

# An abstract behavioral model of distributed concurrent objects (1)

Einar Broch Johnsen

Dept. of Informatics, University of Oslo  
Email: [einarj@ifi.uio.no](mailto:einarj@ifi.uio.no)

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

Viinistu, Estonia, Jan 27 2009

## Creol at a glance

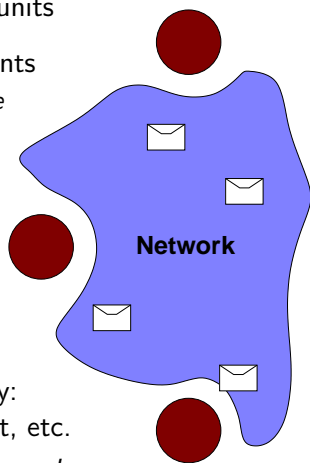
- ▶ An executable OO modelling language
- ▶ Formally defined semantics in rewriting logic
- ▶ Targets open distributed systems
- ▶ Abstracts from the particular properties of the (object) scheduling and of the (network) environment
- ▶ The language design should support verification

### Talk Overview

- ▶ Distributed concurrent objects in Creol
- ▶ Semantics and execution platform
- ▶ Reasoning about Creol models
- ▶ Runtime evolution of Creol models

# Open Distributed Systems

- ▶ Consider systems of communicating software units
- ▶ **Distribution**: geographically spread components
  - ▶ Networks may be *asynchronous* and *unstable*
  - ▶ Component 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*



# Objects, Concurrency, Distribution

## Goal:

Formal object-oriented framework to model and reason about ODS

## Design issues

- ▶ **Interacting software units**: objects exchanging method calls
- ▶ **Concurrency**: threads, race conditions, scheduling, ...
- ▶ **Distribution**: data sharing and synchronization between objects
- ▶ **Openness**: Software units provided by other parties, upgrades, ...

# High-Level Executable Modeling

Specification level	Specification formalism examples
Design-oriented	UML/OCL, State diagrams
Abstract behavioral	Creol
Implementation-oriented	SPEC#, Java+JML
Implementation	Java, ...

Levels of abstraction in (models of) object-oriented software systems

- ▶ **OO**: the dominant paradigm today
- ▶ Model between a **design** and an **implementation** language
- ▶ **Abstraction**: Nondeterministic behavior
  - ▶ Abstracts from specific network solutions and protocols
  - ▶ Abstracts from specific scheduling policies in objects
- ▶ **Abstraction**: Local computation uses ADTs (no rep. objects)
- ▶ **Executable semantics**, directly supports simulation / testing
- ▶ Abstract behavioral specifications will be further developed in **Hats!**

# Shared Variables and Concurrency (1)

⟨**thread 1** :  $x := x + 2$ ⟩

⟨**thread 2** :  $x := x - 3$ ⟩

•  $x = 5$

What are the possible values for  $x$  after execution?

- ▶  $x = 4$
- ▶  $x = 2$
- ▶  $x = 7$

To control the outcomes of the execution, we need a **locking discipline!**

## Concurrency in the OO setting

**Qualified name space:** Threads operate on the object state

**thread 1** :  $o.x := o.x + 2$

•  $o.x = 5$

**thread 2** :  $o.x := o.x - 3$

**Methods:** Threads correspond to method activations

class  $C$

begin

var  $x : T := 5$

op **m1**() ==  $x := x + 2$

•  $o.x = 5$

op **m2**() ==  $x := x - 3$

end

- ▶ May have a **dynamic number** of method activations!
- ▶ To regain control, we need a locking discipline again!

## Shared Variables and Concurrency (2)

- ▶ **Concurrent systems**
- ▶ Challenging to maintain a consistent state!
- ▶ Necessary for **reasoning control**
- ▶ Solution: **locks**
  - ▶ **Interleaved atomic sections**: less non-determinism
  - ▶ Locks enforce very **low-level** style of programming!
  - ▶ Easy to make **mistakes**
  - ▶ Semaphores, monitors
  - ▶ Problem to ensure enough signalling (try to prove this!)
- ▶ **“Synchronized” methods** (Java)
  - ▶ The (synchronized) threads use a **per-object object**
  - ▶ Not all methods need to be synchronized
  - ▶ Very coarse-grained
  - ▶ What happens if a method gets “stuck”?

