An abstract behavioral model of distributed concurrent objects (2)

Einar Broch Johnsen

Dept. of Informatics, University of Oslo
Email: einarj@ifi.uio.no

COST Action IC0701 Winter School
on Verification of Object-Oriented Programs,

Viinistu, Estonia, Jan 29 2009

Plan

- Distributed concurrent objects in Creol: **previous lecture**
- Semantics and execution platform: **previous lecture**
- Reasoning about Creol models: **today**
- Runtime evolution of Creol models: **today**

Note: Today's topics are very much "work in progress".

Flashback

- An **executable** OO **modelling** language
- Formally defined semantics in rewriting logic
- Targets open distributed systems
- Abstracts from the particular properties of the (object) scheduling and of the (network) environment
- The language design should support verification
- **Key concepts**: concurrent objects, interfaces, asynchronous method calls, suspension points, ...

Example: A Bank Account

```
interface Client
begin with Account
    op giveCode (out code : Int)
end

interface DepositAccount
begin with Any
    op deposit (in sum : Int, out return : Bool)
end

interface Account inherits DepositAccount
begin with Client
    op transfer (in sum : Int, acc : Account; out return : Bool)
end
```

Example: A Bank Account (2)

```
class BankAccount implements Account
begin
  var bal : Int := 0; var f : Label[Bool];

  op verify(in code:Int)== ...

  with Any
    op deposit (in sum : Int, out return : Bool) ==
        bal := bal+sum; return := true

  with Client
    op transfer (in sum : Int, acc : Account; out return : Bool) ==
      await caller!giveCode(code);
        if verify(code)
          then await bal ≥ sum ; bal := bal−sum;
            f!acc.deposit(sum); await f?; return := true
          else return := false end
end
```

Typing

- **Context** Γ: interfaces $\Gamma_{\mathcal{I}}$, classes $\Gamma_{\mathcal{C}}$, variables Γ_{V}
- **Context overriding**: $\Gamma + \Delta$ is Γ overridden by Δ
- **Judgments** $\Gamma \vdash s$

The type system (sketch):

$$\text{(Var)}\ \frac{\Gamma(v) = T}{\Gamma \vdash v : T} \qquad \text{(Get)}\ \frac{\Gamma(x) = T \quad \Gamma \vdash v : \mathsf{Label}[T]}{\Gamma \vdash v?(x) : ok} \qquad \text{(New)}\ \frac{\exists T' \in \mathrm{interfaces}(\Gamma_{\mathcal{C}}(C)) \cdot T' \preceq T}{\Gamma \vdash \mathsf{new}\ C(\,) : T}$$

$$\text{(Class)}\ \frac{\begin{array}{c}\forall M \in \overline{\mathbf{with}\ I\ \overline{M}} \cdot \Gamma + [\mathrm{attr}(C)] + [\mathsf{caller} \mapsto_v I] \vdash M : ok \\ \forall I \in \overline{I} \cdot \mathrm{implements}(\Gamma_{\mathcal{C}}(\Gamma_{\mathsf{V}}(self)), I)\end{array}}{\Gamma \vdash \mathbf{class}\ C\ \mathbf{implements}\ \overline{I}\ \mathbf{begin}\ \mathbf{inherits}\ \overline{C}\ \mathbf{var}\ \overline{f\ T};\overline{\mathbf{with}\ I\ \overline{M}}\ \mathbf{end} : ok}$$

Type soundness:

no method-not-understood errors at run-time for well-typed programs

Reasoning about Creol Objects

- Creol objects are typically **non-terminating**
- Object state strictly **encapsulated** by the interfaces
- At most **one active process** at a time inside the object
- **Unspecified** (cooperative) scheduling

- **Basic idea**: Objects as maintainers of invariants

- Local class invariant **i**: maintenance of local state
- Global invariant **I**: properties of futures (method calls)

Behavioral Types

- Annotate interfaces with specs of external properties

interface Account **inherits** DepositAccount
begin with Client
  op transfer (**in** sum : Int, acc : Account; **out** return : Bool) **sat** (p,q)
end

How to specify these properties?

