An Automatic Approach to Model Checking UML State Machines

ZHANG Shaojie, LIU Yang
National University of Singapore

The Fourth IEEE International Conference on Secure Software Integration and Reliability Improvement

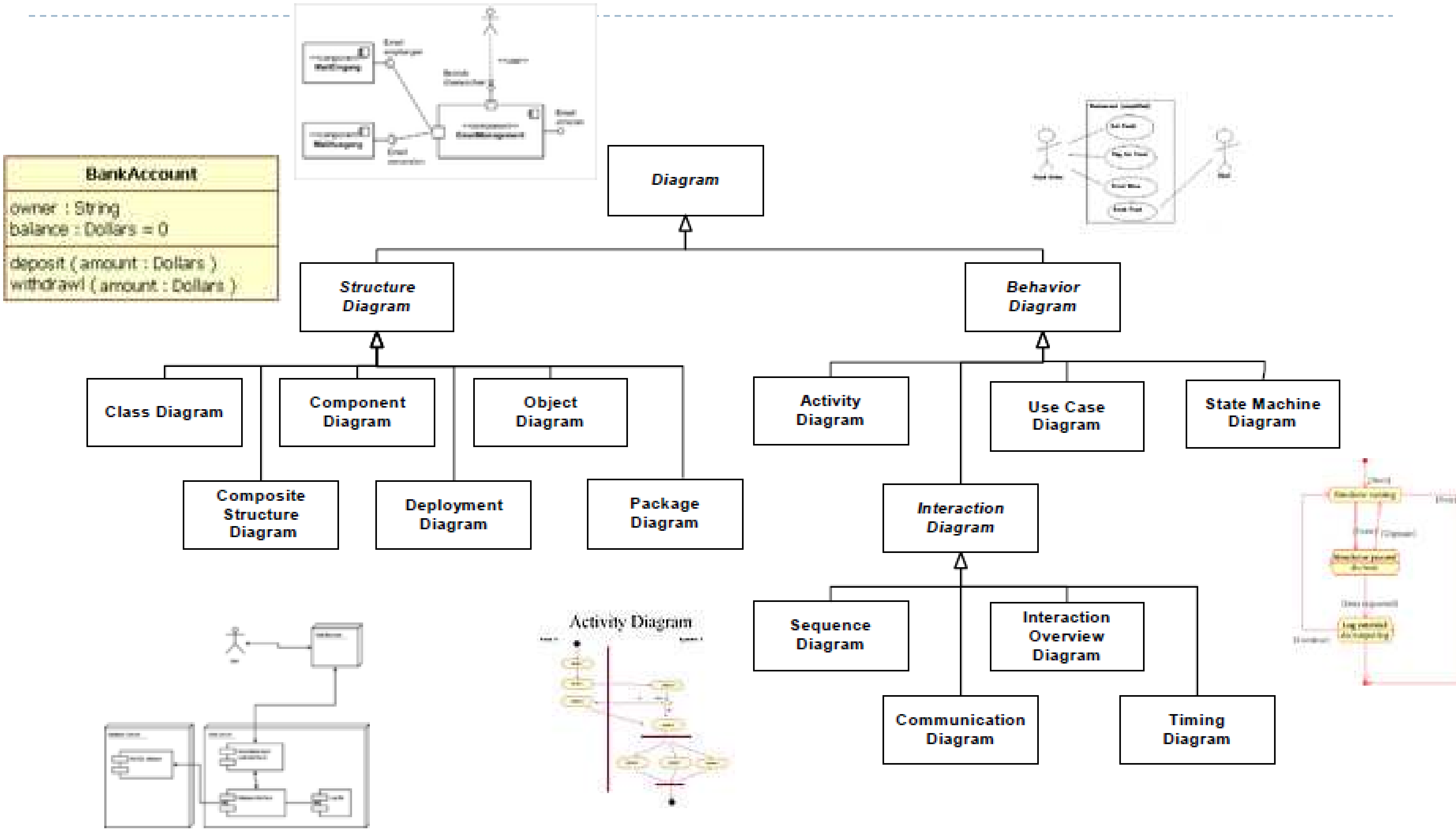
Agenda

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

Introduction

- Unified Modeling Language (UML) is de facto standard 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

