied at the same period of time. These also contain ladder logic and pelican crossing. **Chapter 3** contains the explanation on these subjects. After that several examples such as Stack and Clock were created in Maude followed by the pelican crossing (M_1 is reached). All those work is done before the initial document is submitted. Specification of the ladder logic will be done. Model checking is being studied at the same period of time including underlying theories such as Linear Temporal Logic. Model checking is included in the first case study under section 4.4, safety conditions are done in the pelican crossing example. Then, ladder logic should be formalised with the help of the specification written earlier (M_2 is reached). Case studies should be conducted with ladder logic follows by larger examples (M_3 is reached). Work related to Hi-Maude (*e.g. To model a system*) maybe done (M_4 is reached) and dissertation will be written by the beginning of May 2015.

5.3 Development Model

We can apply a development model to the task. Waterfall model, spiral model and V-model are common methods in the software development process. Waterfall model is not a good choice in this situation as it follows a too tight structure. Every stage is gone through step by step. *i.e. Requirements → Design → Implementation → Verification → Maintenance.* It works like a waterfall. The disadvantage is that changes are often needed. We cannot guarantee everything in a stage is finalized before proceeding to the next stage. It will be cost ineffective if we want to make amendment to the preceding stages, it is especially difficult in modelling. We may experience difficulties in the implementation process, or we may want to specify more features and functions on larger systems. V-model is regarded as a extended waterfall model. The inflexibility is also a massive hindrance in the area of formal methods. Hence waterfall model and V-model are not suitable.

A spiral model would be fit to this project. It is like a combination of waterfall and rapid prototype model. It takes the iterative approach to the software development. The project starts from a small prototype, it may undergoes a certain number of cycles, each cycle follows the stages under waterfall model. One of the main features about spiral model is the emphasis on risk analysis. At the beginning of each cycle, we should aware of the potential risks in that cycle. A review should be done at the end of the cycle. The project can be started with a simple module. More modules will be written after several cycles and these modules can cooperate, more features can be built. We can keep improving the program until we are satisfied. Risk analysis also helps to maintain the stability of the system. Hence larger object-oriented systems can be well-suited with spiral model.

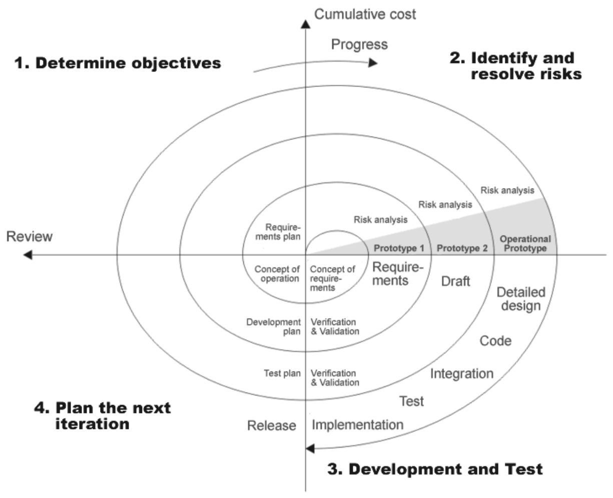


Figure 5.2: A Spiral Model (Boehm, 1988)

Chapter 6

Risk Management

In this chapter we will describe how the risk management would be done.

6.1 Risk Analysis

We will be analysing the risks which may be encountered throughout this project. Below is a table that summarises the possible risks in this project.

Risk number	**Risk**	**Specific relation to this project**
1	Some of the parts overrun.	It may happen due to a task or its theory is too difficult to understand. Extra time may be needed for changes in the implementation.
2	Some tasks are too difficult to complete.	Likely to happen in writing larger examples and ladder logic, (*e.g.* Applying model-checking to ladder logic.) LTL model checking maybe difficult to understand.
3	Machine crashes or accidental data loss.	Possible in any project.
4	Problems with integrating tools.	Possible to happen in Hi-Maude as specific version of Full-Maude is needed for using some extensions.
5	Unsuitable software development model.	Possible to happen while writing programs in Maude.
6	Lack of validation of the program.	Possible to happen where the characteristics of the real system are not really being modeled despite the program itself is correct.

