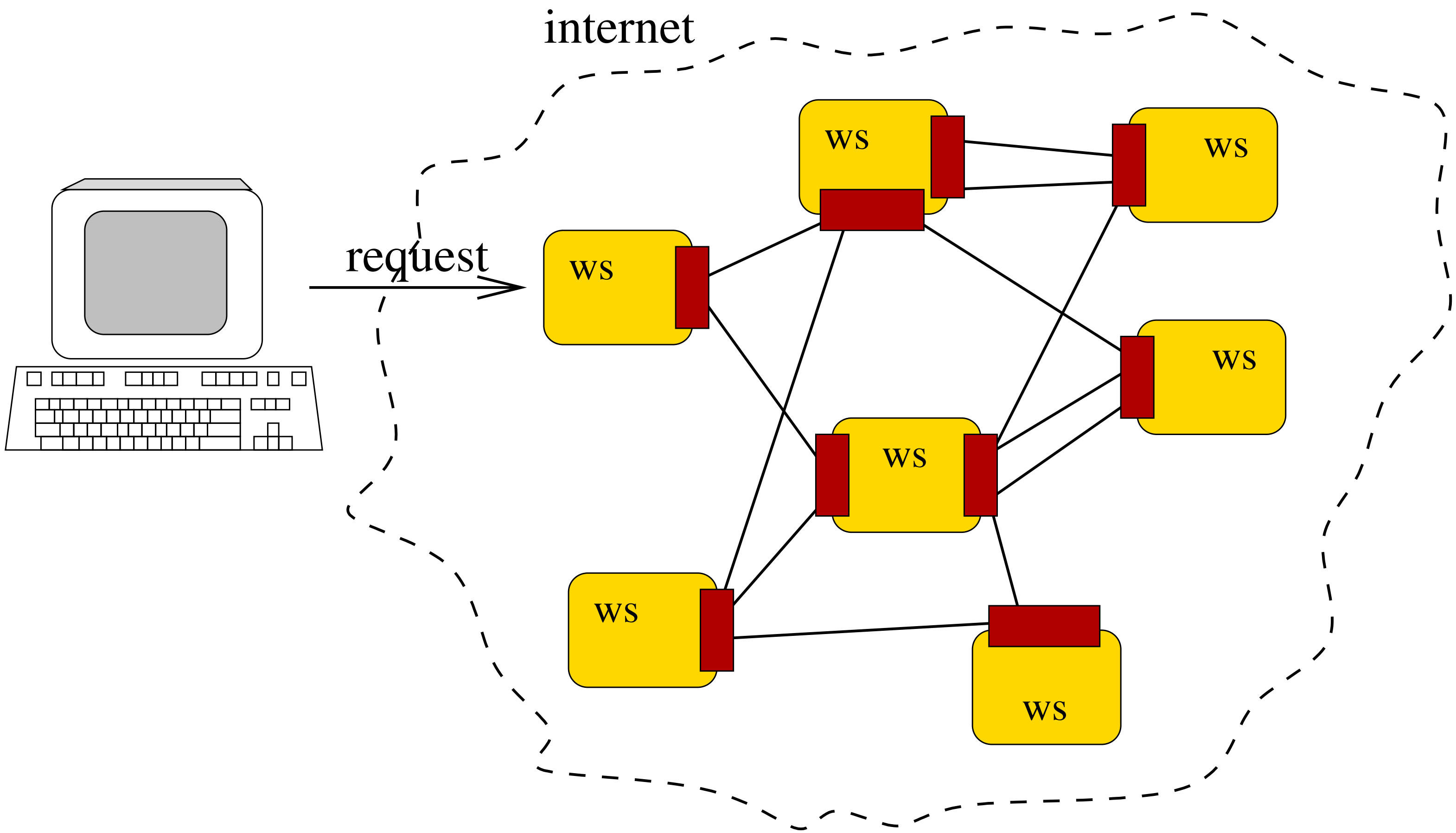
How Formal Methods Can Contribute to the Formal Development of Web Services

Gwen Salaün, VASY, INRIA Rhône-Alpes

joint work with Daniela Berardi, Lucas Bordeaux, Antonella Chirichiello, Andrea Ferrara, Massimo Mecella, Marco Schaerf

