Creol: A Formal Model of Distributed Concurrent Objects

Einar Broch Johnsen

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

KeY symposium, Nomborn in Eisenbachtal, June 14 2007

Creol at a glance

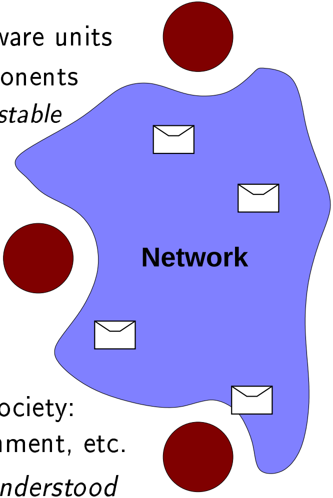
- an executable OO modelling language
- formally defined semantics in rewriting logic
- targets open distributed systems
- abstracts from the particular properties of
  the (object) scheduling and of the (network) environment
- the language design should support verification

Historical Note

- started as Norw. project Creol at UiO by Johnsen and Owe in 2004
- developed into an EU project Credo in 2006:
  W. Yi (Uppsala), C. Baier (Dresden), W-P de Roever (Kiel),
  B. Aicherning (Graz/Macao), F. de Boer (CWI) + industries
- Norw. project Connect 2006: active interfaces to connect objects

Open Distributed Systems

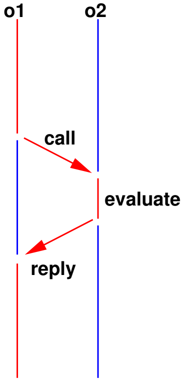
- Consider systems of communicating software units
- Distribution: geographically spread components
  - Networks may be *asynchronous* and *unstable*
- Components are unstable
  - Availability may vary over time
- Evolution: systems change at runtime
  - New requirements / bug fixes
  - Changing environments
  - Mars Rovers reprogrammed 11 times since landing on Mars!
- ODS *dominate* critical infrastructure in society: bank systems, air traffic control, e-government, etc.
- ODS: *complex*, *error prone*, and *poorly understood*



- **Creol / Credo project goal:**

  Formal object-oriented framework to model and reason about ODS

Object orientation: Remote Method Calls



RMI / RPC method call model

- Control threads follow call stack
- Derived from sequential setting
- Hides / ignores distribution!
- Tightly synchronized!

Creol :

- Show / exploit distribution!
- Asynchronous method calls
  - more efficient in distributed environments
  - *triggers* of concurrent activity
- Special cases:
  - *Synchronized communication:* the caller decides to wait for the reply
  - *Sequential computation:* only synchronized computation

Creol: A Concurrent Object Model

- Objects are *concurrent*, encapsulating a processor
- Object variables are *typed by interfaces*
- *No assumptions* about the (network) environment
- Execution in objects should be *flexible*
  - Adapt to delays in the environment
  - **Implicit scheduling** between internal processes inside an object
  - High-level program *flexibility* w.r.t. the environment:
    no need for explicit signaling or thread declarations
  - Process control by *suspension points*
  - Combines *active* and *passive/reactive* behavior
- **Method invocations**: *synchronous* or *asynchronous*
- **Dynamic reprogramming**: Class definitions may *evolve at runtime*

Interfaces as types

- Object variables (pointers) are *typed by interfaces*
  (other variables are typed by data types)
- *Mutual dependency*: An interface may require a *cointerface*
  - Explicit keyword *caller*
  - Supports callbacks to the caller through the cointerface
  - Protocol-like behaviour
- Supports *strong typing*: no "method not understood" errors
- All object interaction is *controlled* by interfaces
  - *No explicit hiding* needed at the class level
  - Interfaces provide aspect-oriented specifications
  - A class may implement a number of interfaces

Example: Authorization Policies (1)

Let interface *Auth* offer methods *grant*, *revoke*, *auth*, and *delay*.

```
interface Auth
begin
with Any                      // cointerface
  op grant(in x:Agent)        // grant authorization to agent x
  op revoke(in x:Agent)       // revoke authorization from agent x
  op auth(in x:Agent)         // check that agent x is authorized
  op delay                    // delay until no agent is authorized
end
```

Internal Processes in Concurrent Objects

- **Process**: code + local variable bindings (method activation)
- **Object**: *state* + *active* process + *suspended* processes
- **Suspension** by means of await statements: **await** *guard*
- **Guards** are combinations of:
  - $wait \in$ Guard (explicit release)
  - $l? \in$ Guard, where l : Label
  - $\phi \in$ Guard, where ϕ : *Local state* $\rightarrow$ Bool
- Inner guards are allowed: $\ldots;$ **await** $g; \ldots$
- If g evaluates to false the active process is *suspended*, with its *local variable bindings*
- If no process is active, any suspended process may be *activated* if its guard evaluates to true.
- Inner guards enable *interleaving* of *active* and *reactive* code
- Remark: No need for signaling / notification / pulse

Object Communication in Creol

- Objects communicate through method invocations *only*
- Methods organized in classes, seen externally via interfaces
- *Different ways to invoke* a method m
- Decided by caller — *not* at method declaration
- **Asynchronous** invocation: $l!o.m(In)$
- **Passive waiting** for method result: **await** $l?$
- **Active waiting** for method result: $l?(Out)$
- **Guarded** invocation: $l!o.m(In); \ldots;$ **await** $l?; l?(Out)$
- Label free abbreviations for standard patterns:
  - $o.m(In; Out) = l!o.m(In); l?(Out)$ — **synchronous call**
  - **await** $o.m(In; Out) = l!o.m(In);$ **await** $l?; l?(Out)$
  - $!o.m(In)$ — no reply needed
- **Internal calls:** $m(In; Out)$, $l!m(In)$, $!m(In)$
  Internal calls may also be asynchronous/guarded

Some Remarks

Asynch. mtd. calls useful to combine OO + distribution:

- Synchronous calls defined by asynchronous calls
- Extends the notion of *future variables* [Yonezawa86, . . .]:

$$l!m(In); \ldots; l?(Out)$$
$$l!m(In); \ldots; \textbf{await } l?; \ldots; l?(Out)$$

- Provides the *efficiency* of message passing
- All inter-object communication by method calls,
  no need for separate concept of message
- Any method may be called *synchronously* or *asynchronously*
- Cointerfaces: mutual dep. / callback / availability restriction
- *Inheritance will be as usual for OO:*
  may inherit/redefine methods in subclasses

Creol Language Constructs

Syntactic categories.

l in Label
g in Guard
p in MtdCall
S in ComList
s in Com
x in VarList
e in ExprList
m in Mtd
o in ObjExpr
ϕ in BoolExpr

Definitions.

$g ::= wait \mid \phi \mid l? \mid g_1 \wedge g_2$
$p ::= o.m \mid m$
$S ::= s \mid s; S$
$s ::= \textbf{skip} \mid (S) \mid S_1 \Box S_2 \mid S_1 \| S_2$
$\mid x := e \mid x := \textbf{new}\ classname(e)$
$\mid \textbf{if}\ \phi\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}$
$\mid\ !p(e) \mid l!p(e) \mid l?(x) \mid p(e; x)$
$\mid \textbf{await}\ g \mid \textbf{await}\ l?(x) \mid \textbf{await}\ p(e; x)$

Example: Combining Authorization Policies (2)

Let classes *SAuth* and *MAuth* define two authorization strategies implementing *Auth*.

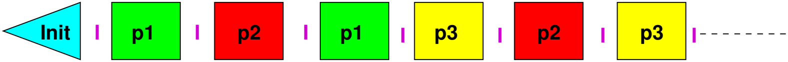
class *SAuth* **implements** Auth
begin **var** *gr*: Agent = null
with Any
  op grant(**in** x:Agent) == delay; *gr* := x
  op revoke(**in** x:Agent) == **if** *gr* = x **then** *gr* := null **fi**
  op auth(**in** x:Agent) == **await** (*gr* = x)
  op delay == **await** (*gr* = null)
end

Let classes *SAuth* and *MAuth* define two authorization strategies implementing *Auth*.

class *MAuth* **implements** Auth
begin **var** *gr*: Set[Agent] = $\emptyset$
with Any
  op grant(**in** x:Agent) == $gr := gr \cup \{x\}$
  op revoke(**in** x:Agent) == $gr := gr \setminus \{x\}$
  op auth(**in** x:Agent) == **await** $(x \in gr)$
  op delay == **await** $(gr = \emptyset)$
end

