LVM - Laboratory 2

October 26, 2017

This tutorial serves as a very basic introduction to the PROMELA language and Spin model checker. It is necessarily very incomplete and you are encouraged to search other sources of information to learn further details about the these topics. If you use more advanced machinery in your submission, you will not be penalized. I reccomend Mordehcai Ben-Ari's excelent "Principles of the Spin model checker" as a reference (from which I shamelesly "borrowed" some examples). Another source used in the preparation of this guide were Sagar Chaki's slides from his lectures at CMU.

Let's start with a simple problem: we want to take a a two-digit number and reverses its digits. There are two approaches that we will consider: Either we gather and store gather the "10's" digit in the "1's" place and gather the "1's" digit and store it in the "10's" place (in either order) or we exchange both digits in a single atomic arithmetical operation. Both approaches are exemplified in the following program graphs:

The atomic version is *deterministic* in the sense that it only has one possible execution. The phased version in *nondeterministic* since it can choose between two possible executions: either we first gather the "1's" digit or the "10's" digit. We will use this simple example to introduce the PROMELA language.

PROMELA

Promela is a simple programming language that we will use mostly as a modelling tool. It stands for PROcess MEta Language and allows features such as dynamic process creation, asynchronous execution of processes or communication by shared variables and message channels. It has a C like syntax for expressions and control statements are in the form of guarded commands.

Sequential programming

The following is a PROMELA program that reverses the digits of a two-digit number. Programs in PROMELA are composed of a set of processes; here we start with a single process declared by the words **active proctype**. Processes may have parameters and even if there are no parameters, the parentheses ( ) must appear. The statements of the process are written between the braces {

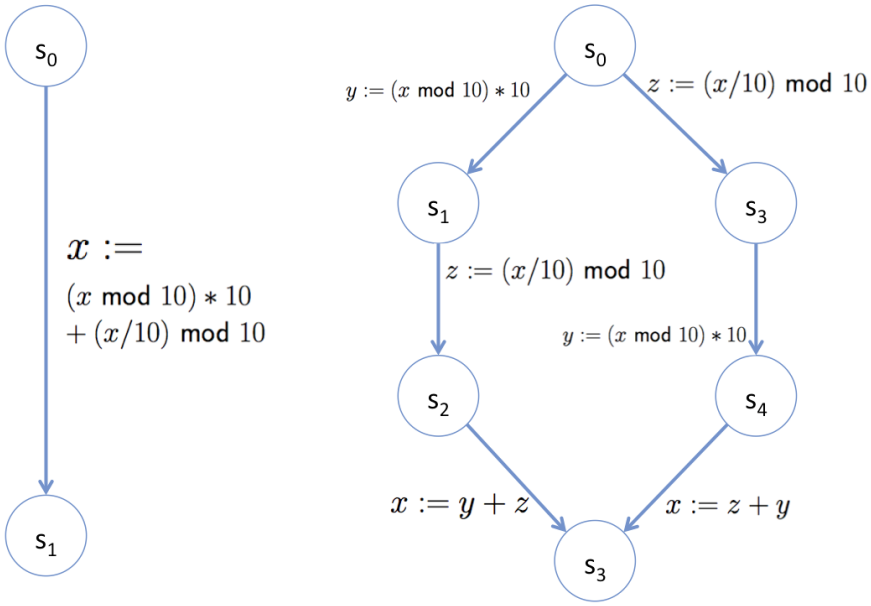

Figure 1: On the left: atomic reversal of digits. On the right: phased reversal of digits.

and }. Comments can be enclosed between /* and */. Notice the similarity with C.

```
1  active proctype P() {
2      int value = 54;
3      int reversed;
4      reversed=
5          (value % 10)*10+
6          (value/10) % 10;
7  printf("value=%d, reversed=%d\n", value, reversed)
8  }
```

If you have a keen eye for debugging, you will notice that the printf command does not end with a ;. This is not a typo. In PROMELA, semicolons are not *statement terminators*, they are *statement separators*, which means that the last statement does not need a semicolon after it[1].

