Recap: Model Checking for model-based testing

From test spec.

LTL Property System Model

Model Checking

No, with Counter-example trace

Generated Test

Encoding test specifications

Def. 1
A trace \( \sigma \) satisfies a test specification \( M \) if \( \sigma \) contains at least one linearization of \( M \) as a contiguous subsequence.

Given MSC \( M \),
- define \( \text{Lin}(M) \) = set of linearizations of \( M \).
- For each linearization \( \sigma = e_1 e_2 \ldots e_n \), define
  - \( \text{prop}_\sigma = \bigwedge (\text{prop}_1 \land X(\text{prop}_2 \land X(\ldots (\text{prop}_{n-1} \land X(\text{prop}_n)))) \)
- Define property \( \phi_M \) corresponding to \( M \) as
  - \( \phi_M = \neg \bigvee_{\sigma \in \text{Lin}(M)} \text{prop}_\sigma \)

A counter-example to \( \phi_M \) is a test satisfying \( M \).

Encoding test specifications

Def. 2
A trace \( \sigma \) satisfies a test specification \( M \) if \( \sigma \) contains at least one linearization of \( M \) as a subsequence.

Given MSC \( M \),
- define \( \text{Lin}(M) \) = set of linearizations of \( M \).
- For each linearization \( \sigma = e_1 e_2 \ldots e_n \), define
  - \( \text{prop}_\sigma = \bigwedge (e_1 \lor e_2 \lor \ldots \lor e_n) \)
- Define prop, \( \phi_M = \neg \bigvee_{\sigma \in \text{Lin}(M)} \text{prop}_\sigma \)

A counter-example to \( \phi_M \) is a test satisfying \( M \).
LTL Model Checking – does M |= ϕ
1. Consider ¬ϕ. None of the exec. traces of M should satisfy ¬ϕ.
2. Construct a finite-state automata A¬ϕ such that
   - Language(A¬ϕ) = Traces satisfying ¬ϕ
3. Construct the synch product M × A¬ϕ.
4. Check whether any exec trace σ of M is an exec trace of the product M × A¬ϕ, i.e., check Language(M × A¬ϕ) = empty-set?
   - Yes: Violation of ϕ found, report counterexample σ
   - No: Property ϕ holds for all exec traces of M.

Recap: finite-state automata
- A = (Q, ∑, Q0, →, F)
  - Q is a finite set of states
  - ∑ is a finite alphabet
  - Q0 ⊆ Q is the set of initial states
  - → ⊆ Q × ∑ × Q is the transition relation
  - F ⊆ Q is the set of final states.
- What is the Language of such an automaton?

Regular languages:
- Accept any finite-length string σ ∈ ∑*, which ends in a final state.
ω-regular languages:
- Accept any infinite-length string σ ∈ ∑ω which visits a final state infinitely many times.

Set of strings accepted = Language of the automata.

Finite automata
- Meaning as a regular language
  - (a+b)*b+
    - All finite length strings ending with b
- Meaning as an ω-regular language
  - All infinite length strings with finitely many a

LTL properties to automata
- Given a LTL property p
  - we want to convert p to an automata A_p, s.t.
    - Language(A_p) = strings / traces satisfying p
- LTL properties are checked over infinite traces.
  - Given an infinite trace σ and a LTL property p, we can check whether σ |= p
  - To convert LTL properties to finite-state automata, consider automata accepting inf-length traces.
    - Language(A_p) is ω-regular, not regular.

LTL properties to automata
- Given a LTL property ϕ
  - We convert it to a ω-regular automata A_ϕ
    - Language(A_ϕ) = {σ ∈ ∑ω | σ |= ϕ}
    - Language(A_ϕ) is defined as per the ω-regular notion of string acceptance, it accepts inf-length strings.
    - All infinite length strings satisfying ϕ form the language of A_ϕ
    - Whether an infinite length string satisfies ϕ (or not) is defined as per LTL semantics.
Example: LTL property to automata

Recall: LTL Model Checking

Example: Verify GFp

Product Automata

Product Automata Construction

4/6/2011
Recall: LTL Model Checking

1. Consider $\neg \phi$. None of the exec. traces of $M$ should satisfy $\neg \phi$.
2. Construct a finite-state automata $A_{\neg \phi}$ such that
   - $\text{Language}(A_{\neg \phi}) = \text{Traces satisfying } \neg \phi$
3. Construct the synch product $M \times A_{\neg \phi}$
4. Check whether any exec trace $\sigma$ of $M$ is an exec trace of the product $M \times A_{\neg \phi}$ i.e. check $\text{Language}(M \times A_{\neg \phi}) = \text{empty-set}$
   - Yes: Violation of $\phi$ found, report counterexample $\sigma$
   - No: Property $\phi$ holds for all exec traces of $M$.

Emptiness Check

- $\text{Language}(M \times A_{\neg \phi}) = \text{empty-set}$?
- Is there any trace which visits one of the accepting states of the product automata infinitely many times?
- Look for accepting cycles.

Emptiness Check

- Perform DFS from initial state until you reach an accepting state $s_{\text{acc}}$.
- When you reach $s_{\text{acc}}$, remember $s_{\text{acc}}$ in a global var, and start a nested DFS from $s_{\text{acc}}$.
- Stop the nested DFS if you can reach $s_{\text{acc}}$.
- If no accepting cycles are found, report yes.
- If accepting cycles are found
  - Concatenate the two DFS stacks and report it as counter-example trace of the LTL property.
  - This algo. is implemented in SPIN model checker.

Nested DFS – step 1

- procedure dfs1(s)
  - push s to Stack1
  - add {s} to States1
  - if accepting(s) then
    - States2 := empty; seed = s; dfs2(s)
  - endif
  - for each transition s → s' do
    - if s' ≠ States1 then dfs1(s')
  - endfor
  - pop s from Stack1
- end

Nested DFS – step 2

- procedure dfs2(s)
  - push s to Stack2
  - add {s} to States2
  - for each transition s → s' do
    - if s' = seed then report acceptance cycle
    - else if s' ≠ States2 then dfs2(s')
  - endif
  - endif
  - pop s from Stack2
- end
Organization

- So Far
  - What is a Model?
  - ATC – Running Example
  - How to model such requirements
  - How to validate the models
    - Simulations,
    - Model-based testing,
    - Model Checking
    - Model Checkers
      - SPIN

**Organization

- So Far
  - What is a Model?
  - ATC – Running Example
  - How to model such requirements
  - How to validate the models
    - Simulations,
    - Model-based testing,
    - Model Checking
    - Model Checkers
      - SPIN

**SPIN**

- A tool for modeling complex concurrent and distributed systems.
- Provides:
  - Promela, a protocol meta language
  - A model checker
  - A random simulator for system simulation
  - Promela models can be automatically generated from a safe subset of C.

**Our Usage**

- Learn Promela, a low-level modeling language.
- Use it to model simple concurrent system protocols and interactions.
- Gain experience in verifying such concurrent software using the SPIN model checker.
- Gives a feel (at a small scale)
  - What are hard-to-find errors?
  - How to find the bug in the code, once model checking has produced a counter-example?

**Features of Promela**

- Concurrency
  - Multiple processes in a system description.
- Asynchronous Composition
  - At any point one of the processes active.
  - Interleaving semantics
- Communication
  - Shared variables
  - Message passing
    - Handshake (synchronous message passing)
    - Buffers (asynchronous message passing)

**Features of Promela**

- Within a process
  - Non-determinism: supports the situation where all details of a process may not be captured in Promela model.
- Standard C-like syntax
  - Assignment
  - Switch statement
  - While loop
  - Guarded command
    - Guard and body may not evaluated together, that is, atomically.
Example

```c
byte state = 0;
proctype A()
{
  byte tmp;
  (state==0) -> tmp = state;
  tmp = tmp+1;
  state = tmp;
}
init { run A(); run A(); }
```

We need to define how processes are scheduled.

SPIN's process scheduling

- All processes execute concurrently
- Interleaving semantics
  - At each time step, only one of the “active” processes will execute (non-deterministic choice here)
  - A process is active, if it has been created, and its “next” statement is not blocked.
  - Each statement in each process executed atomically.
  - Within the chosen process, if several statements are enabled, one of them executed non-deterministically.
  - "We have not seen such an example yet!"

Will this loop terminate?

```c
byte count;
proctype counter()
{
  do
  :: count = count + 1
  :: count = count - 1
  :: (count == 0) -> break
  od;
}
```

Non-determinism within a single process.

Enumerate the reasons for non-termination in this example

This loop will not terminate

```c
active proctype TrafficLightController() {
  byte color = green;
  do
    :: (color == green) -> color = yellow;
    :: (color == yellow) -> color = red;
    :: (color == red) -> color = green;
    od;
}
```

channels

```
chan data, ack = [2] of bit;
```

Array of channels also possible.

```
Example with channels
```

```
chan data, ack = [1] of bit;
proctype node1() {
  do
    :: data!1;
    :: ack!1;
  od
}
proctype node2() {
  do
    :: data?1;
    :: ack?1;
  od
}
init { atomic{
    run node1(); run node2();
  }
}
```

Talking to different processes via dedicated channels.
Example with channels

```
chan data, ack = [1] of bit;

proctype node1() {
    do
        :: data!1;
        :: ack?1;
    od
}

proctype node2() {
    do
        :: ack!1;
        :: data?1;
    od
}

init { atomic{
    run node1();
    run node2();
}
}
```