Table 6.1: Risk Identification

	Consequence		
Probability	Little	Moderate	Serious
High		**1**	
Medium			**2,6**
Low		**5**	**3,4**

increasing severity

increasing severity

Table 6.2: Severity Measurement

In the table, the numbers refer to the Risk number in table 6.1. (1) and (2) are seen as more dangerous risks. (1) may not cause much impact to the overall project, however it is the most common problem in any project, it is very likely to happen therefore this risk must be handled and monitored. (2) is not much common but the completeness of tasks is important. It may affect the following work. Therefore, the structure of the whole project may be different without the completion of certain tasks. (4) and (2) may cause similar impact which (4) is not likely to happen. (3) is happened rarely but loss of data is a disaster. With regard to (5), choosing the wrong software development model to start may affect the planning and management in the process especially in a larger system. The whole program may becomes trivial because of (6). The properties should be modeled and the understanding to the system should be clear.

6.2 Risk Management

In this section we will explain the strategies to be used for each risk we have mentioned before. There are three types of strategies which can be used. We can think of precaution and avoiding it with Avoidance Strategy (A). We can treat it and reducing its impact with Minimization Strategy (M). We can solve the problem alternatively with Contingency Plan (C).

Risk	**Strategy**
(1)Overrun	Concentrate on the important parts, consider giving up tasks that is less important *e.g.* Hi-Maude (M). Regularly monitor the time plan, modify it if necessary (M). Allow more time than expected for tasks in time plan (A).
(2)Difficult task	Follow the spiral model, prepare well at the beginning and review at the end of each cycle (A). Try to think of different approach before start doing the tasks (A). Do enough reading until familiar with the topic related to the task (A).
(3)Data loss	Keep back up files on external hard drive (C). Synchronise the file with network drive, enable real-time updates (C).
(4)Integrating tools	Try to run the tools before deciding to do any task on it (A). Contact the author of the tool if necessary (M).
(5)Software development model	Use another software development model (C). Identify the advantages and disadvantages of various models (A).
(6)Lack of validation	Ask for advice from members of the Railway Verification Group (M). Review the program especially in larger examples, check through whether the properties being modeled are corresponding to the functions created in the program (A).

Table 6.3: Strategies

6.3 Risk Monitoring

Specific strategies have been suggested for each risk. All of the risks above will be monitored. Any potential risk will be added to the list. These risks will be reviewed regularly in order to maintain the effectiveness of the project. All the steps in Risk Management Process has been explained in this chapter.

Chapter 7

Summary

This initial document provides a brief introduction to the project. Background information and the project aims have been discussed at the beginning of the document. The main idea is to investigate the feasibility of modelling Railway Control System in Maude. Chapter 2 includes the syntax and semantics of Full Maude and Real-Time Maude. Stack and Clock are used to demonstrate the ideas in Maude. Basic knowledge on the Railway Domain has been discussed. Ladder logic can be useful in modelling railway interlockings. In order to keep the progress on the right track, a case study on pelican crossing is conducted. Basic instructions on model checking has been applied. This case study is related to Maude and ladder logic. More examples in the railway domain will be built on the basis of ladder logic using Maude. The initial work is done up to this point. The following chapters are related to future work. Project plan, milestones and development model are discussed in these chapters. Towards the end of this document, the outline of risk management procedure are shown. Each of the risk has been analysed. According to the timeline, another set of work is ready to be started. The project plan will be reviewed throughout the project in order to keep good progress.

Bibliography

[1] Seoul train crash blamed on system failure. `http://www.news.com.au/world/seoul-train-crash-blamed-on-system-failure/story-fndir2ev-1226904550085`, May 2014. [Accessed: 20/08/2014].

[2] Kyungmin Bae, Peter Csaba Ölveczky, Thomas Huining Feng, and Stavros Tripakis. Verifying ptolemy ii discrete-event models using real-time maude. In Karin Breitman and Ana Cavalcanti, editors, *Formal Methods and Software Engineering*, volume 5885 of *Lecture Notes in Computer Science*, pages 717–736. Springer Berlin Heidelberg, 2009.

[3] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[4] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L Talcott. The Maude System. `http://maude.cs.uiuc.edu`. [Accessed: 10/08/2014].