- Simple case: relate inputs to outputs
- Strengthen specs with auxiliary variables
- The history of observable communication (local trace)
- Specify restrictions (invariant) on local sequence of interaction
- Alphabet of observables given by interface and caller's cointerface
- deposit and transfer (from interface), giveCode (from cointerface)

Example: More expressive behavioral types (Larch style)

We can assume that

- an invocation is reflected in the history by an invoc message
- a completion is reflected by a comp message
- histories are well-formed

Define balance : Seq[α(Account)] $\rightarrow$ Bool

balance(ε)=0
balance(h $\vdash$ *comp*(*deposit*(*sum*)))= balance(h) + sum
balance(h $\vdash$ *comp*(*transfer*(*sum*, *acc*)))= balance(h) − sum
balance(h $\vdash$ **others**)= balance(h)

transfer_ok(h,sum, o)= balance(h) $\geq$ sum $\wedge$ h/o **ew** comp(giveCode,...)

Now, transfer_ok(h,sum,o) can now be used as a **postcondition** to transfer-calls from o, or as an **invariant** AI(h) at the interface level

AI(h) = h **ew** comp(transfer, sum, o) $\Rightarrow$ transfer_ok(h,sum,o)

Internal Reasoning (1)

- Class invariant
- For each method declaration: pre/postconditions and proof outline

Proof obligation

- A class must satisfy local and global invariants
- Applies to all methods in the class

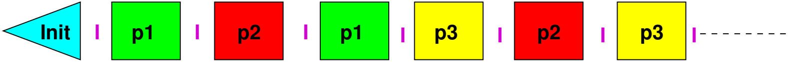
Example

Without histories: $\mathsf{bal} \geq 0$

With histories: $\mathsf{bal} \geq 0 \wedge \mathsf{bal} = \mathsf{balance}(\mathsf{h}/\alpha(\mathsf{Account}))$

Internal Reasoning (2)

- Let us consider a local execution in an object



- Basic idea for the partial correctness proof theory

  Objects as maintainers of local invariants i

- Standard weakest precondition proof rules
- Rule for **await**-statements

$$\frac{i \wedge g \Rightarrow q}{\{i\}\ \textbf{await}\ g\ \{q\}}$$

The Global Invariant

What is the global invariant?

- Imposes restrictions on the values of comp-messages (futures)
- Representation of the behavioral type system
- Relates completions to invocations
- Relates object histories after projection to interface alphabets

Proof obligation: A class does not violate the global invariant

- Induction over the methods again
- The class implements its declared interfaces
- The class does not violate preconditions from other interfaces
- If the global invariant is history-based, then the local invariant will also need to construct a history. This typically relates the internal state with the observable communication (trace) of an object.

Global Reasoning: Example

interface Account **inherits** DepositAccount
begin with Client
  op transfer (**in** sum : Int, acc : Account; **out** return : Bool)
invariant AI(h)
end

Let **H** denote the global history.

$\mathbf{I}(H)$= well-formed(H) $\wedge \ldots \wedge$ AI(H/α(Account)) $\wedge \ldots$

(Composition technique for local reasoning, Soundararajan TOPLAS 1984)

Verification vs. Testing

- Work on testing objects wrt. behavioral interfaces
- Larch-style specs. give **confluent** and **terminating** rewrite system
- Restrictions on object input, requirement on object output
- Use Maude to simulate an **open environment** for an object, based on its interface
- May add **scheduler** to the object to restrict non-determinism in order to comply with the interface requirement

Inheritance and Behavioral Subtyping

The separation of interface and class inheritance allows a flexible form of **code reuse**.

- Behavioral subtyping requirements apply to subinterfaces
- A class must maintain its own invariant and the global invariant
- A class need not maintain superclass' invariants
- Class inheritance may use **lazy behavioral subtyping**, which supports **incremental reasoning**
- LBS tracks exactly which properties need to be maintained by method redefinitions in subclasses

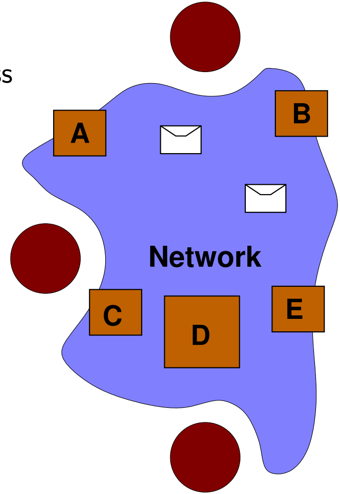
System Evolution in Creol

- Distributed systems need modifications due to
  - Bug fixes
  - New user requirements
  - Changing system environments
- Critical systems need to evolve without compromising availability!
  - E.g., Bank systems and air traffic control systems
- Evolution must happen at runtime
- Modifications must be safe
- Focus so far: type safety

Dynamic Class Upgrades in Creol

- Balance flexibility, ease of use, robustness
- A *modular* OO upgrade mechanism
- Asynchronous upgrades propagate through the dist. system
- Modify class definitions at runtime

- Class upgrade affects:
  - All *future* instances of the class and its subclasses
  - All *existing* instances of the class and its subclasses



Which changes are supported?

