CS4052 Logic and Software Verification

Sizhe Yuen

December 15, 2017

Contents

1 Promela

1.1 Processes

Listing 1: Process definition

```
proctype P1(int id) {
    int n = id;
    do
    :: n < N -> n = n + 1;
    :: else -> break;
    od
}

active [N] proctype Pi() {
    //process is active on start and there are N processes
}

init {
    run P1(0);
}
```

More than one alternative can be given in a loop denoted by `::`. If more than one alternative is possible, choice is non-deterministic.

1.2 Channels

Listing 2: Channel definition

```
chan toP1 = [0] of {byte}; //synchronous channel
chan toP2 = [N] of {byte}; //asynchronous channel with buffer size N

toP1!x; //send value of x down channel toP1
toP1?x; //receive message from channel toP1 into variable x
```

Synchronous channels block until the message is read while asynchronous channels block only when the buffer is full.

There are a number of functions that can be applied to a channel:

- `len(c)` - Number of messages in `c`
- `empty(c)` - Is the channel empty?
- `nempty(c)` - Is the channel not empty?
- `full(c)` - Is the channel full?
- `nfull(c)` - Is the channel not full?

There are also a number of variations of possible received messages:

- `c?x,y` - Received values saved to `x` and `y`
- `c?2,y` - First sent value has to be 2
- `c?eval(x),y` - First sent value must have the value of x
- `c?<x,y>` - Received values saved to x and y, but message is kept in buffer
- `c??2,y` - Get the first message in the buffer which has a first value of 2 (if none, keep waiting)
- `c??<2,y>` - As above, but keep in buffer rather than read to variable y

Additional checks for asynchronous channels:

- `c?[x,y]` - Checks whether the message receipt is possible
- `c?[2,y]` - Is `c?2,y` possible next?
- `c??[2,y]` - Is `c??2,y` possible?

Note that we can also write `c?_` if we do not care about the value being sent.

1.3 Invariant

An invariant is an example of a verifiable safety property which has to be true in all system states. In spin, assertions can be used as follows:

Listing 3: PROMELA assertion

```
assert(<Boolean condition>)

active proctype Invariant() {
    byte y = 0;
    bool b = false;
    assert( !b || y > 42);
}
```

Note the assertion above is only executed once. Labels can be used to block until a process reaches that label. For example:

Listing 4: PROMELA labels

```
active [3] proctype P() {
m1: do
    :: x < 10 -> m2: x = x + 1;
    :: x > 5 -> m3: break;
    od
}

active proctype Inv() {
    P[0]m2;P[1]m3;
    P[2]m3;assert(x <= 11)
```

1.4 Trace

It is possible to impose specific sequences of communication using a **trace**.

Listing 5: Example of a trace

```
mtype = {a,b};
trace {
    do
    :: c1!a; c2?b;
    od
}
```

This above example states that *sendong on channel* `c1` *alternates with receiving on channel* `c2`.

2 LTL

2.1 Linear Time Properties

2.1.1 Safety

Safety properties are about *nothing bad should happen.* An example of this is the **mutual exclusion property** - Always at most one process is in its critical section.

Safety properties refer to all states in the system.

2.1.2 Liveness

Liveness properties are about something good will happen eventually. They can be used to guarantee that progress is made.

- **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 executions.

2.1.3 Fairness

Unconditional fairness

Every process gets its turn infinitely often.

Strong fairness

Every process that is **enabled** infinitely often gets its turn infinitely often.

Weak fairness

Every process that is continuously enabled from a certain point onwards gets its turn infinitely often.