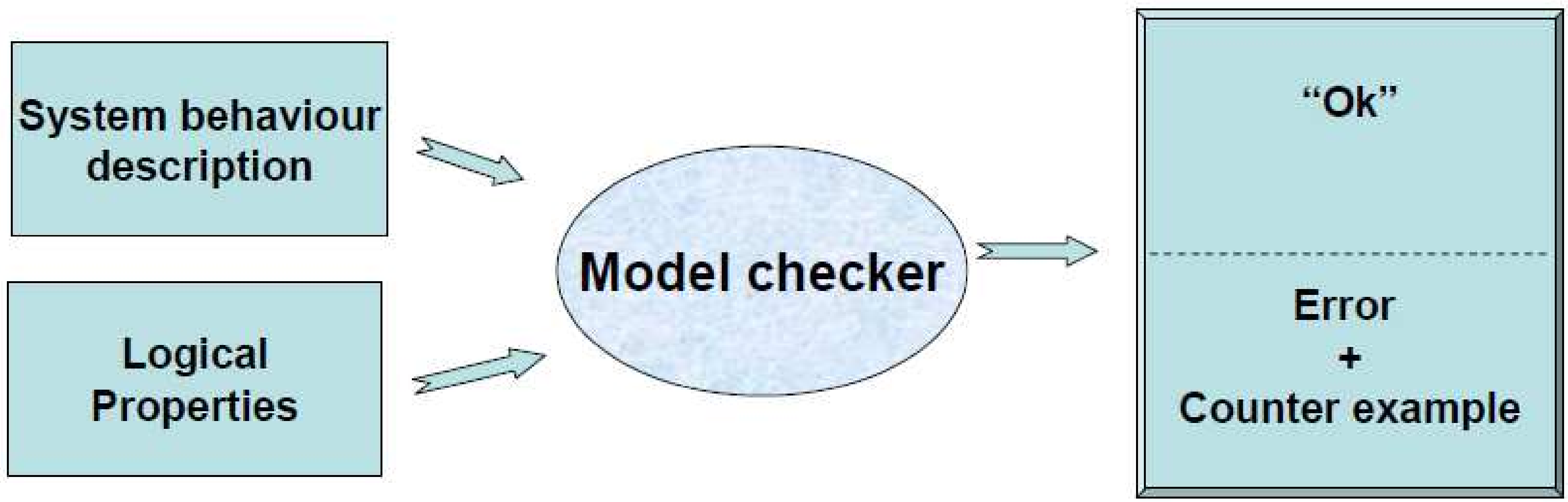
Introduction



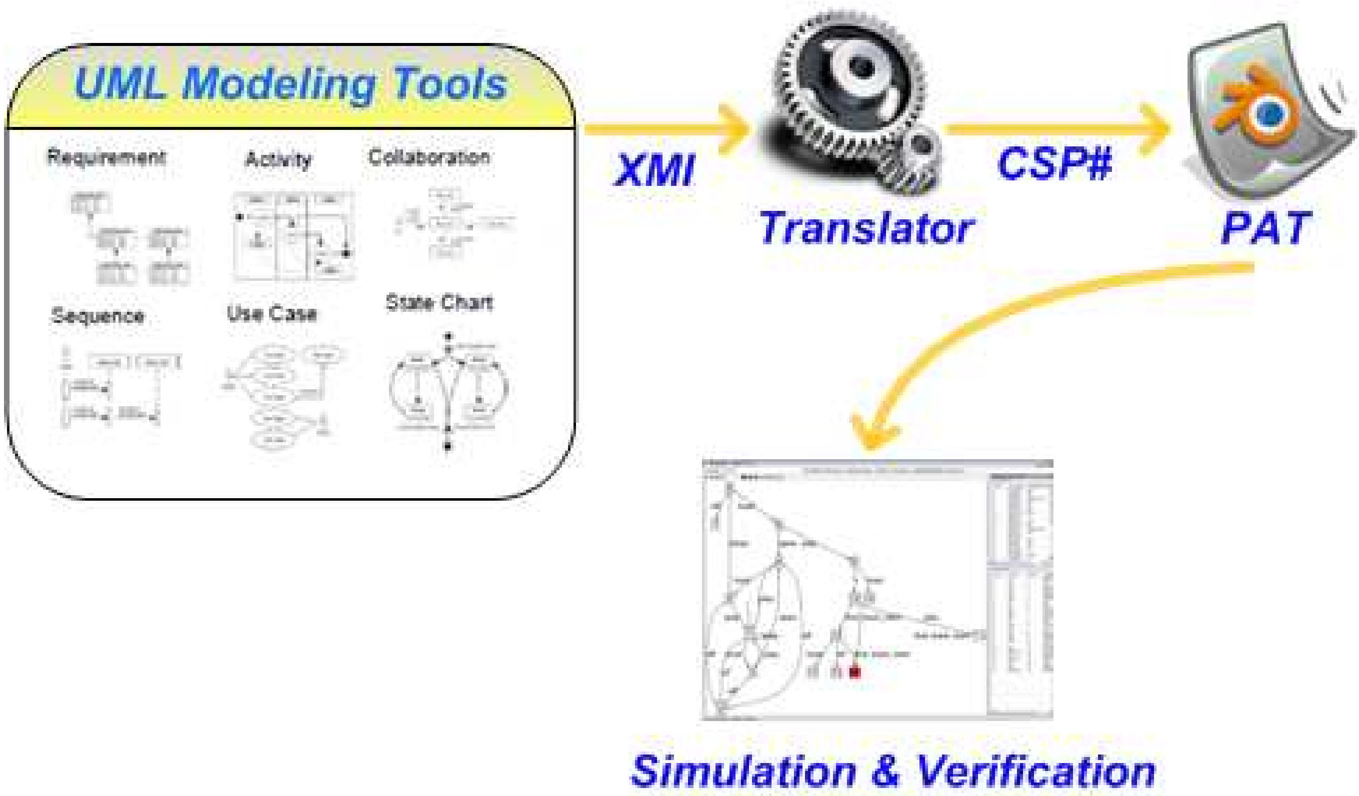
Introduction

- Lack of precise and complete semantics
  - Esp. for dynamic behavior
  - How can we ensure that the models for a system analysis and its design are consistent?
  - How can we check that a design model correctly realizes a system requirement model?
- Take advantage of formal methods to detect model-level errors

Model Checking Principle



UML & Model Checking



Introduction

- Present a translation approach to verifying UML state machines.
  - Fully automatically
  - Independent of any modeling tools
- Verification tool: PAT
  - SPIN, FDR, SMV, UppAal, Chess, Magic, Verisoft, Slam, Blast...
  - Expressive modeling language
  - Simulator
  - Deadlock, reachability, trace refinement relationship, linear temporal logic (LTL) properties with various fairness assumptions.