[5] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, and Carolyn L Talcott. All about maude-a high-performance logical framework, how to specify, program and verify systems in rewriting logic, volume 4350 of lecture notes in computer science. *Springer*, 4:50–88, 2007.

[6] Steven Eker, José Meseguer, and Ambarish Sridharanarayananx. The Maude LTL Model Checker, 2002.

[7] Wan Fokkink and Paul Hollingshead. Verification of interlockings: from control tables to ladder logic diagrams. In *Proceedings of the 3rd Workshop on Formal Methods for Industrial Critical Systems - FMICS'98*, 1998.

[8] Neal Harman and Monika Seisenberger. Lecture Notes on CS-213 System Specification, 2011.

[9] Phillip James. SAT-based Model Checking and its applications to Train Control Systems, 2010.

[10] Phillip James, Andy Lawrence, Faron Moller, Markus Roggenbach, Monika Seisenberger, Anton Setzer, Karim Kanso, and Simon Chadwick. Verification of solid state interlocking programs. In *Software Engineering and Formal Methods*, pages 253–268. Springer, 2014.

[11] Phillip James and Markus Roggenbach. Automatically verifying railway interlockings using sat-based model checking. *10th International Workshop on Automated Verification of Critical Systems*, 35, 2010.

[12] K. Kanso and Swansea University. School of Physical Sciences. Computer Science. *Formal Verification of Ladder Logic*. Swansea University, 2008.

[13] Andy Lawrence. Verification of railway interlockings in scade, 2010.

[14] Andy Lawrence, Ulrich Berger, Markus Roggenbach, and Monika Seisenberger. Safety and performance of the european rail traffic management system-a modelling and verification exercise in real time maude. *22nd International Workshop on Algebraic Development Techniques*, 2014.

[15] Lyndsey Layton. Metro crash: Experts suspect system failure, operator error in red line accident. http://www.washingtonpost.com/wp-dyn/content/article/2009/06/22/AR2009062203261.html, June 2009. [Accessed: 22/08/2014].

[16] Theodore McCombs. Maude 2.0 primer. `http://maude.cs.uiuc.edu/primer/maude-primer.pdf`, August 2003. [Accessed: 15/08/2014].

[17] Peter Ölveczky. The real-time maude tool. `http://heim.ifi.uio.no/peterol/RealTimeMaude/`. [Accessed: 18/08/2014].

[18] Peter Csaba Ölveczky. Real-time maude 2.3 manual. `http://folk.uio.no/peterol/RealTimeMaude/intro.pdf`, August 2007. [Accessed: 18/08/2014].

[19] Peter Csaba Ölveczky and José Meseguer. Specification and analysis of real-time systems using real-time maude. In *Fundamental Approaches to Software Engineering*, pages 354–358. Springer, 2004.

[20] PeterCsaba Ölveczky and Stian Thorvaldsen. Formal modeling and analysis of the ogdc wireless sensor network algorithm in real-time maude. In MarcelloM. Bonsangue and EinarBroch Johnsen, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 4468 of *Lecture Notes in Computer Science*, pages 122–140. Springer Berlin Heidelberg, 2007.

[21] Fernando Puente. Driver error "only cause" of santiago accident, says report. `http://www.railjournal.com/index.php/europe/driver-error-only-cause-of-santiago-accident-says-report.html`, June 2014. [Accessed: 20/08/2014].

Appendix A

Simple examples

The Stack Example in Full Maude

```
fmod STACK is
   sorts Stack Entity NEStack  .
   subsort NEStack < Stack .

   op push : Stack Entity -> NEStack .
   op pop : NEStack -> Stack .
   op top : NEStack -> Entity .
   vars E F : Entity .
   var S : Stack .
   eq pop(push(S,E)) = S .
   eq top(push(S,E)) = E .
endfm
```