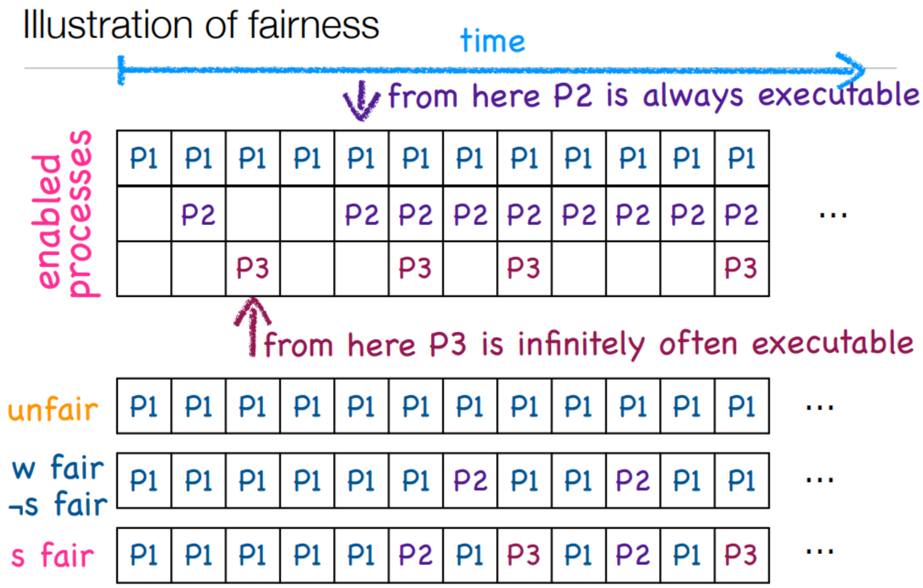

Figure 1: Illustration of fairness

Spin supports weak fairness and can support strong fairness with an LTL statement.

2.2 LTL

2.2.1 Syntax

Given valid formulae p, q and r:

- $\Box p$ - Always p
- $\Diamond p$ - Eventually p
- $\bigcirc p$ - Next p
- $p\ U\ q$ - p until q

There is a difference between strong and weak until.

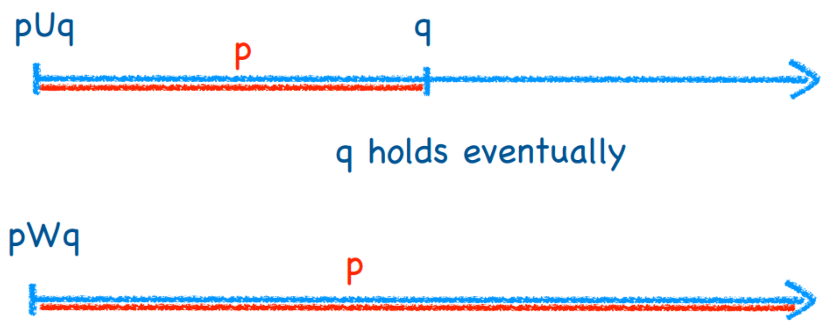

Figure 2: Strong and weak until

2.2.2 Typical LTL Properties

- **Invariant** $\Box p$
- **Reply** $p \to \Diamond q$
- **Guaranteed reply** $\Box(p \to \Diamond q)$
- **Progress** $\Box\Diamond p$ Infinitely often p
- **Stability** $\Diamond\Box p$ Eventually forever p
- **Exclusion** $\Box(p \to\neq q)$

2.2.3 LTL Properties for mutual exclusion

- P_1 and P_2 never simultaneously have access to their critical sections:

$$\Box(\neg \mathrm{crit}_1 \vee \neg \mathrm{crit}_2) \tag{1}$$

- Each process is infinitely often in its critical section:

$$(\Box\Diamond \mathrm{crit}_1) \wedge (\Box\Diamond \mathrm{crit}_2) \tag{2}$$

- Every waiting process will eventually enter its critical section:

$$(\Box\Diamond \mathrm{wait}_1 \to \Box\Diamond \mathrm{crit}_1) \wedge (\Box\Diamond \mathrm{wait}_2 \to \Box\Diamond \mathrm{crit}_2) \tag{3}$$

- Whenever the semaphore y has the value 0, one of the processes is in its critical section:

$$\Box(y = 0 \to \mathrm{crit}_1 \vee \mathrm{crit}_2) \tag{4}$$

2.2.4 Fairness in LTL

Three types of fairness constraints:

- Unconditional $\square\lozenge\text{crit}_i$
- Strong fairness $\square\lozenge\text{wait}_i \rightarrow \square\lozenge\text{crit}_i$
- Weak fairness $\lozenge\square\text{wait}_i \rightarrow \square\lozenge\text{crit}_i$

2.3 Transition System

$$TS = (S, Act, T, I, AP, L) \tag{5}$$

where

S	= set of states
Act	= set of actions
T⊆S ×Act ×S	= transition relation
I⊆S	= set of initial states
AP	= set of atomic propositions
L:S →2^{AP}	= labelling function

2.3.1 Deterministic observable behaviour

Action based: deterministic on the executed observable actions. *At most one outgoing transition labelled with action αper state*

State based: ignore actions and reply on APs that hold in the current state. *At most one outgoing transition from a state with label a to a state with label a*

2.3.2 Execution fragment

Let $TS = (S, Act, T, I, AP, L)$
A finite execution of fragment ρ of TS is an alternating sequence of states and actions ending with a state:

$$\rho = s_0\ \alpha_1\ s_1\ \alpha_2\ s_2\ ...\ \alpha_n\ s_n \tag{6}$$

such that $(s_i, \alpha_{i+1}, s_{i+1}) \in T$ forall $0 \leq i < n$ where $n \geq 0$.