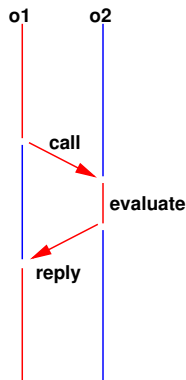


## Shared Variables and Concurrency (3)

- ▶ Modern approaches are more high-level
- ▶ Harder to make mistakes, easier for the programmer (?)
- ▶ **Ownership systems**
- ▶ **Transactional memory systems**
- ▶ **Distribution still a challenge**
  - ▶ Do we keep multiple copies of objects in the state?
  - ▶ Ownership, transactional memory, locking disciplines. . . can be used to allow under-the-hood sharing
  - ▶ We here abstract from data sharing in the distributed setting
  - ▶ Focus on concurrency and synchronization aspects
- ▶ **Distributed systems based on message passing**
  - ▶ No shared state between different objects
  - ▶ All interaction by (asynchronous) message passing
  - ▶ But what about method calls?
  - ▶ . . . and synchronization?

# Method Calls and Distribution

## Our approach



- ▶ Show / exploit distribution!
- ▶ **Asynchronous method calls**
  - ▶ more efficient in distributed environments
  - ▶ *triggers* of concurrent activity
- ▶ **Special cases:**
  - ▶ *Asynchronous message sending:*  
the caller doesn't care about the reply
  - ▶ *Synchronized communication:*  
the caller decides to wait for the reply
  - ▶ *Sequential computation:*  
only synchronized computation

# Active and Passive Objects

## Passive objects

- ▶ Execute their methods in the caller's thread of control (e.g., Java)
- ▶ In multithreaded applications, we must take care of proper synchronization
- ▶ If two objects call the same object, race condition may occur

## Active (or concurrent) objects

- ▶ Execute their methods in their own thread of control (e.g., Actors)
- ▶ Communication is asynchronous
- ▶ Call and return are decoupled (in Creol: future variables)
- ▶ Usually, active objects use cooperative multitasking
- ▶ Cooperative multitasking is specified using schedulers

# Creol: A Concurrent Object Model

## Some underlying assumptions

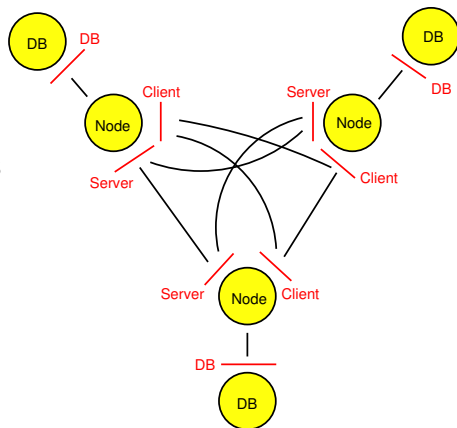
- ▶ All objects are **active** (or concurrent), but may receive requests
  - ▶ Need easy way to combine **active** and **passive/reactive** behavior
- ▶ We don't always know how objects are implemented
  - ▶ Separate **specification** (interface) from **implementation** (class)
  - ▶ Object variables are **typed by interface**, not by class
- ▶ **No assumptions** about the (network) environment
  - ▶ Communication may be unordered
  - ▶ Communication may be delayed
  - ▶ Execution should adapt to possible delays in the environment
- ▶ Execution in objects should be **flexible**
- ▶ Synchronization should be decided by the caller
  - ▶ **Method invocations** may be **synchronous** or **asynchronous**
- ▶ Systems **evolve at runtime**
  - ▶ Need support for **dynamic reprogramming**

## Interfaces as types

- ▶ Object variables (pointers) are *typed by interfaces* (other variables are typed by data types)
- ▶ 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
- ▶ *Mutual dependency*: An interface may require a *cointerface*
  - ▶ Explicit keyword *caller*
  - ▶ Supports callbacks to the caller through the cointerface
  - ▶ Protocol-like behaviour
- ▶ **Type safety**: no “method not understood” errors