1. In the beginning the earth was empty, and then...

[1]Another statement separator is "->", which works exacly as ";". The reason for the existence of this syntatic sugar will become apparent once we go over conditionals

- Open SpinRCP by clicking the "windows" key and writing "spinrcp" in the search field.
- Create a new Spin project, by clicking "File→Other→Other". Check the box in "Show all wizards", click "General→Project→OK". Name your project "LVMLab1" and click "Finish". You have now created a project for this class.
- Right click your project name (on the left) and choose "New→Other→Other →Promela file". Name your file "reverse.pml" and click "Finish". You have now created your first PROMELA file.
- Now copy the above program into "reverse.pml" (the main window in the center).

We typically will use PROMELA to perform *automated verification*, which is done *statically*, without actually executing the program. However, PROMELA also allows us to *simulate* executions of the program, which can be useful to debug our models. There are three ways to simulate executions in PROMELA: *random*, *interactive* and *guided*. A random simulation executes the program until a nondeterministic choice has to be made. At that point, as the name implies, it chooses randomly which command is executed, among those available. Interactive simulations on the other hand, make this decision by asking the user which command should be run next. Finally, if we already have produced an execution in the past, we may have a "trail" file (more on this later) that we may want to re-run in a guided (by the trail file) simulation. Let's simulate our reversor.

2. Simulation is the sincerest form of flattery.

- Click "Simulation" in the main banner (center, top).
- Choose "Random" and any seed you like (since this program is deterministic, the choice of seed is irrelevant). Click "Ok".
- Click "Run" in the right side panel to simulate the execution.

If all went right, you should now see a cryptic piece of output. On the top side of your main window, you will have a yellow block. This represents the one active instance of process P. Since there are no other processes, this poor block is all alone. In the bottom window, on the other hand, there are words! Let's examine this output line by line:

/usr/local/bin/spin -X -p -v -s -r -l -g -n123 -u10000 reverse.pml

This is what actually was sent to the terminal. If you wish to run spin in a command line, or to check which options are active, check this line (more on spin options later).

0: proc - (:root:) creates proc 0 (P)

At state 0 the process *root* (which always exists) spawns a new process, with *process id* 0. It is of type P.

spin: reverse.pml:0, warning, proctype P, 'int reversed' variable is never used (other than in print stmnts)

Ah, a warning in processes of the type P. Apparently variable reversed is never used! This is because printf commands (and other *io* commands) are ignored during verification. PROMELA is designed to model *closed* systems, without outside interaction. Print statements are only executed during simulation and therefore variables used only in Print statements are not considered used.

1: proc 0 (P:1) reverse.pml:4 (state 1) [reversed = (((value%10)*10)+((value/10)%10))]

We now execute the next available command of process with id 0 (of type P). This is the command described in file "reverse.pml", line 4. This command is shown inside [ ].

P(0):reversed = 45

There was a single change in state assignment, variable "reversed" in process with id 0, of type P, became 45.

value= 54, reversed= 45
2: proc 0 (P:1) reverse2.pml:7 (state 2) [printf('value= %d, reversed= %d\n',value,reversed)]

It seems that we are printing before we simulate the printf, which would be absurd, but in reality what is happening is the following: since the printf function has a side effect (printing to the console), when we simulate printf the side effect happens (generating the first line), and the report about the transition effects (second line) is printed only after the transition is done.

2: proc 0 (P:1) terminates
1 processes created
Random simulation trail written to reverse.pml.rnd_sim.out

Having no further commands to execute, process with id 0, terminates (in order to save memory, termination occurs in the same state as the last command). Then, since there are no more active processes, a report of the execution is presented and a trail file is written (it should now be in your project folder), in case we want to replay this specific trace in a guided simulation.

Multiple Threads

We are now going to dive in one of the main features of PROMELA: concurrency. Consider the following PROMELA model, which you can find in file "example1.pml", in the course webpage.

```
1   byte state = 1;
2   proctype A(){
3       byte x = state;
4       x = x+1;
5       state = x;
6   }
7   proctype B(){
8       byte y = state;
9       y = y+2;
10      state = y;
11  }
12
13  init{ run A(); run B() }
```

In this model we are using a *global (or shared)* variable, state, accessible to all processes. We declare two different types of processes A and B. Notice that none of the processes is active this time, which means that when we simulate the execution of this model, there won't be a copy of any process (except "root") immediately spawned. In order to get processes of type A and B we can use the run command. A "run $X(args)$" command *spawns* a new process of type X with arguments $args$. init is a special process that always starts active. We use it to bootstrap our implementations.

