Overview

- How processes, system models and properties are represented in SPIN.
- How LTL properties are verified within SPIN.
• A **process** is given meaning via an **automata**, *i.e.* a finite-state transition system:
  - set of states (unique initial state);
  - set of state-to-state transitions based upon input stimuli.

• $^*_{\text{-}}$-Automata: the conventional notion of automata where there exists explicit initial and final states, *i.e.* recognizes finite sequence of stimuli. Acceptance corresponds to final state.

• $\omega$-Automata: an automata which contains an explicit initial state but no final state *i.e.* recognizes an infinite sequence of stimuli (reactive systems). Acceptance requires a different criteria ... (more on slide 5).

• A **system model** is given meaning via an **asynchronous interleaving product of automata**.

---

```plaintext
bit x=0;
proctype A(){
do :: (x==0) -> x++
od}  
proctype B(){
do :: (x==1) -> x--
od}  
init {atomic{
          run A();
          run B()}  
}
```
LTL Formula via Buchi Automata

- As mentioned above, we are interested in finite state models which give rise to infinite executions.
- A Buchi Automata provides one way of expressing acceptance properties for infinite executions.
- Acceptance for a Buchi automata means that there exists a state which is visited infinitely often.
- Any LTL formula can be expressed as a Buchi automata.

LTL Verification via Buchi Automata

- To verify that model $M$ satisfies LTL formula $F$ generate:
  - $P$ the asynchronous interleaving product for model $M$.
  - $B$ the Buchi automata corresponding to the negation of $F$.
  - $S$ the synchronous product of $B$ and $P$.
- If $S$ contains an acceptance cycle then a counter-example to $F$ exists.
- Note that in a synchronous product automata each transition denotes a joint transition of the component transitions.
- The synchronous product allows one to check whether or not a model exhibits a particular LTL property as expressed via a Buchi automata.
A Simple Safety Example

\[
\text{bit } x=0; \\
\text{proctype } A()\{ \\
\quad \text{do} \\
\quad \quad :: (x==0) \rightarrow x++ \\
\quad \quad \text{od} \\
\} \\
\text{proctype } B()\{ \\
\quad \text{do} \\
\quad \quad :: (x==1) \rightarrow x-- \\
\quad \quad \text{od} \\
\} \\
\text{init } \{ \text{atomic} \{ \text{run } A(); \text{run } B() \} \}
\]

Will the above model satisfy the following safety invariant?

\[ \Box (x == 0 \lor x == 1) \]
byte x=0;
proctype A(){
    do
        :: true -> x++
    od
}
proctype B(){
    do
        :: (x==1) -> x--
    od
}
init {atomic{ run A(); run B()}}

Will the above model satisfy the following safety invariant?

\[
[] (x == 0 \text{ or } x == 1)
\]
Always Eventually

\[ [] \leftrightarrow P \]

Eventually Always

\[ <> [] P \]
Buchi Automata via Promela

- Buchi automata are represented within SPIN via a special process, known as a **never claim**.
- A **never claim** is used to represent a property that should **never** be satisfied during the execution of a model.
- SPIN automatically interleaves the execution of a **never claim** along with the given Promela model.
- SPIN is looking to see if the execution of the **never claim** matches with the execution of the Promela model. A match corresponds to either:
  - an **acceptance cycle** being detected within the **never claim**
  - **termination** of the **never claim** (complete match)

```
never { /* []<> p */
T0_init:
  if
    ::((p)) -> goto accept_S9
    ::(1) -> goto T0_init
  fi;
accept_S9:
  if
    ::(1) -> goto T0_init
  fi;
}
```
Generating Never Claims within SPIN

```plaintext
never { /* []<> p */
T0_init:
  if
    :: ((p)) -> goto accept_S9
    :: (1) -> goto T0_init
  fi;
accept_S9:
  if
    :: (1) -> goto T0_init
  fi;
}
```

Given a LTL formula, the LTL property manager (XSPIN) displays the generated never claim. Using SPIN one can also directly generate a never claim for an arbitrary LTL formula, e.g. the above never claim was generated by the following command line:

```
spin -f '[]<>p'
```
Proving LTL Properties via Never Claims

- It is easier to prove that a model \textbf{does not satisfy} a property than it is to prove that it does, \textit{i.e.} it only takes one counter-example to shown that a property is not satisfied.

- A never claim is therefore typically used to represent the \textbf{negation} of the formula (property) of interest.

- To prove $F$, a never claim is generated for $\neg F$ – the \textbf{negation} of $F$. SPIN then checks the model against $\neg F$:
  - If an \textbf{acceptance cycle} is detected then $\neg F$ is satisfied and a counter-example exists for $F$.
  - If \textbf{no acceptance cycle} is detected then $\neg F$ is not satisfied, and therefore $F$ is satisfied by the model.

Safety Property via Never Claim

```plaintext
never { /* ![]p */
T0_init:
  if
    :: (! ((p))) -> goto accept_all
  :: (1) -> goto T0_init
fi;
accept_all:
  skip
}
```
Response Property via Never Claim

```plaintext
never { /* ![] (p -> <>q) */
T0_init:
  if
  :: (! ((q)) && (p)) -> goto accept_S4
  :: (1) -> goto T0_init
  fi;
accept_S4:
  if
  :: (! ((q))) -> goto accept_S4
  fi;
}
```

Precedence Property via Never Claim

```plaintext
never { /* ![](p -> r U q) */
T0_init:
  if
  :: (! ((q)) && (p)) -> goto accept_S4
  :: (! ((q)) && ! ((r)) && (p)) -> goto accept_all
  :: (1) -> goto T0_init
  fi;
accept_S4:
  if
  :: (! ((q))) -> goto accept_S4
  :: (! ((q)) && ! ((r))) -> goto accept_all
  fi;
accept_all:
  skip
}
```
Summary

Learning outcomes:

- To understand and be able to describe how processes and system models are represented in SPIN.
- To understand and be able to convert between a Buchi automata and an equivalent LTL formula.
- To understand and be able to convert between LTL formulas and never claims.
- To be able to explain LTL reasoning within SPIN at the level of Buchi automata and never claims.

Recommended reading: