Model Testing Asynchronously Communicating Objects using Rewriting Modulo AC

Olaf Owe¹, Martin Steffen¹, and Arild B. Torjesen¹,²

University of Oslo

Norsk Regnesentral

MBT’10, Παφωσ
Structure

1. Introduction
2. Testing Creol objects
3. Rewriting logic implementation and experimental results
4. Conclusion
1 Introduction

2 Testing Creol objects

3 Rewriting logic implementation and experimental results

4 Conclusion
Background

- Project:
  - modelling **asynchronously** communicating components in **open** environments
  - **object-oriented**
  - **behavioral** interface descriptions
  - automated verification and testing techniques

- Challenges
  - **asynchronicity** $\Rightarrow$ non-determinism $\Rightarrow$ state space explosion.

- Approach:
  - tackle complexity by “divide-and-conquer”
  - black-box behavior given by interactions at the **interface**
General setting

**Goal:** Test components under environment assumptions/schedulings

**Approach:** Specification language over communication labels

- **input** interactions: environment *assumptions*.
- **output** interactions: *commitments* of the component.

⇒ expected observable output behavior under the *assumption* of a certain scheduling of input.

**Method:** Specification simulates environment behavior.

- execute component and specification in parallel
- *generate* incoming communication from specification
- *test* actual outgoing communication from the component
Main contributions

1 Theoretical basis:
   - formalization of the interface behavior of an asynchronous OO
     modelling language.

2 Framework for scheduling asynchronous testing of objects.
   - executable specification language
   - method for composing specifications and components under test
   - implementation of a test framework

3 Use Maude’s rewriting modulo AC to test only up to
   observational equivalence.

4 Use Maude’s search for state exploration (rewriting modulo AC).

5 Experimental results, comparing:
   - modulo AC rewriting.
   - explicit reordering of output events.
Creol (www.uio.no/~creol): object-oriented modelling language for distributed systems

- model distributed systems at a high level of abstraction.
- strongly typed, formal operational semantics in rewriting logic
- active concurrent objects
- communication by asynchronous method calls.
- Creol object: acts as a monitor.
- cooperative scheduling, i.e., explicit and conditional release/yields etc.
- non-deterministic selection of suspended processes and incoming calls.
Abstract syntax

\[ C ::= \textcolor{magenta}{0} \mid C \mid C \mid \nu(n:T).C \mid c[F, M] \mid o[c, F, L] \mid n\langle t \rangle \]

\[ F ::= l = f, \ldots, l = f \]

\[ M ::= l = m, \ldots, l = m \]

\[ m ::= \varsigma(n:T).\lambda(x:T, \ldots, x:T).t \]

\[ f ::= \varsigma(n:T).\lambda().v \mid \varsigma(n:T).\lambda().\bot_{n'} \]

\[ t ::= v \mid \text{stop} \mid \text{let } x : T = e \text{ in } t \]

\[ e ::= t \mid \text{if } v = v \text{ then } e \text{ else } e \mid \text{if } \text{undef}(v.l()) \text{ then } e \text{ else } e \]
\[ \quad \mid v@[l(v)] \mid v.l(l) \mid v.l() \mid v.l := \varsigma(s:T).\lambda().v \]
\[ \quad \mid \text{new } n \mid \text{claim@}(n, n) \mid \text{get@}n \mid \text{suspend}(n) \mid \text{grab}(n) \mid \text{release}(n) \]

\[ v ::= x \mid n \mid () \]

\[ L ::= \bot \mid \top \]

- **component**: classes, objects, and (named) threads.
- **active**, executing entities: *named threads* \( n\langle t \rangle \)
- hiding and dynamic scoping: \( \nu \)-operator
Interface interactions

- Steps occurring at the interface.
- Component/environment: exchange information via \textit{call-} and \textit{return}-labels:

\[
\gamma ::= n\langle\textit{call} \ n.l(\bar{v})\rangle \mid n\langle\textit{return}(n)\rangle \mid \nu(n:T).\gamma \quad \text{basic labels}
\]
\[
a ::= \gamma? \mid \gamma! \quad \text{input and output labels}
\]

- External steps

\[
\Xi \vdash C \xrightarrow{a} \Xi \vdash \dot{C}
\]

- $\Xi = \text{“context” of } C$ (assumptions + commitments)
- contains identities + typing of objects and threads known so far
- \textbf{checked} in incoming communication steps
- \textbf{updated} in outgoing communication steps
1 Introduction

2 Testing Creol objects

3 Rewriting logic implementation and experimental results

4 Conclusion
Behavioral interface specification language

Black-box behavior of a component described by a set of traces.

