Summary

Lam Chak Yan

September 11, 2014

This is the summary of my summer project. I studied different techniques which can be used in formal specification.

1 Maude

The first thing I looked at is the Maude system. It is a language which supports equational and rewriting logic specification and programming. It can be used to model computer systems. At the beginning of the project, I have done some background reading on maude, based on the course notes of the course system specification, the maude primer and several other papers from the Internet. There are many examples which are helpful for understanding the semantics and syntax in maude. The vending machine example clearly explained the rewrite rules. It rewrites coins to items and contains rule for the changes return. There are several other interesting examples such as dining philosopher and the game"crossing the river"(shepherd, goat, cabbage, wolf).

1.1 Real-Time Maude

The Real-Time Maude is a tool that supports the formal specification for real-time and hybrid systems. It is suitable for specifying object-oriented real-time systems. It can simulate the concurrent behaviours of the system. On top of it, Model checking can be done with a given time limit.
I have read Andrew Lawrence's work related to Train Control System. This verification of hybrid systems can be fitted into Real-Time Maude programs. The movement of the trains can be modelled by introducing timed object-oriented module. A simple example for the model checking in real-time maude is a clock model.

2 Theorem Prover

I have experienced formal proofing by using COQ, a formal proof management system which uses formal language for machine-checked proofs. I have read the tutorials and done some interactive proofs myself. I was planned to use the coq

SMT solver, however there are some difficulties in the installation probably due to the compatibilty of the SMT solver and coq library.

3 Ladder Logic

Ladder logic is a programming language that represents a program by a graphical diagram. It can contain many rungs and each rung consists of a coil and contacts. I understood the relation between ladder logic and propositional logic while reading Andrew Lawrence's thesis. He suggested that we could implement ladder logic in Real-Time maude. Then we started developing the program. The program can now read a list of rungs and compile the output with given time limits.

4 Pelican Crossing

The MRes thesis of Andrew Lawrence contains a pelican crossing example which helps me to understand the basic concept of ladder logic and verification. I started to write a simple maude program modelling pelican crossing as the one in the thesis was written in SCADE. After that I kept editing the program and improving it. I used Real-Time Maude with built-in LTL model checker module to model the pelican crossing example. The program ensured several conditions in the pelican crossing.

4.1 Specification

```
(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)
```

This is the module that contains all the operations and the rewrite rules for modelling a pelican crossing.

The code will be explained step by step. The module is protecting NAT-TIME-DOMAIN as this predefined module have to be imported for the use of time related command in Real-Time Maude. We have a class called System:

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

The variables follow by the class describes the current state of the system. Their types are all booleans. I started with several variables tlar', tlbr', plar', plbr', tlag', tlbg', plag', plbg' , which represent different states of the traffic lights. e.g. when tlar' is set to true, that means the traffic light is showing red. Lights for pedestrians begin with p and t for traffic. The two operations crossing' and req' compute the new values of themselves while rewriting.

```
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 .
```

There are two rewriting rules extracted from the module above. These two rules are used for model checking. As it can be rewritten into a pressed state or a non-pressed state. These two rules will be executed while using the model checker. Each of the rules costs one time step. In the ladder logic formula for controlling the pelican crossing, we have "$req' \Leftrightarrow (pressed \wedge \neg req)$" . It is translated into an equation "*eq* $req'(A, B) = A$ *and not* B." under Maude. Hence A represents pressed where we have $req : req'(true, R)$ and $req : req'(false, R)$ for the two different rules. The boolean follows by "== false" above simply means the negation of the boolean value itself. Below are the formulae.

