Wrap Your Objects Safely

Olaf Owe Gerardo Schneider

{olaf, gerardo}@ifi.uio.no

Department of Informatics
University of Oslo, Norway

FESCA, 28 March 2009 – York, UK

Motivation

- How to enforce security in open distributed systems?
- Restrict the uploading/downloading of applications compromising data privacy, confidentiality, etc
  - Sandbox model of Java
    - A set of rules to limit an untrusted applet to execute certain operations when arriving to the site whether the browser resides
  - Only download "signed" code
    - Up to the user to allow which code to accept
- Other solutions?
  - Different *boxed* calculi

Motivation

- How to enforce security in open distributed systems?
- Restrict the uploading/downloading of applications compromising data privacy, confidentiality, etc
  - Sandbox model of Java
    - A set of rules to limit an untrusted applet to execute certain operations when arriving to the site whether the browser resides
  - Only download "signed" code
    - Up to the user to allow which code to accept
- Other solutions?
  - Different *boxed* calculi
- We want to address this at the programming language level

Our Proposal

General Aspects

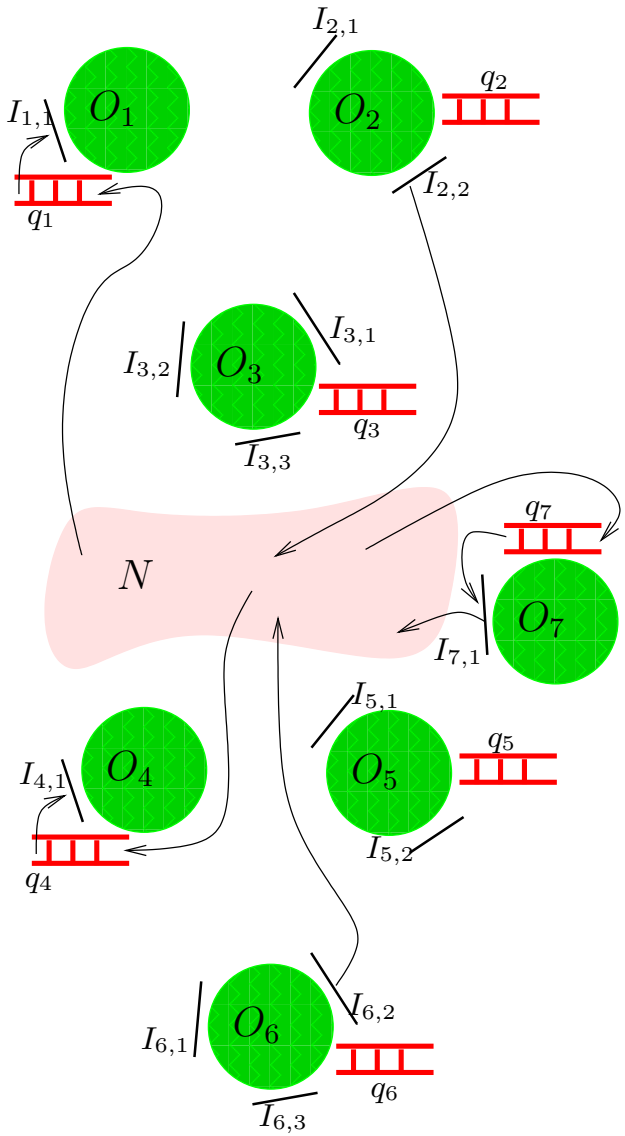
- A programming language primitive to wrap objects (and components)
- A wrapper is a membrane defined *around* an object to isolate it from its environment
  - The membrane itself
  - The operational part —automaton
- Communication between the *inside* and the *outside* of the membrane is controlled by the wrapper automaton
- Two possibilities:
  - The untrusted part is what is inside the wrapper
  - The untrusted part is the environment