Introduction

- Compared with other works
  - Support a larger subset of UML state machines than most other works
    - Esp. advanced modeling constructs
  - Minimize the use of shared variables
    - Directly specify in terms of processes and events

Agenda

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

Modeling language

- CSP# (Communicating sequential programs)
  - Communicating Sequential Processes + shared variables + low-level programming constructs
- Grammar

$$
\begin{aligned}
P ::= & \; Stop \mid Skip \mid e\{prog\} \rightarrow P \mid P_1;\ P_2 \mid P_1 \,\Box\, P_2 \\
& \mid P_1 \;|||\; P_2 \mid P_1 \parallel P_2 \mid [b]P \mid atomic\{P\} \\
& \mid P_1 \,\triangle\, P_2 \mid ch!exp \rightarrow P \mid ch?x \rightarrow P \\
& \mid case\{b1 : P_1;\ b2 : P_2;\ \cdots;\ default : P\} \\
& \mid e ::= name(.exp)*
\end{aligned}
$$

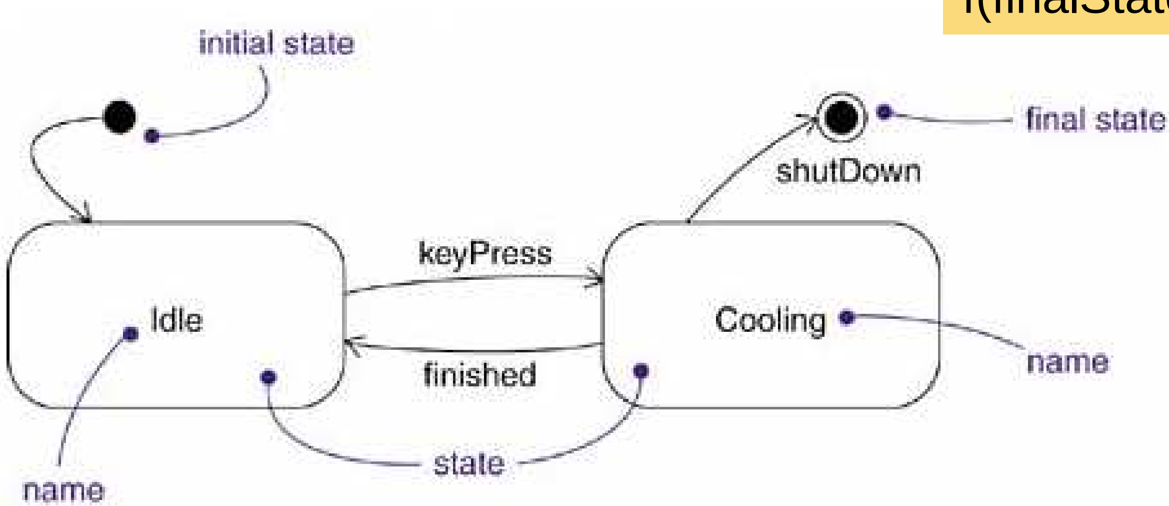
Translation Rules

$$f : UML \rightarrow CSP\#$$

UML State Machines

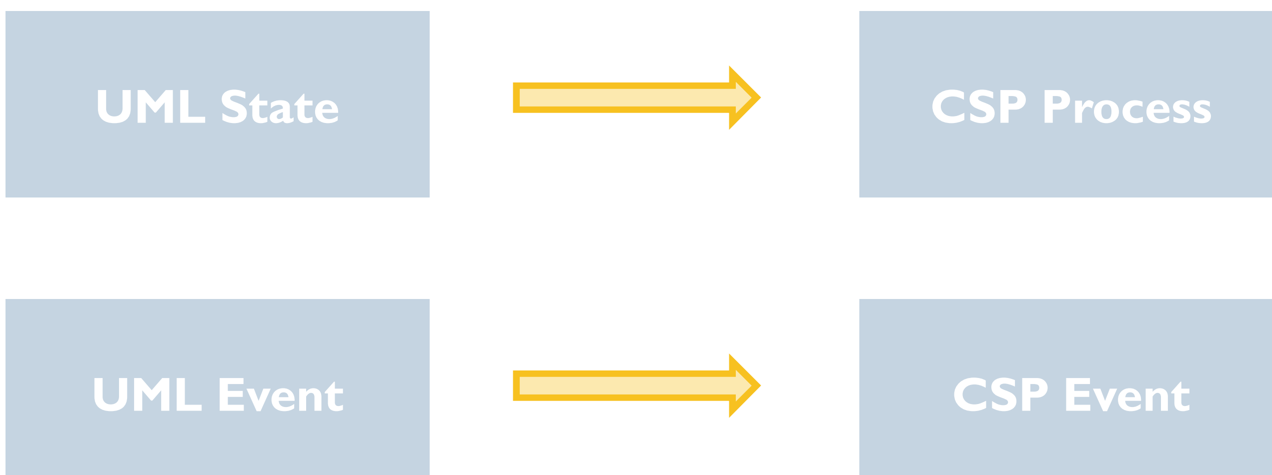
- A state machine describes the lifetime of a single object.
- It contains states and transitions between them.

f(finalState) = Skip



UML state machines

- A state is a condition or situation during the life of an object during which some invariant condition holds.
- An event is an occurrence of a stimulus that can trigger a state transition.