The Clock Example in Real-Time Maude

```
(tomod CLOCK is protecting NAT-TIME-DOMAIN .
   class Clock | hour : Time , minute : Time , second : Time .
   op c1 : -> Oid [ctor] .
   op initstate : -> Configuration .
   eq initstate = < c1 : Clock | hour : 0 , minute : 0 , second : 0 > .
   var O : Oid .
   vars S M H : Time .

   crl [second] : {< O : Clock | second : S >}
      => {< O : Clock | second : S + 1 >} in time 1 if S < 59 .

   crl [minute] : {< O : Clock | minute : M , second : S >}
      => {< O : Clock | minute : M + 1 , second : 0 >}
        in time 1 if S == 59 /\ M < 59 .

   crl [hour] : {< O : Clock | hour : H , minute : M , second : S >}
      => {< O : Clock | hour : H + 1 , minute : 0 ,  second : 0 >}
        in time 1 if S == 59 /\ M == 59 /\ H < 23 .

   crl [reset] : {< O : Clock | hour : H , minute : M , second : S >}
      => {< O : Clock | hour : 0 , minute : 0 ,  second : 0 >}
        in time 1 if S == 59 /\ M == 59 /\ H == 23 .
endtom)
```

Appendix B

The Full Source Code for the Pelican Crossing Example

```
(tomod PELICAN is protecting NAT-TIME-DOMAIN .
 class System | crossing : Bool , req : Bool , tlag : Bool , tlbg : Bool ,
 tlar : Bool , tlbr : Bool , plag : Bool , plbg : Bool , plar : Bool , plbr : Bool .

 var S : Oid .
 vars A B C R : Bool .
 op system1 : -> Oid [ctor] .
 ops crossing' req' : Bool Bool -> Bool .
 eq crossing'(A,B) = A and not B .
 eq req'(A,B) = A and not B .

 rl [pressed] : { < S : System | crossing : C , req : R > } =>
  { < S : System | crossing : crossing'(R,C) , req : req'(true,R) ,
  tlag : (req'(true,R)) and crossing'(R,C) == false ,
  tlar : crossing'(R,C) ,
  tlbg : (req'(true,R)) and crossing'(R,C) == false ,
  tlbr : crossing'(R,C) ,
  plag : crossing'(R,C) , plar : crossing'(R,C) == false ,
  plbg : crossing'(R,C) , plbr : crossing'(R,C) == false > } in time 1 .

 rl [notpressed] : { < S : System | crossing : C , req : R > } =>
  { < S : System | crossing : crossing'(R,C) , req : req'(false,R) ,
  tlag : crossing'(R,C) == false ,
  tlar : crossing'(R,C) ,
  tlbg : crossing'(R,C) == false ,
  tlbr : crossing'(R,C) ,
  plag : crossing'(R,C) , plar : crossing'(R,C) == false ,
  plbg : crossing'(R,C) , plbr : crossing'(R,C) == false > } in time 1 .

 op press : Configuration -> Configuration .
 eq press(< S : System | crossing : C , req : R >) =
  < S : System | crossing : crossing'(R,C) , req : req'(true,R) ,
  tlag : (req'(true,R)) and crossing'(R,C) == false ,
  tlar : crossing'(R,C) ,
  tlbg : (req'(true,R)) and crossing'(R,C) == false ,
  tlbr : crossing'(R,C) ,
  plag : crossing'(R,C) , plar : crossing'(R,C) == false ,
  plbg : crossing'(R,C) , plbr : crossing'(R,C) == false > .
 op notpress : Configuration -> Configuration .
 eq notpress(< S : System | crossing : C , req : R >) =
   < S : System | crossing : crossing'(R,C) , req : req'(false,R) ,
  tlag : crossing'(R,C) == false ,
  tlar : crossing'(R,C) ,
  tlbg : crossing'(R,C) == false ,
  tlbr : crossing'(R,C) ,
  plag : crossing'(R,C) , plar : crossing'(R,C) == false ,
  plbg : crossing'(R,C) , plbr : crossing'(R,C) == false > .
endtom)

(tomod MODELPC is protecting PELICAN .
  op initstate : -> Configuration .
  eq initstate = < system1 : System | crossing : false , req : false ,
  tlag : false , tlbg : false , tlar : true , tlbr : true ,
  plag : false , plbg : false , plar : true , plbr : true > .
endtom)

 (tomod MODEL-CHECK-PELICANCROSS is
  including TIMED-MODEL-CHECKER .
  protecting MODELPC .
  vars tlar' tlbr' plar' plbr' tlag' tlbg' plag' plbg' C : Bool .
  op system1 : -> Oid [ctor] .
  ops safecross safelight counter : Oid -> Prop [ctor] .

  eq {< system1 : System | tlar : tlar':Bool , tlag : tlag':Bool ,
   tlbr : tlbr':Bool , tlbg : tlbg':Bool , crossing : C:Bool ,
   plar : plar':Bool , plag : plag':Bool , plbr : plbr':Bool ,
   plbg : plbg':Bool >} |=
   safecross(system1) = ((C and plag' and plbg' and tlar' and tlbr') or
   (C == false)) .

  eq {< system1 : System | tlar : tlar':Bool , tlag : tlag':Bool ,
   tlbr : tlbr':Bool , tlbg : tlbg':Bool , crossing : C:Bool ,
   plar : plar':Bool , plag : plag':Bool , plbr : plbr':Bool ,
   plbg : plbg':Bool >} |= safelight(system1) = not((tlbg' or tlag') and
   (plag' or plbg')) .

  eq {< system1 : System | tlar : tlar':Bool , tlag : tlag':Bool ,
   tlbr : tlbr':Bool , tlbg : tlbg':Bool , crossing : C:Bool ,
   plar : plar':Bool , plag : plag':Bool , plbr : plbr':Bool ,
   plbg : plbg':Bool >} |= counter(system1) = not(tlbr' and plbr') .
  endtom)
```

