Contents

1 Introduction

The TU Delft course IN4387 System Validation revolves around a single project which looks at the safety controller of a public road drawbridge. The goal of the assignment is to model a simple bridge control system by describing the system requirements and by showing the model using label transition systems. This assignment learns us how to properly use modelling tools and it emphasises the importance of proper and well formulated project documentation.

This document describes the different phases we have gone through in the project. In section 2 the global system requirements are described. Section 3 is dedicated to the interactions between the subsystems defined in section 2. In section 4 the requirements translated into μ-calculus formulae are described. The resulting architecture is given in section 5. Section 6 describes the bridge modeling process. In section 7 the bridge model is verified with the translated requirements. The final conclusions are presented in section 8.

2 Requirements

The first step in the design process is to capture the global requirements of the bridge system we aim to develop. The next sections describe the different system components we have distilled from the assignment description, operation sequences we have distinguished in the two main processes and the system and functional requirements.

2.1 System Components

- Two sets of traffic signs
  - Pre-signs P1 .. P4
  - Stop-signs S1 .. S4
- Four barriers B1 .. B4
- Motor (On, Off)
- Motor Brake
- Motor Sensor - Moving Up, Moving Down, Stopped
- Locking Pins L1 L2
- 8 Barrier Sensors: 4 Open, 4 Closed
- 2 Bridge Sensors: Bridge Up, Bridge Down

2.2 Sequences

Prior to establishing a set of system requirements, the default operating sequence is listed. Whereas this is the normal sequence of operation, any individual action may be called for by the operator or controller.

2.2.1 Bridge opening

1. **UI → Begin sequence open bridge**
2. Traffic Pre-signs on
3. Traffic Stop-signs on
4. Barriers down
5. Remove locking pins
6. Disengage motor brake
7. Motor on
8. Motor off
9. Engage motor brake

2.2.2 Bridge closing

1. **UI → Begin sequence close bridge**
2. Disengage motor brake
3. Motor on
4. Motor off
5. Engage motor brake
6. Insert locking pins
7. Barriers up
8. Traffic Stop-signs off
9. Traffic Pre-signs off

2.3 System Requirements

From the default sequences of actions stated above, a list of system requirements can be established. However, first several definitions will be set:

The state of 'ON' is defined by the action 'Turn on' not followed by the action 'Turn off', which may or may not be preceded by the action 'Turn off'. Likewise, the state of 'OFF' is described by the action 'Turn off' not followed by the action 'Turn on', which may or may not be preceded by the action 'Turn on'. Similarly, the state of 'REMOVED' is defined by the action of 'remove' not followed by the action 'insert'. We define 'INSERTED' to be equivalent to "not REMOVED".

We group the signs and barriers as follows:

- Pre-signs P1 and P2, Stop-signs S1 and S2, Entry Barrier B1 and Exit Barrier B2
- Pre-signs P3 and P4, Stop-signs S3 and S4, Entry Barrier B4 and Exit Barrier B3

We define 'corresponding' as "being in the same group".

1. Signs
   (a) The traffic stop-signs may only be turned on if the corresponding pre-signs are ON.
   (b) The traffic stop-signs may only be turned off if the corresponding barriers are OPEN.
   (c) The traffic pre-signs may only be turned off if the corresponding stop-signs are OFF.
2. Barriers
   (a) The entry barriers (B1, B4) may only be lowered if their corresponding traffic stop-signs are ON.
   (b) The exit barriers (B2, B3) may only be lowered if their corresponding entry barrier is CLOSED.
   (c) The barriers may only be opened if the bridge locking pins are INSERTED.
3. Pins
   (a) The locking pins may only be removed if all barriers are sensed to be CLOSED.
   (b) The locking pins may only be inserted if the motor brake is engaged and the bridge is CLOSED.
4. Motor
   (a) The motor brake may only be disengaged if the locking pins have been REMOVED.
   (b) The motor brake may only be engaged if the motor is STOPPED.
   (c) The motor may only be turned on if the motor brake is disengaged.

2.4 Functional Requirements

1. From the closed state (that is, with the bridge down and locked, all signs off and the barriers open), the bridge must be able to reach the open state (that is, with the bridge open and the motor brake engaged).

2. The closed state (as above) must be reachable from every reachable state.

3 Interactions

During the sequences described in section 2.2 interactions between the various subsystems take place. The following sections give an overview of those interactions.

3.1 User inputs

The following commands can be sent by the user to the safety controller. In this list, the parameter "a" is the target of the command, with functions having specific targets. In this list, the P1 to P4 are the pre-signs, S1 to S4 are the stop-signs, B1 to B4 are the barriers.

The parameter "p" is the corresponding action. For the setMotor function, we have added a parameter to indicate the direction of the motor.

user_setPreSign(a,p):	(a: P1 .. P4 — p: ON, OFF)	Turn the pre-signs on or off.
user_setStopSign(a,p):	(a: S1 .. S4 — p: ON, OFF)	Turn the stop-signs on or off.
user_setBarrier(a,p):	(a: B1 .. B4 — p: OPEN, CLOSED)	Set the barriers to open or close.
user_setLockpin(p):	(p: INSERTED, REMOVED)	Set the lock pins to be removed or inserted.
user_setMotorBrake(p):	(p: ON, OFF)	Turns the motor brake on and off.
user_setMotor(p):	(p: MOTOROFF, UP, DOWN)	Turns the motor on and off and specifies its direction.

3.2 Actuators and Signals

The following commands can be sent by the controller to the actuators and signals. In this list, the parameter "a" is the target of the command, with functions having specific targets. In this list, the P1 to P4 are the pre-signs, S1 to S4 are the stop-signs, B1 to B4 are the barriers.