Our Proposal

`safeNew` and Creol

- `safeNew` C (P;A) creates an instance of class C (and parameters P), wrapped with automaton A
- We need
  - A language for defining the wrapper automaton
  - Extend a programming language with the `safeNew`
  - Enforce the properties of the wrapper at runtime
- Implementation in Creol
  - Asynchronous object-based modeling/programming language
  - Active objects
  - Non-blocking method calls (processor release points)
  - (Executable) operational semantics in Rewriting Logic (Maude)

Creol



- O_i: objects
- $I_{i,j}$ are its interfaces
- q_i its message queue
- N is the network

Syntactic categories	*Definitions*
t in Label	$g ::= wait \mid b \mid t? \mid g \wedge g$
g in Guard	$p ::= x.m \mid m$
p in MtdCall	$\overline{s} ::= \varepsilon \mid s; \overline{s}$
s in Stm	$s ::= (\overline{s})$
v in Var	$\mid \overline{v} := \overline{e} \mid v :=$ **new** $Id(\overline{e})$
e in Expr	$\mid$ **if** b **then** $\overline{s}$ **else** $\overline{s}$ **fi**
m in Mtd	$\mid$ **while** b **do** $\overline{s}$ **od**
x in ObjExpr	$\mid !p(\overline{e}) \mid t!p(\overline{e}) \mid t?(\overline{v}) \mid p(\overline{e}; \overline{v})$
b in BoolExpr	$\mid$ **await** $g \mid$ **await** $p(\overline{e}; \overline{v})$

Creol

Semantics

- Configuration:

  op none : $\longrightarrow$ Config [**ctor**] .
  op _ _ :Config Config $\longrightarrow$ Config [**ctor assoc comm identity**: none] .

- A Creol object:

  $<$o : C | Att: A, Lvar: L, Pr: S, PrQ: P, InQ: Q, Icnt : I , Ocnt: N $>$

- A Creol class:

  $<$C : Cl | Mtd: M, Att: A $>$

- Object creation in Creol:

  (New): $<$C : Cl | Mtd: M, Att: A $>$
  $<$O : C' | Pr: v:= new C; S, Ocnt: N $>$
  $\longrightarrow$ $<$C : Cl | Mtd: M, Att: A $>$
  $<$O : C' | Pr: v:=ob(O,N); S, Ocnt: N+1 $>$
  $<$ob(O,N): C | Att: A+(*this* $\mapsto$ ob(O,N)), Lvar: ε, Pr: run(),
  PrQ: ε, InQ: ε, Icnt: 1, Ocnt: 1 $>$.

Creol

Semantics

- Configuration:

  op none : $\longrightarrow$ Config [**ctor**] .
  op _ _ :Config Config $\longrightarrow$ Config [**ctor assoc comm identity**: none] .

- A Creol object:

  $< o : C \mid Att: A, Lvar: L, Pr: S, PrQ: P, InQ: Q, Icnt: I, Ocnt: N >$

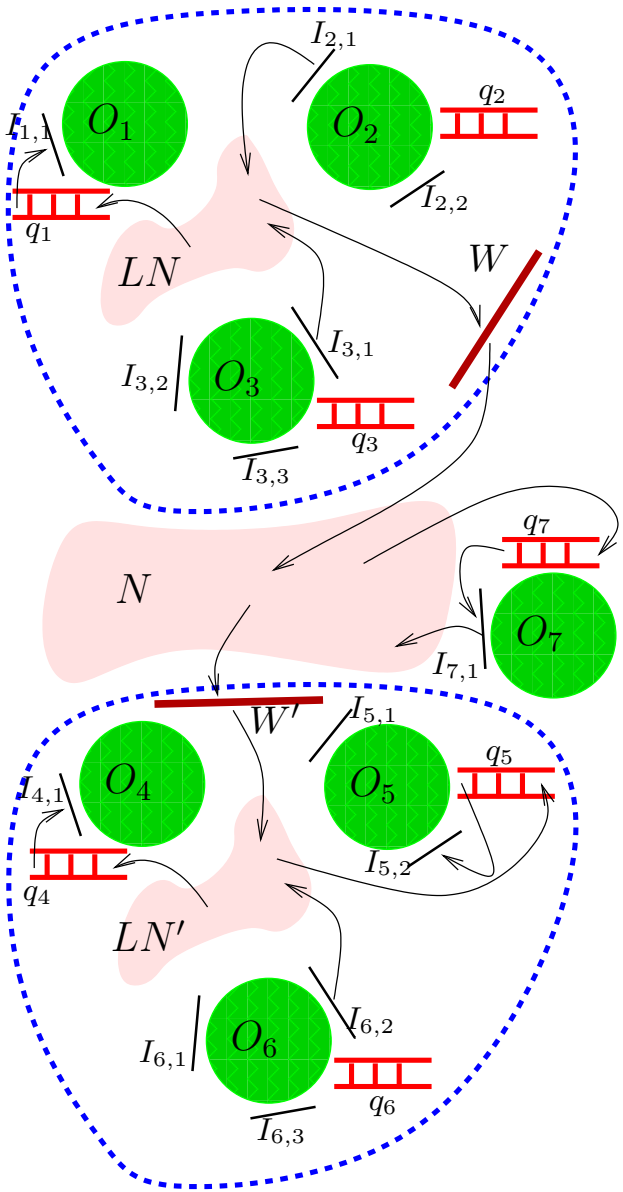
- A Creol class:

  $< C : Cl \mid Mtd: M, Att: A >$

- Object creation in Creol:

  $$\begin{array}{ll} \text{(New)}: & < C : Cl \mid Mtd: M, Att: A > \\ & < O : C' \mid Pr: v{:=}\ \text{new } C; S, Ocnt: N > \\ \longrightarrow & < C : Cl \mid Mtd: M, Att: A > \\ & < O : C' \mid Pr: v{:=}ob(O,N); S, Ocnt: N{+}1 > \\ & < ob(O,N): C \mid Att: A{+}(\mathit{this} \mapsto ob(O,N)), Lvar: \varepsilon, Pr: \text{run}(), \\ & \qquad\qquad PrQ: \varepsilon, InQ: \varepsilon, Icnt: 1, Ocnt: 1 > . \end{array}$$

Enhancing Creol with Wrappers



Enhancing Creol with Wrappers

- Configurations (System: Config / class decl):

  op _+_ : Classes System → Config [**ctor**].

- Wrapper definition:

  sort Wrapper . **subsorts** Wrapper < System .
  op {_|_} :Config Automaton →Wrapper [**ctor**].

- Operational rule for the `safeNew`:

  (safeNew): CL +<O: C' |Pr: v:=safeNew C(FA); S, Ocnt: N >
  ⟶ CL + <O: C' | Pr: v:=ob(O,N); S, Ocnt: N+1 >
  { classes (CL,C) +
  <ob(O,N): C | Att: A+(*this* ↦ ob(O,N)), Lvar:ε, Pr: run(),
  PrQ: ε, InQ: ε, Icnt: 1, Ocnt: 1 >
  | FA } .

- A possible wrapper configuration may then look like:
  $\{< C : CL|... > + < o : C|... > (m \text{ to } o)\,(m' \text{ to } o') \mid FA\}$

Enhancing Creol with Wrappers

- Configurations (System: Config / class decl):

  op _+_ : Classes System → Config [**ctor**].

- Wrapper definition:

  sort Wrapper . **subsorts** Wrapper < System .
  op {_|_} :Config Automaton →Wrapper [**ctor**].

- Operational rule for the `safeNew`:

  (safeNew): CL +<O: C' |Pr: v:=safeNew C(FA); S, Ocnt: N >
  ⟶ CL + <O: C' | Pr: v:=ob(O,N); S, Ocnt: N+1 >
  { classes (CL,C) +
  <ob(O,N): C | Att: A+(*this* ↦ ob(O,N)), Lvar:ε, Pr: run(),
  PrQ: ε, InQ: ε, Icnt: 1, Ocnt: 1 >
  | FA } .

