Using Promela/Spin for Model Checking

- Promela/Spin Prep
- Lab 4
- Assignment 2

~~Lab 4 - Part 0. Reasoning About the Size of State Space~~

We will skip this part for L4.

~~Let M be a Promela model that has two active processes P and Q that communicate with each other via two channels. Each channel has a maximum size of two. A slot in each channel can hold one of five different message types (an mytpe with 5 values). The FSA of P and Q each has 10 local states. P and Q each has a local variable of type bool. M also has a global variable of type byte shared between the two processes.~~

1. ~~What is the maximum number of global system states that M can have?~~
2. ~~By what factor would the maximum size of the global state space increase if you increased the capacity of only one channel by 1? (Repeat calculation 1 for the system with increased channel capacity, and divide it by the original calculation 1)~~

Lab 4 - Parts 1-2. Modeling and Verification

The following defines an increasingly complex series of games. Each game builds on the previous one, so you should tackle them in the given order.
For each game below:

1. Model the described behavior in Promela using the given *Property Interface* and *Promela Template*. Name your .pml files exactly as specified by the game. Obey the property interface given. Your model should be readable/understandable, concise, and be loyal to the game specification (should model players and referee as closely as possible). Follow Promela Coding Style.
2. Simulate the model in Spin. Visualize the behavior in SpinRCP.
3. Verify the specified properties on your Promela model using Spin by running the given verifier script for the game; e.g.,
   ```
   > bash verify BrokenPingPong.pml BrokenPingPong.config
   ```
4. If a property with suffix `MustNotFailForCredit` is failing, you must fix it to receive a non-zero score for the lab component. This is a sanity check for an invalid model defined in such a way that makes all other properties trivially pass (similar to a trivially passing test).

The verifier script is available in this github repo along with all the config files.

5. Analyze the verifier report. If any of the verifications that should pass fails or that should fail passes, use guided simulation to replay the error trail generated. Is the error expected? If not, identify the faulty behavior. Fix the model (or property) if necessary and go to Step 3. If all properties are verified, go to next step.
6. Submit your .pml files to Vocareum for checking.
7. If a reference property with suffix `MustNotFailForCredit` is failing, you must fix it to receive a non-zero score for the lab component. This is a sanity check for an invalid model defined in such a way that makes all other properties trivially pass (similar to a trivially passing test).

Notes

- Promela model file names must have extension **.pml** to be recognized by SpinRCP.
- Download latest versions of lecture notes.
- Download the verifier script and config files for each game from this github repo.

- To see how each verification is run (which Spin options are used), see the corresponding config file.

Resources

- Promela/Spin manual: http://spinroot.com/spin/Man/Manual.html
- Spin options: http://spinroot.com/spin/Man/Spin.html
- Spin root: http://spinroot.com/
- Verifier script repo (with config files): https://github.com/cmusv-svvt/promela-autograder
- SpinRCP information
- Useful spin tools: runspin and parsepan
- Spin Tutorial by G. Holzmann (see under Canvas > Modules > Supplements)

Property Interface

Your Promela models must define the macros given below using the **`#define`** cpp directive, *as applicable* to the game. Your LTL properties must be named exactly as specified under each game and should be expressed only in terms of the macros below (no other global variables permitted in LTLs in these exercises). The macros should contain only expressions referring to these macros; they cannot contain any statements, such as assignments, channel expressions, control structures (e.g., do-od, if-fi) because these are not permitted in an LTL.

- **`allBalls`**: gives total number of balls in the game at a given time (a ball is in the game if either it's in mid-air or with one of the players; initially this is 0) - used in all games
- **`balls(WHITE)`**: gives total number of WHITE balls in the game at a given time (same conditions as **`allBalls`** apply) - used in JugglersBW.pml only
- **`balls(BLACK)`**: total number of BLACK balls in the game at a given time (same conditions as **`allBalls`** apply) - used in JugglersBW.pml only

Caution: **`balls`** is parameterized, e.g.,

```
#define balls(color)  ...some expression here…
```

You cannot define it twice (overload it) with the same signature by substituting values for the parameter:

```
#define balls(WHITE)   …   // WRONG
#define balls(BLACK)   …   // WRONG
```

IMPORTANT - Property Interface Macros and Observation Variables: Your model should not use the above macros or any global variables that the macros depend on and that are defined to track the state of the system (e.g., variables counting the number of different kinds of balls in the game) to change the behavior of the players or the referee. This means these variables or macros cannot be used as guards before receiving or sending a message or controlling another behavior of the players or the referee. They can only be used in the macros (and indirectly in the LTL properties) and updated properly in the processes. Their sole purpose is to specify and verify properties, rather than control the behavior of the system (i.e., they are observation, or state-revealing, variables). Players or referee should not need to

check channel contents or any of the observation variables before receiving or sending: they just need to update the necessary observation variables appropriately upon receiving, sending, or loss.