- Introduce new classes in the running system
- Provide new services by introducing new interfaces
- Modify an existing class in the class hierarchy
- Which modifications can we allow?
  - Add / remove interfaces?
  - Add /remove class parameters?
  - Add / remove fields?
  - Add /remove methods?
  - Redefine methods?
  - Add /remove superclasses?

Example of a Class Upgrade: Bank Account

```
class BankAccount implements Account begin                    // Original
var bal : Int := 0; var f : Label[Bool];
with Any
  op deposit (in sum : Int, out ret : Bool) == bal := bal+sum; ret := true
with Client
  op transfer (in sum : Int, acc : Account; out ret : Bool) ==
      await bal ≥ sum ; bal := bal−sum; f!acc.deposit(sum); ret := true
end

update BankAccount implements ∅ inherits ∅ begin
var overdraft : Nat := 0
with Client
  op transfer (Nat sum, Account acc; out ret : Bool) ==
      await bal ≥ (sum−overdraft); bal := bal−sum;
           f := acc!deposit(sum); ret := true
with Banker
  op setOverdraft (max: Nat) == overdraft := max
end
```

Example of a Class Upgrade: Bank Account

```
class BankAccount implements Account begin                 // New version
var bal : Int := 0; var f : Label[Bool]; var overdraft : Nat := 0
with Any
  op deposit (in sum : Int, out ret : Bool) == bal := bal+sum; ret := true
with Client
  op transfer (Nat sum, Account acc; out ret : Bool) ==
     await bal ≥ (sum−overdraft); bal := bal−sum;
         f := acc!deposit(sum); ret := true
with Banker
  op setOverdraft (max: Nat) == overdraft := max
end
```

Syntax for Dynamic Classes

$$
\begin{array}{lcl}
U & ::= & \textbf{new-class}\ C\ \textbf{implements}\ \overline{I}\ \textbf{inherits}\ \overline{C}\ \textbf{begin}\ \overline{\textbf{var}\ f : T};\overline{\textbf{with}\ I\ \overline{M}}\ \textbf{end} \\
& | & \textbf{new-interface}\ I\ \textbf{inherits}\ \overline{I}\ \textbf{begin with}\ I\ \overline{M_s}\ \textbf{end} \\
& | & \textbf{update}\ C\ \textbf{implements}\ \overline{I}\ \textbf{inherits}\ \overline{C}\ \textbf{begin}\ \overline{\textbf{var}\ f : T};\overline{\textbf{with}\ I\ \overline{M}}\ \textbf{end} \\
& | & \textbf{simplify}\ C\ \textbf{retract}\ \overline{C}\ \textbf{begin}\ \overline{\textbf{var}\ f : T};\overline{\textbf{with}\ I\ \overline{M}}\ \textbf{end}
\end{array}
$$

Challenges:

- The timing of async. upgrade operations at runtime
- New processes must execute on the new object state
- Old processes must execute on the old object state
- The operations may depend on each other!

Example

```
class C1     -- Version 2, Upgrade 1
begin
  op run() == n(); run()
  op n() == var o : I;
      o := new C3; o.m()
end
class C2     -- Version 2, Upgrade 1
begin
  op m() == Body
end
class C2     -- Version 2, Upgrade 1
begin
  op m() == Body
end

class C3     -- Version 3, Upgrade 1
implements I
inherits C2
begin endclass C3     -- Version 3, Upgrade 1
implements I
inherits C2
begin end
```

Versions and upgrades

- At runtime, classes have *version numbers* and *upgrade numbers*
- Upgrading a class directly or indirectly increases the version number

Making Dynamic Class Upgrades Type-Safe

- When can the upgrades be applied safely at runtime?
  - There may be *dependencies* between different upgrades
  - An upgrade may depend on earlier upgrades of the same class
  - An upgrade may depend on the upgrades of superclasses
  - An upgrade may depend on the upgrades of other classes
  - The object state must be upgraded *before* executing new code
- Ensure that execution remains type-safe when classes change asynchronously
  - E.g., a redefined class (C_3) supports its interfaces
  - Methods are available when called
- Even if upgrades are well-typed, runtime errors may still occur if upgrades are applied too early in the distributed setting

