Formal Verification

This document is a guide that intends to put into practice important concepts of formal verification, focusing on model checking of systems that have a finite state representation. There are a number of tutorials and exercises that will teach you how to use available tools. This lab will be evaluated based on the exercises that are proposed along it.

1. Introduction

1.1. Preliminaries

As the complexity of electronic systems increases, the likelihood of subtle errors becomes much greater. A way to cope, up to a certain extent, with the issue of correctness is the use of mathematically-based techniques, known as formal methods.

Correctness plays a key role in many applications. For the levels of complexity typical to modern electronic systems, traditional validation techniques like simulation and testing are neither sufficient nor viable to verify the correctness of such systems. First, these methods may cover just a small fraction of the system behavior. Second, bugs found late in prototyping phases have a negative impact on the time-to-market. Third, as more applications become dependent on computer systems, a failure may lead to catastrophic situations, e.g. in safety-critical systems like transportation, defense, and medical applications.

One of the well-established approaches to formal verification is model checking. It is used to determine whether the model of a system satisfies a set of required properties. Model checking is fully automatic and can produce counterexamples for diagnostic purposes. The main disadvantage of model checking is the state-explosion problem. Thus the key challenges are the algorithms and data structures that allow to handle large search spaces.

In model checking, a number of desired properties (called in this context specification) are checked against a given model of the system. The two inputs to the model checking problem are the system model and the properties that such a system must satisfy, usually expressed as temporal logic formulas.

A temporal logic is a logic augmented with temporal modal operators which allow to reason about how the truth of assertions changes over time. Temporal logics are usually employed to specify desired properties of systems. There are different forms of temporal logics depending on the underlying model of time. We focus on CTL (Computation Tree Logic) because it is a representative example of temporal logics and we will use it in this lab in order to specify properties of systems.

CTL is based on propositional logic of branching time, that is, a logic where time may split into more than one possible future using a discrete model of time. Formulas in CTL are composed of atomic propositions, boolean connectors, and temporal operators. Temporal operators consist of forward-time operators (G globally, F in the future, X next time, and U until) preceded by a path quantifier (A all computation paths, and E some computation
Figure 1 illustrates some of the CTL temporal operators. The computation tree represents an unfolded state graph where the nodes are the possible states that the system may reach. The shaded nodes are those states in which property \( p \) holds. Thus it is possible to express properties that hold in the root node (initial state) using CTL temporal operators. For instance, \( AF \ p \) holds in the initial state if for every possible path, starting from the initial state, there exists at least one state in which \( p \) is satisfied, that is, \( p \) will eventually happen. The other temporal operators might be interpreted in a similar way.

In CTL, time is not mentioned explicitly. Temporal operators only allow to describe properties in terms of “next time”, “eventually”, or “always”.

TCTL is a real-time extension of CTL that allows to inscribe subscripts on the temporal operators to limit their scope in time. For instance, \( AF <n \ p \) expresses that, along all computation paths, the property \( p \) becomes true within \( n \) time units.

### 1.2. Getting Started

The tool that will be used during this lab session is UPPAAL (http://www.uppaal.com). It is installed in the course directory

\[
/home/TDTS30/sw/uppaal-4.0.4
\]

In order to use this model checkers, the module tdts30 must be loaded. This can be done using the following command:

```
module add /home/TDTS30/modules/tdts30
```

If you want the module to load automatically when you log in the next time, enter the
command:

```
module initadd /home/TDTS30/modules/tdts30
```

When you have completed the course and all lab assignments, the module can be unloaded using the following commands:

```
module rm tdts30
module initrm tdts30
```

### 2. UPPAAL Tutorial

UPPAAL is a tool for modeling, validation, and verification of real-time systems. Validation can be performed via simulation whereas verification can be done via automatic model-checking. In UPPAAL, systems are modeled using timed automata (in a simplistic form, timed automata are finite state machines enhanced with time). Time is measured in UPPAAL through real-valued variables called clocks. These clocks should not be confused with hardware clocks. Clocks should rather be considered as stop watches or chronometers. Time is continuous and all clocks advance at the same rate, though it is possible to test the value of a clock or reset it.

