Modelling of Railway Control Systems in the Maude System

Initial Document

Lam Chak Yan - 667271

September 2014

Abstract

This is where an abstract can appear. If you don't have an abstract, just throw out theses lines. The abstract should not exceed 10 lines :-)
3
4
5
6
7
8
9
10

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

Contents

1 Introduction **3**
- 1.1 Motivation . . . 3
- 1.2 Project Aims and Approach . . . 4

2 Maude **5**
- 2.1 Basic Information on Full Maude . . . 5
- 2.2 Real-Time Maude . . . 7
  - 2.2.1 Example: Modelling a clock . . . 8

3 Railway Control Systems **10**

4 Case Study: Pelican Crossing in Maude **11**

5 Development Model **14**

6 Planning **16**
- 6.1 Milestones . . . 16
- 6.2 Timeline . . . 16
- 6.3 Risk Analysis . . . 17

7 Conclusion **18**

Chapter 1

Introduction

1.1 Motivation

The number of computerized systems has been growing rapidly. The demand of correct systems is high. 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. [WASHINGTON] 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. However the deaths are mainly caused by system failure. Another example is a train crash in Seoul[au new]. 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. 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. We will evaluate the feasibility of model checking on Railway using Real-Time

Maude. Some work has been done in the past related to the verification and modelling. ((Timed CSP Simulator has been used to model level crossing [SWANSEA].)) Verification of solid state interlocking programs has been explored by Swansea Railway Verification Group []. Andrew Lawrence has verified railway interlockings by using a software SCADE suite [ANDREW]. Karim Kanso has done a formal verification on ladder logic. 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. (P.14 master)

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 can the Maude system 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.

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.

Chapter 2

Maude

2.1 Basic Information on Full Maude

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 [ICFEM'09]. Discrete event models such as traffic light has been simulated by 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 can be concluded as the module does not have any meaning. It is not completed yet. Therefore we need operations that can handle the sorts in order to archive what we want to compute. Each operator

should have a name and argument. Equations are used to specify how the operation works in different scenario. We need to declare variables for the declaration of equation. Below is the complete version of the stack example.

```
1  fmod STACK is
2     sorts Stack Entity NEStack  .
3     subsort NEStack < Stack .
4
5     op push : Stack Entity -> NEStack .
6     op pop : NEStack -> Stack .
7     op top : NEStack -> Entity .
8
9     vars E F : Entity .
10    var S : Stack .
11
12    eq pop(push(S,E)) = S .
13    eq top(push(S,E)) = E .
14 endfm
```

We can test the program by 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 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 also other types of declaration. Below is a list of some 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 the initial state. This involves term rewriting. The additional functionality provided by this extension is that we can discover whether a point is reachable from the initial state. It 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 of 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 show by modelling 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 .

   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)
```

Above is the whole specification of a simple clock model. 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 c1 as an object identifier, we assume c1 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 c1 after 13880 time units (seconds). The result is shown on the next page.

```
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. We can verified this by the formula $(3 \times 60 \times 60) + (51 \times 60) + 20 = 13880$. With the help of a calculator, we can say that the clock is correct. 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. This may be discussed later.

Chapter 3

Railway Control Systems

Chapter 4

Case Study: Pelican Crossing in Maude

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 Andrew Lawrence[ANDY]. In a pelican crossing control system, there are several lights for pedestrians and traffic. Pedestrians can press the button on the control panel when they want to cross the road. We would like to model how these buttons interact with the traffic lights. There are some rules for the pelican crossing control system. We can present these rules by a set of ladder logic formulae.

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

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

```
(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 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 p. *e.g. plar′ and plbg′*.

The two operations *press* and *notpress* are used to simulate the action of the pedestrian *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 except the red lights are set to false. We can simulate a set of actions by this command as an example:

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

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 instruction. 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 at the end. *req* is set to false.

Chapter 5

Development Model

We can apply a development model to the task. Waterfall model, spiral model and V-model are some common methods used in the software development process. We are not choosing waterfall model as it follows a too tight structure. Every stage is gone through step by step. *i.e. Requirements* $\rightarrow$ *Design* $\rightarrow$ *Implementation* $\rightarrow$ *Verification* $\rightarrow$ *Maintenance.* It works like a waterfall. The disadvantage is that changes are needed in some cases. 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 when we started the implementation process, or we may want to specify more features and functions on larger systems. Hence waterfall model is not a good choice. V-model is regarded as a extended waterfall model. The inflexibility is also a massive hindrance in the area of formal methods. Therefore V-model is not suitable.

Spiral model would be suitable for development in Maude for this project. The concept of spiral model is different from the other models. 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 at the beginning, 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. We should also have a review at the end of the cycle. This model is suitable for development in our project. The project is started with a simple module. More modules will be written after several cycles and these modules can communicate. More and more features can be built. We can keep improving the program until we are satisfied. Risk analysis also helps in maintaining the stability of the system. Hence larger object-oriented systems can be well-suited with spiral model. The spiral model can be represented by the following figure.

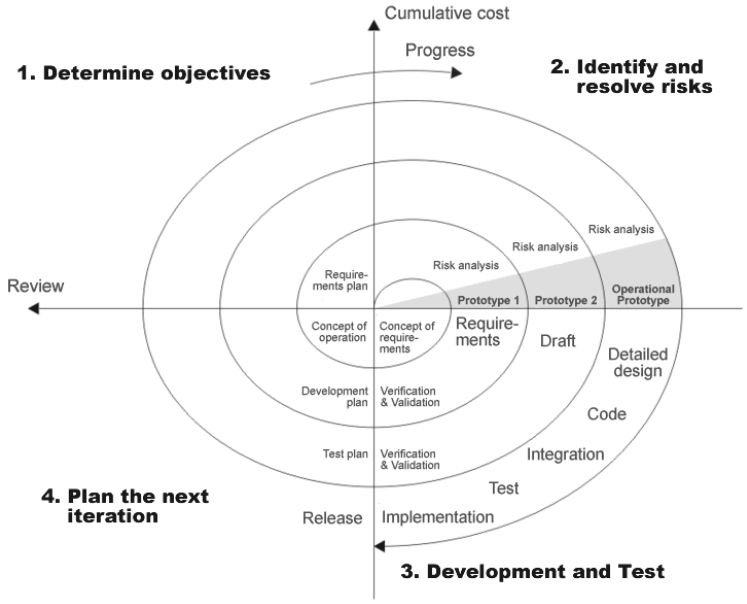

Figure 5.1: A spiral model (Boehm, 2000)

Chapter 6

Planning

6.1 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, resource on the internet.
2. Overview on Railway domain: Study thesis of Andrew Lawrence [ANDREW].
3. First own case study in Maude: pelican crossing
4. Writing initial document.
5. Specification of modelling ladder logic.
6. Study model checking in detail: apply to pelican crossing.
7. Formalisation of ladder logic 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.

Study on Hi-Maude may be introduced only if there is enough time.

6.2 Timeline

MS project-graph

6.3 Risk Analysis

Table

Chapter 7

Conclusion

http://www.koreabang.com/2014/stories/seoul-subway-crash-passengers-ignore-safety-announcement.html

http://www.washingtonpost.com/wp-dyn/content/article/2009/06/22/AR2009062203261.html

http://www.news.com.au/world/seoul-train-crash-blamed-on-system-failure/story-fndir2ev-1226904550085

http://www-verimag.imag.fr/ tripakis/papers/icfem09.pdf

http://en.wikipedia.org/wiki/Santiago$_d e_C ompostela_d erailmenthttp : //www.railjournal.com/index.php/eu error - only - cause - of - santiago - accident - says - report.html$