Type Analysis of Class Upgrades

- A program is type checked in a typing environment
- Runtime updates are type checked in a typing environment
- Consequently: the typing environment must **evolve** to reflect the evolution of the runtime program
- Sequence of typing contexts $\Gamma_0, \Gamma_1, \Gamma_2, \ldots$
- Type analysis of the original program in Γ_0
- Type analysis of an upgrade operation in Γ_i constructs Γ_{i+1}
- Approach: The type analysis uses a **type and effect system** which modifies the typing environment

Typing w/ Dependency Effects

- **Context** extended with *dependencies* Γ_d (class name + version)
- **Judgments** $\Gamma \vdash s \langle \Sigma \rangle$ where Σ is a set of dependencies
- $[\![v]\!]$ represents the dependency information for v

$$\text{(Var)}\ \frac{\Gamma(v) = T}{\Gamma \vdash v : T \langle [\![v]\!] \rangle} \qquad \text{(Get)}\ \frac{\Gamma(x) = T \quad \Gamma \vdash v : \mathrm{Label}[T] \langle \Sigma \rangle}{\Gamma \vdash v?(x) : ok \langle [\![x]\!] \cup \Sigma \rangle}$$

$$\text{(New)}\ \frac{\exists T' \in \mathrm{interfaces}(\Gamma_{\mathcal{C}}(C)) \cdot T' \preceq T}{\Gamma \vdash \mathrm{new}\ C(\,) : T \langle \{\langle C, curr(C, \Gamma)\rangle\} \rangle}$$

$$\text{(Class)}\ \frac{\begin{array}{c}\forall M \in \overline{\mathbf{with}\ I\ \overline{M}} \cdot \Gamma + [\mathrm{attr}(C)] + [\mathrm{caller} \mapsto_v I] \vdash M : ok \langle \Sigma^M \rangle \\ \forall I \in \overline{I} \cdot \mathrm{implements}(\Gamma_{\mathcal{C}}(\Gamma_V(self)), I)\end{array}}{\begin{array}{c}\Gamma + [\langle C, 0\rangle \mapsto_d \bigcup_{M \in \overline{M}} \Sigma^M \setminus \{\langle C, 0\rangle\}] \\ \vdash \mathbf{class}\ C\ \mathbf{implements}\ \overline{I}\ \mathbf{begin}\ \mathbf{inherits}\ \overline{C}\ \mathbf{var}\ \overline{f\ T}; \overline{\mathbf{with}\ I\ \overline{M}}\ \mathbf{end} : ok\end{array}}$$

Typing of Dynamic Class Constructs

(New-Class)

$$\frac{\Delta = [C \mapsto_C (\overline{C}, \overline{I}, \overline{T\ f}, \overline{M})] \qquad C \notin \mathrm{dom}(\Gamma^i_{\mathcal{C}}) \qquad \Gamma^i + \Delta + [\mathsf{this} \mapsto_v C] + \Delta' \vdash \textbf{class}\ C\ \textbf{implements}\ \overline{I}\ \textbf{begin inherits}\ \overline{C}\ \textbf{var}\ \overline{f\ T}; \overline{\textbf{with}\ I\ \overline{M}}\ \textbf{end} : ok}{\Gamma^i + \Delta + [\langle C, 1\rangle \mapsto_d \Delta'_d(\langle C, 0\rangle)] \vdash \textbf{new-class}\ C\ \textbf{implements}\ \overline{I}\ \textbf{begin inherits}\ \overline{C}\ \textbf{var}\ \overline{f\ T}; \overline{\textbf{with}\ I\ \overline{M}}\ \textbf{end} : ok}$$

