An Automatic Approach to Verifying UML State Machines

ZHANG Shaojie, LIU Yang
National University of Singapore
Agenda

• Introduction
• Our Approach
• Case Study
• Conclusion & Future Work
Introduction

• Unified Modeling Language (UML) is the de facto standard modeling language for designing and architecting software systems.

• UML model consists of a set of diagrams that together describes the single system.
  - Specification
  - Visualization
  - Architecture design
  - Construction
  - Simulation and Testing
  - Documentation

• Take advantage of formal methods to verify the correctness of UML models
Introduction – Model Checking Principle
Introduction -- Our Big Picture
Introduction

- Present a translation approach to verifying UML state machines.
  - Fully automatic
  - Independent of any modeling tools
- Compared with other work
- Verification tool: PAT
Introduction

• Present a translation approach to verifying UML state machines.

• Compared with other work
  ➢ **Support a larger subset of UML state machines**
    • Esp. advanced modeling constructs
  ➢ **State space representation efficiency**
    • Minimize the use of shared variables

• Verification tool: PAT
Introduction

• Present a translation approach to verifying UML state machines.

• Compared with other work
  ➢ Support a larger subset of UML state machines
    • Esp. advanced modeling constructs
  ➢ State space representation efficiency
    • Minimize the use of shared variables

• Verification tool: PAT
  ➢ Various simulation mechanisms
  ➢ Deadlock, reachability, trace refinement relationship, linear temporal logic properties with various fairness assumptions.
Agenda

- Introduction
- Our Approach
- Case Study
- Conclusion & Future Work
Our Approach – Modeling language

- CSP#
  - CSP + shared variables + low-level programming constructs

- Programs as events
  - event{program} -> Process

- Process constructs
  - Stop, Skip
  - ch!exp -> P, ch?x -> P
  - P; Q
  - P [] Q, P <> Q, if (b) {P} else {Q}
  - P || Q, P ||| Q
  - P |> Q
  - P = Q
Our Approach – Translation Rules

\[ f : UML \rightarrow CSP\# \]
Our Approach -- UML State Machines

• A state machine describes the lifetime of a single object.
• It contains states and transitions between them.
• Simple, composite and submachine states
Our Approach – Basic Rules

- UML Event → CSP# Event
- UML State → CSP# Process
Our Approach – State

• Three optional behaviors of a state:
  - **Entry** → *Atomic Process*
  - **DoActivity** → *Interrupt Process*
  - **Exit** → *Atomic Process*

\[
f(\text{state}) = f(\text{entry}); \quad f(\text{doActivity}) \quad \Delta \quad (f(\text{trans}_1) \square f(\text{trans}_2) \square \cdots \square f(\text{trans}_N))\]
Our Approach – State

- Composite state
  - Region
  - Process
  - Regions
  - Interleaving (or Parallel) Processes

\[
f(\text{composite}) = \\
f(\text{entry}); \\
(f(\text{doActivity}) || f(r1) || f(r1)|| | | | ... ) \\
\Delta \\
(f(\text{trans}_1) \Box f(\text{trans}_2) \Box ... \Box f(\text{trans}_N))
\]
Our Approach – State

- Submachine state
  - Specifies the insertion of the specification of a state machine

\[ f(\text{ReadAmount}) = f(\text{ReadAmount\text{SM}}) \]
Our Approach – Transition

• A transition has five parts.
  - Source state
  - Target state
  - Event trigger
  - Guard condition
  - Effect

\[ f(\text{trans}) = \text{event} \rightarrow [\text{guard}] f(\text{exit}); f(\text{effect}); f(\text{targetState}) \]
Our Approach – State Machine

• State machine

\[ f(sm) = f(i) \]

Where \( i \) is the topmost initial state of \( sm \).

• System

\[ f(s) = f(sm_1) ||| f(sm_2) ||| \cdots ||| f(sm_n) \]
Advanced Modeling Behavior

- Fork
- Join
- Entry/Exit point
- History
Advanced Modeling Behavior – Fork

- Fork state deals with the transition from a single source state to several substates in different regions of a composite state.