Processes *do not* run sequentially, one after the other (that is the whole point of concurrency). Instead, PROMELA *interleaves* all active processes: at each point in time, it nondeterministically chooses one process and allows it to execute the next command.

3. Stop touching my stuff!

- Import file "example1.pml" into your project.
- Try to predict the result of simulating this model.
- Simulate this model in "Random" mode with several different seeds[2]. Analyse the final assignment for state. Is it what you expected?
- Simulate this model in "Interactive" mode. How many different final values of state can you create?

What you just saw was what something called a *race condition*. Race conditions happen when two or more processes try to access and work on the same resource at the same time. The interactions resulting thereof can be hard to predict. We usually try to avoid race conditions by using semaphores or synchronization, which we will see in a bit.

Of course we can also *cheat*! There is a keyword in PROMELA, atomic, which can be used to wrap a few commands with the effect that a process that

[2]By the way, notice that you now have two yellow blocks (one for each process)! Amazing...

starts running an atomic will retain priority until it finishes the sequence of commands wrapped by the atomic statement. Of course this is an unreasonable restriction of the behaviour of the system. Processes *can* loose priority in the middle of an execution. Atomic is used as a hack by experienced model builders: one *can* build a semaphore system that guarantees that a sequence of statements will behave *as if* wrapped by an atomic. However, the PROMELA model will have to have a few more states as a result of inserting this semaphore system and may become slow to verify. In these cases, we can do a gentlemans agreement that the actual implementation will have a semaphore system, but for efficiency reasons, the PROMELA model uses an atomic, instead. You are *not* "experienced model builders" (yet) and as such, you are *not* allowed to use atomic in your submissions. If you do, you better have a very convincing reason.

4. Teh hackz.

- Import file "example2.pml" into your project. Notice how it wraps the changes to state as atomic.
- Try to predict the result of simulating this model.
- Simulate this model and confirm (or dispel) your suspicions.

Remark: Variables declared inside of a process are *local* to that process. Different processes can declare an use variables with the same name without worrying about them being shared. for example, in the previous example, both A and B could have declared local variable x instead of using x and y.

Process instantiation and parametrization

We have seen two ways of spawning new processes so far, either by declaring them active from the start of by run in them. In an incredibly bad analogy, I'm going to compare active processes with static classes in OOP and processes that are run with instantiated classes[3]. Just like you can instantiate the same object several times, you can also run processes of the same type several times, creating a new instance of the process for each run. You can also call run at any tie, not only inside of init, but inside *any* process.

5. "There are so many 'me's" - Agent Smith.

- Go back to "example1.pml" and substitute line 13 by init{ run A(); run B(); run A();} . Don't forget to save!
- Simulate this model. How many different outputs do you expect?

[3]If you ever show this to my OOP professor, I may have to do a penance quest, but I'll fail you out of spite!

PROMELA is designed to specify *closed systems*, that is, systems without outside interactions. This is the reason why it does not have *io* functions outside of simulation and it's also the reason why it does not allow user inputs (recall that in our reversor, we specified 54 *inside* of the process, not as an instantiation of a symbolic input). However, this does not mean that processes cannot be symbolically parametrized. In "example1.pml", processes A and B are essentially the same. One adds 1 and the other 2, but the structure of the program is the same. Consider the following program, which you can find in file "example3.pml":

```
1  byte state = 1;
2  proctype A(byte inc){
3      byte x = state;
4      x = x+ inc;
5      state = x;
6  }
7
8  init{ run A(1); run A(2); run A(1)}
```

6. Fundamentally the same.

- Import file "example3.pml" into your project.
- Simulate this model and convince yourself that it is functionally equivalent to the one from the previous exercise.

Executability, commands and conditionals

A central concept in PROMELA is *executability*. Depending on the current assignment, commands can (or not) be executable. Declarations and assignments are always executable. Conditionals are only executable when they hold (the executability of other commands will be discussed later). If a conditional is not executable, the process will "wait" until it becomes executable. This allows us to use executability to be used as a means of synchronisation.

7. Wait for it...

- Load and simulate the model in "example4.pml". Notice how we can use the conditional (state==1) to "force" the process of type A to wait for the process of type B.

Noticed how we used "->" instead of ;? They have the same exact semantics, but it feels better to put an arrow after the conditional.

