CS4052 Logic and Software Verification

Sizhe Yuen

October 7, 2018

Contents

A system model can be simulated with formal methods. Violations in the model will give counterexamples of the error in the software logic. There are a few typical properties we want to check for.

- Is the generated result correct (value within bounds)?
- Does the system terminate correctly?
- Can the system reach a **deadlock** situation?
- Can a deadlock occur at a certain point in the system?
- Is a response always received after some time?

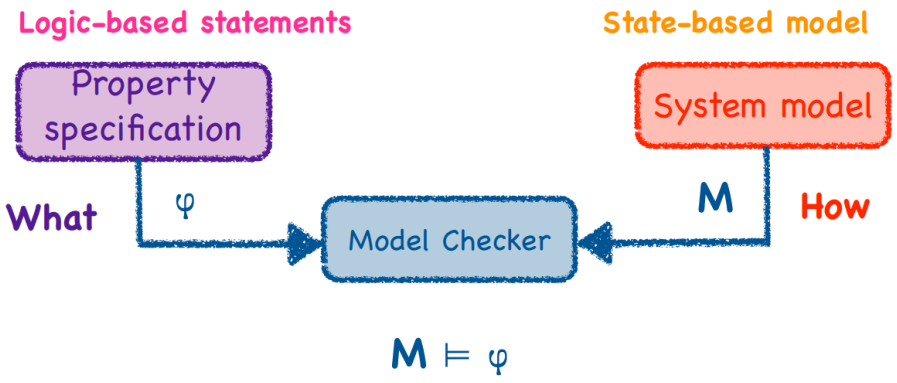

Figure 1: High level idea of a what a model checker does.

The model checker examines all relevant states in M to check if ϕ is satisfied. If a state is found where ϕ is violated, the model checker provides a counterexample as to how that state can be reached. The counterexample describes an execution path that leads from the initial state to the state that violets ϕ.

2.2 Concurrency and Atomicity

Most errors in models are classic **concurrency errors**. This is where unforeseen interleavings between processes caused issues such as deadlock or changing state when the model isn't supposed to. Wrong assumptions are often made as to what is executed atomically.

```
1 || proc Inc = while true do if x < 200 then x += 1
2 || proc Dec = while true do if x > 0 then x -= 1
3 || proc Reset = while true do if x == 200 then x = 0
```

Listing 1: Concurrency example in pseudocode

In the code above, x is a shared variable and all three processes depend on the value of x. Because the instructions between conditional of the `if` and the `then` are not atomic, it is possible for x to become negative when the condition of `Reset` is interleaved with the `Dec` process.

2.3 Temporal Logic

Temporal logic is an extension of basic propositional logic using **temporal operators** that describe the behaviour of the system over time. This is useful for describing different system properties:

- **Functional correctness** - Does the system behave as expected?
- **Reachability** - Is it possible to reach a deadlock state?
- **Safety** - Something bad never happens (e.g. Two processes both in the critical section)
- **Liveness** - Something good will eventually happen (e.g. Printer process eventually prints)
- **Fairness** - Under certain conditions, can an event occur repeatedly?
- **Real time properties** - Is the system acting in time?

A classic issue for model checkers if if the model is too large to be handled. This results in a **state-space explosion** where the number of states needed to model the system accurately easily exceeds the amount of available computer memory. Different techniques must be applied to exploit regularities in the structure of the model to prevent this problem.

3 PROMELA and SPIN

3.1 PROMELA

PROMELA (PROcess MEta LAnguage) is a specification language used for model checking with a tool called **Spin**. It contains the notion of *communication channels* as it was originally developed for specification of communication protocols.