- A possible wrapper configuration may then look like:
  $\{< C : CL|... > + < o : C|... > (m \text{ to } o)\,(m' \text{ to } o')\,|\,FA\}$

Enhancing Creol with Wrappers

- Configurations (System: Config / class decl):

  op _+_ : Classes System → Config [**ctor**].

- Wrapper definition:

  sort Wrapper . **subsorts** Wrapper < System .
  op {_|_} :Config Automaton →Wrapper [**ctor**].

- Operational rule for the `safeNew`:

  (safeNew): CL +<O: C' |Pr: v:=safeNew C(FA); S, Ocnt: N >
  ⟶ CL + <O: C' | Pr: v:=ob(O,N); S, Ocnt: N+1 >
  { classes (CL,C) +
  <ob(O,N): C | Att: A+(*this* ↦ ob(O,N)), Lvar:ε, Pr: run(),
  PrQ: ε, InQ: ε, Icnt: 1, Ocnt: 1 >
  | FA } .

- A possible wrapper configuration may then look like:
  $\{< C : CL|... > + < o : C|... > \ (m \ to \ o)\,(m' \ to \ o') \,|\, FA\}$

Example: Readers and Writers

Without Wrappers

- `rwcons := new RWController(db)`
  - `db` is an interface of the `DataBase` class

Example: Readers and Writers

Without Wrappers

- `rwcons := new RWController(db)`
  - `db` is an interface of the `DataBase` class

class *RWController(db: DataBase)*
begin
var free: Bool = true, readers: ObjSet = $\emptyset$, writer: Obj = null
  pr, pw: Nat = 0 *// pending calls to db.read and db.write*
with *RWClient*
  op OR() == **await** free; **if** writer $\neq$ null **then** free := false;
    await (writer = null); free := true **fi**; readers := readers $\cup$ {caller}
  op CR() == **await** (caller $\in$ readers); readers := readers $\setminus$ {caller}
  op OW() == **await** free; free := false;
    await (readers = $\emptyset$ $\wedge$ pr = 0 $\wedge$ writer = null);
    free := true; writer := caller
  op CW() == **await** (pw = 0 $\wedge$ writer = caller); writer := null
  op read(**in** k: Key **out** x: Data) == **await** (caller $\in$ readers);
    pr := pr + 1; **await** db.read(k; x); pr := pr − 1
  op write(**in** k: Key, x: Data) == **await** (writer = caller);
    pw := pw + 1; **await** db.write(k,x); pw := pw − 1
end