The parameter "p" is the corresponding action. For the setMotor function, we have added a parameter to indicate the direction of the motor.

a_setPre(a,p):	(a : P1 .. P4 — p : ON, OFF)	Turns the pre-signs on or off.
a_setStop(a,p):	(a : S1 .. S4 — p : ON, OFF)	Turns the stop-signs on or off.
a_setBar(a,p):	(a : B1 .. B4 — p : OPEN, CLOSED)	Sets the barriers to open or close.
a_setBrake(p):	(p : ON, OFF)	Puts the motor brake on or off.
a_setLock(p):	(p : REMOVED, INSERTED)	Sets lock pins to be removed or inserted.
a_setMotor(p):	(p : MOTOROFF, UP, DOWN)	Turns the motor on or off and moves the bridge up or down.

3.3 Sensors

The following commands can be used by the safety controller to read out the current data from the sensors. The parameter "a" is the target of the command, and only the Barrier has specific sensors. If a command has no target, there is only one sensor of that type and will be used automatically. "r" means the Return value, the possible values that the sensors can return. For the UNDETECTED value, it is returned in the cases when the DETECTED requirement is not met.

sense_barrierOpen(r):	(a: B1 .. B4 — r : DETECTED, UNDETECTED)	Reads the barrier Open sensor.
sense_barrierClosed(r):	(a: B1 .. B4 — r : DETECTED, UNDETECTED)	Reads the barrier Closed sensor.
sense_Motor(r):	(r : MOTOROFF, UP, DOWN)	Reads the motor sensor, and returns UP if the bridge is opening, DOWN if the bridge is closing, and MOTOROFF if the bridge is not moving.
sense_bridgeOpen(r):	(r : DETECTED, UNDETECTED)	Reads the bridge Open sensor.
sense_bridgeClosed(r):	(r : DETECTED, UNDETECTED)	Reads the bridge Closed sensor.

4 Translated Requirements

4.1 Introduction

In this section the requirements listed in section 2 are translated by means of the interactions described in section 3.

In this section we define:
$B1, B4 \in$ Entry barriers
$B2, B3 \in$ Exit barriers

We also have the following functions: `matchPreSign(P)`, `matchStopSign(S)`, `matchBarriers(B)`, `matchExitBarrier(B)`. These functions state that the elements of the set are each other's corresponding element, as stated in section 2. Note that there is also a function `matchEntryBarrier(B)` which, although used in the model, is not used in any verification step.

4.2 Requirements translation into μ-calculus formulae

Various tests have been split into two because the first action shown in the formula matches the situation in the start state. As such, the first test shows the normal one, used anywhere but from the initial state. The second test concerns a path directly from the start state. Note that requirement 4.c shows four tests, two for each direction of movement.

1. Signs
   (a) The traffic stop-signs may only be turned on if the corresponding pre-signs are ON.
      i. $[true*]\forall S$:StopSignID . $[a\_setPre(matchPreSign(S), OFF)$
      $. (!a\_setPre(matchPreSign(S), ON)) * \ . a\_setStop(S, ON)]false$
      ii. $\forall S$:StopSignID . $[(!a\_setPre(matchPreSign(S), ON)) * \ . a\_setStop(S, ON)]false$
   (b) The traffic stop-signs may only be turned OFF if the corresponding barriers are OPEN:
      i. $[true*]\forall B$:BarrierID . $[a\_setBar(B, CLOSED) . (!a\_setBar(B, OPEN)) *$
      $. a\_setStop(matchStopSign(B), OFF)]false$
   (c) The traffic pre-signs may only be turned off if the corresponding stop-signs are OFF.
      i. $[true*]\forall S$:StopSignID . $[a\_setStop(S, ON) . (!a\_setStop(S, OFF)) *$
      $. a\_setPre(matchPreSign(S), OFF)]false$
2. Barriers
   (a) The entry barriers (B1, B4) may only be lowered if their corresponding traffic stop-signs are ON.
      i. $[true*]\forall S$:StopSignID . $[a\_setStop(S, OFF) . (!a\_setStop(S, ON)) *$
      $. a\_setBar(matchBarrier(S), CLOSED)]false$
      ii. $\forall S$:StopSignID . $[(!a\_setStop(S, ON)) * \ . a\_setBar(matchBarrier(S), CLOSED)]false$
   (b) The exit barriers (B2, B3) may only be lowered if their corresponding entry barrier is sensed to be DOWN.
      i. $[true*]\forall B$:BarrierID . $val(B == B1 || B == B4) => [a\_setBar(B, OPEN)$
      $. (!a\_setBar(B, CLOSED)) * \ . a_setBar(matchExitBarrier(B), CLOSED)]false$
      ii.
   (c) The barriers may only be opened if the bridge locking pins are INSERTED.
      i. $[true*]\forall B$:BarrierID . $[a\_setLock(REMOVED) . (!a\_setLock(INSERTED)) *$
      $. a\_setBar(B, OPEN)]false$
3. Pins
   (a) The locking pins may only be removed if all barriers are sensed to be DOWN.

i. $[true*]\forall B$:BarrierID . $[a\_setBar(B, OPEN)\,.\,(!a\_setBar(B, CLOSED))*\,.\,a\_setLock(REMOVED)]false$

ii. $\forall B$:BarrierID . $[(!a\_setBar(B, CLOSED))*\,.\,a\_setLock(REMOVED)]false$

(b) The locking pins may only be inserted if the motor brake is engaged and the bridge is sensed to be CLOSED.

i. $[true*\,.\,a\_setBrake(OFF)\,.\,(!a\_setBrake(ON))*\,.\,a\_setLock(INSERTED)]false$

ii. $[true*]\forall$MS: $motorState\,.\,val(MS == UP||MS == DOWN) => [a\_setMotor(MS)\,.\,(!sense\_bridgeClosed(DETECTED))*\,.\,a\_setLock(INSERTED)]false$

4. Motor

(a) The motor brake may only be disengaged if the locking pins have been REMOVED.

