System Validation 2014
Homework Part 1 - Model Checking

Stefan Blom, Marieke Huisman and Wojciech Mostowski

Available: Tuesday Sept. 23, 2014
Deadline: 23:59 CET, Tuesday Oct. 7, 2014

- The assignment can be made in pairs.
- Write the names of the team members and their student id's at the top of every file and/or document.
- This exercise is worth 100 points.
- All solutions should be uploaded via BlackBoard.

This assignment has two parts:

- Modelling and verifying a passing area in a single track rail road: using temporal logic specifications to analyse some given models, defining appropriate models, and validating them by temporal logic specifications (using NuSMV).
- Annotating and verifying a file system implementation: finding mistakes in calling conventions and missing error checking (using satabs).

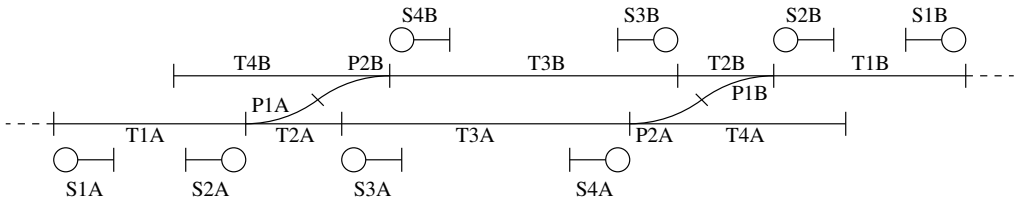

Figure 1: The layout of the passing area.

Part 1 - SMV and Temporal Logics

In this part, we are going to verify the system that controls the signals and switches of a passing area in a single track rail road. Such systems are called *interlockings*.

Modelling Approach

Many systems, interlockings included, may not work correctly given random inputs or depend on controlled hardware behaving in a certain way. E.g. trains do not (dis)appear at random and switches take time to move between their two positions. Thus, to check the correct behaviour of an interlocking, we need to provide not just the formulas against which the interlocking must be validated, but also an environment in which this is done.

This yields NuSMV models with three parts:

Environment This part describes the behaviour of the environment in which the interlocking is to be validated.

Interlocking Describes the behaviour/implementation of the interlocking

Specification Contains all of the properties that the system must satisfy.

We will use the NuSMV module system and put each of these parts in its own module (and file).

Description

An abstract picture of the passing area is given in Fig. 1. The dashed lines on the left and on the right are the outside world where trains arrive from and leave to. The line segments represent tracks and each track is given a label starting with `T`. Each track can be occupied or not. At the end of each track, a signal is placed that tells the trains driver if it is safe to enter the next track, by showing green, or not, by showing red. Finally, the layout contains four switches (formally called sets of points or points) with labels starting with a `P`. A set of points, such as `P1A` can be locked in the straight position, moving or locked in the curved position. When both `P1A` and `P2B` are locked in the curved position train can go from `T1A` to `T3B` and back. When a train goes across a set of points that is not locked or locked in the wrong position, then the train could derail, so the interlocking must prevent this. When `P1A` and `P2B` are locked straight then trains can safely travel from the track `T1A` across the track `T2A` (which includes the points) to track `T3A` and back. The requirement that `P2B` is locked straight is not to prevent derailment, but is meant to prevent a runaway train on `T3B` from hitting the flank of a train moving along `T2A`.

Interface and Atomic Properties

In order to be able to verify different environments, models and set of properties independently, they must be written in different files that adhere to an API and then be put together. The API is fixed by the file `passing-main.smv`. This file contain the `Main` modules, which instantiates two modules: `Environment` and `Interlocking`. Moreover, it defines the atomic properties that can be used in formulas.

The environment controls the variables that model the world. That is, it sets the occupation of track according to what is allowed by the signals and it controls the two signals that control access to the outside world (`S1A` and `S1B`) as well as it simulates the operation of the points. The interlocking controls the other signals directly and it can issue orders to the points that the environment must follow.

Note how the inputs that are passed: the environment gets the old control values as input and the interlocking gets the newly generated environment inputs as inputs. So updates occur in two steps: first, the environment determines the new inputs; second, the interlocking reacts to the new inputs. The advantage of this is that the outputs of the interlocking can match the newly generated inputs and thus the formulas. In other words, the time needed for the interlocking to react to the environment is hidden in the update order.