Example: Readers and Writers

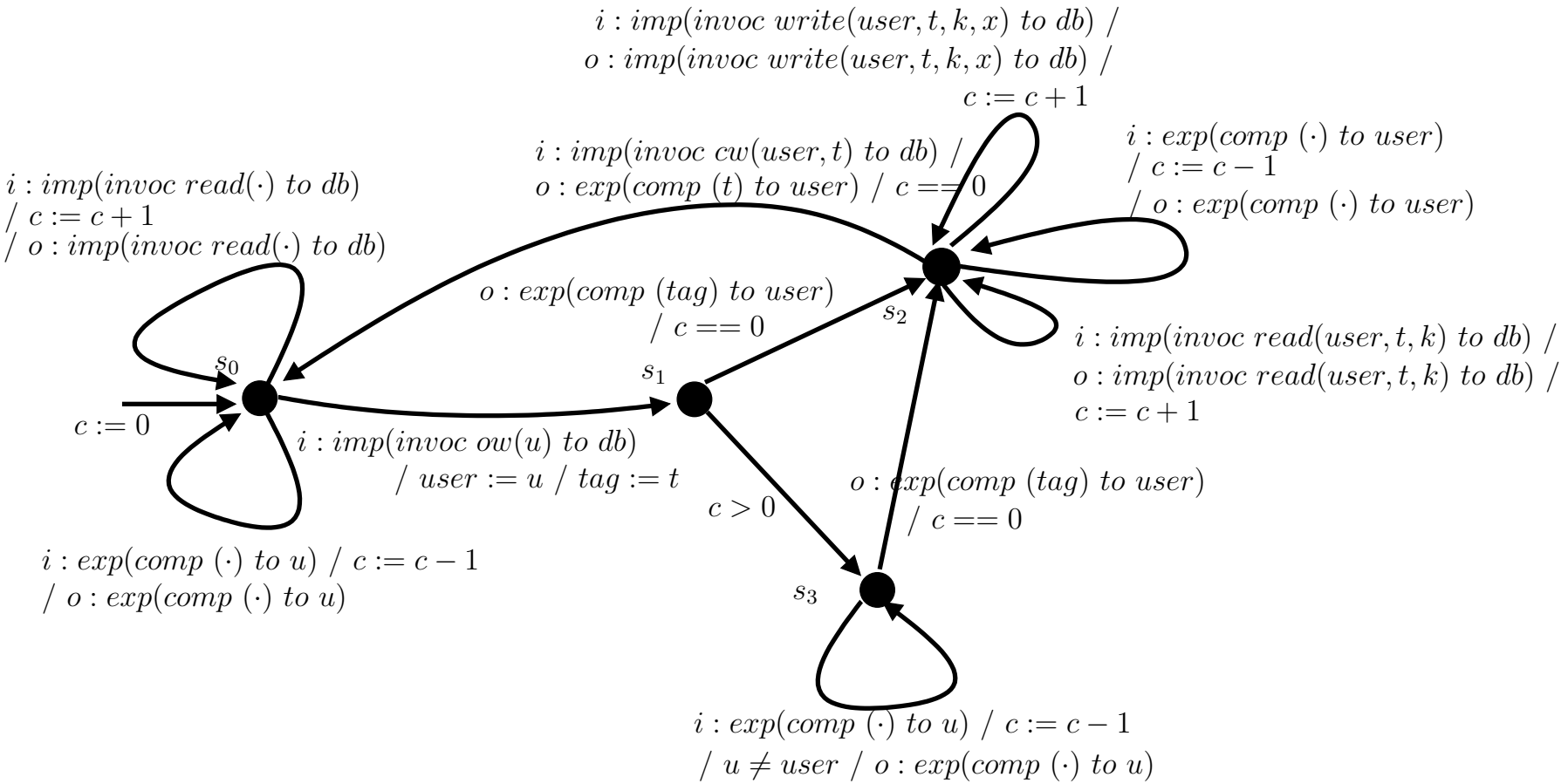
With Wrappers

- `rwcons := safeNew DataBase(;Aut)`
  - No need of all the code above

Example: Readers and Writers

With Wrappers

- `rwcons := safeNew DataBase(;Aut)`
  - No need of all the code above



Conclusions

- Extending Creol with wrappers is easy
  - We needed also to modify some of the *transportation* rules
- Advantages of the (wrapper) automaton over Creol code
  - Separation of concern
  - Facilitate verification
- Components defined as wrapped objects (including classes)
- Localities: wrappers + identifiers
- Wrappers as adaptors

- The automaton could be written using the functional language of Creol
- Need for a library with "standard" wrappers
- Explore applications in smart cards

Conclusions

- Extending Creol with wrappers is easy
  - We needed also to modify some of the *transportation* rules
- Advantages of the (wrapper) automaton over Creol code
  - Separation of concern
  - Facilitate verification
- Components defined as wrapped objects (including classes)
- Localities: wrappers + identifiers
- Wrappers as adaptors

- The automaton could be written using the functional language of Creol
- Need for a library with "standard" wrappers
- Explore applications in smart cards