System Validation 2014
Homework Part 1 - Model Checking

Thom Ritterfeld - s1509101
Tom Vocke - s144002

PART 1 :: SMV AND TEMPORAL LOGICS

Part 1.1 :: Write a set of properties that validate the correctness of the various instances of the model (20pt).

See the file "properties.smv" for our solutions. The added use cases are :

- the interlock should be used to let trains pass (if there are never two trains in the interlock, it has no use)
- a train should not move if both signs are red
- trains should always move at some point in time (trains make progress)

Part 1.2 :: Take a counter-example against and LTL liveness property and explain the order of events in plain english (10pt)

Trains are waiting in t1a and t3b, and can never leave since the lock does not allow passing (stays in curved state while it should go to straight to let the train in 1A pass). This violates the liveness property stating that trains should always keep moving.

See the following trace:

```
-- specification  G (T1A_occupied ->  F !T1A_occupied)  is false
-- as demonstrated by the following execution sequence

--Run starts here, with following initial state:
--none of the tracks are occupied
--the interlocks are not locked into any position
--all the interlock signal show a red light (interlocks are not ready so this is correct)
--the signals to the outside world show green
--no commands have been given to the interlocks

-> State: 1.1 <-
  T1A_occupied = FALSE
  T2A_occupied = FALSE
  T3A_occupied = FALSE
  T4A_occupied = FALSE
  T1B_occupied = FALSE
  T2B_occupied = FALSE
  T3B_occupied = FALSE
  T4B_occupied = FALSE
  S1A_red = FALSE
  S1B_red = FALSE
  S1A_green = TRUE
  S1B_green = TRUE

  …………….. continues on next page
```

```
  P1A_locked_straight = FALSE
  P2A_locked_straight = FALSE
  P1B_locked_straight = FALSE
  P2B_locked_straight = FALSE
  P1A_locked_curved = FALSE
  P2A_locked_curved = FALSE
  P1B_locked_curved = FALSE
  P2B_locked_curved = FALSE
  S2A_red = TRUE
  S3A_red = TRUE
  S4A_red = TRUE
  S2B_red = TRUE
  S3B_red = TRUE
  S4B_red = TRUE
  S2A_green = FALSE
  S3A_green = FALSE
  S4A_green = FALSE
  S2B_green = FALSE
  S3B_green = FALSE
  S4B_green = FALSE
  P1A_goal_straight = FALSE
  P2A_goal_straight = FALSE
  P1B_goal_straight = FALSE
  P2B_goal_straight = FALSE
  P1A_goal_curved = FALSE
  P2A_goal_curved = FALSE
  P1B_goal_curved = FALSE
  P2B_goal_curved = FALSE

--train moves into T0B, so s1B signal turns red
--control gives commands to locks, to go into the straight position

-> State: 1.2 <-
  S1B_red = TRUE
  S1B_green = FALSE
  P1A_goal_straight = TRUE
  P2A_goal_straight = TRUE
  P1B_goal_straight = TRUE
  P2B_goal_straight = TRUE

--train moves from t0b to t1b
--signal s1b turns green since destination track in empty
--P2A and P1B reach straight lock position
--goals are reset for P2A and P1b
--s2b is set to green since P1B has reach straight lock position

-> State: 1.3 <-
  T1B_occupied = TRUE
  S1B_red = FALSE
  S1B_green = TRUE
  P2A_locked_straight = TRUE
  P1B_locked_straight = TRUE
  S2B_red = FALSE
  S2B_green = TRUE
  P2A_goal_straight = FALSE
  P1B_goal_straight = FALSE
```

…………….. continues on next page

--train moves straight from t1b to t2b
--s2b turns red since destination track t2b is occupied

-> State: 1.4 <-
T1B_occupied = FALSE
T2B_occupied = TRUE
S2B_red = TRUE
S2B_green = FALSE

--train moves from t2b to t3b
--p1a and p2b reach locked position
--s1a turns red, meaning train is incoming from outside world t0a
--s2a and s3b turn green since locks are set and destination tracks are empty
--straight goals for p1a and p2b are cleared
--goal for p1a and p2b are set to curved

-> State: 1.5 <-
T2B_occupied = FALSE
T3B_occupied = TRUE
S1A_red = TRUE
S1A_green = FALSE
P1A_locked_straight = TRUE
P2B_locked_straight = TRUE
S2A_red = FALSE
S3B_red = FALSE
S2A_green = TRUE
S3B_green = TRUE
P1A_goal_straight = FALSE
P2B_goal_straight = FALSE
P1A_goal_curved = TRUE
P2B_goal_curved = TRUE

--train moves from t0a to t1a
--s1a signal can turn green again
--P1a p2b lock has started to moved to curved position
--s2a turns red since lock is in moving state

-> State: 1.6 <-
T1A_occupied = TRUE
S1A_red = FALSE
S1A_green = TRUE
P1A_locked_straight = FALSE
P2B_locked_straight = FALSE
S2A_red = TRUE
S2A_green = FALSE

--nothing happens
-> State: 1.7 <-

--P1a reaches locked state, does not change any signals
-> State: 1.8 <-
P1A_locked_curved = TRUE

--p2b reach lock state. Trains are waiting in t1a and t3b, and CAN NEVER LEAVE!! since lock does not allow passing
-- Loop starts here
-> State: 1.9 <-
P2B_locked_curved = TRUE
P1A_goal_curved = FALSE
P2B_goal_curved = FALSE

Part 1.3 :: Change the points module in such a way that the move can take an arbitrary amount of time. (10pts)

This is done in delay-env.smv The result is shown below:

```
MODULE Points(goal_straight,goal_curved)
 VAR
        status : {straight,moving,curved};
 ASSIGN
        init(status):=moving;
        next(status):=case
                        goal_straight & status=curved    : moving;
                         goal_straight & status=moving   : {moving, straight};
                         goal_curved & status=straight   : moving;
                         goal_curved & status=moving     : {moving, curved};
                         TRUE                             : status;
                       esac;
 FAIRNESS
        !(status = moving)
```

Two cases are added, that add non determinism when in the moving state. This means, once moving, the track can either keep on moving (any number of transitions), or reach it destination. A fairness constraint is added to ensure that once the point has started moving, it will also stop moving at some point. This excludes the case where no trains ever enter the system, which is fine.

Part 1.4 :: Describe the trace leading to a violation of a signal safety property after the delay in introduced. (10pts)

The trace is left out since the error only occurred after 53 transitions. It can be found in *nusmv_trace_opg4.txt.* A description of the final state the system is in at this point, can also be found in this file.

The problem that is shown in this trace is that one of the lights (in this case S4B) turns green when this is not allowed. This is a result of a train arriving in T3B, which triggers the curved command for points P1A and P2B. For a system without delay, both the commands will be executed in exactly the same time. This means that if one point has reached its destination, so has the other.

The Simple-Env environment assumes that both points are indeed locked at the same time, and only checks the state of one of the points. This results in an error once the delay is introduced. This can be fixed by checking both lights as follows (in simple-int.smv) :

```
next(S4A_red) := case
   P1B_locked_curved & P2A_locked_curved & !T4A_occupied & ! T2B_occupied & ! T1B_occupied : FALSE;
   TRUE : TRUE;
esac;
next(S4B_red) := case
   P1A_locked_curved & P2B_locked_curved & !T4B_occupied & ! T2A_occupied & ! T1A_occupied : FALSE;
   TRUE : TRUE;
esac;
```

Part 1.5 :: Extend the model until it handles the case of two trains correctly. (20pts)

We based our result on the simple-int.smv and simple-env.smv files. We started off by adding the delay as in Part 1.4 to make the model more realistic. Also we added a second train in the simple-env.smv files as shown below:

```
--this definition was updated to support two trains

 DEFINE
        T1A_occupied := train_A.location = T1A | train_B.location = T1A;
        T2A_occupied := train_A.location = T2A | train_B.location = T2A;
        T3A_occupied := train_A.location = T3A | train_B.location = T3A;
        T4A_occupied := train_A.location = T4A | train_B.location = T4A;
        T1B_occupied := train_A.location = T1B | train_B.location = T1B;
        T2B_occupied := train_A.location = T2B | train_B.location = T2B;
        T3B_occupied := train_A.location = T3B | train_B.location = T3B;
        T4B_occupied := train_A.location = T4B | train_B.location = T4B;

--also an addition train module was loaded as variable in the environment
 VAR
        --see full-env.smv for full contents
        train_A : Train(
                --see full-env.smv for full contents
        );
        train_B : Train(
                --see full-env.smv for full contents
        );
```

In case of two trains it is not enough to check both points for their lock states. The following cases that were not possible before now have to be accounted for:

1. ==
   Train is waiting on T3B
   Train enters at T1A
   ==
   The problem with this case is that the first track is that if the curved lock has priority the trains will always be waiting on each other. Hence the straight lock should have priority over the curved lock commands.

2. ==
   Train 1 is waiting at T3B
   Train 2 is waiting at T1A
   Lock gives straight priority (as needed by first case)
   ==
   Due to delay, train 2 can leave the system and be back at T1A before the lock has moved back to curved state. Straight gets priority again, and the train in T3B is stuck forever. This can be solved by changing the locks simultaneously, as if they are one. This would mean that train 2 can only leave the system when the locks reach curved state, and then train 1 can leave before train 2 arrives again.

3. ==
   Train 1 enters, and gets to first lock
   lock is set to straight
   second train enters on the same side
   first train passes lock
   second train follows after first train, and occupies first track
   ==

   The problem with this case is that the first track is always occupied, so the liveness property ”LTLSPEC G (T1A_occupied -> F !T1A_occupied)” no longer holds.
   Also, since the previous case required the locks to be synchronized, the systems get to a deadlock where straight gets priority, but the train 2 cannot move because train 1 is waiting on the lock to be curved (which will never happen because of synchronization).