## Example: A Peer To Peer Network

- ▶ Consider a P2P file sharing network
- ▶ Nodes in the network exchange files
- ▶ Each node may be involved in several uploads/downloads simultaneously
- ▶ Nodes may appear, disappear, and reappear
- ▶ Interfaces describe the different aspects of the system:  
**DB**, **Client**, **Server**



## P2P Example: The Data Base

The database stores file locally for a **Server** node and provides a **DB** interface

- ▶ `getFile`: retrieve file from database
- ▶ `storeFile`: store file in database
- ▶ `listFile`: list available files in database
- ▶ `getLength`: get length of given file

**interface DB**

**begin**

**with Server**

**op** `getFile`(**in** fld: String; **out** file: List[String])

**op** `getLength`(**in** fld: String; **out** length: Int)

**op** `storeFile`(**in** fld: String, file: List[String])

**op** `listFiles`(**out** fSet: Set[String])

**end**

## P2P Example: The Client

The **Client** interface of the nodes provides services to other **Clients**

- ▶ availFiles: which files are available in P2P network?
- ▶ reqFile: request a given file from a given server

```
interface Client
```

```
begin
```

```
  with Client
```

```
    op availFiles (in sList: List[Server]; out files: List[[Server, Set[String]]])
```

```
    op reqFile(in sld: Server, fld: String)
```

```
end
```



## P2P Example: The Server

A **Server** interface of the nodes provides services to other **Servers**

- ▶ enquire: list available files at the given server
- ▶ getLength: length of a given file in the server
- ▶ getPack: get a part of a given file from the server

```
interface Server
```

```
begin
```

```
  with Server
```

```
    op enquire(out files: Set[String])
```

```
    op getLength(in fld: String; out lth: Int)
```

```
    op getPack(in fld: String, pNbr: Int; out pack: String)
```

```
end
```

## P2P Example: The Node

A **Peer** is both a **Server** and a **Client**

```
interface Peer  
  inherits Client, Server  
begin end
```

## Object Communication in Creol

- ▶ Objects communicate through method invocations *only*
- ▶ *Different ways to invoke* a method *m*
- ▶ Decided by caller — *not* at method declaration time
- ▶ **Asynchronous** invocation: *!o.m(In)*
- ▶ **Polling** for method result: **await** *l?*
- ▶ **Reading** the method result: *l?(Out)*
- ▶ **Guarded** invocation: *!o.m(In); ...; await l?; l?(Out)*
- ▶ **Label free abbreviations** for standard patterns:
  - ▶ *o.m(In; Out) = !o.m(In); l?(Out)* — **synchronous call**
  - ▶ **await** *o.m(In; Out) = !o.m(In); await l?; l?(Out)*
  - ▶ *!o.m(In)* — no reply needed
- ▶ **Internal calls:** *m(In; Out)*, *!m(In)*, *!m(In)*  
Internal calls may also be asynchronous/guarded

## Async. mtd. calls useful to combine OO with distribution

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

$$!m(In); \dots; !?(Out)$$

$$!m(In); \dots; \mathbf{await} \ !?; \dots; !?(Out)$$

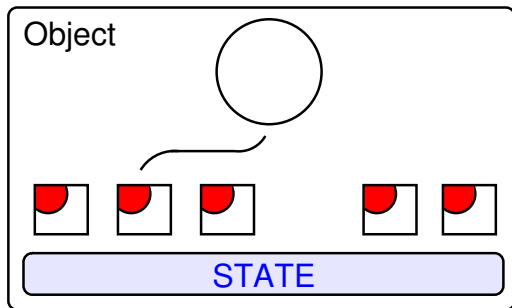
- ▶ Provides the *efficiency* of async. message passing
- ▶ All inter-object communication by method calls,  
*no need for a separate concept of message*
- ▶ Any method may be called *synchronously* or *asynchronously*
- ▶ Labels are first-class values and may be passed around

$$!m(In); \dots; !o.p(l); \dots; !?(Out)$$

- ▶ **Cointerfaces**: mutual dep. / callback / availability restriction
- ▶ **Inheritance will be as usual for OO**:  
may inherit/redefine methods in subclasses