Executability (or lack thereof) can be a useful synchronization tool, but it can also lead to distributed systems related disasters:

8. Locked up and awaiting execution.

- Consider the model in "example5.pml". Notice that none of the processes will finish but both will forever wait for the command (state==1) to become executable.
- By default, *deadlocks* like this are considered failures of the system. Certain spin options allow us to ignore deadlocks during verification.

Control flow

So far, instances of PROMELA processes have been very simple single line programs. Despite all our fancy multiple threading, we cannot yet even model the phased reversing program of Figure 1.

Control flow in PROMELA is slightly different from most programming languages. For example, case selection is *not* necessarily based in sequential testing (like typical ifs, elifs, switches...).

Consider the following program, that implements the second program graph in Figure 1:

```
1   active proctype P() {
2       int value = 54;
3       int y;
4       int z;
5       int reversed;
6       if
7           :: y = (value % 10)*10; z = (value/10) % 10
8           :: z = (value/10) % 10; y = (value % 10)*10
9       fi
10      reversed= y+z
11  printf("value=%d, reversed=%d\n", value, reversed)
12  }
```

This program presents the case selection construct misleadingly named if...fi. Inside of an if...fi block we can have several blocks of code, each starting with ::. If a process reaches an if...fi block and has priority, it will nondeterministically choose one of these blocks to execute *as long as the first command in the block is executable at the time*. The last line bears repeating: *as long as the first command in the block is executable at the time*. This means that blocks of the form

```
if
    :: condition -> option_then
    :: !condition -> option_else
fi
```

behave exactly like if-then-else statements. Cases do not need to be *exhaus-*

tive (all branches may be blocked) and they do not need to be *mutually exclusive* (multiple branches may be executable). Optionally, one may add a branch of the form :: else -> option. An else branch is executable *only* if *all* other branches are blocked[4].

9. Just in case.

- Create a .pml file that runs 3 processes, A(), B()and check(). A() and B() can do whatever you think suitable, but check() should print "Process A gave me permission to go" or "Process B gave me permission to go", depending on the case. Hint: use shared variables to "grant permission".
- Modify your program to also allow "Last I checked, neither A nor B had given me permission".

The other usual control flow option, repetition, also has a few differences from standard repetition statements in other languages (fors, whiles, do-untils...). The most glaring difference is that repetition statements in PROMELA are, in many cases, *supposed* to run forever (the models we design are often for autonomous perpetual systems).

Consider the following ball passing circle program, which you can find in "example6.pml":

```
1  byte ball = 1;
2  proctype Player(byte myname){
3      do
4          :: (ball == myname) -> ball = ball+1% 3
5          :: (ball == myname) -> ball = ball-1% 3
3      od
6  }
7
8  init{ run Player(0); run Player(1); run Player(2);}
```

10. Better than Ronaldo.

- Try to simulate the previous program.
- Convince yourself that the previous program never stops.

Never ending programs are a problem for simulation. Fortunately, the objective of spin is to do verification, simulation is more of a debugging tool. Verification is designed to be performed even in infinitely running systems. Too bad we haven't learned how to do verification yet (patience, young grasshopper).

Let us consider a different process (one that finishes) to compute the GCD of two integers ("example7.pml"):

[4]This is very different from having a :: true -> option branch, since this one will be executable even if some other branch is.

```
1   proctype Euclid(int x, y){
2       do
3           :: (x > y) -> x = x-y
4           :: (x < y) -> y = y-x
5           :: (x == y) -> break
6       od
6   }
7
8   init{ run Euclid(18,30)}
```

This program introduces breaks. A break, if executed, lets us out of the innermost do...od cycle we are in.

11. Do or do not, there is no try.

- Simulate the Euclid algorithm.
- Change initial values, try to break the implementation.

A final note on control flow. Although it is obvious in hindsight, you can nest control flow operators. For example, the following is a valid process, that simulates a killed random walk.

```
1   proctype counter(){
2       do
3           :: (count != 0) ->
4              if
5                 :: count = count +1
6                 :: count = count-1
7              fi
5           :: (count == 0) -> break
6       od
6   }
```

Data types

So far we have used data types more or less carelessly, but PROMELA actually accepts a rather restricted set of data types, mostly to avoid infinite or intractably large internal representations.

The basic types accepted by PROMELA are: bool, byte, short and int.