(Class-Update)

$$\frac{\Gamma^i_{\mathcal{C}}(C) = (\overline{C}_1, \overline{I}_1, \overline{T_1\ f_1}, \overline{\textbf{with}\ I_1\ \overline{M}_1})\} \qquad \underline{n = curr(C, \Gamma^i_d)} \qquad \underline{\mathrm{refines}(\overline{M}_2, \overline{M}_1)} \qquad \Delta = [C \mapsto_C (\overline{C}_1; \overline{C}_2, \overline{I}_1; \overline{I}_2, (\overline{T_1\ f_1}; \overline{T_2\ f_2}), (\textbf{with}\ I_1\ \overline{M}_1 \oplus \textbf{with}\ I_2\ \overline{M}_2))] \qquad \Gamma^i + \Delta + [\mathsf{this} \mapsto_v C] + \Delta' \vdash \textbf{class}\ C\ \textbf{implements}\ \overline{I}_2\ \textbf{begin inherits}\ \overline{C}_2\ \textbf{var}\ \overline{f_2\ T_2}; \overline{\textbf{with}\ I_2\ \overline{M}_2}\ \textbf{end} : ok}{\Gamma^i + \Delta + [(C, n+1) \mapsto_d \Delta'_d(C, 0) \cup \{(C, n)\}] \vdash \textbf{update}\ C\ \textbf{implements}\ \overline{I}_2\ \textbf{begin inherits}\ \overline{C}_2\ \textbf{var}\ \overline{f_2\ T_2}; \overline{\textbf{with}\ I_2\ \overline{M}_2}\ \textbf{end} : ok}$$

After Type Analysis of an Upgrade Operation

- The type analysis gives us a new typing context for the analysis of the next upgrade operation
- The dependency mapping gives us the dependencies of an upgrade operation in terms of versions of other classes

At runtime

- Γ_d enforces an ordering of updates obeying static dependency requirements
  - Ensures appropriate timing for the application of each upgrade
  - Upgrades which do not depend on each other may be applied in any order (or in parallell)
- The requirements are used as an argument to the runtime upgrade

Semantics

Rough idea

- Upgrade messages are injected into the runtime configuration
- Messages propagate asynchronously
- Messages modify class representations when dependencies are resolved
- When to apply changes to objects: processor release!

An Operational Semantics for Class Upgrades

- Recall the operational semantics of Creol in rewriting logic
- The system configuration consists of classes, objects and messages
- Creol classes: $\langle C\#n : Cl \mid Upd : u, Inh : C'\#n'; \ldots, Att, Mtds\rangle$
- Creol objects: $\langle o : Ob \mid Cl : C\#n, Pr, PrQ, Att\rangle$
- Rewrite rules and equations transform sub-configurations

Class upgrade

Given an well-typed upgrade term: $upd(C, Imp, Inh, Var, Mtd)$

- A class upgrade of C is realized through the insertion of a message $upgrade(C, Inh, Var, Mtd, \Gamma_d(\langle C, curr(C, \Gamma_d^i)\rangle))$ in the system configuration at runtime
- Γ is the environment obtained from type checking the upgrade term

Direct class upgrade

$$upgrade(C, \mathrm{I}, \mathrm{A}, \mathrm{M}, ((C' \# n)\ \mathrm{R}))\ \langle C' \# n' : Class \mid Upd : u\rangle$$
$$\longrightarrow upgrade(C, \mathrm{I}, \mathrm{A}, \mathrm{M}, \mathrm{R})\ \langle C' \# n' : Class \mid Upd : u\rangle \textbf{ if } u \geq n$$

$$upgrade(C, \mathrm{I}, \mathrm{A}, \mathrm{M}, \emptyset)$$
$$\langle C \# n : Class \mid Upd : u, Inh : \mathrm{I}', Att : \mathrm{A}', Mtds : \mathrm{M}'\rangle$$
$$\longrightarrow$$
$$\langle C \# (n+1) : Class \mid Upd : u+1, Inh : \mathrm{I}'; \mathrm{I}, Att : \mathrm{A}'; \mathrm{A}, Mtds : \mathrm{M}' \oplus \mathrm{M}\rangle$$

Indirect class upgrade

$$\langle C \# n : Class \mid Inh : \mathrm{I}; (C' \# n'); \mathrm{I}'\rangle\ \langle C' \# n'' : Class \mid \rangle$$
$$= \langle C \# (n+1) : Class \mid Inh : \mathrm{I}; (C' \# n''); \mathrm{I}'\rangle\ \langle C' \# n'' : Class \mid \rangle \textit{ if } n'' > n'$$

Object upgrade

Objects are upgraded in *quiescent* states:

the processor has been released and no pending process is activated yet.