IMPORTANT - Use of atomic or d_step: You must be careful and conservative about these constructs since they can significantly change the behavior by forcing unintended sequential behavior and restricting, or even forbidding, concurrency. Their use should be limited to situations in which:

- An observation variable needs to be updated instantly in response to a process action without interference from other processes (which may attempt to update the same variable in a different way, causing a race condition). Since observation variables should not control the behavior of a process (e.g., by acting as guards), this kind of usage does not affect the behavior of a process. It only serves to provide the right value of the variable to a property that needs it.
- Sections or sub-behaviors of processes are independent and don't communicate so no matter in what order the underlying statements from different processes are executed, the outcomes will still be the same and the properties of interest won't be affected. This is the case, for example, if a sequence of statements in a process only update local variables or global variables not used by any other process and the sequence does not involve communication with any other process (e.g., by sending and receiving messages to another process through a channel).
- The intended behavior of a group of statements is truly indivisible. This means you are modeling a system that internally guarantees that these statements cannot be interleaved.

If you are putting a receive and subsequent send in a process within an atomic or d_step, then you're forbidding another process from doing anything between that send and receive. But this is not the intended behavior of the system -- it destroys the built-in concurrency between the players. This may not matter too much when there is one ball, but it matters a lot when there are multiple balls in the system (as happens in A2): one player must catch and throw a ball while another ball in mid-air and the other player are both frozen. This wouldn't make sense. Of course we can solve all concurrency problems by forbidding concurrency altogether, but we need concurrency because it naturally exists in all distributed systems and without it we wouldn't be able to run multiple applications on a single core, have multi-core systems, or allow a server to serve more than one client at a time.

IMPORTANT - channel capacities

Channel capacities should not be artificially limited. Define a high-enough capacity that accommodates multiple balls to be in the game (air can hold many balls). The number of balls should be controlled by the referee and players.

Promela Model Template for Lab 4 and Assignment 2

```
/* constants and mtypes */
mtype = { ... }
```

```
/* these two macros are for JugglersBW.pml only */
#define  WHITE          0
#define  BLACK          1
...

/* global variables and channels */
...

/* macros for the property interface -- needed for reference properties */
#define  allBalls       ...
#define balls(color)    ... // for JugglersBW.pml only
...

/* Your LTL properties */

ltl singleBallForever { ... } // for PingPong.pml
ltl alwaysAtMostOneBall { ... } // for BrokenPingPong.pml
...

/* Any mandatory sanity-check LTL properties: these must pass for non-zero
grade */

/* for PingPong.pml only */
ltl initiallyNoBallsMustNotFailForCredit { (allBalls == 0) && []true }
ltl allBallsCannotBeConstantMustNotFailForCredit { ![](allBalls == 0) &&
![](allBalls == 1) && ![](allBalls == 2) && []true }

/* for Broken PingPong.pml only */
ltl sanityCheckForAllBallsMustNotFailForCredit { (allBalls == 0) &&
![](allBalls == 0) && ![](allBalls == 1) && ![](allBalls == 2) && []true }

/* proctypes */

/* comment your code only if necessary,
   explaining what each proctype is supposed to be
*/
proctype ... {
...
}

...

init { /* if needed */
...
}

/* monitor processes for invariants (optional) */
```

```
/* explain what this does if you use it */
active proctype myPropertyMonitor() {
 ...
}

/*

REPORT

Your report goes here if required.

*/
```

Mandatory Games

Game 1: PingPong.pml (Lab 4 warm-up - Part 1)

Two players play a game of indefinitely passing a single ball back and forth. A third person, the referee, kickstarts the game by passing the ball to the first player. A ball is **in the game** (the game is on) if it's in mid-air or with one of the players (if it's with the referee, it's not in the game). *In this game, when you count the balls: update the number of balls upon when a player both catches a ball and throws a ball, and consider explicitly how many balls are in mid-air between players.*

- **noDeadlock** (safety check, invalid end states): The game should never stop.
- **`singleBallForever`** (LTL - liveness check): Eventually (once the game starts), the players forever have a single ball to pass around.