We solved all these cases by doing the following :

1. Synchronize the lock commands so that system moves as one (either all locks curved, or all locks straight)
2. Update the signals S4A and S4B so that green is only given when both locks are locked in curved state (need due to random time delay and case 2.
3. Give straight priority over curved commands, but only when a system is not already moving (a.k.a always finish a given command, but initiale straight before curved)
4. Update the liveness property for track 1A and track 1B so that it is allowed to either become free sometime in the future, or if not, the opposite tracks should never be used (this is due to case 3)
5. Update all signals to check on both sides of the lock to avoid invalid greens (see Part 1.4)

Note that there is also a case where a train leaves the system just when another enters the system. In this case the trains collide in track T1B or T1A. This cannot be avoided since there are no tools to check whether a train is coming (non-deterministic).

The final result can be found in full-env.smv, full-int.smv and properties.smv

PART 2 :: SOFTWARE MODEL CHECKING

Part 2.1 :: Perform bounds-check on init_fs. (10pts)

The following command does the bounds-checking in satabs and shows us the claims:

satabs --bounds-check main_bounds.c wrapped_fs.c

You will get the following error:

```
Violated property:
  file wrapped_fs.c line 20 function init_fs
  array `dir_status' upper bound
  i < 3
```

We recognized that the dir_status array is initialized within the file status loop. But the file array is larger than the dir array; the array size of dir_status is 3 (MAX_DIRS - 3) and the loop goes from index 0 till 5 (MAX_FILES - 6). So the dir_status[i] within the loop will be out of bounds.

To fix this error we added another for loop to set the directory status in init_fs().

```
for(i=0; i<MAX_DIRS;i++){
        dir_status[i]=0;
  }
```

We still got the following error:

```
Violated property:
  file wrapped_fs_bounds.c line 29 function open_dir
  array `dir_status' upper bound
  parent < 3
```

This is because our main checks nondeterministically directories, but the MAX_DIRS are only 3. So if when we are trying to open directory 4 we also have an out of bounds error. To fix this issue we've moved MAX_DIRS to wrapped_fs.h and added a line to main_bounds.c.

```
__CPROVER_assume(0 <= n && n < MAX_DIRS);
```

Now stabs only checks the directories within the bounds between 0 and MAX_DIRS and the verification succeeds.

Part 2.2 :: Perform bounds-check on init_fs. (10pts)

Lets start by saying that we lost a lot of time here due to some properties of satabs that we were simply unaware off. In particular, the following case causes a FAIL:

```
file_status[1] = ENTRY_USED;
read_file(1,0,10,(void *)buff);
….

int read_file(int fd,long offset,int len,void*buf){
        assert ( file_status[1] == ENTRY_USED);
        return 0;
}
```

whereas this one does not

```
read_file(1,0,10,(void *)buff);
….

int read_file(int fd,long offset,int len,void*buf){
        file_status[1] = ENTRY_USED;
        assert ( file_status[1] == ENTRY_USED);
        return 0;
}
```

We still don't understand why exactly this is happening, but it seems that it has something to do with the fact that file_status is set in a different file than it is read in the above cases. For this reason we moved the main function to the wrapper_fs.c file and did everything from there.

Also simple checks can already take roughly 15 minutes or more. It is impossible to verify your solutions within an acceptable time frame.

To get around this a little bit, we simplified mainly the close_dir() function to only loop for close_file() when file-dependency is something we want to test. Also, we test the close_file() function separately, and replace calls to this with a simplified version, and an assertion for valid parameters.

Last but not least, we performed all checks separately to reduce the state-space, so tests can be enabled separately. Each test is described in the main() function in wrapped_fs_nested.c. The fixes for each test-case can be enabled or disabled using the appropriate definitions provided at the top of this file.

SATABS also had some difficulty evaluating (file_status[xx] & ENTRY_USED == XX) statements correctly, or uses a different associativity than C. We replaced all similar calls with a more verbose version ((file_status[xx] & ENTRY_USED) == XX)

Bugs that were found :

- Dir status not set anywhere on create, open or close dir
- Parents of directories not registered, so children cannot be checked upon closing folder. This can lead to a child folder being open after its parent is closed
- Any statement containing (xxx & xxx == xx) is evaluated incorrectly and was changed to ((xxx & xxx) == xx). This should not matter, but it does for the satabs checking.
- close_dir allowed closing of the root directory

Command Line used to test was simply :

>> satabs wrapped_fs.c

Several options given in “satabs -h” were explored to reduce test-time, but no suitable ones were found. A bit more guidance / information in the assignment might be useful for future students.

Traces for dir structure proved to be impossible to create. Satabs was set to run with a max of 10000 iterations and after running for 8 hours, it had not yet found an error, even though there cleary is one, since parents of directories are not administered.

We fixed the bug, but cannot verify whether it works. Some advice on how to solve this would be appreciated.