We have not done so thus far, but we can also use fixed size arrays of the previous types, declared and handled in C-style:

```
byte state[20];
state[3] = state[2-i] + 5;
```

Additionally, we can define *symbolic* constants, with no semantics but useful when we want to have suggestive names:

mtype={SEND, RECV};

Finally, there is one additional basic type that we haven't discussed yet: chan, which denotes communication channels. Oh my! What a nice, completely fortuitous, segway:

Message passing

PROMELA allows channels systems as an additional means of inter-process communication.

Channels have *types* and *capacities*, as we have seen in class. They are declared with the following syntax:

chan name = [16] of {short} declares a channel, named "name" with space for up to 16 "shorts"
chan name = [5] of {bit,short} declares a channel, named "name" with space for up to 5 PAIRS (bit,short).

Message sending uses a syntax similar to the one we learned in class:

name ! expression
name ! expression1, expression2 (to send a pair)

Message receiving uses the converse syntax:

name ? variable
name ? variable1, variable2 (to recieve a pair)

12. Hold these for me for a second, please.

- Load "example8.pml". What does it do?
- Simulate it to check your intuition. Pay attention at the yellow boxes, they are finally doing things!

Channels allow a few little tricks. First of all, executability in channels is like what we've seen in class: Sends (!) are executable only when the channel is *not full*, whereas receives (?) are executable only when the channel is *not empty*. This can be exploited to impose some measure of synchronization, in particular when channel size is 0.

13. Still better than Ronaldo.

- Redo the "example6.pml" to use size 0 channels instead of shared variables.

PROMELA receive commands can *optionally* have constant values in the variable place. If this is the case, the channel will only take a message from the channel it it matches exactly the constant.

For example, consider the following model, where a leader process want to signal one of two slave processes but the three processes only have access to a single channel

```
1   chan com = [1] of byte
2
3   active proctype leader(){
4       if
5           :: com ! 1
6           :: com ! 2
7       fi
8   }
9
10  active proctype slave1(){ com ? 1 }
11
12  active proctype slave2(){ com ? 2 }
```

Notice that if the slaves were "listening" to a variable instead of trying to match a constant, slave2 could "steal" a signal intended for slave1. We would need the leader to keep sending messages hoping for the right slave to get them and have the slave acknowledgeing the leader. Quite the hassle.

14. What does it meeeeeean?!

- Load "example9.pml". I'm using a new trick here. Can you figure out what's happening?
- Load "example10.pml". Without simulating it, can you figure out what it does?

Verification

We've learned a lot about modeling systems, a lot about simulating them, but what about actual verification?

Well, that's for next Lab, but let's learn at least how to check the most basic properties with a construct that other languages sometimes also have: an assert statement. If you assert a conditional, you are essentially claiming that "no matter the execution that led here, this conditional *must* always hold".

Of course spin is not going to simulate all possible behaviors (there's an inifnite number of them more often than not), so simulation will not help us now.

We're finally going to press the "Verification" button in the central top ribbon!

Consider the following mutual exclusion protocol, presented in "example11.pml":

```
1   byte in=0 /* no one is in the critical region*/
2   byte perm=0 /* processes start with permission to enter critical */
3
4   active [2] proctype P(){  /* A new trick! 2 equal active processes! */
5       do
6           ::
7               (perm == 0 )-> /* check if anyone requested permission */
8               perm = 1; /* lock permission */
9               in = in + 1; /* enter critical */
10              (assert(in == 1)); /* there's only me in critical */
11              in = in - 1; /* exit critical */
12              perm = 0; /* release permission */
13      od
14  }
```

Let's verify this system! Click the "verification" button in the central banner. You may need to tick the "Assertion violations" box. Click "OK".

Uh Oh... the system is unsafe! Damn it!

15. I can't catch a break!

- Analyse the output of verification. Can you understand the counterexample?
- Replay the counterexample in a Guided simulation.

And that's it, we're done! Since this is the first time I'm writing a Lab guide, this may take too much or too little time. If you still have time, here's another challenge:

16. Beauty, Truth and Spaghetti

- Implement the dining philosophers. Simulate the hell out of them!
- Refine your model until you're confident it works.
- Make a few pertinent assertions and verify them.
- If a tree falls in a forest and no one is there to listen, do we have free will and can we experience objective reality? Pass the cheese, please.