A Specification Language for Coordinated Objects

Gabriel Ciobanu* and Dorel Lucanu**

gabriel@iit.tuiasi.ro, dlucanu@info.uaic.ro

*Institute of Computer Science, Romanian Academy, Iaşi, Romania

**"A.I.Cuza" University, Iaşi, Romania

Outline

- what is coordination
- our approach to coordination: the main idea
- our approach to coordination: details
  - syntax and operational semantics
  - integrated semantics
  - implementation
- conclusion

What is coordination

"Coordination models and languages are meant to close the conceptual gap between the cooperation model of an application and the lower level communication used in its implementation." (F. Arbab, *What Do You Mean, Coordination?*, 1998)

"...: the formalization of the separation of concerns that is known as Coordination"

"Object-oriented systems do not go a long way in supporting that separation."
(J. Fiadeiro, *Categories for Software Engineering*, 2005)

Example

- consider a sender, a receiver, and unreliable communication channels
- we assume that all these are represented as objects
- they should communicate in a safe way
- a protocol is a coordinator that instructs the objects to accomplish a safe communication
- the goal of this work is
  - to specify the protocol and the objects separately, and then
  - to check the properties of the assembled system
- Alternate Bit Protocol (ABP) is just an example of such a coordinator

The main idea

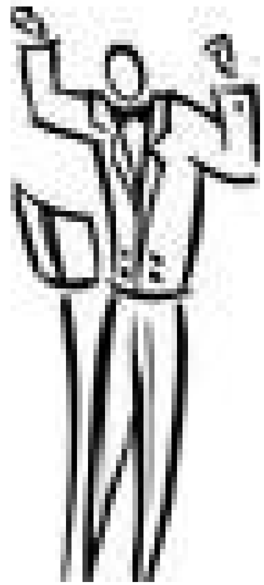
- the three components of the coordination

The main idea

- the three components of the coordination
  - coordinated objects

The main idea

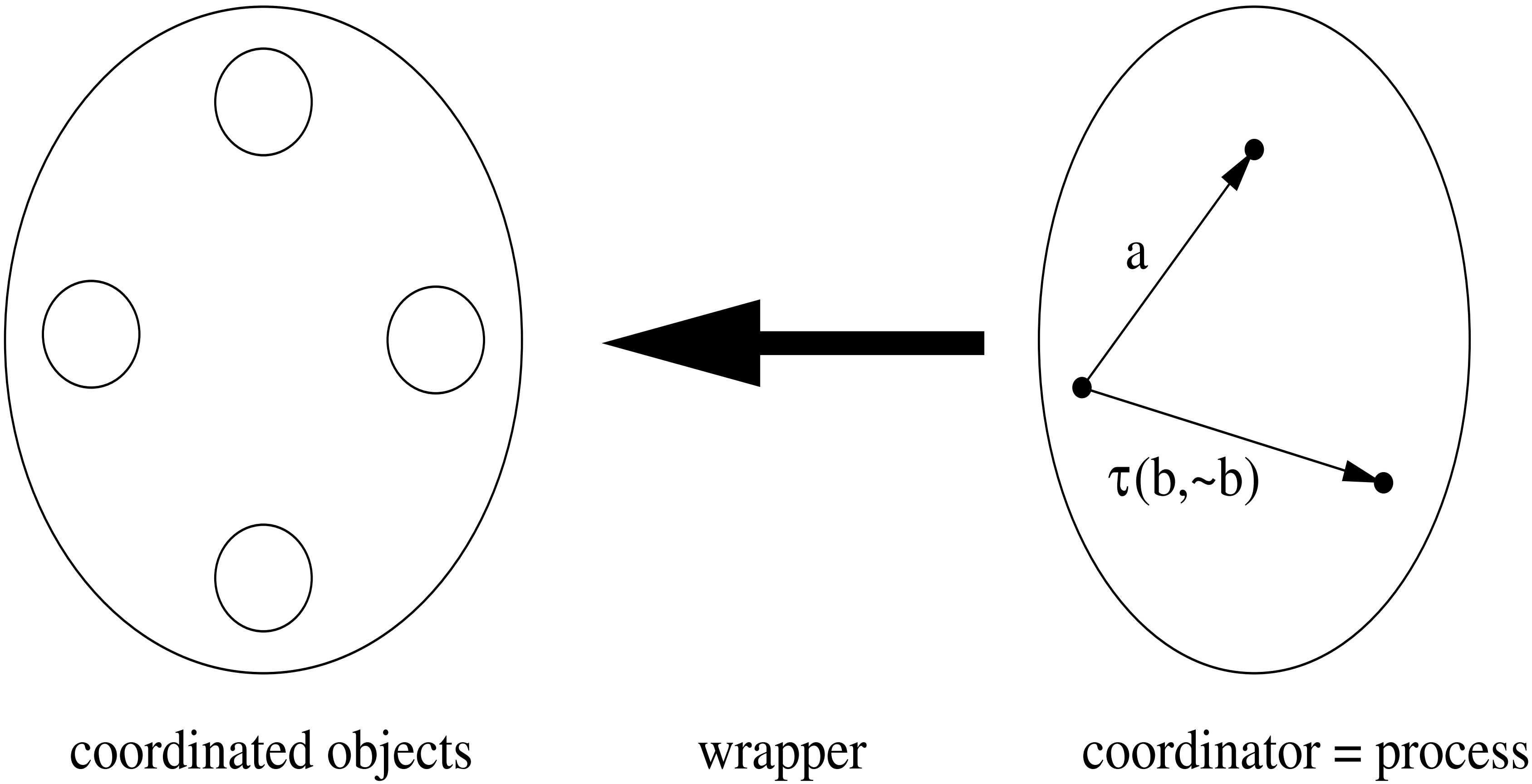
- the three components of the coordination
  - coordinated objects
  - coordinator