Game 2: BrokenPingPong.pml (Lab 4 deliverable - Part 2)

Any player can drop the ball (you have to simulate a player dropping the ball). If the ball is dropped, the referee eventually picks it up and passes it back to the first player to re-start the game. *In this game, you may choose to update the number of balls only when they are reintroduced into the game by the referee and when they are dropped by one of the players.* You may use **timeout** in this model.

- **noDeadlock** (safety check, invalid end states): The game should never stop.
- **`alwaysAtMostOneBall`** (LTL - liveness check): Always, the players have at most one ball to pass around.

Game 3: Jugglers.pml (Assignment 2 warm-up - Part 1)

The referee has two balls. The game gets started as usual with the referee picking up one of the balls and passing it to the first player. At one point after that, she picks up the second ball, and passes it to the first player. The players will then have to juggle one or two balls between them (but they need both hands to catch and throw a ball). They may drop a ball from time to time unexpectedly. When that happens, the referee picks the dropped ball up and passes it back to the first player. Do not use **timeout** in this model. Your model must be properly and effectively parameterized to avoid a deduction.

- **noDeadlock** (safety check, invalid end states): The game should never stop.
- **`alwaysEventuallyGameOn`** (LTL - liveness check): The game always eventually starts with at least one ball in the game.
- **`alwaysAtMostTwoBalls`** (LTL - liveness check): The number of balls the players are juggling between them may never exceed two.

- **`alwaysEventuallyTwoBalls`** (LTL - liveness check, add *weak fairness**): If eventually the game is on and stays on (at least one ball is in the game), the players will always eventually have two balls to juggle between them. This property has a precondition.
    - Make sure that you also verify the possibility of the precondition: it is **possible** that eventually both balls are never dropped. This is <u>not</u> expressible as an LTL formula, but its negation is. Express the negation as an LTL formula **`notEventuallyAlwaysGameOn_FAIL`** and make sure that it fails (thus giving a counterexample that proves the possibility of the precondition).
      /* _FAIL suffix is required to tell the verification script that this property is supposed to fail. */
- **possiblyAllBallsAreDropped** (never claim - liveness check)**:** It is possible that always eventually both balls are dropped.
    - This property is <u>not</u> expressible as an LTL formula, but its negation is. Express the negation as an LTL formula **`notPossiblyAllBallsAreDropped_FAIL`** and verify that it fails (thus giving a counterexample, which proves the original property).
      /* _FAIL suffix is required to tell the verification script that this property is supposed to fail. */

**Weak fairness*: Turning on weak fairness guarantees that when a transition is enabled indefinitely at a given global state, it is eventually executed. This allows a model to exit a possible cycle in those cases where such a global state (one with a constantly enabled transition leading outside the cycle) exists in a cycle. Refer to Appendix in the CheckingProperties slide deck for more information.

Game 4: JugglersBW.pml (Assignment 2 deliverable - Part 2)

One ball is WHITE and the other ball is BLACK. Do <u>not</u> use timeout in this model. Your model must be properly and effectively parameterized to avoid a deduction.

- All the properties of **Juggler.pml,** plus...
- **`alwaysDifferentColors`** (LTL - liveness check): If two balls are in play, they must be of different colors. That is, if the players are passing around two balls, the balls must be of different colors.

Optional Exercises

// No verifier config file is available for these games //

Game 5: SuperJugglers.pml

The referee has a supply of MAX_BALLS. The game is started in the usual way. The referee may pass a new ball to the first player any time, until the number of balls in the game reaches MAX_BALLS. Each ball is of a different color. The players must now be able to juggle multiple balls, keeping track of them. They may drop a ball from time to time.

- The number of balls the players are juggling between them may never exceed MAX_BALLS.
- The game should never stop.
- If eventually no balls are ever dropped, the players will eventually have MAX_BALLS to juggle between them.
- It is possible that eventually all balls are dropped.
- All balls in the game are always of different color.

Game 6: RoundRobin.pml

There can be more than two players in the game, but there is a single ball. The number of players is fixed. The maximum number of players in any game is MAX_PLAYERS. The players stand in a circle. The referee passes the ball to the first player as usual. Each player receives the ball from the previous player and passes it to the next player in the circle. Thus the game proceeds only in one direction. The last player in the circle passes the ball back to the first player in the circle. Any player can drop a ball. The referee then picks the ball up and restarts the game.

- The game should never stop.
- Always, the players have at most one ball to pass around.
- Eventually the game is on.

Game 7: SuperRoundRobin.pml

The referee in RoundRobin now has a supply of MAX_BALLS. Combine RoundRobin with SuperJugglers for this game.