An introductive example

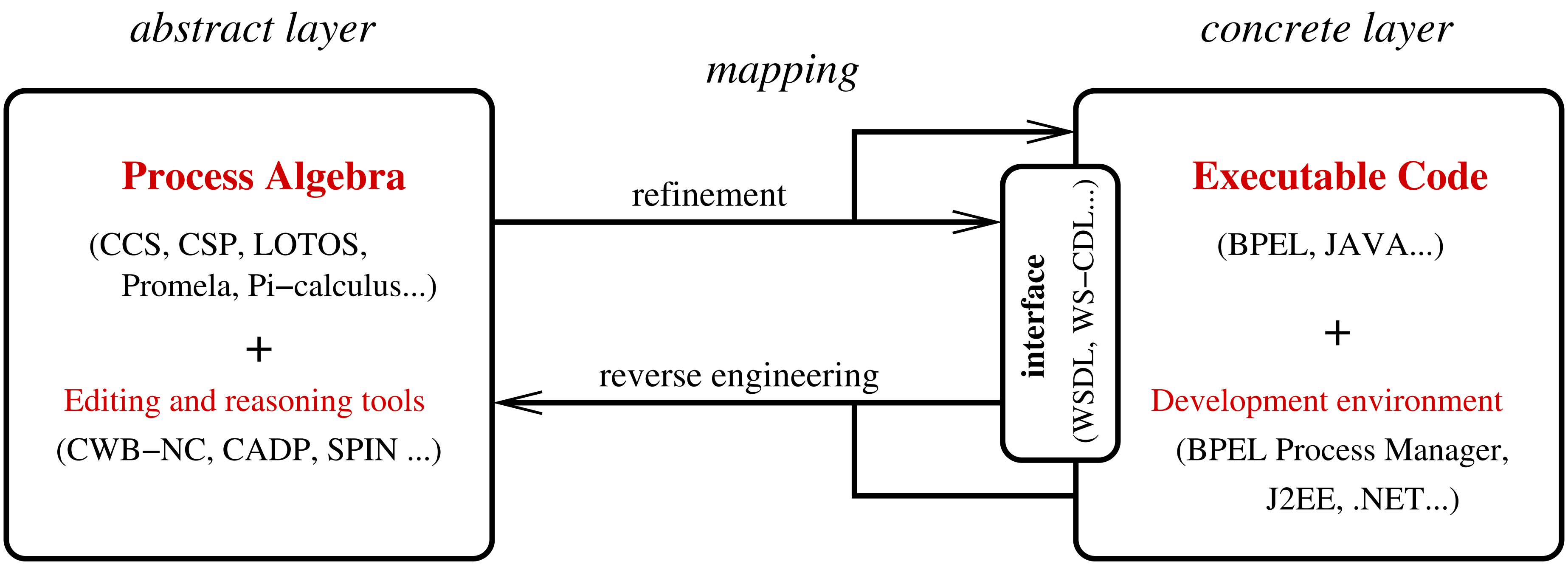
Organization of a trip (airplane tickets, room booking, exhibitions, shows, etc) delegated to interacting WSs



Formal methods for web services

- WSs are distributed processes which communicate through the exchange of messages
- one central question is to make them working together to perform a given task
- WSs and their interaction are best described using behavioural description languages
- we privilege abstract and formal languages to use in a second step existing verification tools
- several candidates, *e.g.* transition system models (LTS, Mealy automata, Petri nets)
- we advocate the use of process algebra (PA) as description means

Overview of the approach



Outline

- Describing WSs using PA
- Automated reasoning on WSs
- Application: negotiating WSs using LOTOS/CADP
- Concluding remarks

Outline

- Describing WSs using PA
  - What is a process algebra?
  - Specifying web services as processes
  - Composing web services
- Automated reasoning on WSs
- Application: negotiating WSs using LOTOS/CADP
- Concluding remarks

What is a process algebra? $\rightsquigarrow$ CCS

- basic entities: input/output actions (*request* and *'confirm*)
- basic contructs:
  - sequence *a.P*
  - nondetermistic choice *P+Q*
  - parallel composition $P_1|\ldots|P_n$
  - restriction $P\backslash\{a_1,\ldots,a_m\}$
- τ for hidden actions, esp. result of a synchronization
- termination using 0 and recursive call *P*
- operational semantics: possible evolutions of a process