Reasoning about Creol Objects

- Observation: All object interaction is by means of method calls
- Let us consider a local execution in an object



- Basic idea for the proof theory

Objects as maintainers of local invariants i

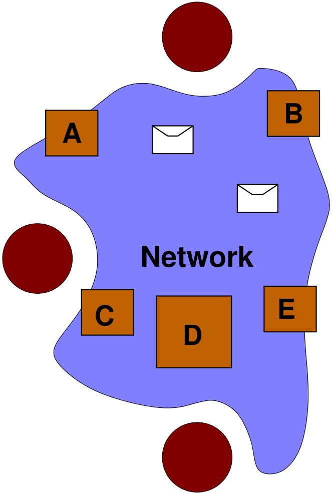
- Standard proof rules
- Rule for await

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

- For *method calls*, we must rely on the interface
  (the class of an object is not statically known)
- Annotate interfaces with pre/postconditions on methods
- For more precise characterizations, we may rely on the
  local history of observable communication
- the *soundness* and *completeness* of the proof system for partial correctness may be shown by
  - an encoding into a standard sequential language (e.g., Apt)
  - extended with a nondeterministic assignment operator
- The completeness is here relative to a sufficiently strong local invariant

Dynamic Classes in Creol

- Dynamic classes: *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



Example of a Class Upgrade: The Good Bank Customer (1)

```
class BankAccount implements Account          -- Version 1
begin var bal : Int = 0
with Any
   op deposit (in sum : Nat) == bal := bal+sum
   op transfer (in sum : Nat, acc : Account) ==
      await bal ≥ sum ; bal := bal−sum; acc.deposit(sum)
end
upgrade class BankAccount
begin var overdraft : Nat = 0
with Any
   op transfer (in sum : Nat, acc : Account) ==
      await bal ≥ (sum−overdraft); bal := bal−sum;
         acc.deposit(sum)
with Banker
   op overdraft_open (in max : Nat) == overdraft := max
end
```

Example of a Class Upgrade: The Good Bank Customer (2)

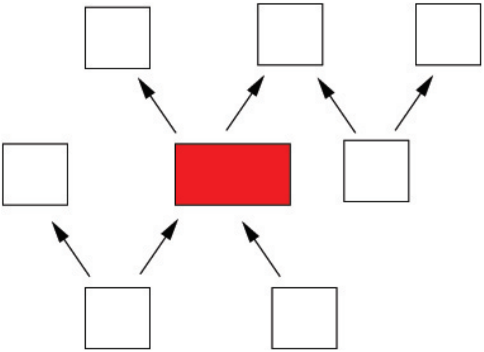
```
class BankAccount implements Account          -- Version 2
begin var bal : Int = 0, overdraft : Nat = 0
with Any
  op deposit (in sum : Nat) == bal := bal+sum
  op transfer (in sum : Nat, acc : Account) ==
    await bal ≥ (sum−overdraft); bal := bal−sum;
      acc.deposit(sum)
with Banker
  op overdraft_open (in max : Nat) == overdraft := max
end
```

A Dynamic Class Mechanism

General case: Modify a class in a class hierarchy

Type correctness: Method binding should still succeed!

- Attributes may be added (no restrictions)
- Methods may be added (no restrictions)
- Methods may be redefined (subtyping discipline)
- Superclasses may be added
- Formal class parameters may *not* be modified



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

Present and Future Work

- Operational semantics in rewriting logic
- Maude interpreter
- Type system
- Dependent upgrades
- Distributed interpreter running on JVM
- Reasoning support
- Parametrization, packages, components, ...
- Testing / Validation / Lightweight verification
- Web services / XML

Most papers available from

`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. *Inheritance in the Presence of Asynchronous Method Calls*. Proc. HICSS-38. IEEE, 2005.

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*. Submitted to journal, 2007.

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. *Verification of Concurrent Objects with Asynchronous Method Calls*. Proc. SwSTE, 141–150. IEEE, 2005.

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

E. B. Johnsen, O. Owe, A. B. Torjusen. *Validating Behavioral Component Interfaces in Rewriting Logic*. Fundamenta Informaticae 2007. To appear.