$crossing' \Leftrightarrow (req \wedge \neg crossing)$,
$req' \Leftrightarrow (pressed \wedge \neg req)$,
$tlag' \Leftrightarrow ((\neg crossing') \wedge (\neg pressed \vee req'))$,
$tlbg' \Leftrightarrow ((\neg crossing') \wedge (\neg pressed \vee req'))$,
$tlar' \Leftrightarrow crossing'$,
$tlbr' \Leftrightarrow crossing'$,
$plag' \Leftrightarrow crossing'$,
$plbg' \Leftrightarrow crossing'$,
$plar' \Leftrightarrow (\neg crossing')$,
$plbr' \Leftrightarrow (\neg crossing')$

4.2 Example: running the module

```
(tomod EXAMPLE is protecting PELICAN .
  ops initstate danger : -> 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 > .

  eq danger = < system1 : System |  tlag : true , tlbg : true , plag : true , plbg : tru
endtom)
```

This module contains the concrete configuration as an example. The two operations press and notpress in module PELICAN are for simulating the action of the pedestrian. 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:

```
(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 >

It simulates a press action follows by notpress. Crossing is allowed, request is changed to false. All the pedestrian lights are green and traffic lights are red.

4.3 Model checking

```
(tomod MODEL-CHECK-PELICANCROSS is
 including TIMED-MODEL-CHECKER .

 protecting PELICAN .
 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)
```

This is the module used for model checking, we have two safety properties safecross and safelight. For the safecross property, if crossing is allowed, all pedestrian lights should be in green and all traffic lights should be in red.For the safelight property, we ensure that the four lights must not be green at the same time.

safecross(system1) = ((C and plag' and plbg' and tlar' and tlbr') or (C == false)) .
safelight(system1) = not((tlbg' or tlag') and (plag' or plbg')) .

We can test the safety properties by these commands:

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

This is the result obtained:

rewrites: 23378 in 25ms cpu (25ms real) (935120 rewrites/second)
Model check {*initstate*} ⊨t[]safecross(system1)in MODEL-CHECK-PELICANCROSS in time ≤ 100 with mode deterministic time increase
Result Bool : true
rewrites: 23954 in 18ms cpu (17ms real) (1330777 rewrites/second)
Model check {*initstate*} ⊨t[]safelight(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 safety properties as both results are being true in 100 time steps. In order to show that the result is not always be true, we have a counter example which the property assumes that the traffic and pedestrian light b cannot be showing the same signal at any time.

counter(system1) = not(tlbr' and plbr') .

We can test it by using this command:

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

This is the result obtained:

rewrites: 6340 in 12ms cpu (12ms real) (495080 rewrites/second)
Model check{*initstate*} ⊨t[]counter(system1)in MODEL-CHECK-PELICANCROSS in time ≤ 3 with mode deterministic time increase
Result ModelCheckResult : counterexample({{<system1 : System | crossing : false,plag : false,plar : true,plbg : false,plbr : true,req : false,tlag : false,tlar : true,tlbg : false,tlbr : true >} in time 0,'notpressed}{{<system1 : System | crossing : false,plag : false,plar : true,plbg : false,plbr : true,req : false,tlag : true,tlar : false,tlbg : true,tlbr : false >} in time 1,'notpressed}{{<system1 : System | crossing : false,plag : false,plar : true,plbg : false, plbr : true,req : false,tlag : true,tlar : false,tlbg : true,tlbr : false >} in time 2,'notpressed},{{<system1 : System | crossing : false,plag : false,plar : true,plbg : false,plbr : true,req : false,tlag : true,tlar : false,tlbg : true,tlbr : false >} in time 3,'notpressed})

The result shows a counter example, it states that there is violation to the property within three time steps. It is due to the configuration of the initial state, in which all attributes are set to false. If we set the value of tlbr' and plbr' to be different in the configuration initstate, it will satisfy the counter property.

5 Ladderlogic in Maude

```
(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)
```

The module RUNG describes the basic component of a ladder logic. It consists of rungs, contacts and coils. We have to assign a number to each of coils and contacts in order to distinguish them and be mapped to a boolean value. e.g. $cont(1)/coil(2)$.

The view Rungs is used for obtaining single Rung to be an element of a list. A ladder logic normally consists of more than one Rung, hence we need a list of rungs.