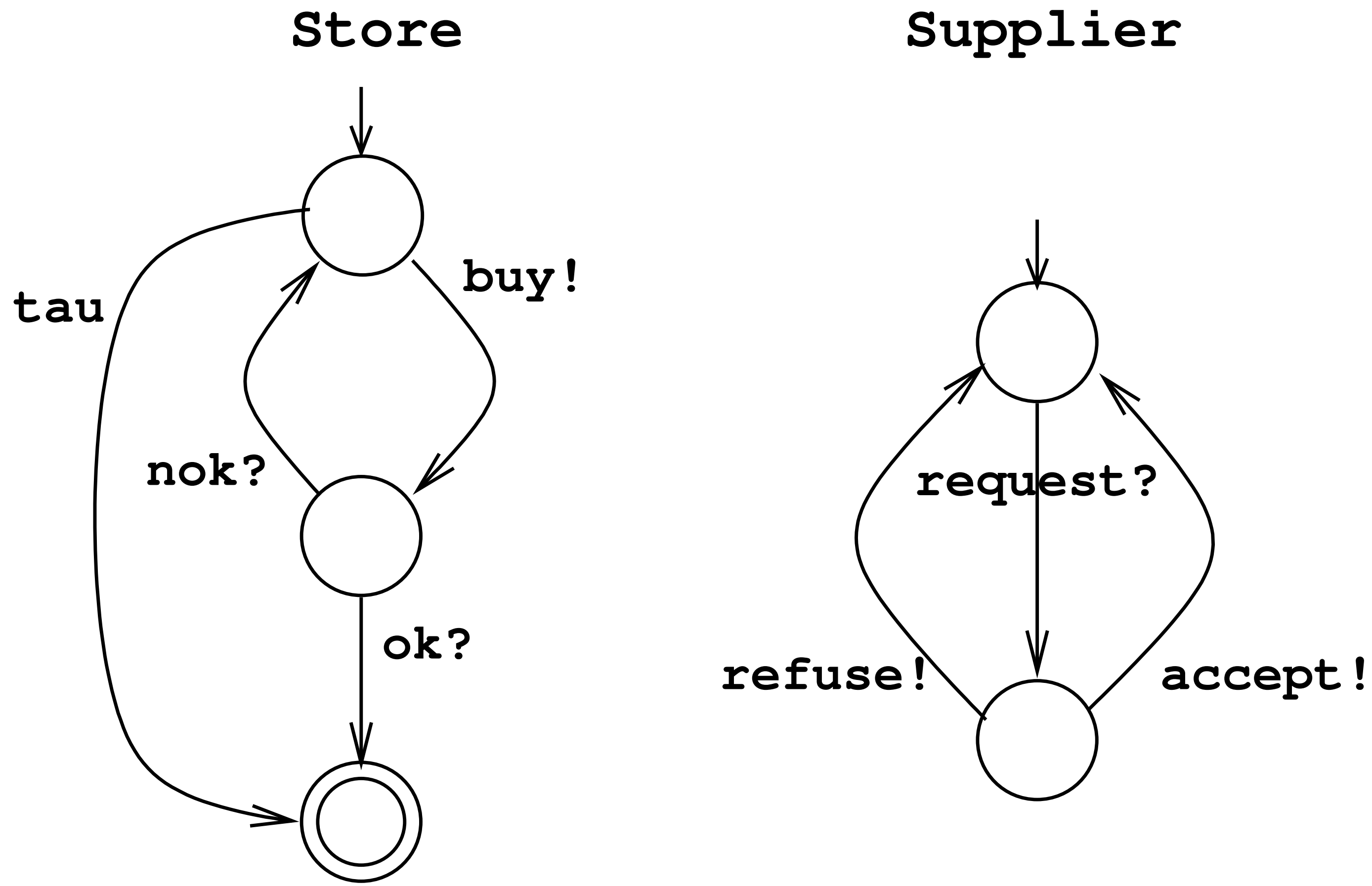
$$((b.a.0 + c.a.0)|'a.'c.0)\backslash\{a\} \xrightarrow{b} (a.0|'a.'c.0)\backslash\{a\} \xrightarrow{\tau} ('c.0)\backslash\{a\} \xrightarrow{'c} 0$$

Specifying web services as processes

- WSs are essentially processes
- PAs are an unambiguous way to represent such behaviours
- processes can describe the body of WSs or their interfaces
- levels of abstraction to have a more faithful representation of a service, *e.g.* data (LOTOS) or mobility (π-calculus) (++ wrt Automata-based Models)
- PAs are compositional notations, then adequate to compose services (++)
- description of real-size problems thanks to textual notations (++)

