Creol as formal model for distributed, concurrent objects

Martin Steffen

IfI UiO

Flacos, Malta

27. November 2008
Structure

Creol
  Distributed Communication in Creol
  Basic Language Constructs

Open semantics and observable interface behavior

Dynamic Class Upgrades

Lazy behavioral subtyping
  Subtyping, late binding, and incremental program development
  Examples
  Basic idea

Conclusion
Creol: a concurrent object model

- executable oo modelling language **concurrent** objects
- formal semantics in **rewriting logics** /Maude
- strongly **typed**
- method invocations: synchronous or **asynchronous**
- targets open distributed systems
- recently: concurrent objects by (first-class) futures/promises
- **dynamic reprogramming**: class definitions may *evolve at runtime*
- the language design should support verification
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

(Flacos’08)
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(ln)`
- *Passive waiting* for method result: `await l?`
- *Active waiting* for method result: `l?(Out)`
- *Guarded* invocation: `l!o.m(ln); . . . ; await l?; l?(Out)`
Language Constructs

Syntactic categories.

Definitions.

\[ g ::= \text{wait} | \phi | l? | g_1 \land g_2 \]

\[ p ::= o.m | m \]

\[ S ::= s | s; S \]

\[ s ::= \text{skip} | (S) | S_1 \circ S_2 | S_1 \parallel S_2 \]

\[ x ::= e | x ::= \text{new} \text{classname}(e) \]

\[ \text{if } \phi \text{ then } S_1 \text{ else } S_2 \text{ fi} \]

\[ e ::= !p(e) | l!p(e) | l?(x) | p(e; x) \]

\[ m ::= \text{await } g | \text{await } l?(x) | \text{await } p(e; x) \]

...
Futures

- introduced in the concurrent Multilisp language [7] [2]
- originally: transparent concurrency compiler annotation
- future e:
  - evaluated potentially in parallel with the rest $\Rightarrow$ 2 threads (producer and consumer)
  - future variable dynamically generated
  - when evaluated: future identified with value
- supported by Oz, Alice, MultiLisp, ... (shared state concurrency), Io, Joule, E, and most actor languages (Act1/2/3 ..., ASP), Java
Async. method calls and futures
**Syntax**

- $o@/v$: asynchronous method call, non-blocking

**execution:**

1. create a “placeholder”/reference to the eventual result: future reference (“label”)
2. initiate execution of method body
3. continue to execute (= non-blocking, asynchronous)

$$ e ::= \ldots | o@/v, \ldots, v | \text{claim}@/(n, o) | \text{get}@n | \ldots $$
Claiming a future

$$t_2 = v$$

$$t_2 \neq v$$

$$\text{release}$$

$$\text{get}$$

$$\text{claim}$$

$$\text{grab}$$
Futures and promises

- terminology is not so clear
- relation to handled futures
- promises [9], I-structures [1]

⇒ 2 aspects of future var:
  - write = value of e “stored” to future
  - read by the clients

- promises: separating the creation of future-reference from attaching code to it\(^1\)
- good for delegation

\(^1\)as in for async. calls
Syntax (promise)

- instead of $o@l(\vec{v})$
- split into
  1. create a promise\(^2\)
  2. fulfill the promise = bind code to it.

\[ e ::= \ldots \mid \text{promise } T \mid \text{bind } o.l(\vec{v}) : T \leftrightarrow n \mid \ldots \]

\(^2\)or a handle to the future.
\[
n'\langle\text{let } x: T' = \text{promise } T \text{ in } t\rangle \rightsquigarrow \nu(n:T').(n'\langle\text{let } x: T' = n \text{ in } t\rangle)
\]

\[
\ldots n_1\langle\text{let } x: T = \text{bind } o.l(\vec{v}) : T_2 \leftrightarrow n_2 \text{ in } t_1\rangle \overset{\tau}{\rightarrow} \\
\ldots n_1\langle\text{let } x: T = n_2 \text{ in } t_1\rangle \\
\| (n_2\langle\text{let } x: T_2 = \text{grab}(o); M.l(o)(\vec{v}) \text{ in } \text{release}(o); x\rangle)
\]
Open semantics and observable interface behavior

Interface description: Task

• characterize possible interface behavior
• possible = adhering to the restriction of the language
  • well-typed
• basis of a trace logic / interface description
• abstraction process:
  • not $C \xrightarrow{t} \hat{C}$?
  • rather: consider $C$ in a context / environment

\[ C \parallel E \xrightarrow{t} \hat{C} \parallel \hat{E} \]

for some environment $E$

⇒ open semantics

\[ \Delta \vdash C : \Theta \xrightarrow{t} \hat{\Delta} \vdash C : \hat{\Theta} \]

• assumptions $\Delta$ abstracts environments $E$

(Flacos’08) Creol as formal model for distributed, concurrent objects
Open semantics and observable interface behavior

One step further: legal traces

- open semantics

\[ \Delta \vdash C : \Theta \xrightarrow{t} \Delta' \vdash C : \Theta' \]

abstracts the environment

- existential abstraction of component, as well:

- characterization of *principally possible* interface behavior

\[ C \parallel E \xrightarrow{t} \dot{C} \parallel \dot{E} \]

for some component \( C \) + some environment \( E \)

\[ \Rightarrow \text{legal trace} \]

\[ \Delta \vdash t : \text{trace} :: \Theta \]
Behavioral interface description

- type system for futures, especially resource aware (linear) type system for promises
- standard soundness results (subject reduction, \ldots)
- formulation of an open semantics plus characterization of possible interface behavior by abstracting the environment
- soundness of the abstractions
- basis for testing Creol objects/components
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
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!

(Flacos’08) Creol as formal model for distributed, concurrent objects 2008 18 / 32
Example of a Class Upgrade: The Good Bank Customer (1)

class BankAccount implements Account

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
Substitutability and subtype polymorphism

Problem:
When can some expression $e_1$ replace some other expression $e_2$?
classical answer: subtyping

Example 1: Assignment

$$x := e \quad \frac{\Gamma \vdash e : T \quad T \leq \Gamma(x)}{\Gamma \vdash x := e : ok}$$

Example 2: Method Calls

$$x := m(e) \quad m : T_1 \rightarrow T_2$$

Want: $m(e)$
Get: $m'(e)$

(contravariance) $m' : T'_1 \rightarrow T'_2$ (covariance)

(contravariance) $m' : T'_1 \rightarrow T'_2$ (covariance)
Behavioral subtyping

Extend subtyping to **behavioral properties:**

“any property proved about supertype objects also holds for subtype objects” [Liskow & Wing 94]

Consider an assertion language on local state variables, a programming language, and some program logic.

Assertions $p_1, p_2, q_1, q_2, \ldots$ used for pre- and postconditions

**When can we replace $e_1$ by $e_2$?**

$$\{p_1\} e_1 \{q_1\}$$  \hspace{1cm} **Applicability:**  $p_1 \Rightarrow p_2$ (ref. contravariance)

$$\{p_2\} e_2 \{q_2\}$$  \hspace{1cm} **Predictability:**  $q_2 \Rightarrow q_1$ (ref. covariance)
Late Binding of Method Calls

Object-oriented programming

- incremental program development
- **Substitutability** is exploited to organize programs by means of *inheritance*
  - *object substitutability*: a subclass object may be bound to a superclass variable
  - *method substitutability* (late binding): subclass methods may be selected instead of superclass methods

Late binding of method calls

- code bound to a call depends on the **actual** class of the object
- decided at **runtime**
- Not statically decidable
Example

```java
class C {
    m() {...}
    n() {...; m(); ...}
}

class D extends C {
    m() {...}
}
```

- the binding of `m()` depends on the *actual class of the object*
- Incremental development: *the class D may be added later*
- *late binding and incremental development pose a challenge for program verification*
Verifying late-bound method calls

- two main approaches in the literature

  - **Open world** [America 91, Liskow & Wing 94, Leavens & Naumann 06, ...]
    - Behavioral subtyping: supports incremental reasoning
    - Subtyping constraints: too restrictive in practice

  - **Closed world** [Pierik & de Boer 05, ...]
    - Complete reasoning method
    - Breaks incremental reasoning

- **Lazy behavioral subtyping** [6]
  - supports incremental reasoning
  - less restrictive than behavioral subtyping
Example: Closed World Approach

```java
class C {
    m():(p_1, q_1) { ... }
    n() { ... ; \{p\}m()\{q\} ; ... }
}

class D extends C {
    m():(p_2, q_2) { ... }
}
```

Closed world approach

- Assumes all commitments of a method known at reasoning time
- Sufficiently expressive: complete reasoning system
- **redo** proofs if a new class is added to the program
- breaks with incremental development principle (proof reuse)

Commitment (declaration site)

Requirement (call site)

PO: \( p \Rightarrow p_1 \land p_2, q_1 \lor q_2 \Rightarrow \)
Example: Open World Approach

```java
class C {
    m(): (p_1, q_1) { ... } 
    n() { ... ; {p}m() {q}; ... }
}
```

**Commitment (declaration site)**

**Requirement (call site)**

**PO:** $p \Rightarrow p_1, q_1 \Rightarrow q$

```java
class D extends C {
    m(): (p_2, q_2) { ... }
}
```

**Commitment (declaration site)**

**PO:** $p_1 \Rightarrow p_2, q_2 \Rightarrow q_1$

### Behavioral subtyping

- $(p_1, q_1)$ acts as a commitment (contract) for declarations of $m$
- Redefinitions relate to the contract, not to the call site
- **Incremental:** Proof reuse when the program is extended
- **Restriction:** $(p_1, q_1)$ too strong requirement for redefinitions

(Flacos'08)

Creol as formal model for distributed, concurrent objects
Lazy behavioral subtyping

Examples

Example: Lazy Behavioral Subtyping

class C {
    m(): (p₁, q₁) { ... }
    n() { ... ; {p}m() {q}; ... }
}

Commitment (declaration site)
Requirement (call site)
PO: p ⇒ p₁, q₁ ⇒ q

class D extends C {
    m(): (p₂, q₂) { ... }
}

Commitment (declaration site)
PO: p ⇒ p₂, q₂ ⇒ q

Lazy behavioral subtyping

- POs depend on requirements, not on commitments (contracts)
- irrelevant parts of old commitments may be ignored
- more flexible than behavioral subtyping approach
- incremental: proof reuse when program is extended

(Flacos’08) Creol as formal model for distributed, concurrent objects 2008 28 / 32
Lazy Behavioral Subtyping

- Distinguish method **use** and method **declarations**
- **track** call site requirements and declaration site commitments
- Proof **reuse**: Impose these requirements on method overridings in new subclasses to ensure that **old proofs** remain valid
- Declaration site proof obligations wrt. superclass’ requirements
  - Many, but **weaker** POs than with behavioral subtyping for superclass declarations
- Formalize how commitments and requirements propagate as subclasses and proof outlines are added
  - Proof environment tracks commitments and requirements
  - Syntax-driven inference system for program analysis
  - Independent of a particular program logic

(Flacos’08) Creol as formal model for distributed, concurrent objects
Conclusion and prospect

- testing Creol-components
- FP7 project HATS “highly-adaptable and trustworthy software”
  - software evolution
  - software families
References I

I-structures: Data-structures for parallel computing.

The incremental garbage collection of processes.

Service, asynchrony and wait-by-necessity.

Towards a method of object-oriented concurrent programming.

Concurrent objects à la carte.

Lazy behavioral subtyping.

Multilisp: A language for concurrent symbolic computation.

Creol: A type-safe object-oriented model for distributed concurrent systems.
References II

Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. 

Components, objects, and contracts. 

A formal language for electronic language. 