The atomic properties that are available are:

T*id*`_occupied`	Track *id* is occupied.
S*id*`_red`	Signal *id* is red.
S*id*`_green`	Signal *id* is green.
P*id*`_locked_straight`	The set of points *id* is locked in the straight position.
P*id*`_locked_curved`	The set of points *id* is locked in the curved position.
P*id*`_goal_straight`	The interlocking orders the environment to move the set of points *id* into the straight position and lock it.
P*id*`_goal_curved`	The interlocking orders the environment to move the set of points *id* into the curved position and lock it.

Provided Models and Running the Tool

The file `assignment1.zip` contains the given files for this exercise. Open a new terminal, unzip this file and cd into the folder nusmv. In this folder, the central file `main.smv` can be found, as well a trivial environment `trivial-env.smv`, a trivial interlocking `trivial-int.smv` a file with a few simple use case formulas `properties.smv` and a short file that includes the already mentioned parts in the proper order `trivial-case.smv`. This is done by using the C Pre Processor, so we need to pass the options `-pre cpp` to NuSMV to invoke it.[1] Another useful flag is `-dcx` which suppresses counterexample generation. Finally, if you use the `TRANS` keyword you should check if your model is deadlock free with the `-ctt` option.

You should try all of these options by issuing the command

```
NuSMV -pre cpp -dcx -ctt trivial-case.smv
```

[1] Comments can lead to warnings from cpp because they are not legal C code. The work-around is to start comments with `--//`.

The output ends with

```
############################################################
The transition relation is total: No deadlock state exists
############################################################
-- specification EF T3A_occupied  is false
-- specification EF T3B_occupied  is false
```

if all is well.

Exercises

To help with the formula writing, we provide three models:

trivial Perfectly safe model because all signals are red and nothing ever happens. See `trivial-case.smv` and included files.

simple Model with a bad interlocking, that work in the given limited environment only. See `simple-case.smv` and included files.

error Model which has an undisclosed error built in. See `error-case.smv` and included files.

i. Write a set of properties that validates the correctness of the various instances of the model. **(20 pt)**
Put these properties in a file `properties.smv`.
Use comment lines to briefly describe the intention of the formulas in plain English (or Dutch).
Also classify the properties as safety, liveness, or other.
The set of properties should include:

- Proper behaviour of the signals. E.g.
  - The signal show either green or red.
  - The signal shows red if passing the signal is unsafe due to occupied tracks or points that are not locked.
- Proper behaviour of the points. E.g.
  - The system never issues conflicting commands.
  - The points always follow the given commands.
- Trains always make progress.
- Several use cases for the environment: E.g.
  - There is a trace in which track `T3A` is occupied.
  - There is a trace in which trains pass each other in the middle.

  Invent at least two other use cases yourself.

ii. The file `error-combined.smv` contains a model with an error. This model is combined with the formulas in the file `error-case.smv` that can be checked with the command **(10 pt)**

```
NuSMV -pre cpp -ctt error-case.smv
```

Take one of the counter-examples against an LTL liveness property and explain the order of events in plain English. That is, use phrases like 'train moves from track x to track y' and 'signal z turn red/green' to describe what happens in each step of the counterexample.

iii. The module that simulates the points in the given simple environment moves between positions in precisely two steps. Copy the file `simple-env.smv` to the file `delay-env.smv` and change the points module in such a way that the movement between positions takes an arbitrary finite amount of steps. Verify that the points work as intended, ignore other errors. The resulting file `delay-env.smv` has to be handed in. The file `delay-case.smv` can be used for running NuSMV on the model. **(10 pt)**

iv. Among the errors when testing with the random delay points there have to be safety errors for the signals. Take the LTL counter-example to one of these errors and explain the order of events in plain English. That is, use phrases like 'train moves from track x to track y' and 'signal z turn red/green' to describe what happens in each step of the counterexample. **(10 pt)**

v. Extend the model until it handles the case of two trains correctly: **(20 pt)**

- Extend or rewrite the environment until it supports two trains. Put the result in the file `full-env.smv` and hand it in.
- Extend or rewrite the interlocking controller until it correctly controls the passing area in the presence of two trains. Put the result in the file `full-int.smv` and hand it in.