Specifying web services as processes

A classical example of communication between a store and several suppliers



Specifying web services as processes

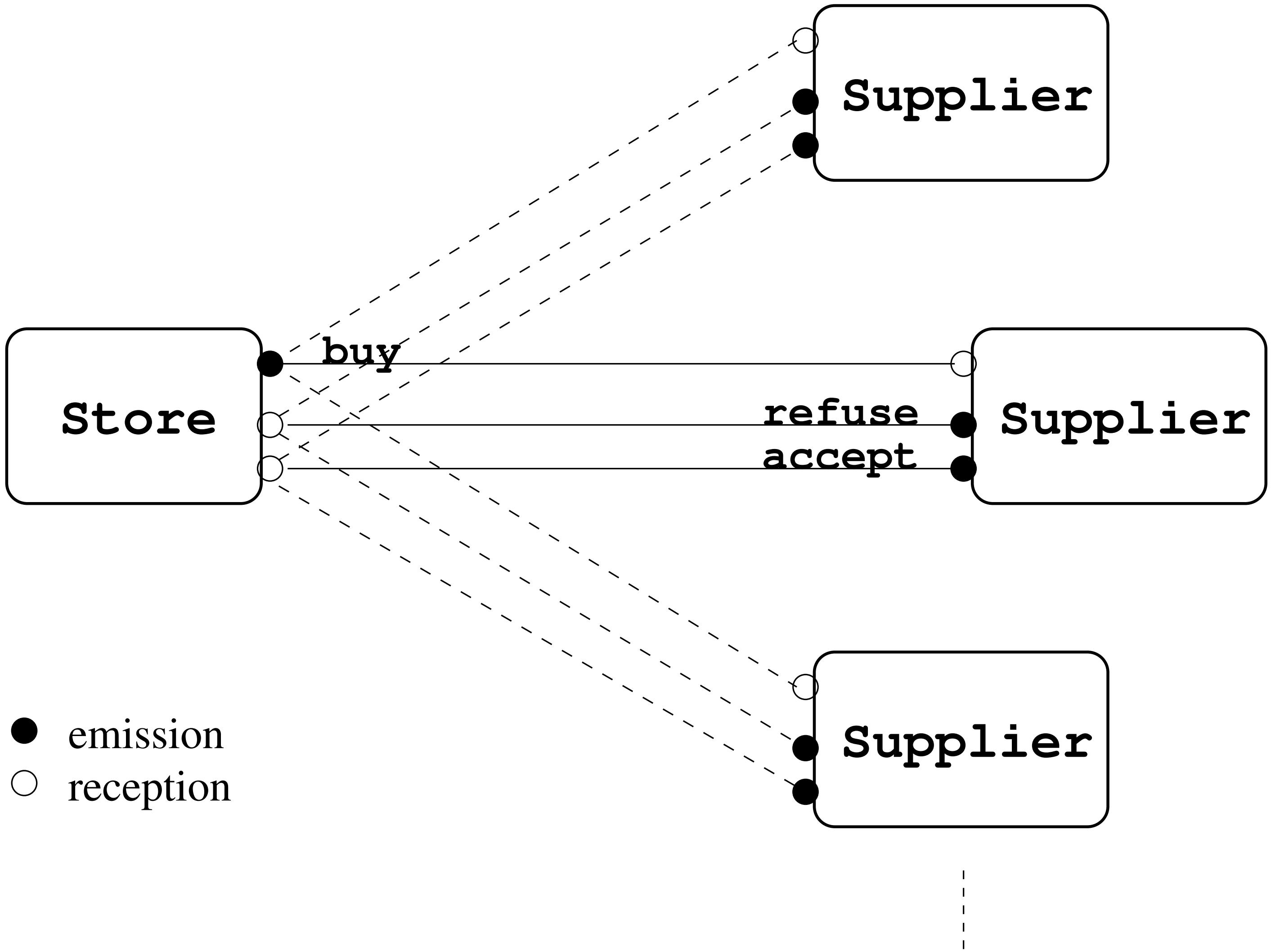
A classical example of communication between a store and several suppliers

```
proc Store =
    'buy.  ( ok.nil + nok.Store )
    + t.nil

proc Supplier =
    request.
       ( 'refuse.Supplier + 'accept.Supplier )
```