## Execution inside a Creol Object

- ▶ **Concurrent objects** encapsulate a processor
- ▶ Execution in objects should *adapt* to environment delays
- ▶ At most *one active process* at a time
- ▶ *Implicit scheduling* between internal processes inside an object



# 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:
  - $l? \in \text{Guard}$ , where  $l$  : Label
  - $\phi \in \text{Guard}$ , where  $\phi$  : *Local state*  $\rightarrow$  Bool
- ▶ If  $g$  evaluates to false the active process is *suspended*
- ▶ 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

# Creol Language Constructs

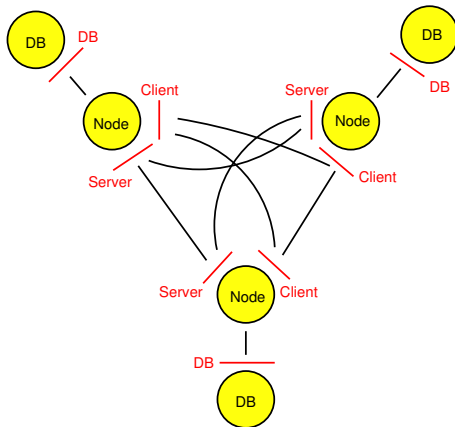
*Syntactic categories.*    *Definitions.*

$l$  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  
 $b$  in BoolExpr

$g ::= \phi \mid l? \mid g_1 \wedge g_2$   
 $p ::= o.m \mid m$   
 $S ::= s \mid s; S$   
 $s ::= \text{skip} \mid \text{begin } S \text{ end} \mid S_1 \square S_2$   
 $\mid x := e \mid x := \text{new } \textit{classname}(e)$   
 $\mid \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ end}$   
 $\mid \text{while } b \text{ do } S \text{ end}$   
 $\mid !p(e) \mid !!p(e) \mid l?(x) \mid p(e; x)$   
 $\mid \text{await } g \mid \text{await } p(e; x)$   
 $\mid \text{release}$

- ▶ Omit the functional language for expressions  $e$  here:  
 this, **caller**, strings, integers, lists, sets, maps, etc

## Recall the P2P Example



We now look at how to model the classes **DataBase** and **Node**.



## The Database

**interface** DB

**begin with** Server

**op** getFile(**in** fld: String; **out** file: List[String])

**op** getLength(**in** fld: String; **out** length: Int)

**op** storeFile(**in** fld: String, file: List[String])

**op** listFiles(**out** fSet: Set[String])

**end**

### One possible implementation of the DB interface:

**class** DataBase(db : Map[String, List[String]]) **implements** DB

**begin with** Server

**op** getFile(**in** fld: String; **out** file: List[String]) == file := get(db, fld)

**op** getLength(**in** fld: String; **out** length: Int) ==  
length := #(get(db, fld))

**op** storeFile(**in** fld: String, file: List[String]) ==  
db := insert(db, fld, file)

**op** listFiles(**out** fSet: Set[String]) == fSet := keys(db)

**end**

## The Nodes

```

class Node(db: DB, admin:P2P, file:String) implements Peer
begin
  var catalog : List[[Server, Set[String]]];

  op findServer(in fld:String, catalog : List[[Server, Set[String]]];
                out server:Server)==
    if isempty(catalog) then server := null
    else if (fld in snd(head(catalog))) then server := fst(head(catalog))
    else findServer(fld, tail(catalog); server) end end

  op run ==
    var l:Label[List[[Server, Set[String]]]];
    var neighbors: List[Server]; var server:Server;
    admin.getNeighbors(;neighbors);
    !this.availFiles(neighbors); await l?; l?(catalog);
    findServer(file,catalog; server); this.reqFile(server,file;)

```

## Implementing the Server interface

### with Server

```
op enquire(out files: Set[String]) == await db.listFiles(; files)
op getLength(in fld: String; out lth: Int) == await db.getLength(fld; lth)
op getPack(in fld: String, pNbr: Int; out pack: String) ==
  var f: List[String]; await db.getFile(fld; f); pack := nth(f, pNbr)
```

