

Modeling and Testing Multi-Threaded Asynchronous Systems with Creol

Rudolf Schlatte

UNU-IIST Seminar
March 16, 2009

Credo Project

- ▶ EU-funded research project
- ▶ model-checking + testing (models of) distributed systems
 - ▶ with some excursions into industrial code . . .

Motivation

- ▶ Modeling
 - ▶ Gain more confidence in requirements, design
 - ▶ Typically near start of project

Motivation

- ▶ Modeling
 - ▶ Gain more confidence in requirements, design
 - ▶ Typically near start of project
- ▶ Testing
 - ▶ Gain more confidence in implementation
 - ▶ Typically near end of project

What to test for

- ▶ Robustness
- ▶ Stability
- ▶ Performance
- ▶ Functionality

Testing Functionality

What to test against?

Test suite = (incomplete / abstract) formal model of the system

We have a formal model from the design phase! Can we use it as test oracle?

⇒ Model-based testing

Slogan: “A language for modeling Distributed Active Objects”

- ▶ Object-oriented modeling language
- ▶ Each method call is its own process within an object
 - ▶ Cooperative scheduling within object
 - ▶ Asynchronous method calls - get results via Future variables
- ▶ Operational semantics given in rewrite logic
 - ▶ Executed on Maude platform
 - ▶ Can do model checking, etc.

Creol as modeling language

- ▶ Familiar syntax

```
op dispatchTask(out index: Int) ==  
  await openCounter > 0;  
  index := index(states, "OPEN");  
  states := replace(states, "BUSY", index);  
  openCounter := openCounter - 1
```

- ▶ Easier modeling through cooperative scheduling

- ▶ Safe communication via member variables

```
if (member >= 1) {  
  member := member - 1;  
  // in C++, member can be negative here!  
}
```

How we used it

- ▶ Re-model an existing system about 100kLOC of C
- ▶ Methods, overall structure turned out “visually similar” to C code
 - ▶ Can we re-use the model for testing?
 - ▶ What does it mean for model + implementation to “behave the same”?

Extracting Event Traces

- ▶ Instrumenting the source code of the implementation
- ▶ Observe internal events of the implementation
- ▶ Record sequence of *events* happening inside the implementation
 - ▶ Events are not only input or output!

Conformance relation

$$conf_I \xrightarrow{I} \{events_I\} \quad (1)$$

Conformance relation

$$\begin{aligned} \text{conf}_M &\xrightarrow{M} \{\text{events}_M\} \\ \text{conf}_I &\xrightarrow{I} \{\text{events}_I\} \end{aligned} \tag{1}$$

Conformance relation

$$\begin{array}{ccc} \text{conf}_M & \xrightarrow{M} & \{\text{events}_M\} \\ \uparrow \pi & & \uparrow \pi \\ \text{conf}_I & \xrightarrow{I} & \{\text{events}_I\} \end{array} \quad (1)$$

Conformance relation

$$\begin{array}{ccc} \mathit{conf}_M & \xrightarrow{M} & \{\mathit{events}_M\} \\ \uparrow \pi & & \uparrow \pi \\ \mathit{conf}_I & \xrightarrow{I} & \{\mathit{events}_I\} \end{array} \quad (1)$$

Every trace observable in the implementation must be replayable in the model:

$$\pi(I(\mathit{conf}_I)) \subseteq M(\pi(\mathit{conf}_I)) \quad (2)$$

The recorded trace

```
<trace>
  <createThreads thread="3079972528"
    time="501911878" number="10"/>
  <starting thread="3075214224"
    time="501911929" info=""/>
  <waiting thread="3066821520" time="501911999"
    info=""/>
  ...
  <enqueue thread="3079972528" time="501912403"
    info="Sabbey-shepherd (Sabbey.c 353)"/>
  ...
</trace>
```

The created code

```
class TestAdapter contracts TestActions
begin
  op init ==
    skip      // TODO: implement test case setup
  op run ==
    See next slide ...

  with Any op enqueue(in thread:Int, time:Int,
                      info:String) ==
    skip      // TODO: implement enqueue action
end
```

Replaying traces in the model

- ▶ An implementation event log is translated into Creol
- ▶ The tester blocks until an event has occurred before allowing the next one

	<code>op run ==</code>
<code>createThread</code>	<code>this.allow("createThread");</code>
↓	<code>await pending_CreateThread = 0;</code>
<code>starting</code>	<code>this.allow("starting");</code>
↓	<code>await pending_Starting = 0;</code>
...	...
↓	<code>this.enqueueTask(3079972528,</code>
<code>enqueueTask</code>	<code>501912403,</code>
	<code>"Sabbey-shepherd (Sabbey.c 353)";</code>
	<code>done := true</code>

What about user input, partial models?

- ▶ Model can be *partial*, e.g. no modeling of user input
- ▶ Generate stub methods and call them from the tester
- ▶ This generates an environment for partial models based on implementation behavior

Inserting events into the model

```
op dispatchTask(out index: Int) ==  
  
  await openCounter > 0;  
  index := index(states, "OPEN");  
  states := replace(states, "BUSY", index);  
  openCounter := openCounter - 1
```

Inserting events into the model

```
op dispatchTask(out index: Int) ==  
  await tc.request("dispatchTask");  
  await openCounter > 0;  
  index := index(states, "OPEN");  
  states := replace(states, "BUSY", index);  
  openCounter := openCounter - 1
```

Reaching a test verdict

- ▶ Run model || tester
- ▶ Assertions in model violated \Rightarrow Model Fail

Reaching a test verdict

- ▶ Run model || tester
- ▶ Assertions in model violated \Rightarrow Model Fail
- ▶ Tester deadlocks: implementation trace not reproducible by model \Rightarrow Implementation Fail

Reaching a test verdict

- ▶ Run model || tester
- ▶ Assertions in model violated \Rightarrow Model Fail
- ▶ Tester deadlocks: implementation trace not reproducible by model \Rightarrow Implementation Fail
- ▶ Tester terminates \Rightarrow Pass

Thank You!