# Formal Verification

Paul Wild

Monday 28th October, 2024

# Introduction to Promela

## Manuals

Promela is the modelling language of the Spin model checker. Documentation can be found here:

- `http://spinroot.com/spin/Man/`

The GitHub release also contains offline documentation that can be used without having to deal with the annoying Captcha.

Promela's syntax is similar to that of C, but the semantics often differ, so you will often want to refer back to the documentation.

# Introduction to Promela

## Process definitions

- `proctype` defines a new process type
- `active` specifies that there should be one or more active instances of this process type

# Introduction to Promela

## Process definitions

- `proctype` defines a new process type
- `active` specifies that there should be one or more active instances of this process type

## Other constructions

- `mtype` is similar to an `enum`, more on that later
- `do :: ... od` is a loop construct
- `(cond)` *blocks* execution of the process until `cond` is true, the `->` is just alternative notation for `;`
- `printf` is the same as in C and other languages

# Introduction to Promela

## Process definitions

- `proctype` defines a new process type
- `active` specifies that there should be one or more active instances of this process type

## Other constructions

- `mtype` is similar to an `enum`, more on that later
- `do :: ... od` is a loop construct
- `(cond)` *blocks* execution of the process until `cond` is true, the `->` is just alternative notation for `;`
- `printf` is the same as in C and other languages

## Simulation

Simulate a random run of the model using `spin model.pml`

# Introduction to Promela

```
mtype = { C, D };
mtype turn = C;

active proctype chef() {
        do
        :: (turn == C) ->
                printf("Cooked␣a␣delicious␣meal!\n");
                turn = D
        od
}

active proctype diner() {
        do
        :: (turn == D) ->
                printf("Ate␣the␣meal!\n");
                turn = C
        od
}
```

# Introduction to Promela

## Branching loops

- `do` loops support branching by adding multiple options, each preceded by a `::`.
- In each iteration, the process nondeterministically chooses a branch among those that are not blocked.
- For instance, we may have the chef prepare different kinds of food.

# Introduction to Promela

## Branching loops

- `do` loops support branching by adding multiple options, each preceded by a `::`.
- In each iteration, the process nondeterministically chooses a branch among those that are not blocked.
- For instance, we may have the chef prepare different kinds of food.

## More chefs and diners

- Let's modify our model by adding another chef and another diner.
- To do this, just write `active [2] proctype` instead. What happens?

# Introduction to Promela

## Branching loops

- `do` loops support branching by adding multiple options, each preceded by a `::`.
- In each iteration, the process nondeterministically chooses a branch among those that are not blocked.
- For instance, we may have the chef prepare different kinds of food.

## More chefs and diners

- Let's modify our model by adding another chef and another diner.
- To do this, just write `active [2] proctype` instead. What happens?
- The model is now suffering from *race conditions* and the diners may eat meals that haven't even been made!

# Introduction to Promela

## Atomic sequences

- We fix this by adding a third item `N` (none) to `mtype` and set `turn` while one of the processes is printing to the console.
- But how to ensure that no other process can read the value of `turn` before we change it?
- Use `atomic` to force a sequence of statements to be executed without other processes intervening (note that execution still takes multiple steps).

# Mutex

Consider the following two processes that repeatedly enter and exit a critical section:

```
active [2] proctype user() {
  bool crit;
again:
  printf("Enter␣critical\n");
  crit = true;
critical:
  printf("Exit␣critical\n");
  crit = false;
  goto again
}
```

Two new language features here: labels and `goto`.

# Mutex

Consider the following two processes that repeatedly enter and exit a critical section:

```
active [2] proctype user() {
  bool crit;
again:
  printf("Enter␣critical\n");
  crit = true;
critical:
  printf("Exit␣critical\n");
  crit = false;
  goto again
}
```

Two new language features here: labels and `goto`.

They may both simultaneously enter their critical section. But can we also get Spin to tell us about this?

# LTL model checking

- Specify the property that we want to check as an LTL formula.

# LTL model checking

- Specify the property that we want to check as an LTL formula.
- Say $p$ specifies that the first process is in the critical section and $q$ specifies this for the second. What can we use?

# LTL model checking

- Specify the property that we want to check as an LTL formula.
- Say $p$ specifies that the first process is in the critical section and $q$ specifies this for the second. What can we use?
- $\mathsf{G}\neg(p \wedge q)$, or equivalently $\neg\mathsf{F}(p \wedge q)$

# LTL model checking

- Specify the property that we want to check as an LTL formula.
- Say $p$ specifies that the first process is in the critical section and $q$ specifies this for the second. What can we use?
- $\mathsf{G}\neg(p \wedge q)$, or equivalently $\neg\mathsf{F}(p \wedge q)$
- In Promela: `ltl ok { [] ! (user[0]@critical && user[1]@critical) }`

# LTL model checking

- Specify the property that we want to check as an LTL formula.
- Say $p$ specifies that the first process is in the critical section and $q$ specifies this for the second. What can we use?
- $\mathsf{G}\neg(p \wedge q)$, or equivalently $\neg\mathsf{F}(p \wedge q)$
- In Promela: `ltl ok { [] ! (user[0]@critical && user[1]@critical) }`
- Compile and run the model checker from the command line:

```
spin -a critical.pml
cc pan.c -o pan
./pan
```

# LTL model checking

- Specify the property that we want to check as an LTL formula.
- Say $p$ specifies that the first process is in the critical section and $q$ specifies this for the second. What can we use?
- $\mathsf{G}\neg(p \wedge q)$, or equivalently $\neg\mathsf{F}(p \wedge q)$
- In Promela: `ltl ok { [] ! (user[0]@critical && user[1]@critical) }`
- Compile and run the model checker from the command line:

```
spin -a critical.pml
cc pan.c -o pan
./pan
```

- Spin reports an assertion violation and writes a trace file. We can examine the trace violating the LTL formula as follows:

```
spin -t -p critical.pml
```

# Mutex

```
active [2] proctype user() {
        bool crit;
again:
        printf("Enter␣critical\n");
        crit = true;
critical:
        printf("Exit␣critical\n");
        crit = false;
        goto again
}
```

- How can we change the model to prevent them from simultaneously entering their critical section?