## Implementing the Client interface

### with Client

```

op availFiles (in sList: List[Server]; out files: List[[Server, Set[String]]]) ==
  var l1: Label[Set[String]]; var l2: Label[List[[Server, Set[String]]]] ;
  var fList: Set[String];
  if (sList = nil) then files := nil
  else l1!head(sList).enquire(); l2!this.availFiles(tail(sList));
    await l1?  $\wedge$  l2?; l1?(fList); l2?(files);
    files := files  $\vdash$  (head(sList), fList) end

op reqFile(in sld: Server, fld: String) ==
  var file: List[String] := nil; var pack: String; var lth: Int;
  await sld.getLength(fld; lth);
  while (lth > 0) do await sld.getPack(fld, lth; pack);
    file := pack  $\dashv$  file; lth := lth - 1 end;
  !db.storeFile(fld, file)

end

```

Semantics defined in **Rewriting Logic** (RL) [Meseguer 92]:

- ▶ *Static specification*: algebraic specification with equations  $E$
- ▶ *Dynamic specification*: (conditional) transition rules:

*subconfiguration*  $\longrightarrow$  *subconfiguration if condition*

- ▶ Rules applicable to non-overlapping subconfigurations, can be executed in a *concurrent rewrite step*

$$\frac{[t_1]_E \longrightarrow t'_1 \quad \dots \quad [t_n]_E \longrightarrow t'_n}{f([t_1]_E, \dots, [t_n]_E) \longrightarrow f(t'_1, \dots, t'_n)} \text{ (Par)}$$

- ▶ Rewriting modulo equational theory
- ▶ Many concurrency models represented in RL:  
Petri nets, CCS, Actors, Unity, ...
- ▶ RL supported by execution engine: **Maude** [Clavel et al. 02]

# The Runtime Syntax (1)

- ▶ **Substitutions** bind variable names to values
- ▶ **Expressions**

**sort** Label Data Expr .

**subsort** Label < Data < Expr .

**op** label: Oid Nat -> Label [ctor] .

- ▶ **Statements**

**sorts** Stmt StmtList .

**subsort** Stmt < StmtList .

**op** await\_ : Expr -> Stmt .

**op** nil : -> StmtList .

**op** \_;\_ : StmtList StmtList -> StmtList [ctor] .

**op** if\_th\_el\_fi : Expr StmtList StmtList -> Stmt [ctor] .

**op** \_[]\_ : StmtList StmtList -> SuspStmt [ctor comm assoc] .

## The Runtime Syntax (2)

### ► Processes

**sort** Process .

**op** idle : -> Process [ctor] .

**notFound** : -> Process [ctor] .

**op** { \_ | \_ } : Subst StmtList -> Process [ctor] .

**var** L : Subst .

**eq** { L | noStmt } = idle .

### ► Objects, Methods, Classes, etc

**op** < \_ : \_ | Att: \_ , Pr: \_ , PrQ: \_ , Lcnt: \_ > :

Oid Cid Subst Process MProc Nat -> Object [ctor] .

**op** < \_ : Class | Inh: \_ , Param: \_ , Att: \_ , Mtds: \_ , Ocnt: \_ > :

Cid InhList VidList Subst MMtd Nat -> Class [ctor] .

### ► Messages between objects and classes

# Multiset Rewriting

**Basic idea:** rewriting applies to multisets of objects, classes, and messages

**sorts** Msg Class Object Configuration .

**subsorts** Class Msg Object < Configuration .

**op** none : -> Configuration [ctor] .

**op** \_\_ : Configuration Configuration -> Configuration  
[ctor assoc comm id: none] .

## AC-rewriting on multisets

- ▶ Matching of the rewrite rules is modulo AC
- ▶ We don't need to reorder the multisets to fire rules!

## Rules rewrite multisets to multisets and typically have the form

- ▶  $\text{obj} \longrightarrow \text{obj}'$
- ▶  $\text{obj} \longrightarrow \text{obj}' \text{ msg}$
- ▶  $\text{obj msg} \longrightarrow \text{obj}'$



# Rewriting Creol Configurations

**System state:** a **multiset** of terms of the following types

- ▶ Classes:  $\langle Id : Class \mid Inh, Param, Att, Mtds, Ocnt \rangle$
- ▶ Messages :  $invoc(l, o_2, m, ln)$  **to**  $o_1$  and  $comp(l, Out)$
- ▶ Objects:  $\langle Id : Cl \mid Att, Pr, PrQ, Lcnt \rangle$  and queues,  
where the process  $Pr = \{Lvar \mid Code\}$

## Transition rules capture object activity

- ▶ Rules that execute code from the active process
- ▶ Rules that activate pending processes
- ▶ Object creation
- ▶ Transport rules (for messages between objects and classes)

## Method Calls

All invocation mechanisms are handled *uniformly* using the primitives for asynchronous communication.