Composing WSs: choreography

- choreography is the problem of guaranteeing that WSs can interact properly
- this problem is especially tricky when independently developed services are put together
- it typically involves situations where the design of services is fixed and their implementation private
- then, services are viewed through their interfaces (encoded using PA)
- automated tools are needed to perform compatibility checks

Composing WSs: choreography



Composing WSs: choreography

Parallel compositions and restriction sets are used to describe interactions between a store and several suppliers

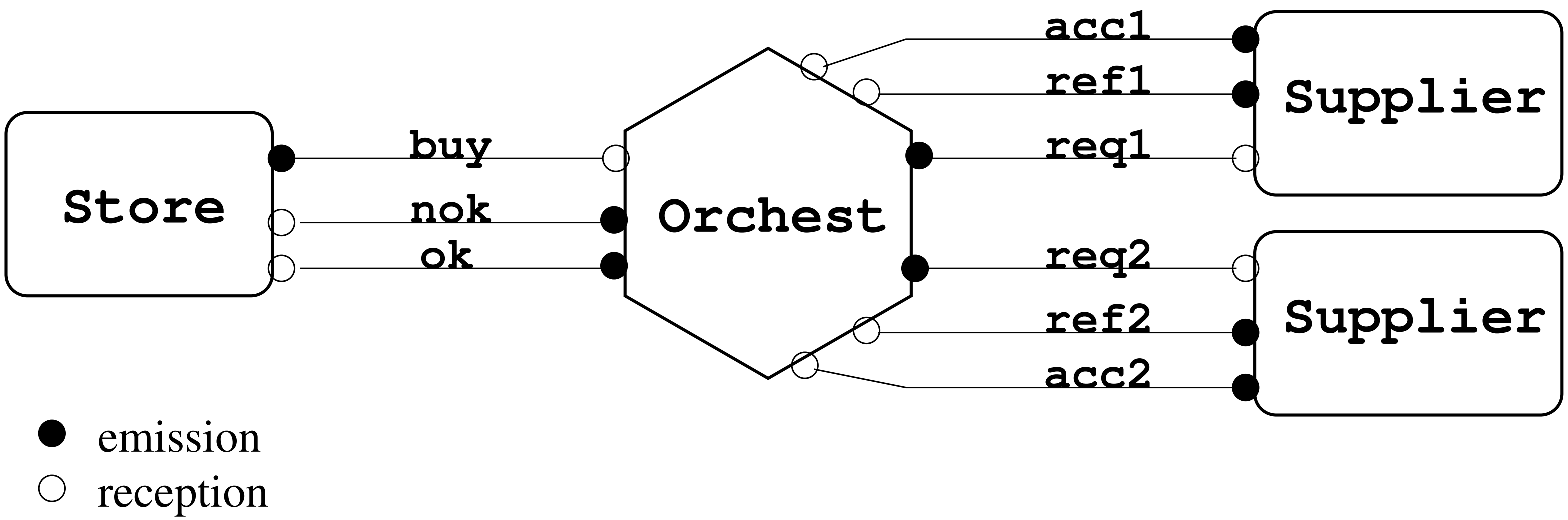
```
*** synchronization set
set restSetC = { request, refuse, accept }

*** composition of 1 store and 3 suppliers
proc SystemC =
     (
          Store [request/buy, refuse/nok, accept/ok] |
          Supplier | Supplier | Supplier
     ) \ restSetC
```

Composing WSs: orchestration

- orchestration aims at developing a new service using existing ones
- the role of the new service (orchestrator) is to manage some existing services by exchanging messages with them
- abstract descriptions in PA can be used in two ways:
  - during the design stage (abst. $\rightarrow$ conc.)
  - for reverse engineering purposes (abst. $\leftarrow$ conc.)
- automated reasoning is useful to validate the orchestrator service

Composing WSs: orchestration

For instance, iterating the request on both suppliers, and terminating if a positive answer is received or both suppliers reply negatively.