State

- A state has three kinds of optional behavior:
  - Entry
  - DoActivity
  - Exit

P1 Δ *P2:* behaves as *P1 until* the occurrence of the first event from *P2*

f (state) =

f (entry); //atomic process

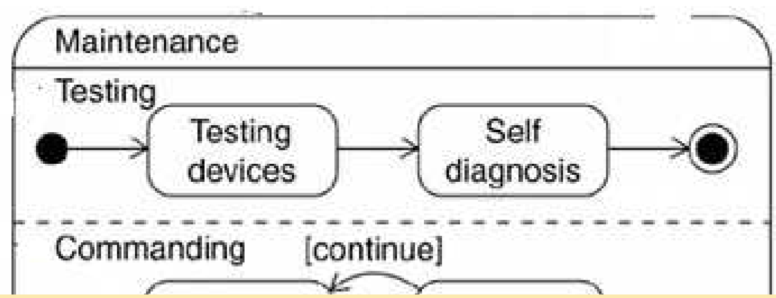
f (doActivity)

Δ

(f (trans1) □ f (trans2) □ · · · □ f (transN))

State

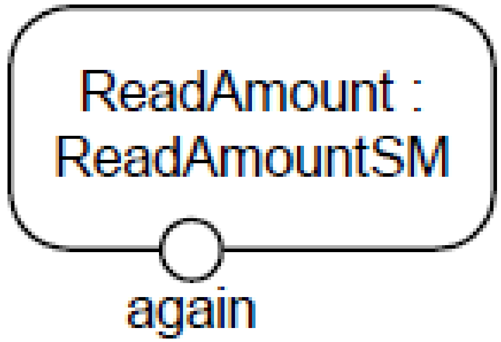
- Three kinds of states
  - Simple
  - Composite
  - Submachine
- Composite state



$$f\,(\text{compositeState}) = f\,(\text{entry});\ (f\,(\text{doActivity}) \;|||\; f(r1) \;|||\; f(r1) \;|||\ldots)\ \Delta\ (f\,(\text{trans1}) \;\square\; f\,(\text{trans2}) \;\square\; \cdots \;\square\; f\,(\text{transN}))$$

State

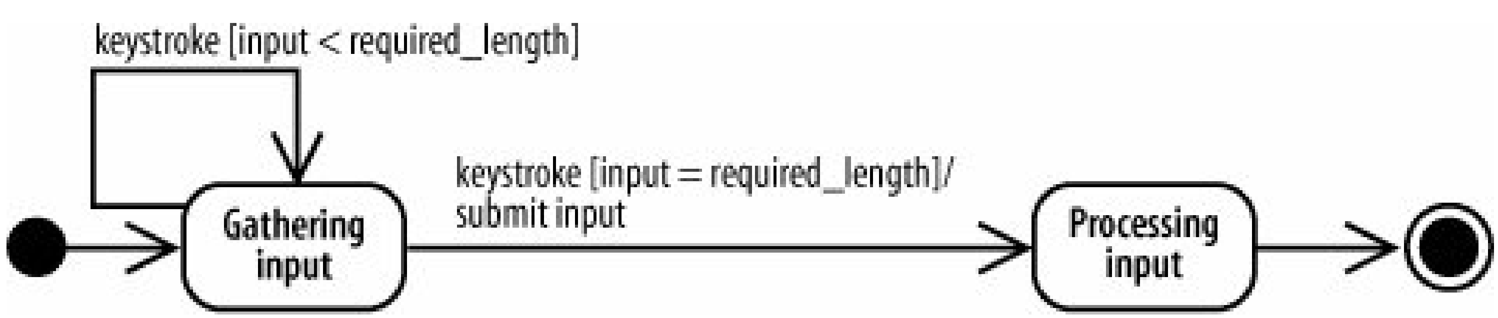
- Submachine state
  - Specifies the insertion of the specification of a submachine state machine



f (ReadAmout) = f (ReadAmountAM)