i. $[true*\,.\,a\_setLock(INSERTED)\,.\,(!a\_setLock(REMOVED))*\,.\,a\_setBrake(OFF)]false$

ii. $[(!a\_setLock(REMOVED))*\,.\,a\_setBrake(OFF)]false$

(b) The motor brake may only be engaged if the motor is sensed to be STOPPED.

i. $[true*\,.\,a\_setMotor(UP)\,.\,(!a\_setMotor(MOTOROFF))*\,.\,a\_setBrake(ON)]false$

ii. $[true*\,.\,a\_setMotor(DOWN)\,.\,(!a\_setMotor(MOTOROFF))*\,.\,a\_setBrake(ON)]false$

(c) The motor may only be turned on if the motor brake is disengaged.

i. $[true*\,.\,a\_setBrake(ON)\,.\,(!a\_setBrake(OFF))*\,.\,a\_setMotor(UP)]false$

ii. $[true*\,.\,a\_setBrake(ON)\,.\,(!a\_setBrake(OFF))*\,.\,a\_setMotor(DOWN)]false$

iii. $[(!a\_setBrake(OFF))*\,.\,a\_setMotor(UP)]false$

iv. $[(!a\_setBrake(OFF))*\,.\,a\_setMotor(DOWN)]false$

5. Functional Requirements

(a) From the start state, the bridge must be able to reach the open state (bridge open, motor brake engaged).

i. $\langle true*\,.\,sense\_bridgeOpen(DETECTED)\rangle\, true$

(b) From the closed state (bridge down and locked, signs off, barriers open), the bridge must be able to reach the open state (as above).

i. $[true*]\forall P$:PreSignID . $[a\_setPre(P, OFF)]\,\langle true*\,.\,sense\_bridgeOpen(DETECTED)\rangle\, true$

(c) From the open state (as above), the bridge must be able to reach the closed state.

i. $[true*\,.\,sense\_bridgeOpen(DETECTED)]\forall P$:PreSignID . $\langle true*\,.\,a\_setPre(P, OFF)\rangle\, true$

(d) The closed state (as above) must be reachable from every reachable state.

i. $[true*]\forall P$:PreSignID . $\langle true*\,.\,a\_setPre(P, OFF)\rangle\, true$

5 Architecture

5.1 Global System Architecture

In the user interface, the interactions between human and machine take place. Through this interface, the bridge operator gives his instructions to the system. The instructions are processed in the controller subsystem and are forwarded to the safety controller subsystem. The safety controller subsystem checks whether the required security conditions are met. Part of this process is the reading and evaluation of the sensors as described in section 2. When the proper safety conditions are met the relevant actuators will be activated. For simplicity reasons one of the features of the safety controller is not implemented. This concerns the submission of an error message to the controller when the correct safety conditions are NOT met. This error message should be displayed in the user interface and the bridge operator can then take the appropriate actions to make the desired control instructions possible.

Figure 1 depicts the logical structure of the bridge safety controller. As can be seen, the bridge has been modelled in seven component-sets being processed in parallel. Several of these component-sets have multiple components, namely the pre-signs, stop-signs, entry- and exit-barriers and the locking pins.

As depicted in the figure through the green arrows, the user interactions are allowed at any point. As such, the user can call for any change of state. However, the components themselves will decide whether this is safe. The user interactions have been previously explained textually in section 3.1. The names link directly to the component names and as such have been omitted from the figure for clarity

The blue arrows depict the communication between components. The communicated information as well as the possible states have been given. In some cases, not only information from other components is used but sensor data is also needed. This is shown by the red arrows. Note that some arrows show the same sensor being used.

Lastly, the purple arrows show the actual actions being communicated to the outside world. Once again, the names link directly to the component names and as such have been omitted from the figure for clarity.