- An **execution fragment** ρ of TS can also be infinite.
- A **maximal execution fragment** is either finite ending in a terminal state, or infinite.
- An **initial execution fragment** starts in an initial state.

A state $s \in S$ is **reachable** in TS if there exists an initial, finite execution fragment:

$$\rho = s_0\ \alpha_1\ s_1\ \alpha_2\ s_2\ ...\ \alpha_n\ s_n = s$$

with $s_0 \in I$ and $n \geq 0$

2.4 Program Graphs

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

$$PG = (Loc, Act, \text{Effect}, C_\dagger, Loc_0, g_0) \tag{7}$$

where

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

The **Effect** indicates how the evaluation η of variables is changed by performing an action.

2.4.1 Transition System for a Program Graph

The $TS(PG)$ of a $PG = (Loc, Act, \text{Effect}, C_\dagger, Loc_0, g_0)$ over Var is the following tuple:

$$TS(PG) = (S, Act, T, I, AP, L) \tag{8}$$

where

$$
\begin{aligned}
&\mathrm{S} && = Loc \times Eval(Var) \\
&\mathrm{T} \subseteq \mathrm{S} \times \mathrm{Act} \times \mathrm{S} && = \frac{l_1 \xrightarrow{g:\alpha} l_2 \land \eta \vDash g}{\langle l_1, \eta \rangle \xrightarrow{\alpha} \langle l_2, \mathrm{Effect}(\alpha, \eta) \rangle} \\
&\mathrm{I} && = \{ \langle l, \eta \rangle \mid l \in Loc_0 \land \eta \vDash g_0 \} \\
&\mathrm{AP} && = Loc \cup Cond(Var) \\
&L(\langle l, \eta \rangle) && = \{ l \} \cup \{ g \in Cond(Var) \mid \eta \vDash g \}
\end{aligned}
$$

2.5 Parallel Composition of Transition Systems

$$TS = TS_1 \parallel TS_2 \parallel ... \parallel TS_n \tag{9}$$

$\parallel$ is the parallel composition operator and is usually **commutative** and **associative** but depends on the kind of communication supported.

$|||$ is the **interleaving** operator where the actions from different processes are interleaved and the system is made of a set of independent processes (no communication).

The interleaving operator for transition systems simply constructs the **Cartesian product** of the individual state spaces without considering the potential conflicts from *shared* variables. For programs with shared variables, we define interleaving directly on the program graph level: $PG_1 \; ||| \; PG_2$.

2.5.1 Composed Program Graph $PG_1 \; ||| \; PG_2$

- Local variables of PG_1 are $x_1 \in Var_1 \setminus Var_2$
- Local variables of PG_2 are $x_2 \in Var_2 \setminus Var_1$
- **Global** variables are $x \in Var_1 \cap Var_2$

Actions that access global variables are **critical** and critical actions *cannot be executed simultaneously.*

2.5.2 Handshaking $TS_1 \|_H TS_2$

Handshaking allows for processes to interact at the same time through synchronous communication (shared actions). The composition of two transition systems handshaking on actions H is as follows:

$$TS_1 \|_H TS_2 = (S_1 \times S_2, Act_1 \cup Act_2, T, I_1 \times I_2, AP_1 \cup AP_2, L) \tag{10}$$

where $H \subseteq Act_1 \cap Act_2$.

Transitions for synchronisation means the actions in H are taken *synchronously* by both processes:

$$\frac{(s_1 \xrightarrow{\alpha}_1 s_1') \wedge (s_2 \xrightarrow{\alpha}_2 s_2')}{\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle s_1', s_2' \rangle} \tag{11}$$

Note that if there is no synchronisation of actions, it is the same as interleaving:

$$TS_1 \|_\emptyset TS_2 = TS_1 \; ||| \; TS_2 \tag{12}$$

3 Timed Automata

Timed automata add clock variables to the program graph. A clock constraint over a set of C of clocks is formed as follows:

$$g ::= x < c | x \leq c | x > c | x \geq c | g \wedge g \tag{13}$$

where c is a natural number and $x \in C$.