Design goals:

- concise
- formally justified
- executable in rewriting logic.

\[
\begin{align*}
\gamma & ::= x\langle call \ x.l(x) \rangle \mid x\langle return(x) \rangle \mid \nu(x:T).\gamma \mid (x:T).\gamma \\
\alpha & ::= \gamma? \mid \gamma! \\
\varphi & ::= X \mid \epsilon \mid \alpha . \varphi \mid \varphi + \varphi \mid \text{rec } X.\varphi
\end{align*}
\]

- specification language: uses variables
- \textbf{two} kinds of variable \textbf{binders}
- Creol communication labels: concrete names/references.
Well-formedness

- Restrict specifications to traces actually possible at the interface.

- four three main restrictions:
  - typing
  - scoping
  - communication patterns
  - polarity: specifications either well-formed input or well-formed output.

- given as derivation/type system over trace specs.
Asynchronicity—"Observational blur"

- Asynchronicity: message order not preserved in communication.

- The specification is relaxed up-to observational equivalence

- Testing of output only up-to observability.

\[
\nu(\Xi) \cdot \gamma_1! \cdot \gamma_2! \cdot \varphi \equiv_{\text{obs}} \nu(\Xi) \cdot \gamma_2! \cdot \gamma_1! \cdot \varphi
\]

\[
\nu(\Xi) \cdot \gamma_1! \cdot \gamma_2! \cdot \varphi \equiv_{\text{obs}} \nu(\Xi) \cdot \gamma_2! \cdot \gamma_1! \cdot \varphi
\]
Operational semantics of specifications

Given $\equiv_{obs}$, the meaning of a specification is given operationally and straightforwardly, e.g.: 

\[
\Xi = \Xi + a \quad \text{R-PREF} \\
\Xi \vdash a.\varphi \xrightarrow{a} \Xi \vdash \varphi \\
\Xi \vdash \varphi_1 \xrightarrow{a} \Xi \vdash \varphi' \\
\Xi \vdash \varphi_1 + \varphi_2 \xrightarrow{a} \Xi \vdash \varphi' \\
\varphi \equiv_{obs} \varphi' \quad \Xi \vdash \varphi' \xrightarrow{a} \Xi \vdash \varphi'' \\
\Xi \vdash \varphi \xrightarrow{a} \Xi \vdash \varphi'' \\
\text{R-EQUIV}
\]
Asynchronous testing of Creol objects

- Combine:
  - external behavior of object
  - intended behavior given by specification
- interaction defined by synchronous parallel composition
- specification $\varphi$ and component must engage in corresponding steps:
  - For *incoming* communication, this schedules the order of interactions with the component
  - For *outgoing* communication, the interaction will take place only if it matches an outgoing label in the specification
  - **Error** if the specification requires input and the component could do output.
Parallel composition

\[\Xi \vdash C \xrightarrow{\tau} \Xi \vdash \hat{C}\]  \hspace{1cm} \Xi \vdash a \lesssim_\sigma b\n
\[\Xi \vdash C \parallel \varphi \to \Xi \vdash \hat{C} \parallel \varphi\]  \hspace{1cm} \Xi_1 \vdash C \parallel \varphi \to \Xi_1 \vdash \hat{C} \parallel \varphi\]

\[\Xi \vdash \varphi : \text{wf}^2\]  \hspace{1cm} \Xi \vdash \nu(\Xi').(C \parallel n(\text{let } x : T = o.l(\vec{v}) \text{ in } t) \parallel \varphi) \to \$\]

\[\Xi \vdash \varphi : \text{wf}^2\]  \hspace{1cm} \Xi \vdash \nu(\Xi').(C \parallel n(\nu) \parallel \varphi) \to \$

- **Matching** of \(\varphi\)'s step and components step (\(\vdash a \lesssim_\sigma b\))
- As said: specification contains:
  - freshness assertions (\(\nu(x : T)\))
  - standard variable declarations (\(x : T\))
1 Introduction

2 Testing Creol objects

3 Rewriting logic implementation and experimental results

4 Conclusion
Implementation in rewriting logic

- Semantics of Creol is executable in Maude

- Implementation of the spec. language in Maude, too

- Execution of Creol components synchronized with specifications
  - generate input from specification
  - test component behaviour for conformance

- Random generation of input parameters from predefined sets or interval.

- No input queue, specified method calls are answered immediately

- Reentering suspended methods may interfere.
Implementation in rewriting logic

- Creol configuration:

  \( r_l \text{ Cfg} \Rightarrow \text{Cfg'} \).