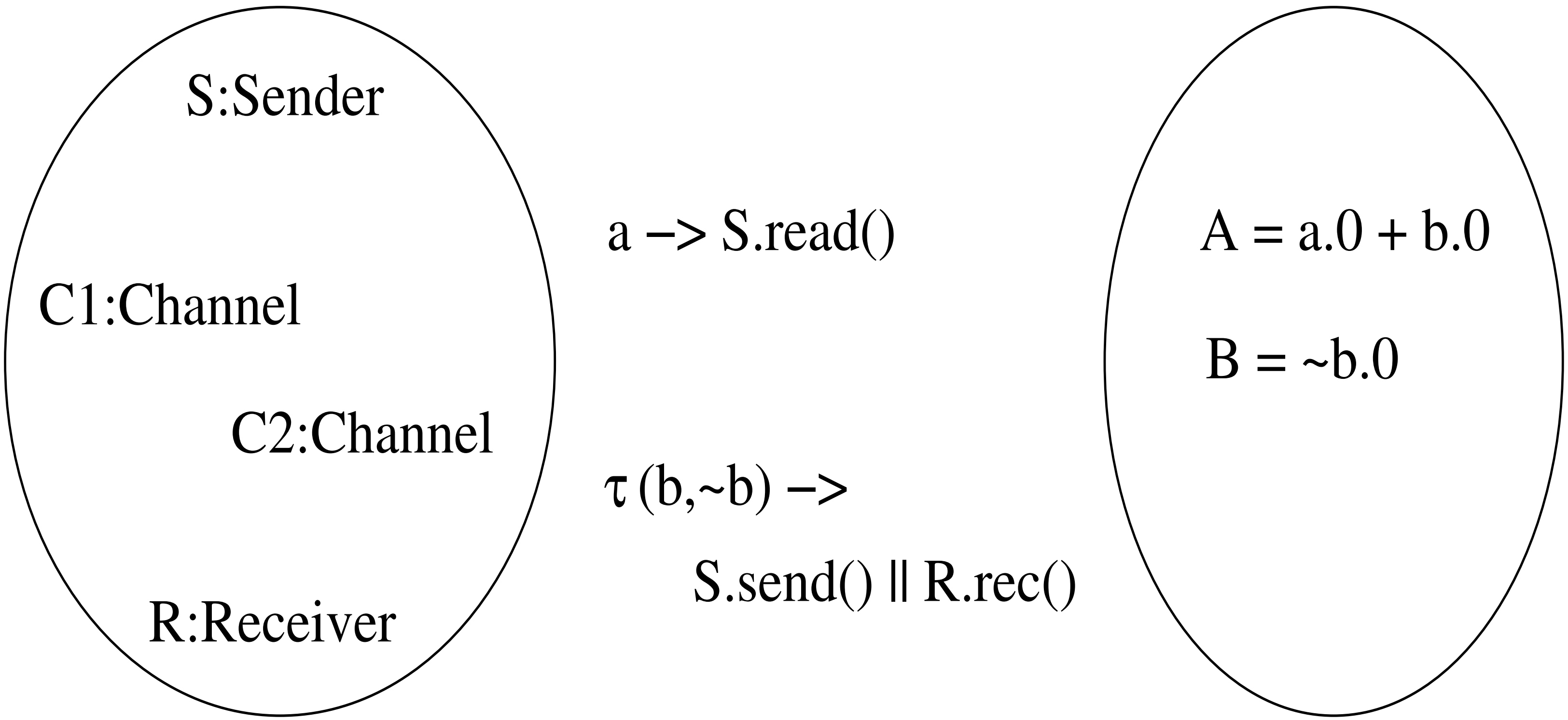
The main idea

- the three components of the coordination
  - coordinated objects
  - coordinator
  - a means to coordinate

The main idea



The main idea



coordinated objects   wrapper   coordinator = process

Classes and objects: syntax

```
class AbsComp
{
  Bool bit;
  Data data;
  Bool ack;
}
```

```
class Sender extends AbsComp
{
  Bool chBit() {
    bit' = not bit;
    data' = data;
    ack' = ack;
  }
  void read() {
    bit' = bit;
    ack' = ack;
  }
  ...
}
```

Classes and objects: configurations

- object state: $(att_1, val_1), \ldots, (att_n, val_n)$
- an execution of a method may change the state:

$$\mathtt{S.chBit}()((\mathtt{bit}, \mathtt{true}), (\mathtt{ack}, \mathtt{false}), (\mathtt{data}, d)) =$$
$$((\mathtt{bit}, \mathtt{false}), (\mathtt{ack}, \mathtt{false}), (\mathtt{data}, d)).$$

- object instance : $(object\ reference \mid object\ state)$
- configuration: a multiset of object instances s.t. an object reference occurs at most once