PROMELA can model concurrent systems made of processes which can:

- Behave **independently** (interleaving)
- Communicate indirectly using **shared global variables**
- Communicate directly using **channels** (synchronously or asynchronously)

Here is an example of a simple PROMELA specification:

```
#define N 20

int res

proctype odd() {
    int x = 1;
    do
    :: x <= N -> res = res + x;
                  x = x + 2;
    :: x > N -> break;
    od
}

proctype even() {
    int x = 0;
    do
    :: x <= N -> res = res + x;
                  x = x + 2;
    :: x > N -> break;
    od
}

init {
    res = 0;
    run odd();
    run even();
}
```

Listing 2: Simple PROMELA specification

More than one alternative in a `do` loop can be given and denoted by `::`. An alternative can only be executed if its guard passes. If there are multiple branches that pass, one is chosen non-deterministically.

Boolean conditions in the middle of code blocks until it becomes true.

```
x < 4;
x > 6;
res = 20;
```

Listing 3: Boolean conditions in PROMELA

Here, the specification must waited until `x` is smaller than 4, then wait until `x` is greater than 5 before proceeding to assign `res`.

It is possible to have an `else` guard in a do loop.

```
1 active proctype ifdo() {
2     do
3     :: x < 10 -> x = x + 1;
4     :: (x > 0 && x < 10) -> x = x - 1;
5     :: x < 9 -> x = x + 3;
6     :: else -> x = x + 4;
7                break;
8     od
9 }
```

Listing 4: PROMELA Do loops in detail

An `else` in a `do` block is only executed if all the other guards fail. Additionally, the `break` statement is the only way to leave a `do` loop in PROMELA.

3.1.1 Communication channels

In PROMELA, two processes can communicate directly using channels.

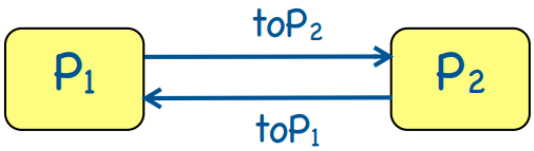

Figure 2: PROMELA channels.

3.2 Linear-Time Properties

3.2.1 Safety Properties

Safety properties refer to all states in the system.

Another safety property that can be checked with spin is whether at the end, all processes have reached a valid end state. A process terminates correctly if it reaches the closing bracket of its `proctype` declaration.

3.2.2 Liveness Properties

Liveness properties describe the requirement that a process makes progress towards a certain goal, the achievement of which depends on the fairness of the system. They are about something which should eventually happen. For example:

- **Eventually** - Each process will eventually enter its critical section.
- **Repeated eventually** - Each process will enter its critical section infinitely often.
- **Starvation freedom** - Each waiting process will eventually enter its critical section.

Liveness properties need to be checked for all possible system runs/executions. It is important to check for liveness both with and without fairness as they can tell us if the system is unfair or not by design.

Programs in spin can be annotated with labels such as:

$$\text{progress}[a-zA-Z0-9]*$$

These labels are examples of a liveness property where all infinite execution paths have to pass a progress label infinitely often. There is an option in spin to tick the box for "non-progress cycles" before running the verification.

3.2.3 Fairness

A **fairness constraint** is imposed on the system that it fairly selects the next process to be executed. These constraints are not a property to be verified, but they can be expressed in temporal logic to verify behaviour of a system with and without fairness constraints. Fairness assumptions are there to remove what can be considered unrealistic behaviours as it may be unrealistic for only `P1` to be executing all the time.

There are different types of fairness:

- **Unconditional fairness** - Every process gets its turn infinitely often. This ignores that a process might not be ready to execute sometimes.
- **Strong fairness** - Every process that is infinitely often enabled should be executed infinitely often.
- **Weak fairness** - Every process that is always enabled from a certain point onwards should be executed infinitely often.