```
(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)
```

This module contains all the operations and rewriting rules for a ladder logic. All ladder logic can be computed based on this module. The configuration of ladder logic contains a list of Rungs, inout and output. A Rung is composed by lland and/or llor, e.g. $lland(cont(1), coil(2))$ means this rung is composed by a contact and a coil. Natural number 1 will be mapped to a boolean value. It will be the input in the configuration. Natural number 2 will be mapped to a boolean value. It will be mapped to a boolean value depending on the input i.e. $cont(1)$. It will be the output in the configuration. e.g. $(2|->true/false)$

This program uses recursion and a base case to solve the Rung from the top to the bottom. The coil number will be identified in each Rung. Then lland and llor will be evaluated. The boolean value obtained in every rung will be mapped to its coil value as output. There can be a closed contact which the negation of the mapped value will be used.

```
(tomod example is protecting NAT-TIME-DOMAIN .
protecting LLINTERLOCKING .
ops ladder1 ladder2 ladder3 ladder4 ladder5 : -> Rung .
ops map1 map2 : -> MapNB .
ops conf1 conf2 : -> Configuration .
eq ladder1 = lland(cont(1),lland(necont(2),coil(8))) .
eq ladder2 = lland(cont(3),lland(cont(4),(lland(cont(5),coil(9))))) .
eq ladder3 = lland(cont(6),lland(cont(7),coil(10))) .
eq map1 = 1 |-> true, 2 |-> false , 3 |-> true, 4 |-> false, 5 |-> true,
 6 |-> false, 7 |-> true .
eq conf1 = < myinter : Inter | ladderlogic : ladder1 ladder2 ladder3 ,
 inputs : map1 , outputs : empty > .

eq ladder4 = lland(llor(cont(1),cont(2)),lland(cont(3),coil(9))) .
eq ladder5 = lland(lland(cont(4),llor(lland(cont(5),necont(6)),cont(7))),
lland(cont(8),coil(10))) .
eq map2 = 1 |-> true, 2 |-> false , 3 |-> false, 4 |-> true, 5 |-> true,
 6 |-> false, 7 |-> false, 8 |-> true .
eq conf2 = < myinter : Inter | ladderlogic : ladder4 ladder5 , inputs : map2 ,
 outputs : empty > .
endtom)
```

This module is an example for the ladder logic in maude. ladder1 to ladder5 are Rungs, they can be combined as a list of Rungs. The configuration conf1 and conf2 store all the information of the ladder logic.

We can test it by using this command:

```
(trew {conf1} in time <= 3 .)

(trew {conf2} in time <= 2 .)
```

This is the result obtained:

rewrites: 7056 in 13ms cpu (13ms real) (528143 rewrites/second)
Timed rewrite conf1 in example with mode deterministic time increase in time ≤ 7
Result ClockedSystem : {<myinter : Inter | inputs :(1 $\mapsto$ true, 2 $\mapsto$ false, 3 $\mapsto$ true, 4 $\mapsto$ false, 5 $\mapsto$ true, 6 $\mapsto$ false, 7 $\mapsto$ true),ladderlogic : nil,outputs :(8 $\mapsto$ true, 9 $\mapsto$ false, 10 $\mapsto$ false)>} in time 7
rewrites: 7028 in 9ms cpu (9ms real) (735300 rewrites/second)
Timed rewrite conf2 in example with mode deterministic time increase in time ≤ 5
Result ClockedSystem : {<myinter : Inter | inputs :(1 $\mapsto$ true, 2 $\mapsto$ false, 3 $\mapsto$ false, 4 $\mapsto$ true, 5 $\mapsto$ true, 6 $\mapsto$ false, 7 $\mapsto$ false, 8 $\mapsto$ true),ladderlogic : nil, outputs :(9 $\mapsto$ false, 10 $\mapsto$ true)>} in time 5

Below is the full ladder logic program for pelican crossing, natural number in the mappings are changed to the exact signal name for readability.