- Creol configuration: objects, classes, and messages:

  \( r_l \text{ O C Cfg} \Rightarrow \text{O'} \text{ C M Cfg} \).

- Test framework: introduce \( \text{Spec} \) for specifications.

  \( c_r l \text{ Spec} \mid\mid \text{O Cfg} \Rightarrow \text{Spec'} \mid\mid \text{O'} \text{ M Cfg if Cond} \).

- Implementation is close to the operational semantics which is easily coded into Maude.

- “Observational blur”, output prefixes of specifications defined to be AC.
Experimental results

- **testing** by executing parallel composition of component and specification.
  
  \[
  \text{rew } \text{spec} \parallel c \text{ cClass .}
  \]

- outcomes:
  - error reported
  - stop

- Conformance relation is input-output conformance Execution of \( c \) should only lead to output foreseen by \( \text{spec} \).

- **verification** by *searching* for error configurations
  
  search in PROGRAM :
  
  \[
  \begin{align*}
  \text{spec} \parallel c \text{ cClass} &\Rightarrow+ \\
  \text{spec}' \parallel \text{conf errorMsg(S:String)} &\Rightarrow \\
  \text{such that} &\ldots
  \end{align*}
  \]
Experiments

- Experiments to demonstrate usefulness of approach

- Compare rewriting specifications with same semantics but:
  
  1. using Maude’s built in AC rewriting.
  2. equivalent, expanded version of specifications.

- AC rewriting pays off wrt. time and number of rewrites.
Example 1

- Component under test consists of one object with \( n \) methods.
- Specification: all methods must have been called before any method may return.
- Tests parametrized over \( n \): spec for \( n = 3 \):

\[
\text{spec3} \quad = \quad n_1 \langle \text{call c.m}_1(x_1) \rangle \cdot n_2 \langle \text{call c.m}_2(x_2) \rangle \cdot n_3 \langle \text{call c.m}_3(x_3) \rangle \cdot (n_1 \langle \text{return}(y_1) \rangle \cdot n_2 \langle \text{return}(y_2) \rangle \cdot n_3 \langle \text{return}(y_3) \rangle) \cdot \epsilon
\]
Example 1

<table>
<thead>
<tr>
<th>n</th>
<th>ms CPU time</th>
<th>AC</th>
<th>Non AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>16</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>38</td>
<td>379</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>198</td>
<td>1.498</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1.030</td>
<td>6.782</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5.407</td>
<td>49.311</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>27.894</td>
<td>NA</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>153.316</td>
<td>NA</td>
<td></td>
</tr>
</tbody>
</table>
Example 2 - broker

- a broker is an intermediary between client and several providers

```
Client

getP(x,k) ➔ Broker

return(v) ➔ Provider

Broker

getQ(x) ➔ Provider

return(v) ➔ Provider

Provider

Provider
```

- specification: broker must query a certain number of providers before returning

\[ \text{specb}_k = n_{c_1} \langle \text{call b.getP}(x, k) \rangle? \]
Example 2

<table>
<thead>
<tr>
<th>$k$</th>
<th>ms CPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$AC$</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>21</td>
</tr>
<tr>
<td>5</td>
<td>33</td>
</tr>
<tr>
<td>6</td>
<td>48</td>
</tr>
<tr>
<td>7</td>
<td>63</td>
</tr>
<tr>
<td>8</td>
<td>86</td>
</tr>
<tr>
<td>9</td>
<td>106</td>
</tr>
</tbody>
</table>
Summary

- **formalization** of interface behavior of a concurrent OO language (Creol) + a behavioral interface specification language.

- how to use this specification language for black-box testing of models for asynchronously communicating objects.

- a **RW logic** formalization of the testing framework for Creol
  - using rewriting for conformance testing and search for verification

- one way to deal with potential reordering of communication

- using modulo **AC** rewriting reduces resource consumption
1. Introduction

2. Testing Creol objects

3. Rewriting logic implementation and experimental results

4. Conclusion
Future work

- from objects to multi-object components
- extensive case study, testing model for Wireless Sensor Networks.
- extend approach to C# or Java
- narrowing
- use traces from real programs
Related work

- [Tre96] ioco testing
- [VCG+08] observable and controllable actions, conformance based on alternating simulation
- [JOT08] assumption/commitment style verification of components
- [GKST10] formal basis of the approached studied here
- [SAdB+08] testing internal state of Creol objects intra-object scheduling
- [AGSS08] case study for model based testing, using Creol
References