- **UPPAAL** provides a quite user-friendly interface that allows graphical editing of systems and their validation/verification. In order to start the tool, type in any terminal window

```
uppaal &
```

#### 2.1. The First Example

- You will start with a pre-defined example. It corresponds to the system shown in Figure 3. In the menu bar of UPPAAL main window select *File -> Open Project* and open the file `/home/TDTS30/tutorial/uppaal/simple.xml`.

```
Figure 3. Example of Timed Automata
```

The model consists of two automata $A$ and $B$. You can check each of them by clicking on the respective name. Each of the automata has three locations $\{a_1, a_2, a_3\}$ and $\{b_1, b_2, b_3\}$ respectively. Their initial locations are, respectively, $a_1$ and $b_1$. While in *Select Mode* (click the big arrow in the toolbar), in the *Editor* tab, you can double-click on the different elements of the automata in order check their properties. For example, if you double-click on the location $a_1$ of automaton $A$, you will easily find out that it is the initial location of that automaton.

- Select the automaton $B$, and double-click on the edge (transition) $b_2\rightarrow b_3$. A window will pop up indicating that its *guard* is $y==1$ and its *update* is $c_b=0$. A guard is a set of con-
ditions that must be satisfied to allow the automaton to fire the transition and change its location. In this case, the automaton \( B \) can change from location \( b_2 \) to location \( b_3 \) only if variable \( y \) is equal to 1. Additionally, when \( B \) changes from \( b_2 \) to \( b_3 \), the clock \( c_b \) is reset (in this example, \( c_a \) and \( c_b \) are clocks).

- Now select the automaton \( A \), and double-click on the location \( a_3 \). The window that pops up indicates that \( a_3 \) has a location invariant \( c_a \leq 3 \). This means that the automaton \( A \) only can stay in \( a_3 \) as long as the clock \( c_a \) is less than or equal to 3. Otherwise, it is forced to change its location.

- You may see there is a label \( t? \) on the edge \( a_2-a_3 \) and a label \( t! \) on the edge \( b_1-b_2 \). This signifies that \( t \) synchronizes automata \( A \) and \( B \) so that a transition from location \( a_2 \) to location \( a_3 \) must be accompanied by a transition from \( b_1 \) to \( b_2 \). Synchronized transitions are consequently always taken simultaneously.

- On the left part of the window, click on Declarations to find out that there are two global “variables” (well, \( y \) is an integer variable while \( t \) is a synchronization label declared by the keyword chan). The clock declarations for \( c_a \) and \( c_b \) are local to each automata. You may see this by expanding the trees of the automata and then clicking on Declarations.

- By clicking on System declarations, on the left part of the window, you may see that the system is composed of the two automata \( A \) and \( B \).

- Once you have understood the different elements of the timed automata model, based on this simple example, you may validate the system via simulation. Click on the Simulator tab to start the simulator, and if a window pops up, asking whether you would like to update the model on the server, click on the Yes button. Figure 4 shows the view of the simulator at this point.

On the left you see the simulation controls where you can select an enabled transition to fire (upper part) and replay/open/save a trace (lower part). The middle corresponds to the valuation of variables at the current state. On the upper right you see the system and its current state. The lower right illustrates the simulation trace graphically.

- You can see that the only transition initially enabled is \( a_1-a_2 \). Click on the Next button and observe how the automata locations change. Note that now the transition \( a_2-a_3 \), which must be taken at the same time as \( b_1-b_2 \) due to the synchronization label \( t \), is enabled. Click again on Next and observe the valuation of variables: these are given by the relations \( y=1 \), \( c_a \in [0,3] \), \( c_b \geq 0 \) and \( c_a \leq c_b \). In general, valuation of clocks are not given as equalities but as intervals. At this point there are two transitions enabled, namely \( a_3-a_1 \) and \( b_2-b_3 \), so that you can select either. Continue the simulation until you have thoroughly understood the behavior of the system.