Transition

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



State machine

- State machine

  $f(sm) = f(i)$ *where i is the topmost initial state of sm.*

- System

  $f(s) = f(sm1) \; ||| \; f(sm2) \; ||| \; \cdot\cdot\cdot \; ||| \; f(smn)$

  $f(s) = f(sm1) \; || \; f(sm2) \; || \; \cdot\cdot\cdot \; || \; f(smn)$

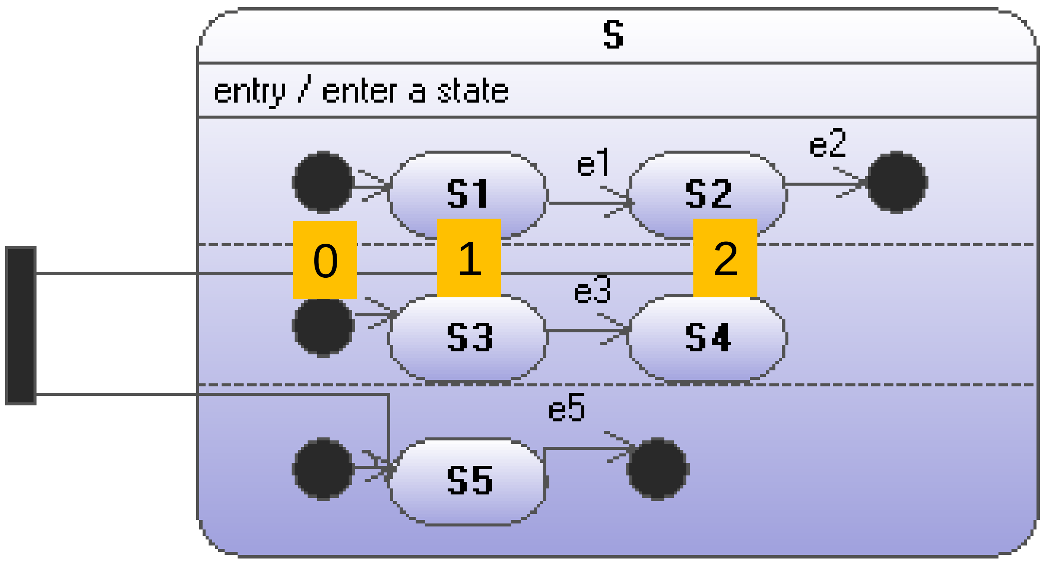
Advanced States and Transitions

- Fork
- Join
- Entry/Exit point
- History

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.

Fork



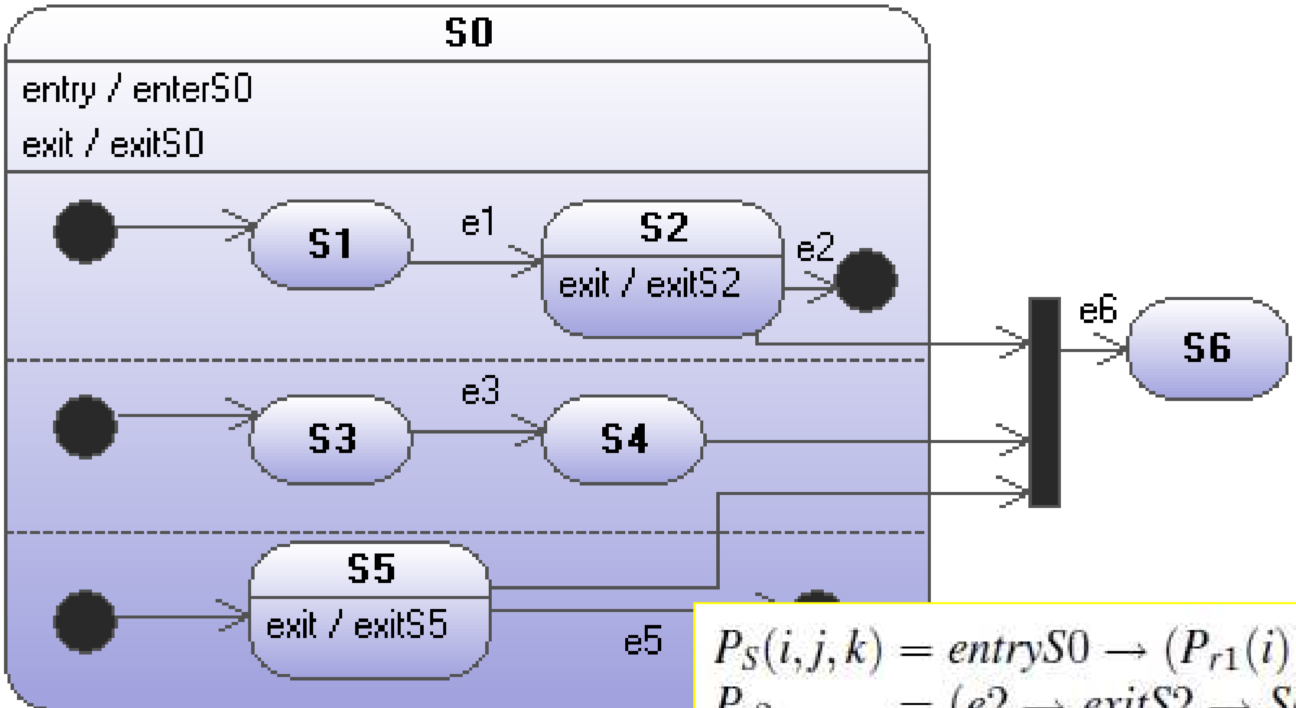
$$P_S(i,j,k)^2 = enter\_a\_state \rightarrow (P_{r1}(i) \;|||\; P_{r2}(j)) \;|||\; P_{r3}(k));$$

$$P_{Fork} = P_S(2,0,1);$$

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.