Commands: syntax

$$
\begin{aligned}
\langle cmd \rangle ::= {} & R \ = \ \texttt{new}\ C(\mathbf{d}) \mid \\
& \texttt{delete}\ R \mid \\
& R.m(\mathbf{d}) \mid \\
& R_1.m_1(\mathbf{d}_1) \| R_2.m_2(\mathbf{d}_2) \ \mid \\
& \langle cmd \rangle ; \langle cmd \rangle \mid \\
& \texttt{if}\ \langle bexpr \rangle\ \texttt{then}\ \langle cmd \rangle\ \texttt{else}\ \langle cmd \rangle \mid \\
& \texttt{throw}\ \texttt{error}()
\end{aligned}
$$

Commands: operational semantics

- labeled transition system, where the labels are given by commands
- $cnfg \xrightarrow{R = \texttt{new}\ C(d_1,\ldots,d_n)} cnfg, (R|(att_1, d_1), \ldots, (att_n, d_n))$;
- ...
- $cnfg \xrightarrow{R_1.m_1(\mathbf{d}_1)\|R_2.m_2(\mathbf{d}_2)} cnfg'$ iff $R_1 \neq R_2$, $cnfg'$ is obtained from $cnfg$ by replacing the object instance $(R_i \mid state_i)$ with $(R_i \mid state'_i)$, where $state'_i = R_i.m_i(\mathbf{d}_i)(state_i)$, $i = 1, 2$;
- ...