- The game should never stop.
- The number of balls the players are passing around the circle may never exceed MAX_BALLS.
- Eventually the game is on.
- If eventually no balls are ever dropped, the players will eventually have MAX_BALLS to juggle among them.
- It is possible that eventually all balls are dropped.

L4 / A2 Submission and Grading

No report is necessary for L4. Just submit your pml file to Vocareum. Ignore the instructions below.

~~You must only finish Part 1 and Part 2 - **BrokenPingPong** for your report. Complete the report template below, paste it in your Promela model, and submit your **pml** file to Vocareum. Remove the instructions starting with "//" in report. See vocareum submission instructions.~~

> ~~Lab 4 Report Template~~
> ~~---~~
> ~~1.1. Answer to Part 1.1 (.5 pt)~~
> ~~// Describe your calculation.~~
> ~~---~~
> ~~1.2. Answer to Part 1.2 (.5 pt)~~
> ~~// Describe your calculation.~~

L4/A2 deductions:

Complexity/Style

- Overly/unnecessarily complex model: -1 or -2 depending severity
- Unnecessary guards (e.g., checking length of channel before receive): -1
- Unnecessary variables and/or unnecessary use of them in guards: -1
- Poor readability/format/Promela style: -1 or -2 (see Promela Coding Style in spin-public repo)
- Debugging code left in the model: -1
- Poor understandability (cryptic constructions): -1
- Model too verbose (not parameterized enough): -2 (permissible only in L4 warmup, otherwise Player processes and counts should be parameterized/indexed)
- Ineffective parameterization (parameterized, but pointless and ends up making the model more complicated, less understandable): -1
- Model not loyal to spec: see below, minor deviations get -1

Property Interface

- Property interface violation: -1 (no partial credit due to syntax error for violating property interface)
- Macro definition wrong (missing variables, counting wrong channels, not counting balls in channels, etc.): -1

LTL Properties

- Trivially/Improperly passing properties: -1 per property
- Wrong/missing properties: -1 per property if failing, -2 if passing (L4)
- Wrong/missing properties: 0 per property if failing, -1 if passing (A2)

Unsafe/Improper Behavior

- Improper use of atomic/d_step causing too much serialization (atomic/d_step only permitted for observation variables/counts used in properties, when they are attached to a send/receive operation): -1 or -2 depending on severity (big mistake!)
- Using observation variables as guards in Player processes): -2 (big mistake!)
- Using observation variables as guards to control behavior in Referee or init process representing the referee): -1 (smaller mistake)

- Wrong/improper use of channels (rendez-vous, channels of capacity 1, bi-directional channels, too many channels, e.g., separate channels for white/black balls): -1
- Wrong/improper update of observation variables (see description on how balls should be counted): -1 or -2 depending of severity
- Observation variables missing: -2 (big mistake)
- Use of timeout in L4-Warmup (Part 1) or in any part of A2 (timeout permitted only in L4-Deliverable, Part 2): -2

MIsunderstandings

- Misuse/misunderstanding of Promela constructs (e.g., atomic with a single statement, if with a single choice, if within a do-od having a single choice): -1
- Model deviating from problem, not loyal to description: -1, -2 or -3 depending on severity (no penalty if already covered above under another deduction)

L4 Properties to be defined by student in their model

- Part 1 - PingPong
    - alwaysAtMostOneBall
- Part 2
    - singleBallForever

A2 Properties to be defined by student in their model

- Part 1 - Jugglers
    - alwaysEventuallyGameOn
    - alwaysAtMostTwoBalls
    - alwaysEventuallyTwoBalls
    - notEventuallyAlwaysGameOn_FAIL
    - notPossiblyAllBallsAreDropped_FAIL
- Part 2 - JugglersBW
    - alwaysEventuallyGameOn
    - alwaysAtMostTwoBalls
    - alwaysEventuallyTwoBalls
    - notEventuallyAlwaysGameOn_FAIL
    - notPossiblyAllBallsAreDropped_FAIL
    - alwaysDifferentColors

Resources

Spin and SpinRCP (instructions updated Oct 2020)

Installing Spin and SpinRCP on Mac

(Linux and Windows - check spin and SpinRCP installation instructions at respective web sites):

FIRST INSTALL Spin AND SpinRCP DEPENDENCIES:

- Download latest spin distribution (**latest version is 6.5.1**)
    - Goto https://github.com/nimble-code/Spin/tree/master/Bin and download the correct binary for the latest version for your platform OR build Spin using the installation instructions at http://spinroot.com/ for your platform
    - In you downloaded the binary, unzip/untar file and move the Spin executable (*spin*) to a desired location, preferably to */usr/local/bin* (on MacOS and Linux)
    - On MacOs: easiest way is to install homebrew and run "*brew install spin*' (or upgrade to the latest version by '*brew upgrade spin*' if Spin is already installed)
    - MacOS and Linux: To test where the Spin binary was installed (and whether it was installed in a folder where your OS looks for binaries), try '*which spin*'
- Test that binary works with '*spin -V*' (capital *V*) -- version printed should be the latest one indicated above in bold
- Install the remaining dependencies (*dot* library from the *graphviz* package) by following the instructions at http://lms.uni-mb.si/spinrcp/Download.html
    - MacOS: if you have homebrew, simply do '*brew install graphviz*' to install *graphviz dot* library required by SpinRCP (or to update to latest version do '*brew upgrade graphviz*')
    - MacOS and Linux: To find out where dot was installed (and whether it was installed in a folder where your OS looks for binaries), try *'which dot*'
- You also need
    - The C compiler *gcc* (which should already be installed) for spin to work, and
    - Java 8 (see below) for SpinRCP to run -- this is important!

THEN INSTALL SpinRCP:

- Follow the instructions to install the latest SpinRCP release SpinRCP3.1.1 at http://lms.uni-mb.si/spinrcp/Download.html
    - Follow the instructions carefully - there are many steps
    - You need Java 8 for SpinRCP to work. Make sure you install the proper version of Java 8 first by following the instructions provided at the top portion of Download page
    - Once all the dependencies are installed, download the latest SpinRCP executable for your platform using the yellow buttons [Linux], [Windows], [macOS] in the middle part of the page
    - Scroll down past the yellow [Linux], [Windows], [macOS] download buttons, and follow the instructions for your platform

- Go to http://lms.uni-mb.si/spinrcp/Instructions.html, and follow the instructions to configure launch and configure SpinRCP
- MacOS and Linux: You need to set the paths for *gcc*, *spin*, and *dot* (Step 2 on the Instructions page). Try *'which gcc*', '*which spin*', and '*which dot*' to determine the path names for these binaries.

IF YOU ARE RUNNING MacOSX BIG SUR

- You may get an alert "Failed to create the Java Virtual Machine" when launching the SpinRCP.
  - Edit the */Applications/SpinRCP3.1.1/SpinRCP.app/Contents/Info.plist* file to use your local JVM.
  - A valid configuration file looks something like following:

```
<array>
   <string>-vm</string>
      <string>/Library/Java/JavaVirtualMachines/jdk1.8.0_221.jdk/Co
ntents/Home/bin/java</string>
   <string>-keyring</string>
    <string>~/.eclipse_keyring</string>
   <string>-showlocation</string>
</array>
```

Alternative to SpinRCP (especially for Windows users)

You can always run spin from the command line, but it may get tedious. An alternative GUI for Spin is iSpin, which is included in Spin distribution. Check Spin documentation at spinroot.com. Some Windows users found iSpin easier to install and work than SpinRCP. SpinRCP runs well on Mac OS and Linux, but we haven't tested it on Windows. I do not have iSpin and will use SpinRCP for all examples.

Spin and SpinRCP Known Bugs

- spin (all versions)
  - Model and `pan.c` variable clashes: When a variable is defined in a model (in the def-use sense) only, but never used (i.e., in a guard, conditional, assert, LTL, or on the LHS of an assignment; only appearing on the RHS of an assignment is not a use, it's another def), it doesn't affect the behavior or verification. Spin then doesn't take any extra precautions to treat it properly. But if it detects it has a real purpose, it stashes it in a different scope. So useless variables may end up freaking out spin when compiling the verifier C file `pan.c`, resulting in errors that look like: `./pan.h:463:30: note: previous definition is here`
  - One variable name that may cause this is `done`, used in Tiny Models, which is also a variable used in the generated verifier `pan.c`.

    - Fix: change variable name or make sure the var is used in an assert or LTL inside the model.

Runspin & Parsepan

Two useful scripts by Theo Ruys, to help with setting up (and remembering) configurations for multiple verification runs (the *verify* script that you will use in labs, assignments, and Vocareum submissions takes advantage of these two tools):

- runspin: this is a bash script to automate the verification of Promela models using the model checker Spin. runspin automates the complete verification of a Promela model. Apart from verifying the Promela model, runspin adds valuable extra information to pan's verification report.
- parsepan: this is a Python script to parse the results of SPIN's pan verifier, and to selectively retrieve information from verification reports.
- More info on runspin and parsepan: Check Canvas > Modules > Supplements & Readings

---