The file `full-case.smv` can be used for running NuSMV on the model.

Part 2 - Software Model Checking

Introduction

In this exercise, we will use satabs to expose errors in a library that wraps a low level file system library. The wrapping layer has several tasks:

- Check that every operation is legal and return an error code otherwise.
- Hide directory data by working with directory identities.
- Hide file data by working with file descriptors.

The concept of this exercise is that you expose bugs in the given code using satabs and fix them. The given code is supposed to be a library, so you will have to write the test case(s) yourself. Also, you will have to insert instrumentation, such as assertions, to detect problems.

You are free to modify the given code in order to make it easier to verify. For example, if you inline a procedure call then satabs tends to work a bit better. Similarly, global variables work better than procedure arguments. Restricting recursive procedure calls to a fixed depth to make global arguments work is also acceptable.

In other words, to solve this exercise it is *strongly recommended* to first simplify the problem until you can solve it and only then generalize the solution.

Provided files

The provided files for this exercise are:

`fs.h`	The headers for the low level file system library.
`wrapped_fs.h`	The headers for the wrapping library.
`wrapped_fs.c`	The (initial) implementation of the wrapping library.

Exercises

These exercises require you to develop a main procedure that is used to test the wrapper library. This procedure should use random values to test thoroughly. However, every test case should initialize once at the start and never again.

For example, the simple test case

```
int d1;
init_fs();
d1=open_dir(2,"test");
assert(d1<0);
```

can be made non-deterministic by replacing the `2` with a non-deterministically chosen value.

vi. The wrapping library uses fixed size arrays for mapping directory descriptors to directories and for mapping file descriptors to files. Use `satabs` to find at least one array out of bounds error and fix this error. **(10 pts)**

In the report you should describe

- The command line you used for satabs.

- An analysis of the counter-example that was produced by satabs.

You should hand in as files:

- The version of the main test procedure as `main_bounds.c`.
- The input for the test case as `wrapped_fs_bounds.c`.
  (Please hand in this file even if it is a copy of `wrapped_fs.c`.)
- The output of the test run as `bounds_output.txt`

vii. Starting with the version in which the detected array bounds bug(s) have been fixed, use satabs to verify that open/close are properly *nested*: **(20 pts)**

- A file can only be read or written when it is open.
- A file can only be open when the parent directory is open.
- A directory may only be open if all of its ancestors are open. In other words, all of the directories on the path from the root to the directory must be open.

In the report describe

- The key idea behind your approach to this verification.
- A description of one of the bugs that you found with the tool, complete with the command line to find it.
- The command line(s) needed to validate the final improved version.
  Note: the final version has to be improved (at least one more bug fixed), not bug-free.

Hand in as files:

- The version of the wrapper library in which you found the bug as `wrapped_fs_nesting.c`.
- The version of the test procedure used to find the bug as `main_nesting.c`.
- The output of satabs that found the bug as `nesting_output.txt`.
- The final improved version of the wrapped library as `wrapped_fs_final.c`.
- The final version of the test procedures as `main_final.c`.

1 Summary of files to be handed in

You should hand in is a single ZIP file with the following files:

report.pdf	The report with all of the explanations specified in the questions.
	passing area
`properties.smv`	Properties for validating the passing area models.
`delay-env.smv`	Environment with one train and arbitrary delay points.
`full-env.smv`	Environment with two trains.
`full-int.smv`	Interlocking for environment with two trains.
	file system wrapper
`wrapped_fs_bounds.c`	The version of the wrapper library in which you detected the bound error.
`main_bounds.c`	The version of the main procedure used for detecting the bounds error.
`bounds_output.txt`	The output of the tool for the run that detected the array bounds error.
`wrapped_fs_nesting.c`	The version of the wrapper library in which you detected the bound error.
`main_nesting.c`	The version of the main procedure used for detecting the bounds error.
`nesting_output.txt`	The output of the tool for the run that detected the nesting bug.
`wrapped_fs_final.c`	The final improved version of the wrapper library.
`main_final.c`	Final version of the main test procedure.

In all files, and on the first page of the report, write your names and student numbers.