Appendix C

Ladder Logic: First Attempt

This is our first attempt to model ladder logic in Maude. This work will be continue and worked out more precisely.

```
(tomod RUNG is protecting NAT-TIME-DOMAIN .
  sorts Rung Coil Contact .
  subsorts Coil Contact < Rung .

  ops lland llor : Rung Rung -> Rung .
  op coil : Nat -> Coil [ctor] .
  ops cont necont : Nat -> Contact [ctor] .
endtom)

(view Rungs from TRIV to RUNG is
  sort Elt to Rung .
endv)

(tomod LLINTERLOCKING is protecting NAT-TIME-DOMAIN .
  protecting MAP{Nat,Bool}  * (sort Map{Nat,Bool} to MapNB,
                             sort Entry{Nat,Bool} to EntryNB) .
  protecting RUNG .
  protecting LIST{Rungs} .
  protecting LIST{Bool} .

  class Inter | ladderlogic : List{Rungs}, inputs : MapNB, outputs : MapNB .

  op evalll : Rung MapNB MapNB -> MapNB .
  op evalrung : Rung MapNB MapNB -> Bool .
  op getcoil : Rung -> Nat .
  op myinter : -> Oid .

  var rlist : List{Rungs} .
  var O : Oid .
  vars R R1 R2 : Rung .
  vars M1 M2 : MapNB .
  vars E1 E2 : EntryNB .
  vars N1 N2 : Nat .

  crl [eval] :  {< O : Inter | ladderlogic : rlist, inputs : M1, outputs : M2 >}
   => {< O : Inter | ladderlogic : tail(rlist),  inputs : M1,
   outputs : evalll(head(rlist),M1,M2) >} in time 1 if size(rlist) > 0 .
  crl [eval2] :  {< O : Inter | ladderlogic : rlist, inputs : M1, outputs : M2 >}
   => {< O : Inter | ladderlogic : rlist,  inputs : M1, outputs : M2 >}
   in time 1 if size(rlist) == 0 .

  rl [compcoil] : evalll(R,M1,M2) => insert(getcoil(R) ,evalrung(R,M1,M2) , M2) .

  rl [comprung] : evalrung(coil(N1),M1,M2) => true  .
  rl [comprung2] : evalrung(lland(R,R1),M1,M2) => evalrung(R,M1,M2) and
   evalrung(R1,M1,M2) .
  rl [comprung3] : evalrung(llor(R,R1),M1,M2) => evalrung(R,M1,M2) or
   evalrung(R1,M1,M2) .
  rl [comprung4] : evalrung(cont(N1),M1,M2) => M1[N1] .
  rl [comprung5] : evalrung(necont(N1),M1,M2) => not M1[N1] .

  rl [getcoil] : getcoil(lland(R1,R2)) => getcoil(R2) .
  rl [getcoil2] : getcoil(coil(N1)) => N1 .
endtom)
```