Composing WSs: orchestration

```
proc Orchest = buy.Orch1
proc Orch1  = 'req1. ( acc1.'ok.nil + ref1.Orch2 )
proc Orch2  = 'req2. ( acc2.'ok.nil + ref2.'nok.nil )

set restSetO =            *** synchronization set
   {buy, ok, nok, req1, req2, acc1, acc2, ref1, ref2}

*** we rename channels of the two suppliers
proc SystemO =
 (
     Store
   | Supplier [req1/request, ref1/refuse, acc1/accept]
   | Supplier [req2/request, ref2/refuse, acc2/accept]
   | Orchest
 ) \ restSetO
```

Outline

- Describing WSs using PA
- Automated reasoning on WSs
- Application: negotiating WSs using LOTOS/CADP
- Concluding remarks

Outline

- Describing WSs using PA
- Automated Reasoning on WSs
  - Verifying properties
  - Verifying equivalences
  - Verifying compatibility
- Application: negotiating WSs using LOTOS/CADP
- Concluding remarks

Automated reasoning on web services

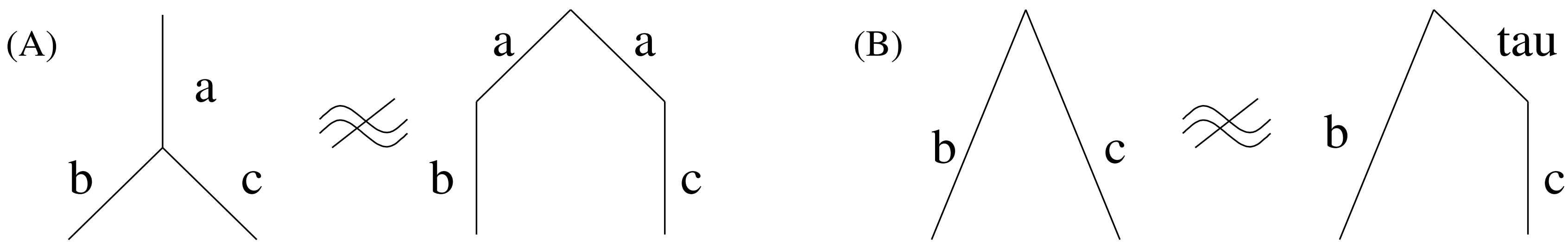
- formally-grounded languages enable one to use automated tools to check that a system matches its requirements and works properly
- these tools can help
  - checking that a service satisfies desirable properties – *e.g.* the property that the system will never reach some unexpected state
  - checking that two processes are equivalent – typically one abstract process expresses the specification of the problem, while the other is a composition of services as a possible solution
  - checking compatibility of services then ensuring correct interactions

Verifying properties

- properties of interest in concurrent systems typically involve reasoning on the possible scenarii that the system can go through
- established formalism for expressing such properties is given by temporal logics
- the most noticeable classes of properties are:
  - safety properties, which state that an undesirable situation will never arise
  - liveness properties, which state that something good must happen

Verifying equivalences

- two processes are considered to be equivalent if they are indistinguishable from the viewpoint of an external observer
- trace equivalence: they produce the same set of traces
- observational equivalence is a more appropriate notion of process equivalence