Coordinators (processes): syntax

```
proc ABP
{
  global actions: in, out, alterS, alterR;
  local actions: ch1, ch2;
  processes: A, A′, V, B, B′, T;
  guards: sok, rok;
  equations:
    A  = in.A′;
    A′ = ~ch1.ch2.V;
    V  = [sok]alterS.A + [not sok]A′;
    B  = ch1.T;
    T  = [rok]B′ + [not rok]out.alterR.B;
    B′ = ~ch2.B;
}
```

Coordinators: operational semantics

- labeled transition system, where the labels are given by action names

$$\frac{}{gact.E \xrightarrow{gact} E} \qquad \frac{E \xrightarrow{act} E'}{E + F \xrightarrow{act} E'}$$

$$\frac{E \xrightarrow{act} E'}{E|F \xrightarrow{act} E'|F} \qquad \frac{E_A \xrightarrow{act} E',\ A = E_A}{A \xrightarrow{act} E'}$$

$$\frac{E \xrightarrow{act} E',\quad \gamma(guard\_id) = true}{[guard\_id]E \xrightarrow{act} E'}$$

$$\frac{}{\tilde{\ }lact.E \ \mid\ lact.E' \xrightarrow{\tau(lact)} E|E'}$$

Wrapper: syntax

```
wrapper w(Sender S, Receiver R) implementing ABP
{
  in -> S.read();
  alterS -> S.chBit();
  alterR -> R.chAck();
  tau(ch1) ->
    R.recFrame(S.data(), S.bit()) ||
    S.sendFrame();
  tau(ch2) ->
    S.recAck(R.ack()) || R.sendAck();
  out -> R.write();
  sok -> S.bit == S.ack;
  rok -> R.bit =/= R.ack;
}
```

Wrapper: operational semantics

- labeled transition system, where the labels are given by action names

$$cnfg \xrightarrow{act} cnfg' \text{ iff } cnfg \xrightarrow{w(\mathbf{R})(act)} cnfg'$$

$$cnfg \xrightarrow{\tau(\mathtt{ch2})} cnfg' \text{ iff } cnfg \xrightarrow{\mathtt{S.recAck(R.ack())}\|\mathtt{R.sendAck()}} cnfg'$$

Integrated semantics

- labeled transition systems as coalgebras
  - Set is the category of sets
  - A is the set of action names
  - $\mathsf{T}_{\mathsf{LTS}} : \mathsf{Set} \longrightarrow \mathsf{Set}$ is the functor given by

$$\mathsf{T}_{\mathsf{LTS}}(X) = \{Y \subseteq A \times X \mid Y \text{ finite}\}$$

  - a coalgebra representing a l.t.s. is a function $\gamma : X \to T_{LTS}(X)$

$$x \xrightarrow{a} y \text{ iff } (a, y) \in \gamma(x)$$

Integrated semantics

- operational semantics of the coordinator: $\pi : Proc \to \mathsf{T}_{\mathsf{LTS}}(Proc)$
- operational semantics of the wrapper:
$$w(\mathbf{R}) : Config \to \mathsf{T}_{\mathsf{LTS}}(Config)$$
- operational semantics of the integrated system consists of a partial supervising operation $proc : Config \to Proc$ and a coalgebra $\gamma : dom(proc) \to \mathsf{T}_{\mathsf{LTS}}(Config)$ s.t. the following diagram commutes:

$$\begin{tikzcd}
Config \arrow[d, "w(\mathbf{R})"'] & dom(proc) \arrow[l, "\mathrm{id}"'] \arrow[r, "proc"] \arrow[d, "\gamma"] & Proc \arrow[d, "\pi"] \\
\mathsf{T}_{\mathsf{LTS}}(Config) & \mathsf{T}_{\mathsf{LTS}}(dom(proc)) \arrow[l, "\mathsf{T}_{\mathsf{LTS}}(\mathrm{id})"] \arrow[r, "\mathsf{T}_{\mathsf{LTS}}(proc)"'] & \mathsf{T}_{\mathsf{LTS}}(Proc)
\end{tikzcd}$$