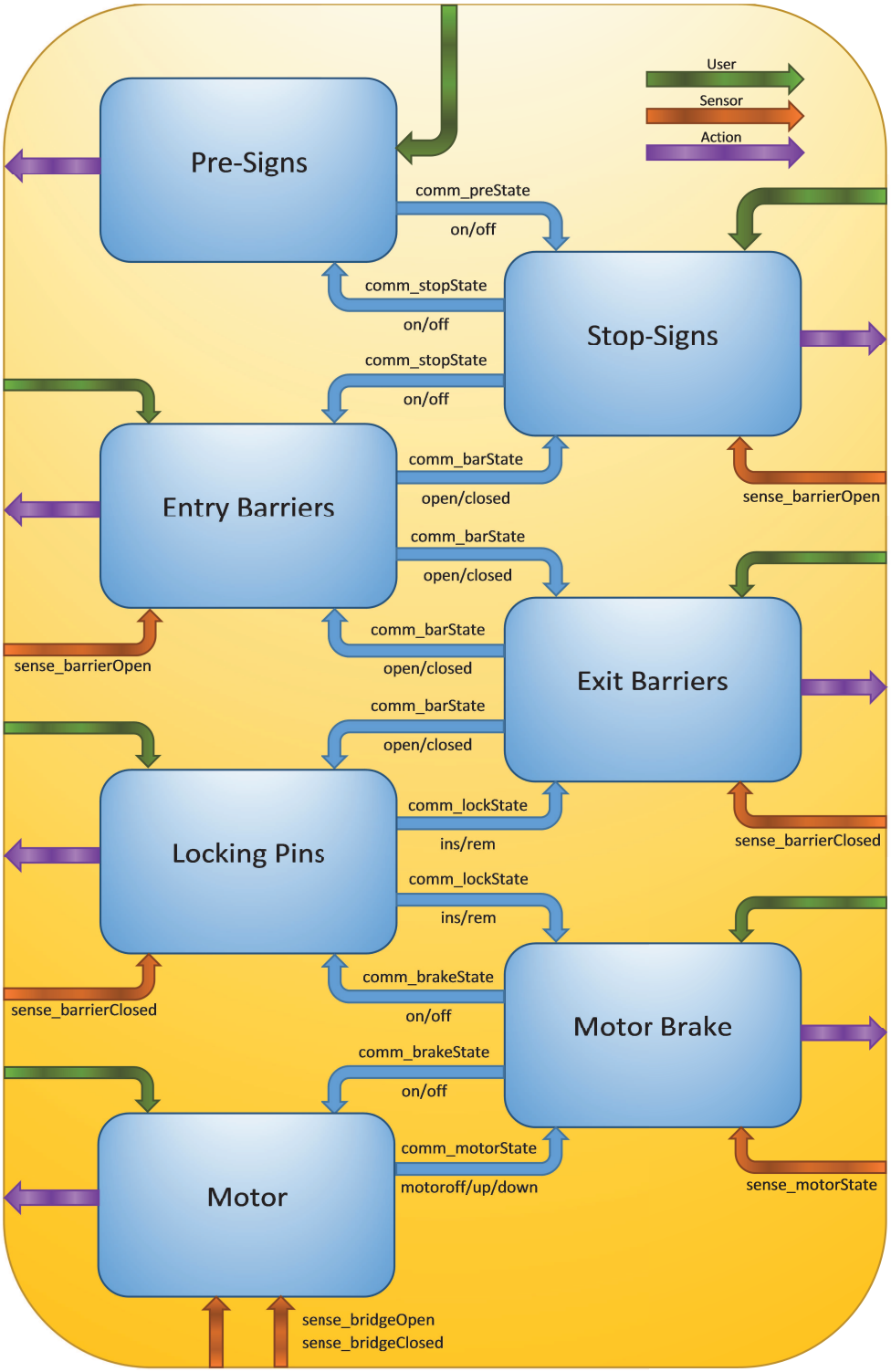


Figure 1: Visual representation of the architecture

6 Modeling the bridge

Although initially planned to only parallelize the lights, the barriers and the bridge as three main components, ultimately we have parallelized every individual component. This, however has caused the system to contain more than 50 million states and billions of transitions, which we were unable to compile on our machines. As such, during our analysis and verifications, we have decided to see the pre-sign, stop-sign and locking pin pairs as one. We are left with two sets of pre-signs (P12, P34), two sets of stop-signs (S12, S34) and one set of locking pins. Since their actions are consistently identical (but could possibly only slightly vary in real life), this should not cause any problem in validation. This actually modelled system has 603 648 states and 16 630 784 transitions.

The actual code is listed in the appendices with descriptive comments for clarification. The figure below shows the `ltsview` visualization of the system. One thing that can be noted before validation is that no deadlocks occur in the system.

Figure 2: Visual representation of the system states

7 Verification

Verification can be done through a visual check and through the use of μ-calculus formulae. For the μ-calculus method, we have written the translated requirements as presented in chapter 4 in mcf files. These can be found in Appendix B.

7.1 Tools

The following versions have been used:

- mCRL2 toolset 201202.0
- mCRL2 toolset 201409.0

In order to simplify tests, we have constructed a makefile (which can be found in the accompanying zip-archive) which allows for running all tests in series and choosing between whether to use lps2bool or lts2bool. Prior to running the tests we have timed which tools were fastest and checked whether they gave the same results for some sample formulae. The latter has shown to hold and we have seen that verification through lps is faster than through lts in our usecases (typically, lps2pbes followed by pbes2bool completes before lts2pbes finishes running).

As hoped and expected, all verifications are tested to hold true for our model.

7.2 Deadlocks

Lastly, a verification to check for deadlocks has been done in two ways. Firstly, a visual check using the `ltsview` application was done. As can be seen in figure 2, no parts of the model representation are marked red when selecting the 'mark deadlocks' option. Had there been deadlocks, the deadlocked state and its containing cluster should have been marked red.
Secondly, μ-calculus has been used to perform verification as shown above. One of the applied verifications was:

$$[true*]\,\langle true\rangle\, true$$

This is the general formula for a deadlock-free environment. Once again, this verification returned a true value. As such, the system has been shown to be deadlock free.

7.3 Verification checks

We tested our functional tests with a variant of our model where no communication was possible (The comm function had an empty set as arguments). As expected, the tests returned *false*.

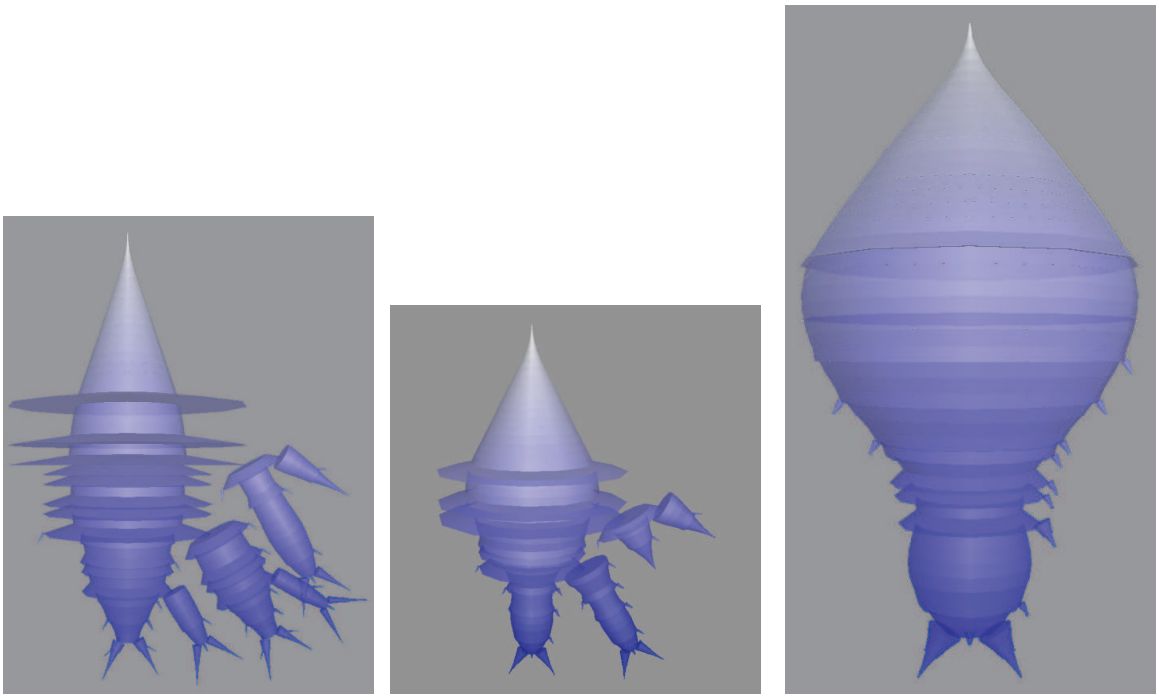
We then tested our safety tests with a variant of our model where a single state could execute any action and loop back onto itself. Most of our tests failed this. We then investigated the other tests. This turned out to be a flaw in our mini-model. Fixing this flaw (a missing action) made all our safety tests return *false*.