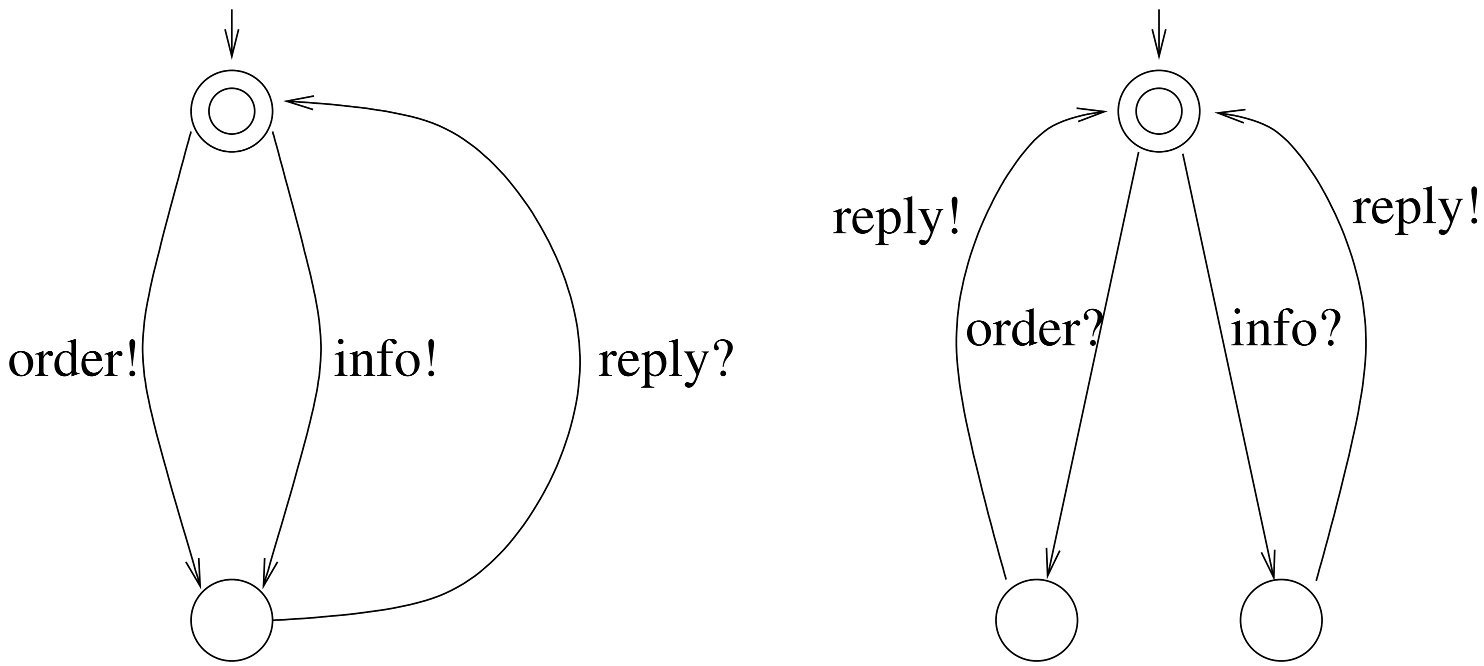
- strong bisimulation too restrictive: strict matching of the τ actions

When are two WSs compatible?

- compatibility: ensuring that WSs will be able to interact properly
- substitutability: replacing one WS by another without introducing flaws
- it depends not only on static properties but also on their dynamic behaviour (service interface)
- compatibility checking can be automated (CADP, SPIN) if defined in a sufficient formal way

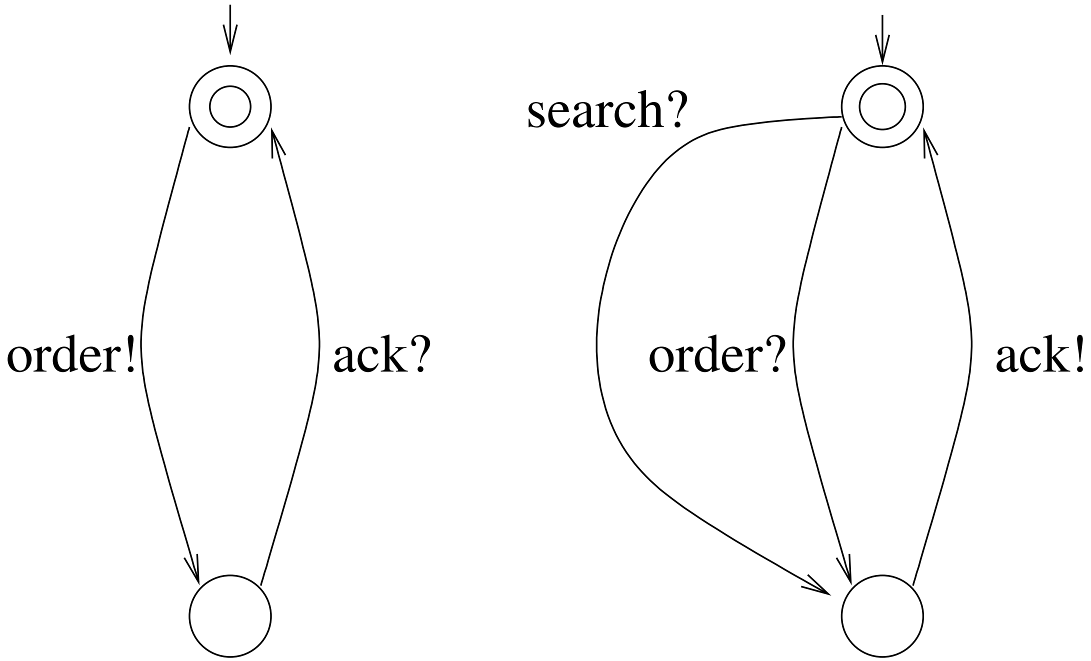
Compatibility 1: opposite behaviours

Two services are compatible if they have opposite behaviours (observational equivalence)