**Synchronous invocations** are expanded by an equation

$$\text{eq } o.m(E; V) = !o.m(E); !?(V)$$

for some fresh label identifier *!*.

### Asynchronous invocation

$$\langle o \mid \text{Att} : A, \text{Pr} : \{L \mid (t!x.m(E); S)\}, \text{Lcnt} : n \rangle$$

→

$$\langle o \mid \text{Att} : A, \text{Pr} : \{L \mid (t := n; S)\}, \text{Lcnt} : \text{next}(n) \rangle$$

$$\text{invoc}(m, (o \text{ label}(o, n) \text{ eval}(E, (A; L)))) \text{ to } \text{eval}(x, (A; L))$$

# Method Calls

**Method loading** — Late binding

$\langle o : C \mid PrQ : W \rangle \text{ invoc}(m, E) \text{ to } o$

$\langle C : \text{Class} \mid \text{Mtds} : M \rangle$

→

$\langle o : C \mid PrQ : (W \text{ bind}(m, M, E)) \rangle \langle C : \text{Class} \mid \text{Mtds} : M \rangle$

(Remark: omitted inheritance, need to traverse the inheritance tree)

**Method completion** — Active waiting

$\langle o : C \mid \text{Att} : A, Pr : \{L \mid (t ? (V); S)\} \rangle \text{ comp}(n, E)$

→

$\langle o : C \mid Pr : \{L \mid (V := E; S)\} \rangle \text{ comp}(n, E)$

*if*  $n = \text{eval}(t, A; L)$

## Suspension Points

Let  $Q$  collect all comp-messages in the configuration

### Reply guard — Proceed

$$\langle o : C \mid Att : A, Pr : \{L | (\textit{await } t?; S)\} \rangle$$

$$\longrightarrow$$

$$\langle o : C \mid Att : A, Pr : \{L | S\} \rangle$$

*if contains*(eval( $t, A; L$ ),  $Q$ )

### Reply guard — Suspend (passive waiting)

$$\langle o : C \mid Att : A, Pr : \{L | (\textit{await } t?; S)\}, PrQ : W \rangle$$

$$\longrightarrow$$

$$\langle o : C \mid Att : A, Pr : \textit{idle}, PrQ : (W \{L | (\textit{await } t?; S)\}) \rangle$$

*if not contains*(eval( $t, A; L$ ),  $Q$ )

## Object Creation

$$\begin{aligned}
 & \langle o : C' \mid \text{Att} : A, \text{Pr} : \{L \mid (x := \text{new } C(E); S)\} \rangle \\
 & \langle C : \text{Class} \mid \text{Param} : V, \text{Att} : A', \text{Ocnt} : n \rangle \\
 & \quad \longrightarrow \\
 & \langle o : C' \mid \text{Att} : A, \text{Pr} : \{L \mid (x := (C; n); S)\} \rangle \\
 & \langle (C; n) : C \mid \text{Att} : \varepsilon, \\
 & \quad \text{Pr} : \{\varepsilon \mid (V := \text{eval}(E, (A; L))); \text{self} := (C; n); \text{init}(A'); !\text{run}\}, \\
 & \quad \text{PrQ} : \varepsilon, \text{Lcnt} : 0 \rangle \\
 & \langle C : \text{Class} \mid \text{Param} : V, \text{Att} : A', \text{Ocnt} : \text{next}(n) \rangle
 \end{aligned}$$

With inheritance, the instantiation of the local state must be handled with more care...

- ▶ pass parameters from subclass to superclass
- ▶ initialize from superclass to subclass

## Guided simulation

- ▶ Simulation
- ▶ Custom rewriting strategies
- ▶ Search for reachable state
- ▶ States may become large
- ▶ Visualization of the runtime state...

DEMO!