7.4 Visualization tools

We checked slices of our system for proper communication by using ltsgraph to visually inspect the statespace, which helped in finding a number of bugs as we were still writing our model.

When the model was complete, the statespace was far too big to be viewed with ltsgraph, so we used ltsview instead. The graph looked rather asymmetric, with some odd branches on the side. Using ltsview we were

able to discover that the requirements made by stop signs and the barriers subsystems were not symmetrical (the stop signs would only turn off if all barriers were up, while barriers could go down if the stop signs in the same lane were on). Once we understood this, the problem was easy to fix. The two tails of the final model correspond to the bridge moving up and down.



(a) Model first version (b) Model second version (c) Model final version

Figure 3: Initial models with extraneous branches and the final modal with said branches removed

8 Conclusions

Our assignment consisted of the modeling of a bridge, as described in the assignment description. The project consisted of five phases.

In the first phase we have determined the system components and captured the requirements (systems & functional). It took us some time to specify the envisioned system into concise requirement specifications without going into too much detail or overexpanding the system.

In the second phase we have defined the interactions between the systems defined in the first phase. Based on the system components, the requirements and the interactions, an architecture was defined. This turned out be a small challenge due to degrees of freedom related to the parallel processes to cluster.

In the third phase we have translated the requirements into μ-calculus formulae.
In the fourth phase the bridge was modeled with the mcrl2 tool. In this phase a lot of time was spent eliminating deadlocks. On of the results of the modeling exercise was a changed architecture.

In the fifth phase the model was verified by translating the μ-calculus formulae from phase three into mcf code. The outcome of this verification process can be grouped in three types:

1. Errors due to incorrect modeling
2. Errors due to incorrect μ-calculus formulae
3. Errors due to unimplemented (inter-)actions

The problems we encountered were mainly of the second type and we decided to translate certain requirements directly from the requirements from the first phase. Sometimes, splitting up the requirements in order to more accurately verify the requirements

Finally, we were able to correctly specify the requirements and modify the model in such way that all fifteen translated requirements were met.

While working on the assignment we have learned to develop a system based on a verifiable model. While working with the mclr2 tool we experienced the benefits of applying the design, verification and refinement cycle. It also require