Join

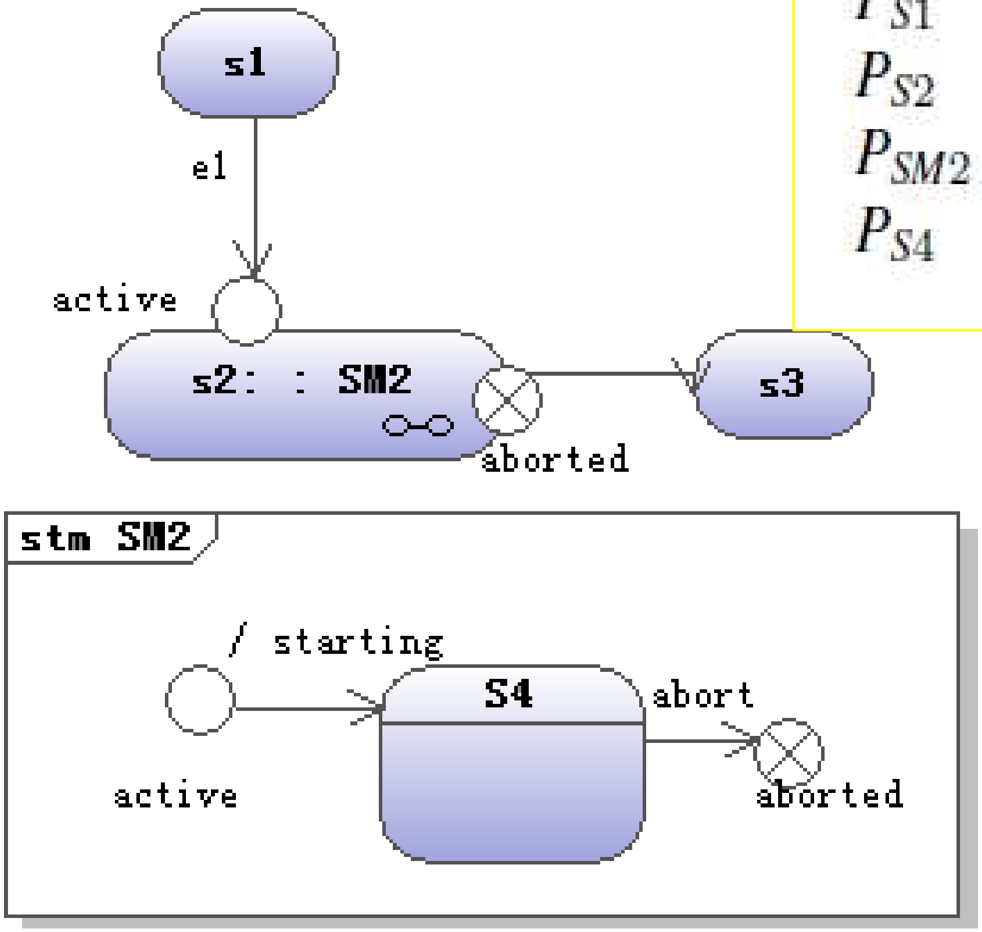


$$
\begin{aligned}
P_S(i,j,k) &= entryS0 \rightarrow (P_{r1}(i) \parallel P_{r2}(j)) \parallel P_{r3}(k));\\
P_{s2} &= (e2 \rightarrow exitS2 \rightarrow Skip) \;\Box\\
&\quad\; (join \rightarrow exitS2 \rightarrow exitS0 \rightarrow P_{join});\\
P_{s4} &= join \rightarrow exitS0 \rightarrow P_{join};\\
P_{s5} &= (e5 \rightarrow exitS5 \rightarrow Skip) \;\Box\\
&\quad\; (join \rightarrow exitS5 \rightarrow exitS0 \rightarrow P_{join});\\
P_{join} &= e6 \rightarrow P_{S6};
\end{aligned}
$$

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

Entry/Exit point



$$
\begin{aligned}
P_{S1} &= e1 \rightarrow ch!0 \rightarrow Skip; \\
P_{S2} &= ch?1 \rightarrow P_{S3}; \\
P_{SM2} &= ch?0 \rightarrow starting \rightarrow P_{S4}; \\
P_{S4} &= abort \rightarrow ch!1 \rightarrow Skip;
\end{aligned}
$$

History

- 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.