# Mutex

```
active [2] proctype user() {
        bool crit;
again:
        printf("Enter␣critical\n");
        crit = true;
critical:
        printf("Exit␣critical\n");
        crit = false;
        goto again
}
```

- How can we change the model to prevent them from simultaneously entering their critical section?
- One way is to use atomic sequences. How?

# Mutex

```
active [2] proctype user() {
        bool crit;
again:
        printf("Enter␣critical\n");
        crit = true;
critical:
        printf("Exit␣critical\n");
        crit = false;
        goto again
}
```

- How can we change the model to prevent them from simultaneously entering their critical section?
- One way is to use atomic sequences. How?
- Can you also find a way to do it without?

# River Puzzle

## Ferryman, Wolf, Goat, and Cabbage

Let us try to formalize and solve this popular puzzle in `Promela`. For those who don't know: A ferryman is supposed to take a wolf, a goat, and a cabbage to the other side of a river. There is a single boat which he can use to cross the river while taking at most one passenger with him. Only the ferryman can operate the boat. However, his passengers are governed by natural instincts (well, except for the cabbage), so if he leaves the wolf together with the goat, or the goat together with the cabbage, on one side, he will have one passenger less. The question is: can he fulfil his task while avoiding this scenario?

- Why could model checking be of any use here?

# River Puzzle

## Ferryman, Wolf, Goat, and Cabbage

Let us try to formalize and solve this popular puzzle in `Promela`. For those who don't know: A ferryman is supposed to take a wolf, a goat, and a cabbage to the other side of a river. There is a single boat which he can use to cross the river while taking at most one passenger with him. Only the ferryman can operate the boat. However, his passengers are governed by natural instincts (well, except for the cabbage), so if he leaves the wolf together with the goat, or the goat together with the cabbage, on one side, he will have one passenger less. The question is: can he fulfil his task while avoiding this scenario?

- Why could model checking be of any use here?

! Output of a counter-example in case a specification is false

# Creating the model and specification

- We can model everything inside a single process, using Boolean variables to denote which side the four passengers are on. Promela has a shorthand `init { }` that we can use for this process.

# Creating the model and specification

- We can model everything inside a single process, using Boolean variables to denote which side the four passengers are on. Promela has a shorthand `init { }` that we can use for this process.
- The choice of passengers for each ride can be modelled via a nondeterministic `do` loop.

# Creating the model and specification

- We can model everything inside a single process, using Boolean variables to denote which side the four passengers are on. Promela has a shorthand `init { }` that we can use for this process.
- The choice of passengers for each ride can be modelled via a nondeterministic `do` loop.
- When shipping multiple passengers over, we want this to happen in a single *atomic* step. However, recall that Promela's `atomic` sequences still execute in multiple steps. Instead, we use a *deterministic step* or `d_step`, which is always executed as if it were a single statement.

# Creating the model and specification

- We can model everything inside a single process, using Boolean variables to denote which side the four passengers are on. Promela has a shorthand `init { }` that we can use for this process.
- The choice of passengers for each ride can be modelled via a nondeterministic `do` loop.
- When shipping multiple passengers over, we want this to happen in a single *atomic* step. However, recall that Promela's `atomic` sequences still execute in multiple steps. Instead, we use a *deterministic step* or `d_step`, which is always executed as if it were a single statement.
- Add some print statements so that you can see what happens.

# Creating the model and specification

- We can model everything inside a single process, using Boolean variables to denote which side the four passengers are on. Promela has a shorthand `init { }` that we can use for this process.
- The choice of passengers for each ride can be modelled via a nondeterministic `do` loop.
- When shipping multiple passengers over, we want this to happen in a single *atomic* step. However, recall that Promela's `atomic` sequences still execute in multiple steps. Instead, we use a *deterministic step* or `d_step`, which is always executed as if it were a single statement.
- Add some print statements so that you can see what happens.
- Finally, create an LTL formula that states that there is *no* path that corresponds to a solution to the puzzle.

# Model checking strategies

- If we run this model with Spin, it will output a solution.
- Investigation of the trace shows that it might be quite long. How can we shorten it?

# Model checking strategies

- If we run this model with Spin, it will output a solution.
- Investigation of the trace shows that it might be quite long. How can we shorten it?
- Method 1: Using iterative deepening depth first search:

```
gcc -DREACH pan.c -o pan
./pan -i
```

# Model checking strategies

- If we run this model with Spin, it will output a solution.
- Investigation of the trace shows that it might be quite long. How can we shorten it?
- Method 1: Using iterative deepening depth first search:

```
gcc -DREACH pan.c -o pan
./pan -i
```

- Method 2: Using breadth first search:

```
gcc -DBFS pan.c -o pan
./pan
```

# Model checking strategies

- If we run this model with Spin, it will output a solution.
- Investigation of the trace shows that it might be quite long. How can we shorten it?
- Method 1: Using iterative deepening depth first search:

```
gcc -DREACH pan.c -o pan
./pan -i
```

- Method 2: Using breadth first search:

```
gcc -DBFS pan.c -o pan
./pan
```

- Verify that the path generated in this way is indeed as short as possible.