- You can invoke the verifier by clicking on the Verifier tab. There are two properties to verify, given by \( AG \neg B.b_3 \) and \( EF A.a_3 \). The first one expresses that the state in which the automaton \( B \) is in location \( b_3 \) will never be reached. The second says there is a computation path that leads to the automaton \( A \) being in \( a_3 \). In the Options menu, be sure that the Diagnostic Trace option is set to a different value than none, e.g. some. Then, select the first property and click on the Check button. If you get a message asking if you want to destroy the old trace, answer Yes. It turns out that \( AG \neg B.b_3 \) does not hold in the system model and the tool generates a path that makes it fail. Go to the simulator to follow the trace generated by the model checker: the left lower part of the window contains the trace. Select its beginning by clicking on \((a_1,b_1)\) and then click the Next button (in the lower part) to advance the simulation of the generated trace.
After a few steps, you will see why the property given by $\text{AG } \neg B. b_3$ is not satisfied. You may go back to the verifier and check the second property.

2.2. Drawing Automata

- Select File -> New System in order to clear the example simple and define a new system. Go to Editor tab. You will see that the initial location of your first automaton has already been drawn for you. In order to draw a second location, click on the add location button (the big circle) in the toolbar. Click anywhere in the drawing area to get the location. Select the select tool (the big arrow) from the toolbar, and double-click on these locations to name them $s_0$ and $s_1$.

- Select the add transition button (the small arrow), click on the location $s_0$ and then on $s_1$. Now click on $s_1$, then on an intermediate point (different from the existing locations), and finally on $s_0$. Select the select tool again. Right-click on location $s_0$ and make sure that it is marked as initial. Double-click on the edge $s_0$-$s_1$ and write $a=1$ in the Update: field. Similarly, double-click on the edge $s_1$-$s_0$ and write $a=0$ in the Update: field. Rename your automaton template to $P$ by modifying the contents in the name field and press enter. Your automaton $P$ should look like the one in Figure 5(a).

- Select Edit -> Insert template to add a new automaton template. Rename the default template $P_0$ to $Q$. Complete the automaton $Q$ so that it looks like the one in Figure 5(b). Note that $c<5$ corresponds to location invariants in this case. You can also experiment using the middle mouse button when drawing automata.
• Declare globally an integer variable \( a \). Declare, locally to \( Q \), a clock \( c \).

• Click on System declarations on the left. You can now see the syntax on how to create several identical automata. The templates can be parameterized, and this is where you provide the actual arguments to these templates. However, in our case, we only want to create one instance of each template. Therefore, remove all the contents, and write `system P, Q;` indicating that the system consists of these two processes (templates).

### 2.3. Fischer’s Mutual Exclusion Protocol

Another pre-defined example, modeling a realistic problem, is Fischer’s mutual exclusion protocol. This example is included as a demo in the distribution of UPPAAL.

• Load the project `/home/TDTS30/tutorial/uppaal/fischer.xml` and save it in your work directory. The system consists of \( n \) processes \( P \) (in this particular case \( n=4 \), Nonetheless it can easily be extended), each performing read and write operations on a shared variable \( id \). Each process \( P_i \) executes the following algorithm:

```plaintext
repeat
  await \( id=0 \)
repeat
  \( id:=i \)
  delay
until \( id=i \)
Critical section
\( id:=0 \)
forever
```

• Study the global and local declarations, as well as the way the template \( P \) is instantiated (click on System declarations, on the left part of the Editor window), in order to clearly understand the definition of the system. Check the invariants of locations, and guards and assign statements of edges. Once you have got a good understanding of the system, go to the simulator and run a few traces in order to get a better feeling of the system behavior.

• Invoke the model checking engine and verify the mutual exclusion requirement, i.e. no two processes should be simultaneously in their critical sections.