Supervising means bisimulation

Proposition.
Let $\gamma^{\sim} : graph(proc) \rightarrow \mathsf{T}_{\mathsf{LTS}}(graph(proc))$ be the coalgebra given by $(a, \langle cnfg_2, p_2\rangle) \in \gamma^{\sim}(\langle cnfg_1, p_1\rangle)$ iff $proc(cnfg_1) = p_1$, $proc(cnfg_2) = p_2$, and $(act, cnfg_2) \in \gamma(cnfg_1)$. Then $\gamma^{\sim}$ is a bisimulation between $w(\mathbf{R})$ and π.

- γ: $cnfg_1 \xrightarrow{act} cnfg_2$ iff
  p_1 supervises $cnfg_1$ ($proc(cnfg_1) = p_1$) and
  p_2 supervises $cnfg_2$ ($proc(cnfg_2) = p_2$) and
  ...
- $\gamma^{\sim}$: $\langle cnfg_1, p_1\rangle \xrightarrow{act} \langle cnfg_2, p_2\rangle$ iff ...

Hidden algebra based semantics

- we use hidden algebra to give semantics to classes and objects
  - *visible sorts* for data values (`Bool, Data`)
  - *hidden sorts* for state space (`Sender`)
  - operations for methods:

    `recAck : Sender Bool -> Sender`
  - operations for attributes:

    `bit : Sender -> Bool`
  - constants for particular states: `initS : -> Sender`
- *behavioural abstraction*
  - a subset Γ of methods and attributes (behavioural ops)
  - Γ*-behavioural equivalence*: two states are Γ-behavioural equivalent iff they cannot be distinguished under Γ-experiments

    if $S \equiv S'$ iff $\mathtt{bit}(S) \equiv \mathtt{bit}(S') \wedge \mathtt{data}(S) \equiv \mathtt{data}(S')$

    then `read` is not Γ-behavioural congruent (it does not preserve $\equiv$)

Hidden algebra based semantics

- the objects and configurations can also be specified using hidden algebra
- the models, i.e., implementations, for hidden specifications are algebras
- $w(\mathbf{R})$ and γ can be defined over hidden algebra models, i.e., implementations
- we get a framework suitable to investigate
  - initial semantics (syntax)
  - final semantics (behaviour)

Temporal properties

- we may use temporal logics for describing behavioural properties of the integrated systems
- the atomic propositions are given by attributes (operations with results of visible sorts)

$$\mathtt{AG}((\mathtt{S.bit}()(\_) = \mathtt{true}\ \wedge \mathtt{R.ack}()(\_) = \mathtt{false}\ \wedge \mathtt{S.data}()(\_) = \mathtt{d}) \rightarrow$$

$$\mathtt{AF}(\mathtt{S.bit}()(\_) = \mathtt{false}\ \wedge \mathtt{R.ack}()(\_) = \mathtt{true}\ \wedge \mathtt{R.data}()(\_) = \mathtt{d}))$$

- since we have an algebraic semantics, algebraic expressions over attributes are also allowed
- the underscore symbol _ is used for the current configuration

Implementation

- joined work with M. Daneş (SYNASC 2005)
- hidden algebra framework for classes and objects is encoded in Maude
- the processes are encoded using rewrite rules
- the wrapper is encoded as a Maude functional module
- we extend Maude to extract a Kripke structure from the integrated specification
- we use an existing model checker to verify temporal properties
- ABP integrated specification is verified under the fairness assumption using SMV

Conclusion

- a specification language for coordinated objects with a syntax closer to OOP languages
- rigorously defined operational semantics based on labeled transition systems and bisimulation
- use of the temporal logics to describe behavioural properties
- an automated procedure extracting a finite-state machine model
- use of the existing model checking algorithms and tools