$$\langle o \mid Cl : C \;\#\; n, Pr : \varepsilon\rangle \; \langle C \;\#\; n' : Class \mid Att : \mathrm{A}\rangle$$
$$= \langle o \mid Cl : C \;\#\; n', Pr : \mathrm{idle}\rangle \; \langle C \;\#\; n' : Class \mid Att : \mathrm{A}\rangle$$
$$(getAttr(o, \mathrm{A}) \;\textbf{to}\; C) \quad \textbf{\textit{if}} \;\; n' > n$$

getAttr traverses the inheritance graph above C and collects the (new) object state, which is returned in a message *gotAttr*

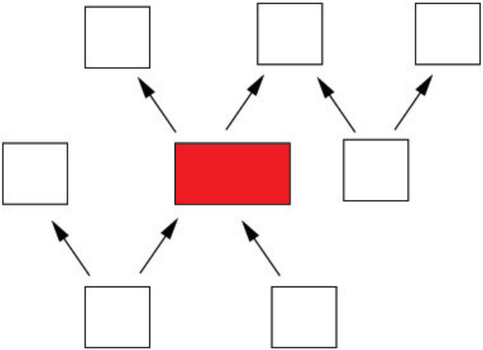
$$(gotAttr(\mathrm{A}') \;\textbf{to}\; o) \; \langle o \mid Att : \mathrm{A}\rangle = \langle o \mid Att : \mathrm{A}'\rangle$$

Type-Safe Upgrades

General case: Modify a class in a class hierarchy

Type correctness: Method binding should still succeed!

- Add attributes, methods, interfaces, superclasses
- Redefine methods (subtyping discipline)
- Remove fields, methods
- Remove interfaces: *not* supported
- Formal class parameters may *not* be modified



Theorem. Dynamic class extensions are type-safe in Creol's extended type system

Conclusion

- Formal framework for distributed concurrent objects
- Asynchronous method calls, interfaces, process scheduling, ...
- Operational semantics, rewriting logic, Maude
- Proof systems based on invariant reasoning
- System evolution through dynamic classes
- Use of static analyis for runtime constraints gives type safe upgrades
- Reasoning about dyn. classes: open issue!

http://www.ifi.uio.no/˜creol

Creol — Some Selected References

The communication model.

E. B. Johnsen, O. Owe. *An Asynchronous Communication Model for Distributed Concurrent Objects*. Software and System Modeling 6(1): 39-58, 2007.

F. S. de Boer, D. Clarke, E. B. Johnsen. *A Complete Guide to the Future*. Proc. ESOP'07. LNCS 4421, pp. 316–330. Springer 2007.

Multiple inheritance, method binding.

E. B. Johnsen, O. Owe. *A Dynamic Binding Strategy for Multiple Inheritance and Asynchronously Communicating Objects*. Proc. FMCO'04. LNCS 3657, pp. 274–295. Springer 2005.

Typing, static analysis.

E. B. Johnsen, O. Owe, I. C. Yu. *Creol: A Type-Safe Object-Oriented Model for Distributed Concurrent Systems*. Theoretical Computer Science 365: 23–66, 2006.

E. B. Johnsen, I. C. Yu. *Backwards Type Analysis for Asynchronous Method Calls*. J. of Logic and Algebraic Programming 77: 40-59, 2008.

Dynamic class upgrades.

E. B. Johnsen, O. Owe, I. Simplot-Ryl. *A Dynamic Class Construct for Asynchronous Concurrent Objects*. Proc. FMOODS'05. LNCS 3535, 15–30. Springer 2005.

I. C. Yu, E. B. Johnsen, O. Owe. *Type-Safe Runtime Class Upgrades in Creol*. Proc. FMOODS'06. LNCS 4037, 202–217. Springer 2006.

Analysis.

J. Dovland, E. B. Johnsen, O. Owe. *Observable Behavior of Dynamic Systems: Component Reasoning for Concurrent Objects*. Proc. FInCo'07. ENTCS 203. Elsevier 2008.

J. Dovland, E. B. Johnsen, O. Owe, M. Steffen. *Lazy Behavioral Subtyping*. Proc. FM'08. LNCS 5014. Springer 2008.

E. B. Johnsen, O. Owe, A. B. Torjusen. *Validating Behavioral Component Interfaces in Rewriting Logic*. Fundamenta Informaticae 82 (4): 341-359, 2008.

R. Schlatte, B. Aichernig, F. de Boer, A. Griesmayer, E. B. Johnsen. *Testing Concurrent Objects with Application-Specific Schedulers*. Proc. ICTAC'08. LNCS 5060, 319–333. Springer 2008