A mcrl2 Model

```
% Complete Bridge LTS Version 1.
% This file models the bridge's components:
% P Presigns, S Stopsigns, B Barriers,
% L Locking Pins

sort    % Multi-Objects
        PreSignID               = struct P12 | P34;
        StopSignID              = struct S12 | S34;
        BarrierID               = struct B1 | B2 | B3 | B4;

        % States
        State           = struct ON | OFF;                      % Pre-signs, Stop-signs, Motor brake
        barState                = struct OPEN | CLOSED;         % Barriers
        motorState      = struct UP | DOWN | MOTOROFF;          % Motor direction
        lockState               = struct INSERTED | REMOVED;            % Locking pins
        sensorState     = struct DETECTED | UNDETECTED; % Sensors

        % Semaphore
        %Semaphore      = struct PINBRAKE | BRAKEMOTOR;
        %semState               = struct LOCK | UNLOCK;

        % Sensors


        % Current action state
        Changing                = struct CHANGING | STATIC;


map     % Matching Signs
        matchStopSign:  PreSignID -> StopSignID;
        matchPreSign:   StopSignID -> PreSignID;

        % Barrier -> StopSign
        matchStopSign:  BarrierID -> StopSignID;

        % StopSign -> Barrier
        matchBarrier1:  StopSignID -> BarrierID;
        matchBarrier2:  StopSignID -> BarrierID;

        % Exit <-> Entry
        matchExitBarrier:       BarrierID -> BarrierID;
        matchEntryBarrier:      BarrierID -> BarrierID;

        % Check Entry/Exit
        isEntry:                BarrierID -> Bool;

eqn     % PreSign <-> StopSign
        matchStopSign(P12) = S12;
        matchStopSign(P34) = S34;
        matchPreSign(S12) = P12;
        matchPreSign(S34) = P34;

        % Barrier -> StopSign
        matchStopSign(B1) = S12;
        matchStopSign(B2) = S12;
        matchStopSign(B3) = S34;
        matchStopSign(B4) = S34;

        % Exit <-> Entry
        matchExitBarrier(B1) = B2;
        matchExitBarrier(B4) = B3;
        matchEntryBarrier(B2) = B1;
        matchEntryBarrier(B3) = B4;

        % Check Entry/Exit
        isEntry(B1) = true;
        isEntry(B2) = false;
        isEntry(B3) = false;
        isEntry(B4) = true;
```

```
act
        % External commands
        user_setPreSign         : PreSignID     # State;
        user_setStopSign        : StopSignID    # State;
        user_setBarrier         : BarrierID     # barState;
        user_setLockpin         : lockState;
        user_setMotorBrake                      : State;
        user_setMotor                   : motorState;

        % Internal stuff
        % Actions
        a_setPre                : PreSignID     # State;
        a_setStop               : StopSignID    # State;
        a_setBar                : BarrierID     # barState;
        a_setLock               : lockState;
        a_setBrake                      : State;
        a_setMotor                      : motorState;

        % Communication
        s_preState,     r_preState,     comm_preState   : PreSignID     # State;
        s_stopState,    r_stopState,    comm_stopState  : StopSignID    # State;
        s_barState,     r_barState,     comm_barState   : BarrierID     # barState;
        s_lockState,    r_lockState,    comm_lockState  : lockState;
        s_brakeState,   r_brakeState,   comm_brakeState                          : State;
        s_motorState,   r_motorState,   comm_motorState                          : motorState;

        % Sensors
        sense_barrierOpen,      sense_barrierClosed     : BarrierID     #sensorState;
        sense_bridgeOpen,       sense_bridgeClosed      : sensorState;
        sense_motorState                        : motorState;

proc
        PreSign(p: PreSignID, state: State, c: Changing) =
         % Receive command, start changing
          (state == ON) -> user_setPreSign(p, OFF) . PreSign(c = CHANGING)
         % Receive confirmation, do stuff
        + ((state == ON) && (c == CHANGING)) -> r_stopState(matchStopSign(p), OFF) . a_setPre(p,
            OFF) . PreSign(state = OFF, c = STATIC)
         % receive conflicting state, stop changing
        + ((state == ON) && (c == CHANGING)) -> r_stopState(matchStopSign(p), ON) . PreSign(c =
            STATIC)
         % turning on pre-signs shall always be allowed, forever, for now into eternity
        + (state == OFF) -> user_setPreSign(p, ON) . a_setPre(p, ON) . PreSign(state = ON, c =
            STATIC)
         % no-ops, mostly
        + (state == OFF) -> user_setPreSign(p, OFF) . PreSign(c = STATIC)
        + (state == ON) -> user_setPreSign(p, ON) . PreSign(c = STATIC)
         % send state, stop changing
        + s_preState(p, state) . PreSign(c = STATIC);

        StopSign(s: StopSignID, state: State, c: Changing) =
         % Receive command, start changing
          (state == OFF) -> user_setStopSign(s, ON) . StopSign(c = CHANGING)
        + (state == ON) -> user_setStopSign(s, OFF) . StopSign(c = CHANGING)
         % Receive confirmation, do stuff
        + ((state == OFF) && (c == CHANGING)) -> r_preState(matchPreSign(s), ON) . a_setStop(s, ON)
            . StopSign(state=ON, c = STATIC)
        + ((state == ON) && (c == CHANGING)) -> r_barState(B1, OPEN) . r_barState(B4, OPEN)
. a_setStop(s, OFF) . StopSign(state = OFF, c = STATIC)
         % Receive conficting state, stop changing
        + ((state == OFF) && (c == CHANGING)) -> r_preState(matchPreSign(s), OFF) . StopSign(c =
            STATIC)
        + ((state == ON) && (c == CHANGING)) -> r_barState(B1, CLOSED) . StopSign(c = STATIC)
        + ((state == ON) && (c == CHANGING)) -> r_barState(B4, CLOSED) . StopSign(c = STATIC)
         % no-ops, mostly
        + (state == OFF) -> user_setStopSign(s, OFF) .StopSign(c = STATIC)
        + (state == ON) -> user_setStopSign(s, ON) . StopSign(c = STATIC)
         % send state, stop changing
        + s_stopState(s, state) . StopSign(c = STATIC);


        EntryBarrier(b: BarrierID, state: barState, c: Changing) =
         % Receive a command, start changing
```

```
          (state == OPEN) -> user_setBarrier(b, CLOSED) . EntryBarrier(c = CHANGING)
        + (state == CLOSED) -> user_setBarrier(b, OPEN) . EntryBarrier(c = CHANGING)
         % receive confirmation, do stuff
        % A barrier may only be closed if the corresponding stop-signs are turned on.
        + ((state == OPEN) && (c == CHANGING)) -> r_stopState(matchStopSign(b), ON) . a_setBar(b,
            CLOSED) . EntryBarrier(state = CLOSED, c = STATIC)
        % A barrier may only be opened if the exit barrier is open.
        + ((state == CLOSED) && (c == CHANGING)) -> r_barState(matchExitBarrier(b), OPEN) .
            sense_barrierOpen(matchExitBarrier(b), DETECTED) . a_setBar(b, OPEN) .
            EntryBarrier(state = OPEN, c = STATIC)
         % receive incompatible states, stop changing
        + ((state == OPEN) && (c == CHANGING)) -> r_stopState(matchStopSign(b), OFF) .
            EntryBarrier(c = STATIC)
        + ((state == CLOSED) && (c == CHANGING)) -> r_barState(matchExitBarrier(b), CLOSED) .
            EntryBarrier(c = STATIC)
        + ((state == CLOSED) && (c == CHANGING)) -> sense_barrierOpen(matchExitBarrier(b),
            UNDETECTED) . EntryBarrier(c = STATIC)
         % no-ops, mostly