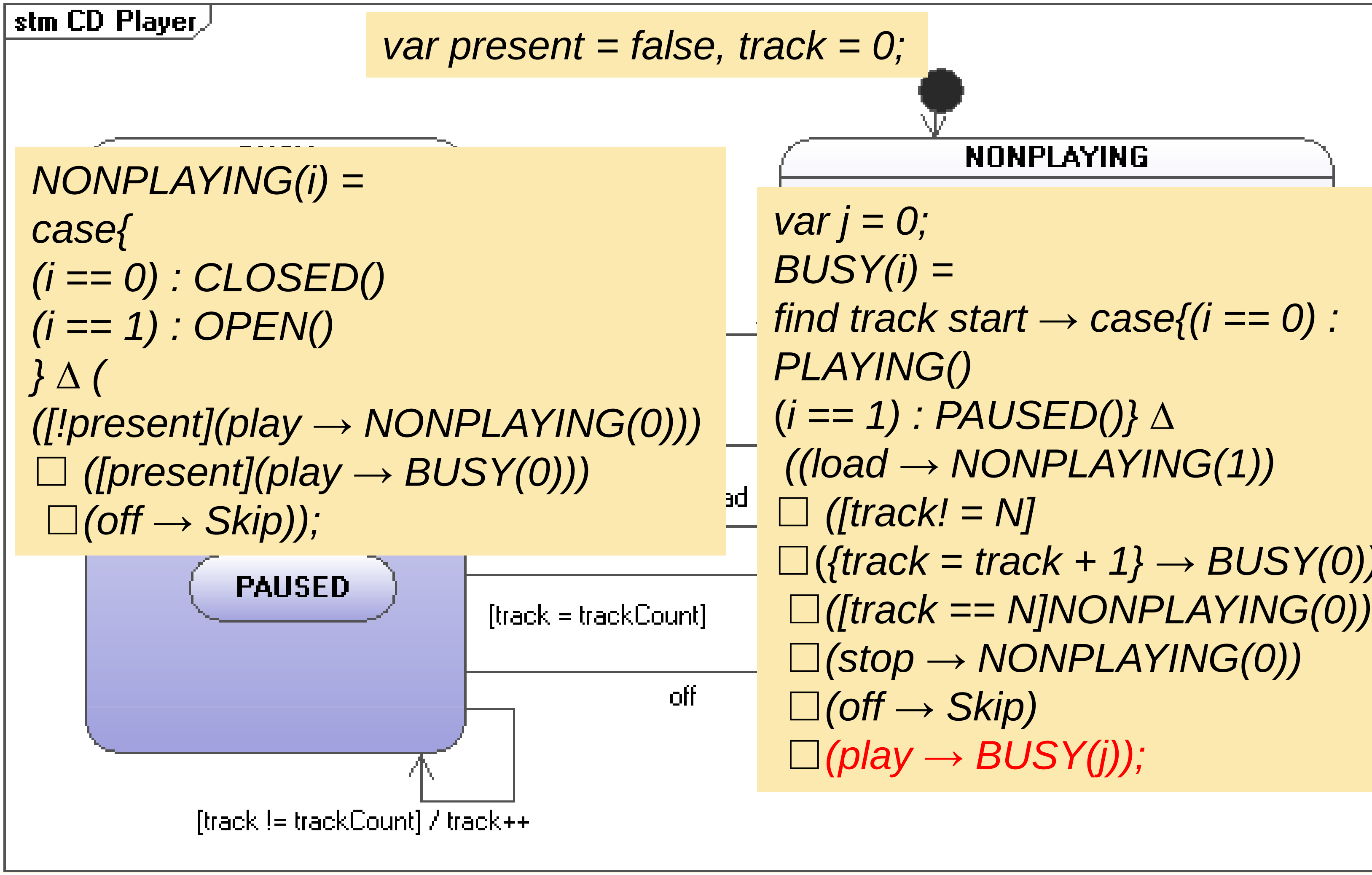
Agenda

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

Case Study

CDPLAYER() = NONPLAYING(0);

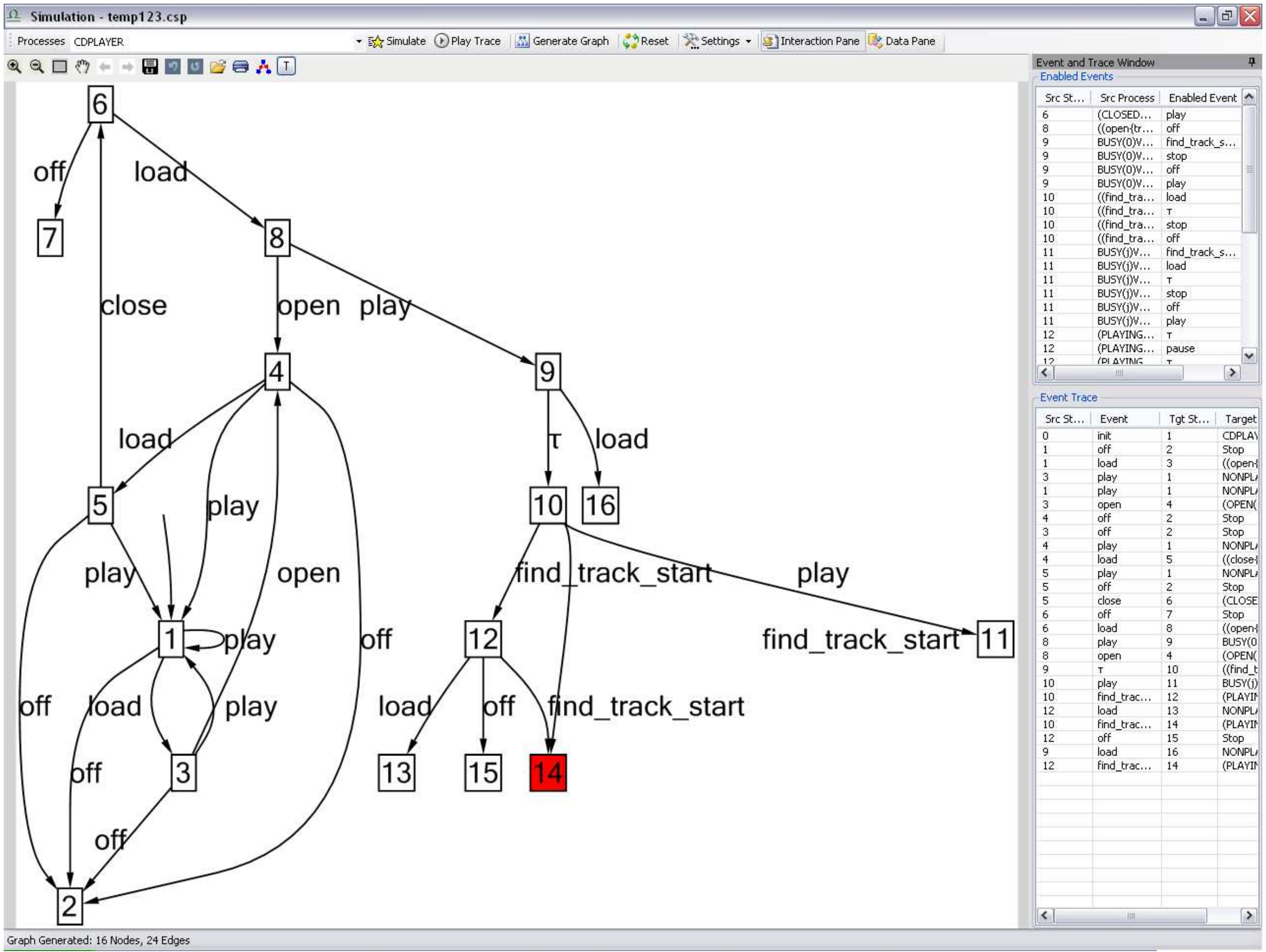
var present = false, track = 0;