- When a transition from a fork state is fired, control passes to all the target states.
Advanced Modeling Behavior – Fork

\[ P_S(i, j, k)^2 = \text{enter a state} \rightarrow (P_r_1(i) \parallel P_r_2(j) \parallel P_r_3(k)); \]

\[ P_{Fork} = P_S(2, 0, 1); \]
Advanced Modeling Behavior – Join

- Join state specifies the transition from substates in different regions of a composite state to a target state outside the composite state.
- A join transition is effective only if all the source states are active.
Advanced Modeling Behavior – Join

\[ P_{S(i,j,k)} = entryS0 \rightarrow (P_{r1}(i) \parallel P_{r2}(j)) \parallel P_{r3}(k)) \]
\[ P_{s2} = (e2 \rightarrow exitS2 \rightarrow Skip) \square \\
\quad (join \rightarrow exitS2 \rightarrow exitS0 \rightarrow P_{join}) \]
\[ P_{s4} = join \rightarrow exitS0 \rightarrow P_{join} \]
\[ P_{s5} = (e5 \rightarrow exitS5 \rightarrow Skip) \square \\
\quad (join \rightarrow exitS5 \rightarrow exitS0 \rightarrow P_{join}) \]
\[ P_{join} = e6 \rightarrow P_{S6} \]
Advanced Modeling Behavior – Entry/Exit point

- Entry/exit point is the entry/exit point of a state machine referred by a submachine state.
- Behaviorally analogous to a subroutine
Advanced Modeling Behavior – Entry/Exit point

\[ P_{S1} = e1 \rightarrow ch!0 \rightarrow Skip; \]
\[ P_{S2} = ch?1 \rightarrow P_{S3}; \]
\[ P_{SM2} = ch?0 \rightarrow starting \rightarrow P_{S4}; \]
\[ P_{S4} = abort \rightarrow ch!1 \rightarrow Skip; \]
Advanced Modeling Behavior – History State

• History state adds “memory "to composite state by recording the last substate that was active prior to a transition from the composite state.

• An integer shared variable is used to record which substate is currently active.
Agenda

- Introduction
- Our Approach
- Case Study
- Conclusion & Future Work
Case Study – A CD Player

CDPLAYER() = NONPLAYING(0);
Case Study – A CD Player

```
NONPLAYING(i) =
case{
(i == 0) : CLOSED()
(i == 1) : OPEN()
} \triangleq
(play -\rightarrow ([\neg]) present] NONPLAYING(0)))
□(play -\rightarrow ([present] BUSY(0)))
□(off -\rightarrow Skip));

CLOSED() = load
→ open{track = 0; present = false}
→ OPEN();

OPEN() = load
→ close{track = 1; present = true}
→ CLOSED();
```
Case Study – A CD Player

BUSY(i) =
find track start →
case {i == 0 : PLAYING()}
   (i == 1) : PAUSED()} \(\Delta\)
((load \rightarrow\) NONPLAYING(1))
   \(\Box\) (track! = N)
   (\{track = track + 1\} \rightarrow BUSY(0)))
   \(\Box\) (track == N) NONPLAYING(0))
   \(\Box\) (stop \rightarrow\) NONPLAYING(0))
   \(\Box\) (off \rightarrow\) Skip)
   \(\Box\) (play \rightarrow BUSY(j)));

PLAYING() = \{i = 0; \} \rightarrow (\text{play track} \rightarrow \text{Skip}) \(\Delta\)(\{pause \rightarrow PAUSED()\} \(\Box\) Skip);

PAUSED() = \{i = 1; \} \rightarrow \text{pause} \rightarrow\) PLAYING();
Case Study

- Variable Declaration
  - var present = false, track = 0, j = 0;
Case Study
Agenda

• Introduction
• Our Approach
• Case Study
• Conclusion & Future Work
Conclusion

• Present an automatic translation-based approach to verifying UML state machines
  - Defined a translation scheme from a UML model composed of hierarchical state machines to CSP#.
  - Effectively handle advanced modeling techniques in state machines.
Future work

• Look for large industrial cases
• Support deferred events and time events
• Provide an easier way to specify properties
The End

Thank you for your kind attention!

Q & A