Modeling and Verification of MPI based distributed software (using ASMs)

Igor Grudenić, Nikola Bogunović

Department of Electronics, Microelectronics, Computer and Intelligent Systems
Faculty of Electrical Engineering and Computing, Zagreb, Croatia
http://www.fer.hr
Presentation overview

- Motivation
- Formal verification – three interesting properties
- Introduction to ASMs (really short one)
- Using ASMs to model basic MPI communication
- Mutual exclusion case study
- Conclusion & Future work
Motivation

- Complex distributed systems are hard for design and analysis
- Typical problems

<table>
<thead>
<tr>
<th></th>
<th>Synchronous communication</th>
<th>Asynchronous communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallelism (scalability)</td>
<td>HARD</td>
<td>EASY?</td>
</tr>
<tr>
<td>Deadlock</td>
<td>POSSIBLE</td>
<td>NOT POSSIBLE</td>
</tr>
<tr>
<td>System complexity</td>
<td>MODERATE</td>
<td>HUGE</td>
</tr>
</tbody>
</table>
Formal verification

- Checks whether system conforms to specification
- Theorem proving & model checking
- Model checking with SPIN (Siegel 2005)
  - theorems and manipulation to avoid state explosion
  - three interesting properties for model checking and distributed systems
    - Safety (e.g. deadlock absence)
    - Liveness
    - Fairness
Introduction to ASM (abstract state machines)

- State machines – known concept
- Abstract state machines – similar to state machine but
  - You need to define only the initial state and give the ‘recipe’ how to evolve (no enumeration)

  ![Evolve recipe diagram]

  - Math:
    - Initial state (and every other state it evolved to) is a static algebra (set $X$, signature $\Sigma=\{f \mid X^r \rightarrow X\}$)
    - State evolution:
      \[
      \text{If } \text{Condition then Update} \\
      \text{Update: } f(t_1, \ldots, t_2) = t, \ (f \in \Sigma \ \ t_x, t \in X)
      \]
Introduction to ASM (example)

\[ A = 5 \quad B = 4 \quad C = 3 \]

IF \( A = 5 \) THEN \( C = 2 \)
IF \( B = 4 \) THEN \( A = 3 \)
IF \( C = 2 \) THEN \( A = 10 \)

\[ A = ? \quad B = 4 \quad C = 2 \]
Using ASMs to model basic MPI communication

- **ASML** (abstract state machine language) – object oriented
- Model exploration – Spec Explorer
- **How to define MPI processes?**
  - every process is instance of the special *process class*
  - Every method in the process class is considered as atomic action
  - Example: process that executes sequence of actions: 
    Add(), Subtract(), Add(), ...

```plaintext
class process()
    var state as Boolean=true

Add()
    require state=true
    state:=false

Subtract()
    require state=false
    state:=true
```

![Diagram showing state transitions between True and False states with methods Add() and Subtract()](image)
Using ASMs to model basic MPI communication

- Encoding MPI communication

<table>
<thead>
<tr>
<th></th>
<th>BLOCKING</th>
<th>NON BLOCKING</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SEND</strong></td>
<td>Blocking Send()</td>
<td>Make the same transfer as blocking but only when MPI_WAIT is posted?</td>
</tr>
<tr>
<td></td>
<td>require state=DESIRED STATE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>receiver.messageQueue+=[Message]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>state=MESSAGE SENT</td>
<td></td>
</tr>
<tr>
<td><strong>RECEIVE</strong></td>
<td>AnnounceMessageReceive()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>require state=DESIRED STATE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>me.announces+={Announce()}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>state:=DESIRED STATE + RECEIVE ANNOUNCED</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ReceiveMessage()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>require messageQueue contains message</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(BL) require DESIRED STATE + RECEIVE ANNOUNCED</td>
<td></td>
</tr>
<tr>
<td></td>
<td>applicationBuffer=getMessage(messageQueue)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>state:=MESSAGE RECEIVED</td>
<td></td>
</tr>
</tbody>
</table>
Mutual exclusion – case study

- Lamport 1978, only non-blocking communication
- Three properties (safety, liveness, fairness)

\[
\text{requestResource()}
\begin{align*}
\textbf{require} & \quad \text{state}=\text{tState.sleeping} \\
\text{state}:= & \quad \text{tState.requesting} \\
\textbf{foreach} & \quad \text{iProcess in processes} \\
& \quad \text{where iProcess}<>\text{me} \\
& \quad \text{iProcess.messageQueue}+=\text{[NewRequest]} \\
& \quad \text{requestQueue}+=\text{[NewRequest]} \\
& \quad \text{clock}+=1, \text{nOfReceivedAck}:=0
\end{align*}
\]

\[
\text{acceptMessage()}
\begin{align*}
\textbf{require} & \quad \text{Size(messageQueue)}>0 \\
\textbf{require} & \quad \text{state}=\text{tState.requesting} \\
\text{messageQueue}:= & \quad \text{Tail(messageQueue)} \\
\text{clock}:= & \quad \text{max(messageQueue(0).clock+1, clock+1)} \\
\textbf{match} & \quad \text{messageQueue(0).messageType} \\
\text{REQ:} & \quad \text{requestQueue}+=\{\text{messageQueue(0)}\} \\
& \quad \text{sender.messageQueue}+=\text{[NewAcknowledge]} \\
\text{REL:} & \quad \text{requestQueue}-=\text{request(sender)} \\
\text{ACK:} & \quad \text{nOfReceivedAck}+=1
\end{align*}
\]
Mutual exclusion – exploration & verification

[Diagram of mutual exclusion states and transitions]
Future Work

- Standalone model of MPI (only basic communication)
- Configurations (made several by hand now)
- Host processes (support execution of MPI functions supported by the model)
Conclusion

- ASMs are powerful for capturing requirements
- ASMs can give an insight to a distributed system
- Some basic (but important) properties can be verified using available tools
- Questions?