NONPLAYING(i) =
case{
(i == 0) : CLOSED()
(i == 1) : OPEN()
} Δ (
([!present](play → NONPLAYING(0)))
□ ([present](play → BUSY(0)))
□(off → Skip));

var j = 0;
BUSY(i) =
find track start → case{(i == 0) :
PLAYING()
(i == 1) : PAUSED()} Δ
((load → NONPLAYING(1))
□ ([track! = N]
□({track = track + 1} → BUSY(0)))
□([track == N]NONPLAYING(0))
□(stop → NONPLAYING(0))
□(off → Skip)
□(play → BUSY(j));

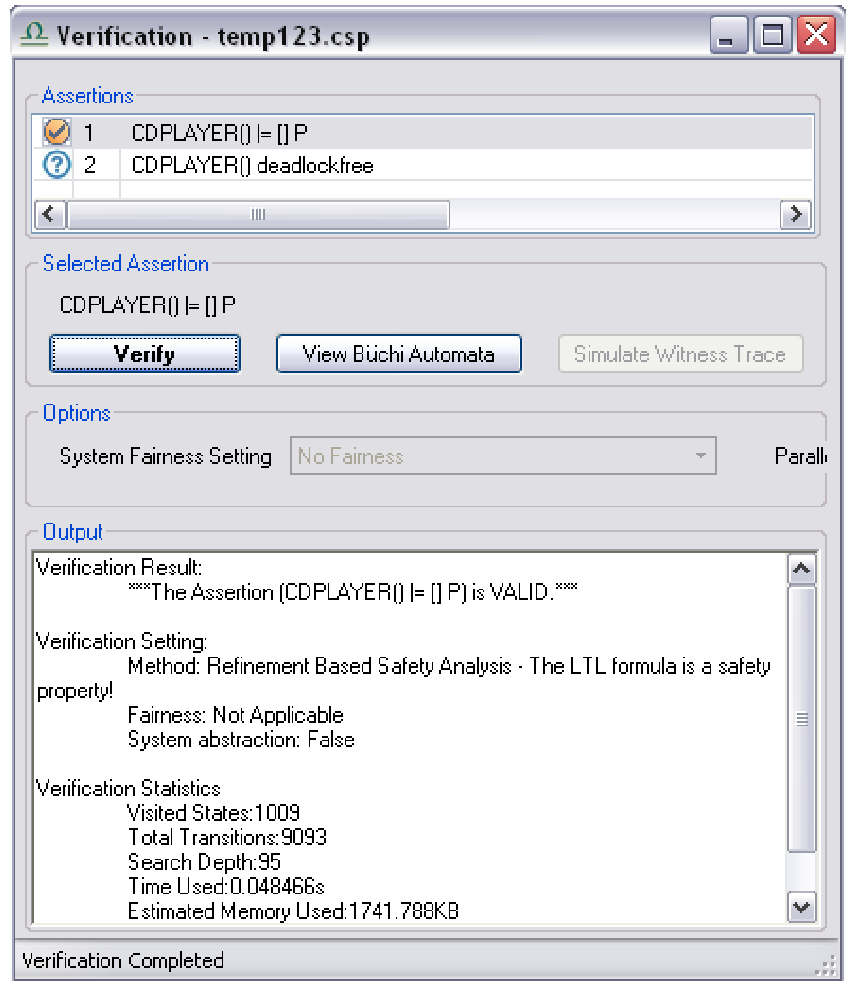
Case Study



Simulation - temp123.csp
Processes CDPLAYER
Simulate
Play Trace
Generate Graph
Reset
Settings
Interaction Pane
Data Pane
Event and Trace Window
Enabled Events
Event Trace
Graph Generated: 16 Nodes, 24 Edges

Case Study

- Two basic requirements
  - $\Box \sim((track == 0) \wedge (play\_track))$
  - $\Box ((present == true) \wedge play \rightarrow \Diamond$



Agenda

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

Conclusion

- Defined a translation scheme for a UML model composed of asynchronously executing, hierarchical state machines.
  - Effectively handle advanced modeling techniques in state machines.
  - Provide a automatic approach to transforming a model of state machines to the input model of PAT model checker

Future work

- Looking for more industrial cases
- Support deferred events and time events
- Sequence diagrams, activity diagrams, …..
- Provide an easier way to specify properties.

The End

Thank you for your kind attention!