        + (state == OPEN) -> user_setBarrier(b, OPEN) . EntryBarrier(c = STATIC)
        + (state == CLOSED) -> user_setBarrier(b, CLOSED) . EntryBarrier(c = STATIC)
         % send state, stop changing
        + s_barState(b, state) . EntryBarrier(c = STATIC);

        ExitBarrier(b: BarrierID, state: barState, c: Changing) =
         % Receive a command, start changing
          (state == OPEN) -> user_setBarrier(b, CLOSED) . ExitBarrier(c = CHANGING)
        + (state == CLOSED) -> user_setBarrier(b, OPEN) . ExitBarrier(c = CHANGING)
         % receive confirmation, do stuff
        % A barrier may only be closed if the entry barrier is closed
        + ((state == OPEN) && (c == CHANGING)) -> r_barState(matchEntryBarrier(b), CLOSED) .
            sense_barrierClosed(matchEntryBarrier(b), DETECTED) . a_setBar(b, CLOSED) .
            ExitBarrier(state = CLOSED, c = STATIC)
        % A barrier may only be opened if the locking pins are both inserted.
        + ((state == CLOSED) && (c == CHANGING)) -> r_lockState(INSERTED)
. a_setBar(b, OPEN) . ExitBarrier(state=OPEN, c = STATIC)
         % receive incompatible states, stop changing
        + ((state == OPEN) && (c == CHANGING)) -> r_barState(matchEntryBarrier(b), OPEN) .
            ExitBarrier(c = STATIC)
        + ((state == OPEN) && (c == CHANGING)) -> sense_barrierClosed(matchEntryBarrier(b),
            UNDETECTED) . ExitBarrier(c = STATIC)
        + ((state == CLOSED) && (c == CHANGING)) -> r_lockState(REMOVED) . ExitBarrier(c = STATIC)
         % no-ops, mostly
        + (state == OPEN) -> user_setBarrier(b, OPEN) . ExitBarrier(c = STATIC)
        + (state == CLOSED) -> user_setBarrier(b, CLOSED) . ExitBarrier(c = STATIC)
         % send state, stop changing
        + s_barState(b, state) . ExitBarrier(c = STATIC);


        Lockpin(state: lockState, c: Changing) =
         % receive command, set changing state
          (state == INSERTED) -> user_setLockpin(REMOVED) . Lockpin(c=CHANGING)
        + (state == REMOVED) -> user_setLockpin(INSERTED) . Lockpin(c=CHANGING)
         % receive confirmation, do stuff
% FIXME! possible deadlock here!!
        + ((state == REMOVED) && (c == CHANGING)) -> r_brakeState(ON) .
            sense_bridgeClosed(DETECTED) . a_setLock(INSERTED) . Lockpin(state = INSERTED, c =
            STATIC)
% FIXME combining thing here
        + ((state == INSERTED) && (c == CHANGING)) -> r_barState(B2, CLOSED) . r_barState(B3,
            CLOSED) . a_setLock(REMOVED) . Lockpin(state = REMOVED, c = STATIC)
         % receive incompatible state: TODO return error
        + ((state == REMOVED) && (c == CHANGING)) -> r_brakeState(OFF) . Lockpin(c = STATIC)
        + ((state == REMOVED) && (c == CHANGING)) -> sense_bridgeClosed(UNDETECTED) . Lockpin(c =
            STATIC)
        + ((state == INSERTED) && (c == CHANGING)) -> r_barState(B1, OPEN) . Lockpin(c = STATIC)
        + ((state == INSERTED) && (c == CHANGING)) -> r_barState(B2, OPEN) . Lockpin(c = STATIC)
        + ((state == INSERTED) && (c == CHANGING)) -> r_barState(B3, OPEN) . Lockpin(c = STATIC)
        + ((state == INSERTED) && (c == CHANGING)) -> r_barState(B4, OPEN) . Lockpin(c = STATIC)
         % no-ops, usually
        + (state == INSERTED) -> user_setLockpin(INSERTED) . Lockpin(c = STATIC)
        + (state == REMOVED) -> user_setLockpin(REMOVED) . Lockpin(c = STATIC)
         % send state, stop changing
        + s_lockState(state) . Lockpin(c = STATIC);
```

```
MotorBrake(state: State, c: Changing) =
 % receive command, set changing state
  (state == ON) -> user_setMotorBrake(OFF) . MotorBrake(c = CHANGING)
+ (state == OFF) -> user_setMotorBrake(ON) . MotorBrake(c = CHANGING)
 % receive confirmation, do stuff
+ ((state == ON) && (c == CHANGING)) -> r_lockState(REMOVED) . a_setBrake(OFF) .
    MotorBrake(state = OFF, c = STATIC)
+ ((state == OFF) && (c == CHANGING)) -> r_motorState(MOTOROFF) .
    sense_motorState(MOTOROFF) . a_setBrake(ON) . MotorBrake(state = ON, c = STATIC)
 % recieve incompatible state: TODO return error
+ ((state == ON) && (c == CHANGING)) -> r_lockState(INSERTED) . MotorBrake(c = STATIC)
+ ((state == OFF) && (c == CHANGING)) -> r_motorState(UP) . MotorBrake(c = STATIC)
+ ((state == OFF) && (c == CHANGING)) -> r_motorState(DOWN) . MotorBrake(c = STATIC)
 % no-ops, usually
+ (state == ON) -> user_setMotorBrake(ON) . MotorBrake(c = STATIC)
+ (state == OFF) -> user_setMotorBrake(OFF) . MotorBrake(c = STATIC)
 % send state, stop changing
+ s_brakeState(state) . MotorBrake(c = STATIC);


Motor(state: motorState, c: Changing, dir: motorState) =
 % receive command to start moving, set changing state and direction
  (state == MOTOROFF) -> user_setMotor(UP) . Motor(c = CHANGING, dir = UP)
+ (state == MOTOROFF) -> user_setMotor(DOWN) . Motor(c = CHANGING, dir = DOWN)
 % receive confirmation, do stuff
+ (c == CHANGING) -> r_brakeState(OFF) . a_setMotor(dir) . Motor(state = dir, c = STATIC)
 % receive incompatible state: TODO return error
+ (c == CHANGING) -> r_brakeState(ON) . Motor(c = STATIC)
 % motor off
+ (state != MOTOROFF) -> user_setMotor(MOTOROFF) . a_setMotor(MOTOROFF) . Motor(state =
    MOTOROFF, c = STATIC)
 % simple direction changes
+ (state == UP) -> user_setMotor(DOWN) . a_setMotor(DOWN) . Motor(state = DOWN, c = STATIC)
+ (state == DOWN) -> user_setMotor(UP) . a_setMotor(UP) . Motor(state = UP, c = STATIC)
 % no-ops, usually
+ (state == MOTOROFF) -> user_setMotor(MOTOROFF) . Motor(c = STATIC)
+ (state == UP) -> user_setMotor(UP) . Motor(c = STATIC)
+ (state == DOWN) -> user_setMotor(DOWN) . Motor(c = STATIC)
 % send state, stop changing
+ s_motorState(state) . Motor(c = STATIC);


init
        hide({comm_preState, comm_stopState},

                allow({ user_setPreSign, user_setStopSign, user_setBarrier, user_setLockpin,
                    user_setMotorBrake, user_setMotor,
                        a_setPre, a_setStop, a_setBar, a_setLock, a_setBrake, a_setMotor,
                        comm_preState, comm_stopState, comm_barState, comm_lockState,
                            comm_brakeState, comm_motorState,
                        sense_barrierClosed, sense_barrierOpen, sense_bridgeOpen,
                            sense_bridgeClosed, sense_motorState},

                        comm({  r_preState|s_preState -> comm_preState,
                                r_stopState|s_stopState -> comm_stopState,
                                r_barState|s_barState -> comm_barState,
                                r_lockState|s_lockState -> comm_lockState,
                                r_brakeState|s_brakeState -> comm_brakeState,
                                r_motorState|s_motorState -> comm_motorState},

                                   PreSign(P12, OFF, STATIC) || PreSign(P34, OFF, STATIC)
                                || StopSign(S12, OFF, STATIC) || StopSign(S34, OFF, STATIC)
                                || EntryBarrier(B1, OPEN, STATIC) || EntryBarrier(B4, OPEN, STATIC)
                                || ExitBarrier(B2, OPEN, STATIC) || ExitBarrier(B3, OPEN, STATIC)
                                || Lockpin(INSERTED, STATIC)
                                || MotorBrake(ON, STATIC)
                                || Motor(MOTOROFF, STATIC, MOTOROFF)
                        )
                )
        );
```