Properties to be verified (queries in UPPAAL terms) are not saved together with the model. They are saved in separate files with the extension `.q`. Choose Queries -> Save queries in the menu in order to save the properties. If the query file is given the same name as the model (just with a different extension), it is loaded automatically together with the model.
2.4. Assignments

1. Load the project /home/TDTS30/tutorial/uppaal/exercise.xml. Verify the two simple properties included in the system. Explain the results.

2. Load the project /home/TDTS30/tutorial/uppaal/fischer.xml and save it in your work directory. Modify it in order to describe instances of Fischer’s mutual exclusion protocol for \( n = 8, 9, 10, 11 \) and verify the mutex requirement for each of them. Note that the formula expressing the mutex requirement varies according to the size of the problem (number of processes).
Present, in a table, the verification time in terms of the number of processes. Note that the graphical interface in UPPAAL does not report the verification time. Therefore, you need to measure this time “by hand” (for instance, use your wristwatch). How long time would it take to verify 12 processes?

3. Load the project /home/TDTS30/tutorial/uppaal/fischer.xml and save it in your work directory. Locally to the process \( P \), declare a constant \( m \) with value 1 (note that a constant \( k \) with value 2 has already been declared). Change the guard of the edge \( \text{wait-CS} \) to \( x > m \), \( id = i \) (instead of \( x > k \), \( id = i \)). Now verify the mutual exclusion requirement and explain the results. Change the values of \( k \) and \( m \) and verify again the mutex requirement. Can you come up with a relation between \( k \) and \( m \) such that the mutex property holds?

4. Traffic Light Controller: Using the timed automata in UPPAAL, describe the model of a traffic light controller. The system must control the lights in a road crossing. There are four lights, one for vehicles traveling in direction North-South (NS), one for vehicles travelling SN, one for the direction West-East (WE), and one for EW, and respectively four sensors that detect the vehicles in a particular direction. The lights shall work independently. For example, if the light NS is green, the light SN is red if there are no cars coming in the direction SN. Of course, safety constraints apply, for instance, SN should not be green at the same time WE is green. Make use of clocks to represent timers that you may need for the controller.
Prove that the following two properties hold in your system model:
a) Liveness: If a vehicle arrives at the crossing (as detected by the respective sensor) it will eventually be granted green light.
b) Safety: Lights on perpendicular directions must not simultaneously be green.

5. Alternating Bit Protocol: The alternating bit protocol is a well-known communication protocol, based on the sliding window technique, which is intended to provide reliable transfer on a lossy and noisy channel. The sending and receiving of messages alternate between two modes (0 and 1).
In order to send a message, the sender sends the message together with its current mode (\( s0 \)). The sender then awaits an acknowledgment from the receiver; if the receiver acknowledges reception in mode 0 (so that the sender gets \( \text{rack0} \)) the sender switches to mode 1 and behaves in a similar way (but in the alternative mode); if the receiver acknowledges reception in the opposite mode (the sender receives \( \text{rack1} \)) the sender retransmits \( s0 \); the sender may also time out, in which case the message \( s0 \) is also retransmitted.
The receiver receives the message \( r0 \) (message sent by the sender in mode 0), or the message \( r1 \) (message sent by the sender in mode 1), or times out. In the last two cases the receiver sends the acknowledgment \( \text{sack1} \) (corresponding to an error message). In the first case the receiver sends an acknowledgment \( \text{sack0} \) and switches to mode 1.
Using timed automata, model the system consisting of one sender, one receiver, and one unreliable channel. Check whether the system satisfies the following properties:

a) messages sent by the sender are eventually received by the receiver;

b) the receiver might send an acknowledgment;

c) the system can not deadlock.

(Note: make the necessary assumptions about features of the system that can not be deduced from the above description).

You must hand in a report with your solutions to the assignments in this section. Your assistant may ask you to show a demo of your solutions. Excessively unnecessarily complex solutions, and unintuitive solutions with few or no explanations and comments, will be returned without consideration.