SPIN Execution Semantics

- Select an enabled transition of any thread, and execute it.
- A transition corresponds to one statement in a thread.
- Handshakes must be executed together.

```
chan x = [0] of {...};
x!1             ||   x?data
```

SPIN Execution Engine

```
while ( (E = executable(s))  != {}) {
    for some (p,t) ∈ E {
        s' = apply(t.effect, s);
        /* execute the chosen statement */
        if (handshake == 0) {
            s = s' ;
            p.curstate = t.target
        }
        else {  ...
        }
    }
    handshake = 0
} /*  for some (p, t) ∈ E */
```

Model Checking in SPIN

- \((P_1 || P_2 || P_3) \models \phi\)
- \(P_1, P_2, P_3\) are Promela processes
- \(\phi\) is a LTL formula
- Construct a state machine via
  - \(M\), asynchronous composition of processes \(P_1, P_2, P_3\)
  - \(A_{\neg \phi}\), representing \(\neg \phi\)
- Show that "language" of \(M \times A_{\neg \phi}\) is empty
- No accepting cycles.
- All these steps have been studied by us !!

Specifying properties in SPIN

- Invariants
  - Local: via assert statement insertion
  - Global: assert statement in a monitor process
- Deadlocks
- Arbitrary Temporal Properties (entered by user)
  - SPIN is a LTL model checker.
  - LTL properties can be entered as input to the checker!
  - Shown in the lab hour of the last lecture!
Connect system & property in SPIN

- **System model**
  - `int x = 100;
  - active proc type A()
  - { do
  - :: x %2 -> x = 3^x+1
  - od
  - }
  - active proc type B()
  - { do
  - :: 1(x%2) -> x = x/2
  - od
  - }

- **Property**
  - `GF (x = 1)`
  - Insert into code
  - `#define q (x == 1)`
  - Now try to verify GF q

More Involved Example

- **Alternating Bit Protocol**
  - Reliable channel communication between sender and receiver.
  - Exchanging msg and ack.
  - Channels are lossy
  - Attach a bit with each msg/ack.
  - Proceed with next message if the received bit matches your expectation.

ABP Architecture

Implemented as SPIN processes

Sender & Receiver code

- `chan datachan = [2] of { bit };`

Sender()

```
active proc type Sender()
{ bit out, in;
  do
  :: datachan!out ->
  active proc type Receiver()
  { bit in;
    do
      :: datachan?in -> ackchan!in
      :: timeout -> ackchan!in
    od
  }
}
```

Receiver()

```
active proc type Receiver()
{ bit in;
  do
    :: datachan?in -> ackchan!in
    :: timeout -> ackchan!in
  od
}
```

Timeouts

- Special feature of the language
  - Time independent feature.
  - Do not specify a time as if you are programming.
  - True if and only if there are no executable statements in any of the currently active processes.
  - True modeling of deadlocks in concurrent systems (and the resultant recovery).

Model Checking in SPIN

- SPIN performs model checking by Nested DFS
  - Discussed in the past lecture !!
  - Find acceptance states reachable from initial states (DFS).
  - Find all such acceptance states which are reachable from itself (DFS).
  - Counter-example evidence (if any) obtained by simply concatenating the two DFS stacks.
Exercise – Model Checking – (1)

Two Computer Engineering students are taking the CS4271 exam. We must ensure that they cannot leave the exam hall at the same time. To prevent this, each student reads a shared token before leaving the hall. The shared token is an arbitrary natural number. The global state of the system is given by $s_1, s_2, n$ where $s_1$ and $s_2$ are the local states of students 1 and 2 respectively. Note that $s_1 \in \{\text{in, out}\}$ and $s_2 \in \{\text{in, out}\}$.

The pseudo-code executed by the two students are:

```plaintext
\begin{align*}
\text{do forever}{ }&\text{if } s_1 = \text{in and } n \text{ is odd } \\
&\{ s_1 := \text{out} \} \\
\text{else if } s_1 = \text{out} \& \& \text{do nothing} \\
\text{else } &\text{do nothing} \\
\end{align*}
\begin{align*}
\text{do forever}{ }&\text{if } s_2 = \text{in and } n \text{ is even } \\
&\{ s_2 := \text{out} \} \\
\text{else if } s_2 = \text{out and } n \text{ is even} \& \& \text{do nothing} \\
\text{else } &\text{do nothing} \\
\end{align*}
```

The two student processes are executed asynchronously. Every time one process is scheduled, it atomically executes one iteration of its loop. The above system is an infinite state system. Design a finite state abstraction and draw the global automata for the abstracted system. Your abstraction should be refined enough to prove mutual exclusion. Initially $s_1 = \text{in}$ and $s_2 = \text{in}$.

---

Exercise – Model Checking – (2)

Consider the mutual exclusion property.

Using the LTL model checking algorithm discussed in class, follow a step by step process to check the correctness of the property on the example given in the previous slide.