B MCF Verification formulae

Signs
1.a.i

```
[true*] forall S: StopSignID . [a_setPre(matchPreSign(S), OFF) . (!a_setPre(matchPreSign(S), ON))*
    . a_setStop(S, ON)]false
```

1.a.ii

```
forall S: StopSignID . [(!a_setPre(matchPreSign(S), ON))* . a_setStop(S, ON)]false
```

1.b

```
[true*] forall B: BarrierID . [a_setBar(B, CLOSED) . (!a_setBar(B, OPEN))* .
    a_setStop(matchStopSign(B), OFF)]false
```

1.c

```
[true*] forall S: StopSignID . [a_setStop(S, ON) . (!a_setStop(S, OFF))* .
    a_setPre(matchPreSign(S), OFF)]false
```

Barriers
2.a.i

```
[true*] forall S: StopSignID . [a_setStop(S, OFF) . (!a_setStop(S, ON))* .
    a_setBar(matchBarrier(S), CLOSED)]false
```

2.a.ii

```
forall S: StopSignID . [(!a_setStop(S, ON))* . a_setBar(matchBarrier(S), CLOSED)]false
```

2.b

```
[true*] forall B: BarrierID . val(B == B1 || B == B4) => [a_setBar(B, OPEN) . (!a_setBar(B,
    CLOSED))* . a_setBar(matchExitBarrier(B), CLOSED)]false
```

2.c

```
[true*] forall B: BarrierID . [a_setLock(REMOVED) . (!a_setLock(INSERTED))* . a_setBar(B,
    OPEN)]false
```

Locking Pins
3.a.i

```
[true*] forall B: BarrierID . [a_setBar(B, OPEN) . (!a_setBar(B, CLOSED))* .
    a_setLock(REMOVED)]false
```

3.a.ii

```
forall B: BarrierID . [(!a_setBar(B, CLOSED))* . a_setLock(REMOVED)]false
```

3.b.i

```
[true* . a_setBrake(OFF) . (!a_setBrake(ON))* . a_setLock(INSERTED)]false
```

3.b.ii

```
[true*] forall MS: motorState . val(MS == UP || MS == DOWN) => [a_setMotor(MS) .
    (!sense_bridgeClosed(DETECTED))* . a_setLock(INSERTED)]false
```

Motor

4.a.i

```
[true* . a_setLock(INSERTED) . (!a_setLock(REMOVED))* . a_setBrake(OFF)]false
```

4.a.ii

```
[(!a_setLock(REMOVED))* . a_setBrake(OFF)]false
```

4.b.i

```
[true* . a_setMotor(UP) . (!a_setMotor(MOTOROFF))* . a_setBrake(ON)]false
```

4.b.ii

```
[true* . a_setMotor(DOWN) . (!a_setMotor(MOTOROFF))* . a_setBrake(ON)]false
```

4.c.i

```
[true* . a_setBrake(ON) . (!a_setBrake(OFF))* . a_setMotor(UP)]false
```

4.c.ii

```
[true* . a_setBrake(ON) . (!a_setBrake(OFF))* . a_setMotor(DOWN)]false
```

4.c.iii

```
[(!a_setBrake(OFF))* . a_setMotor(UP)]false
```

4.c.iv

```
[(!a_setBrake(OFF))* . a_setMotor(DOWN)]false
```

Functional Requirements

5.a

```
<true*.sense_bridgeOpen(DETECTED)>true
```

5.b

```
[true*] forall P: PreSignID. [a_setPre(P, OFF)]<true*.sense_bridgeOpen(DETECTED)>true
```

5.c

```
[true*.sense_bridgeOpen(DETECTED)] forall P: PreSignID. <true*.a_setPre(P,OFF)>true
```

5.d

```
[true*] forall P: PreSignID. <true*.a_setPre(P,OFF)>true
```