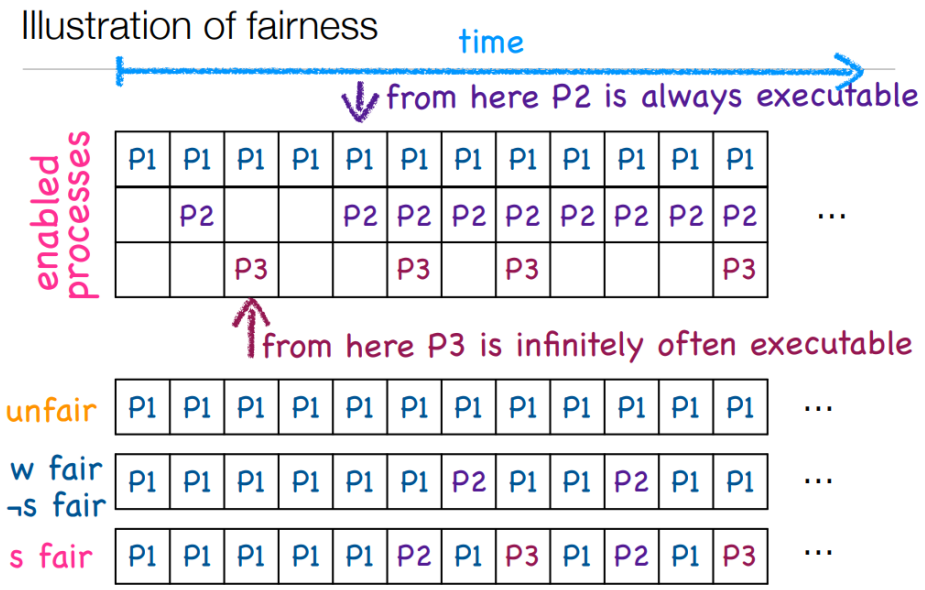

Figure 3: Illustration of different types of fairness.

3.3 Trace constraints

For some systems, we may want to reflect on the **patterns of communication** used. It is possible to impose sequences of communication using a **trace**. Traces have to be valid on all executions. Trace conditions are used to impose an order on the allowed sequences of communications and so **synchronises** with the specification.

```
1 mtype = {a, b};
2
3 trace {
4     do
5     :: c1!a;
6        c2?b;
7     od
8 }
```

Listing 5: Example of a trace in PROMELA

In the example above, sending on channel `c1` alternates with receiving on channel `c2`. All messages sent on `c1` are of type `a`, and all messages received

on `c2` are of type `b`. The trace here synchronises with the specified processes.

Trace specifications are only concerned about *communication actions* and must be deterministic. As such, each specification can only contain one trace constraint.

3.4 Never claims

To check the *correctness* of a specification against certain properties, it is useful to use a **never claim**. With a never claim, we are interested in the executions that **do not** satisfy the property. A never claim *succeeds* (finds an error) if it terminates or it passes infinitely often through `accept` labels. It will always be executed first after global variables are initialised and before any other processes.

In essence, disallowed behaviour is specified and run against the specification to check that it cannot occur. Additionally, a never claim *is interleaved with the specification*. Never claims are always executed first.

The issue with never claims is that it can be time consuming to construct them. They involve searching and thinking about general counter examples. An easier way would be to do the opposite and specify properties that can be used to verify that the system behaves correctly. This is **linear temporal logic** (LTL).

4 Linear Temporal Logic

To express properties in LTL, **atomic propositions** must first be defined:

```
1 #define p (state == ok)
2 #define q (alaram != red)
```

Listing 6: Atomic propositions in PROMELA for LTL

LTL formulae are constructed from atomic propositions combined with *boolean* and *temporal* operators. To be able to describe a property holding on a **state**, only propositional logic is needed. However, to describe properties about **execution** traces, temporal operators are needed.

Given p and q are valid LTL formulae, there are the following temporal operators:

1. $\Box\ p$ - Always p
2. $\Diamond\ p$ - Eventually p
3. $X\ p$ - Next p
4. $p\ U\ q$ - p Until q

Formulae can be combined with normal boolean operators such as $\wedge(and), \vee(or)$.

There are also two forms for until:

- **Strong until** - q must hold eventually in the execution
- **Weak until** - q might never hold in the execution, in which case if p holds for the whole execution, weak until still passes.

4.1 Modelling concurrent systems

4.1.1 Transition systems

A transition system is a standard class of models that are used to represent both hardware and software systems. They are used to describe the behaviour of systems. In essence, a transition system is a directed graph where **nodes** represent states and **edges** represent transitions.

Atomic propositions are simple known facts about the states of the system. These are things such as "$x == 0$" or "$x < 20$".

Formal definition

$$TS = (S, Act, T, I, AP, L) \tag{1}$$

where

$$\begin{aligned} S &= \text{set of states} \\ Act &= \text{set of actions} \\ T \subseteq S \times Act \times S &= \text{transition relation} \\ I \subseteq S &= \text{set of initial states} \\ AP &= \text{set of atomic propositions} \\ L : S \to 2^{AP} &= \text{labelling function} \end{aligned}$$

A transition system is called finite if S, Act and AP are finite.

The labelling function is the essential link for proving properties in a transition system.

$$s \models \phi \text{ iff } L(s) \models \phi \tag{2}$$

Execution fragment

A **finite execution fragment** of a transition system is an alternating sequence of states and actions ending with a state. An execution fragment can also be infinite.

$$\rho = a \xrightarrow{\text{getb}} b \xrightarrow{\tau} c \xrightarrow{\text{getj}} d$$

Figure 5: Example execution fragment.

A **maximal execution fragment** is either finite and ends in a terminal state, or it is infinite. An **initial execution fragment** starts in an initial state.

An *execution* of a transition system TS is an initial, maximal execution fragment. A state s is called reachable if there is some execution fragment that ends in s and starts in some initial state.

4.1.2 Data Dependent systems

Often times executable actions come from conditional branching in systems. To be able to deal with conditionals, conditional transitions can be added to a transition system in the form of a guard. Labels in conditional transitions have the form:

$$g : \alpha$$

where g is a boolean guard and α is an action that occurs if g holds.

However, a graph containing conditional transitions is **not** a transition system. To get a transition system from such a graph, it has to be unfolded and additional information added to the state based on the resources it needs. This is where we introduce the concept of program graphs.

4.1.3 Program graphs

A program graph over a set `Var` of typed variables is defined as follows:

$$PG = (Loc, Act, \text{Effect}, C_\dagger, Loc_0, g_0) \tag{3}$$

where

$$\begin{aligned} Loc &= \text{set of locations} \\ Act &= \text{set of actions} \\ \text{Effect}: \ Act \times Eval(var) \rightarrow \ Eval(var) &= \text{effect function} \\ C_\dagger \subseteq Loc \times Cond(var) \times Act \times Loc &= \text{conditional transition relation} \\ Loc_0 \subseteq Loc &= \text{set of initial locations} \\ g_0 \in Cond(Var) &= \text{initial condition} \end{aligned}$$

The **Effect** indicates how the evaluation η of variables is changed by performing an action. For example, if there was a vending machine that spits out beer or juice depending on the action, and we want to keep track of the number of beers and juices in the machine, the effects may look like this:

$$\begin{aligned} \text{Effect}(\text{getJuice}, \eta)(\text{juiceCount}) &= \eta(\text{juiceCount}) - 1 \\ \text{Effect}(\text{getJuice}, \eta)(\text{beerCount}) &= \eta(\text{beerCount}) \\ \text{Effect}(\text{getBeer}, \eta)(\text{juiceCount}) &= \eta(\text{juiceCount}) \\ \text{Effect}(\text{getBeer}, \eta)(\text{beerCount}) &= \eta(\text{beerCount}) - 1 \end{aligned}$$

The transition system of a program graph - $TS(PG)$ - is the tuple (S, ACt, T, I, AP, L).

5 Timed automata and UPPAAL

5.1 CTL

6 Petri nets