3.1 Definition

A Timed Automata is the tuple $TA = (Loc, Act, C, C_t, Loc_0, Inv, AP, L$ where:

$$\begin{aligned} C &= \text{Finite set of clock} \\ Inv &= Loc \rightarrow CC(C) \end{aligned}$$

3.2 Handshaking with timed automata

Synchronisation

$$\frac{l_1 \xrightarrow{g_1:\alpha, D_1}_1 l_1' \wedge l_2 \xrightarrow{g_2:\alpha, D_2}_2 l_2'}{\langle l_1, l_2 \rangle \xrightarrow{g_1 \wedge g_2:\alpha, D_1 \cup D_2} \langle l_1', l_2' \rangle} \tag{14}$$

Interleaving

$$\frac{l_1 \xrightarrow{g:\alpha,D}_1 l_1'}{\langle l_1, l_2\rangle \xrightarrow{g:\alpha,D} \langle l_1', l_2\rangle} \text{ or } \frac{l_2 \xrightarrow{g:\alpha,D}_2 l_2'}{\langle l_1, l_2\rangle \xrightarrow{g:\alpha,D} \langle l_1, l_2'\rangle} \tag{15}$$

3.3 CTL Syntax

State formula of a set of AP are formed according to the following grammar:

$$\Phi ::= \mathit{true} \mid a \mid \Phi \wedge \Phi \mid \neg\Phi \mid \exists\phi \mid \forall\phi \tag{16}$$

where ϕ is a path formula:

$$\phi ::= \bigcirc\Phi \mid \Phi \cup \Phi \tag{17}$$

3.3.1 CTL properties

- There exists an execution along which p will eventually hold:

$$\exists\Diamond p \tag{18}$$

- There exists an execution along which p is always true:

$$\exists\Box p \tag{19}$$

- Necessarily p will eventually hold

$$\forall\Diamond p \tag{20}$$

- p is always true

$$\forall\Box p \tag{21}$$

3.3.2 CTL examples

- P_1 and P_2 are never simultaneously in their critical sections:

$$\forall\Box(\neg\text{crit}_1 \vee \neg\text{crit}_2) \tag{22}$$

- Each process is infinitely often in its critical section:

$$(\forall\Box\forall\Diamond\text{crit}_1) \wedge (\forall\Box\forall\Diamond\text{crit}_2) \tag{23}$$

3.3.3 Derivations for eventually and always

Eventually

$$\exists\Diamond\Phi = \exists(true \cup \Phi)$$
$$\forall\Diamond\Phi = \forall(true \cup \Phi)$$

Always

$$\exists\Box\Phi = \neg\forall\Diamond\neg\Phi$$
$$\forall\Box\Phi = \neg\exists\Diamond\neg\Phi$$

3.4 Timed CTL (TCTL)

Eventually

$$\exists\Diamond^J\Phi = \exists(true \cup^J \Phi)$$
$$\forall\Diamond^J\Phi = \forall(true \cup^J \Phi)$$

Always

$$\exists\Box^J\Phi = \neg\forall\Diamond^J\neg\Phi$$
$$\forall\Box^J\Phi = \neg\exists\Diamond^J\neg\Phi$$

J denotes the time interval, for example:

$$\Diamond^{\leq 2} \text{ denotes } \Diamond^{[0,2]}$$
$$\Diamond^{>8} \text{ denotes } \Diamond^{[8,\infty[}$$
$$\Diamond \text{ denotes } \Diamond^{[0,\infty[}$$

3.4.1 Definition

There exists a path in which Φ holds during interval J:

$$\exists\Box^J\Phi \tag{24}$$

In all paths Φ must hold during interval J:

$$\forall\Box^J\Phi \tag{25}$$

3.4.2 Example TCTL Properties

The light cannot be continuously on for more than 2 time units:

$$\forall\Box(on \rightarrow \forall\Diamond^{\leq 2}\neg on) \tag{26}$$

The light will stay on for at least 1 time unit and then switch off:

$$\forall\Box(on \rightarrow (\forall\Box^{\leq 1} on \wedge \forall\Diamond^{>1}\text{off})) \tag{27}$$

The gate is always closed when the train is at the crossing:

$$\forall\Box(crossing \rightarrow closed) \tag{28}$$

Once the train is far, within one minute the gate is up for at least 1 minute:

$$\forall\Box(\text{far} \rightarrow \forall\Diamond^{\leq 1}\forall\Box^{\leq 1} up) \tag{29}$$

3.5 UPPAAL

4 Petri Nets

4.1 Definition

A Petri net is a tuple $N = (P, T, G)$ where:

- P is a set of **places**
- T is a set of **transitions**
- G is a directed graph linking places and transitions

4.2 Reachability

Let M ba a marking for a Petri net $N = (P, T, G)$:

$$\text{Reachable}(N, M) = \{M' \mid M' \text{ is a marking for } N\} \tag{30}$$

such that there is a sequence of enabled transitions $t_i \in T$ with $0 \leq i \leq n$ which leads to M'

The set can be finite and infinite and a marking for which there is no enabled transition indicates a deadlock.