```
(tomod RUNG is protecting NAT-TIME-DOMAIN .

  sorts Rung Coil Contact .
  subsorts Coil Contact < Rung .

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

endtom)

(tomod PELICANCROSSING is protecting NAT-TIME-DOMAIN .

  sort signal .
  ops req crossing crossing' pressed req' tlag tlbg tlar tlbr plag plbg plar plbr : ->
  signal .

endtom)

(view PELI from TRIV to PELICANCROSSING is
    sort Elt to signal .
endv)

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

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

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

  op evalll : Rung MapNB MapNB -> MapNB .
  op evalrung : Rung MapNB MapNB -> Bool .
  op getcoil : Rung -> signal .
  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 : signal .

  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)

(tomod example is protecting NAT-TIME-DOMAIN .
protecting LLINTERLOCKING .
vars A B : Bool .
vars req2 crossing2 pressed2 : Bool .
ops crossing' req' : Bool Bool -> Bool .
eq crossing'(A,B) = A and not B .
eq req'(A,B) = A and not B .

ops ladder1 ladder2 ladder3 ladder4 ladder5 ladder6 ladder7
ladder8 ladder9 ladder10 : -> Rung .
ops map1 : Bool Bool Bool -> MapNB .
ops conf1 : Bool Bool Bool -> Configuration .
eq ladder1 = lland(cont(req),lland(necont(crossing),coil(crossing'))) .
eq ladder2 = lland(cont(pressed),lland(necont(req),coil(req'))) .
eq ladder3 = lland(lland(llor(necont(pressed),cont(req')),
 necont(crossing')),coil(tlag)) .
eq ladder4 = lland(lland(llor(necont(pressed),cont(req')),
 necont(crossing')),coil(tlbg)) .
```

```
eq ladder5 = lland(cont(crossing'),coil(tlar)) .
eq ladder6 = lland(cont(crossing'),coil(tlbr)) .
eq ladder7 = lland(cont(crossing'),coil(plag)) .
eq ladder8 = lland(cont(crossing'),coil(plbg)) .
eq ladder9 = lland(necont(crossing'),coil(plar)) .
eq ladder10 = lland(necont(crossing'),coil(plbr)) .

eq map1(pressed2,req2,crossing2) = pressed |-> pressed2 ,
 crossing' |-> evalrung(ladder1, (req |-> req2,crossing |-> crossing2),
 empty) , req' |-> evalrung(ladder2, (req |-> req2,pressed |-> pressed2),
 empty) , req |-> req2 , crossing |-> crossing2 .
eq conf1(pressed2,req2,crossing2) = < myinter : Inter | ladderlogic : ladder1
 ladder2 ladder3 ladder4 ladder5 ladder6  ladder7 ladder8 ladder9 ladder10,
 inputs : map1(pressed2,req2,crossing2) , outputs : empty > .

endtom)
```

We can test it by using this command:

```
(trew {conf1(true,false,true)} in time <= 10 .)
```

pressed: true, req: false, crossing: true .

This is the result obtained:

rewrites: 11308 in 19ms cpu (19ms real) (594001 rewrites/second)
Timed rewrite {conf1(true,false,true)} in example with mode deterministic time increase in time ≤ 10
Result ClockedSystem : {<myinter : Inter | inputs :(crossing $\mapsto$ true, crossing' $\mapsto$ false, pressed $\mapsto$ true, req $\mapsto$ false, req' $\mapsto$ true),ladderlogic : nil,outputs :(crossing' $\mapsto$ false, plag $\mapsto$ false, plar $\mapsto$ true, plbg $\mapsto$ false, plbr $\mapsto$ true, req' $\mapsto$ true, tlag $\mapsto$ true, tlar $\mapsto$ false, tlbg $\mapsto$ true, tlbr $\mapsto$ false)>} in time 10