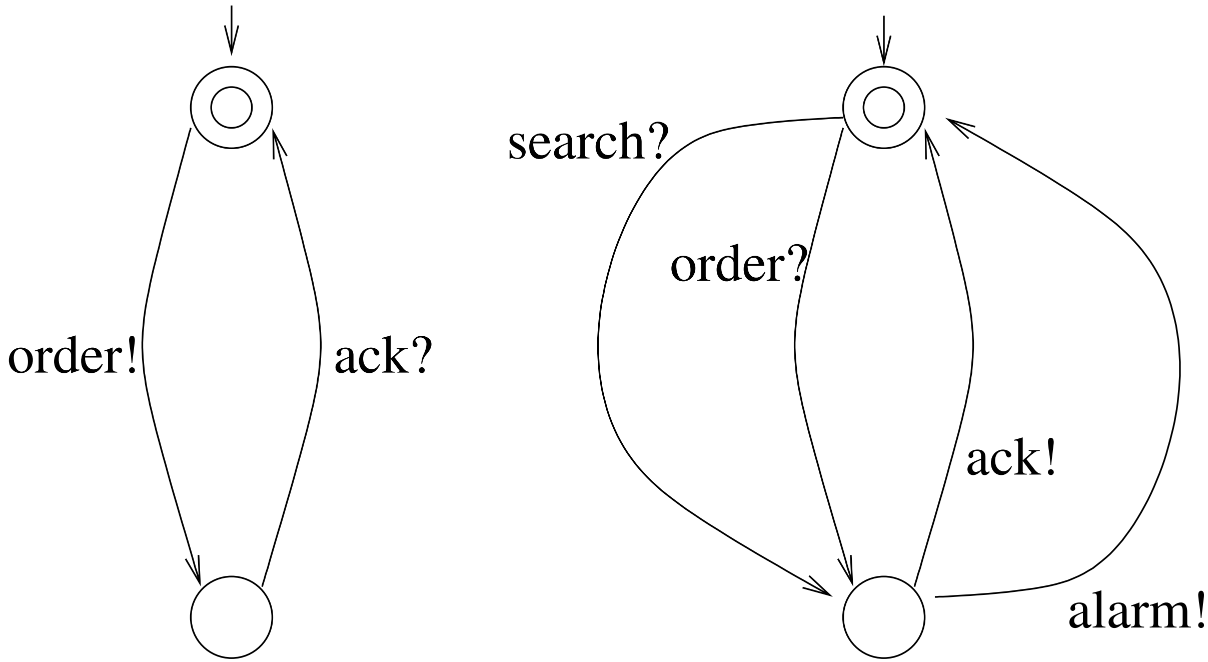
Compatibility 2: unspecified receptions

Two services are compatible if they have no unspecified receptions



Compatibility 3: one-path existence

Two services are compatible if there is at least one execution leading to a pair of final states



Outline

- Describing WSs using PA
- Automated reasoning on WSs
- Application: negotiating WSs using LOTOS/CADP
- Concluding remarks

LOTOS/CADP for Web Services

- in some cases, a less abstract level of description is needed
- LOTOS and CADP to abstractly describe and reason on WSs handling data
- negotiation is a typical example of services involving data (prices, products, stocks)
- clients and providers have to reach an agreement beneficial to all of them
- involved aspects: variables, constraints, exchanged information, strategies

LOTOS in a nutshell

- abstract data types: sorts, operations, generators, axioms

LOTOS in a nutshell

```
type BasicNaturalNumber is
     sorts Nat
     opns 0 (*!  constructor *) :   -> Nat
          Succ (*!  constructor *) :  Nat -> Nat
          _+_ :  Nat, Nat -> Nat
     eqns
          forall m, n :  Nat
          ofsort Nat
               m + 0 = m;
               m + Succ(n) = Succ(m) + n;
endtype
```

LOTOS in a nutshell

- abstract data types: sorts, operations, generators, axioms
- basic LOTOS: gates, **exit**, *g;B*, **[]**, $B_1|[g_1, ..., g_n]|B_2$

```
cc; exit
[]
(
     bb; inter; exit
          |[inter]|
     inter; aa; exit
)
```

LOTOS in a nutshell

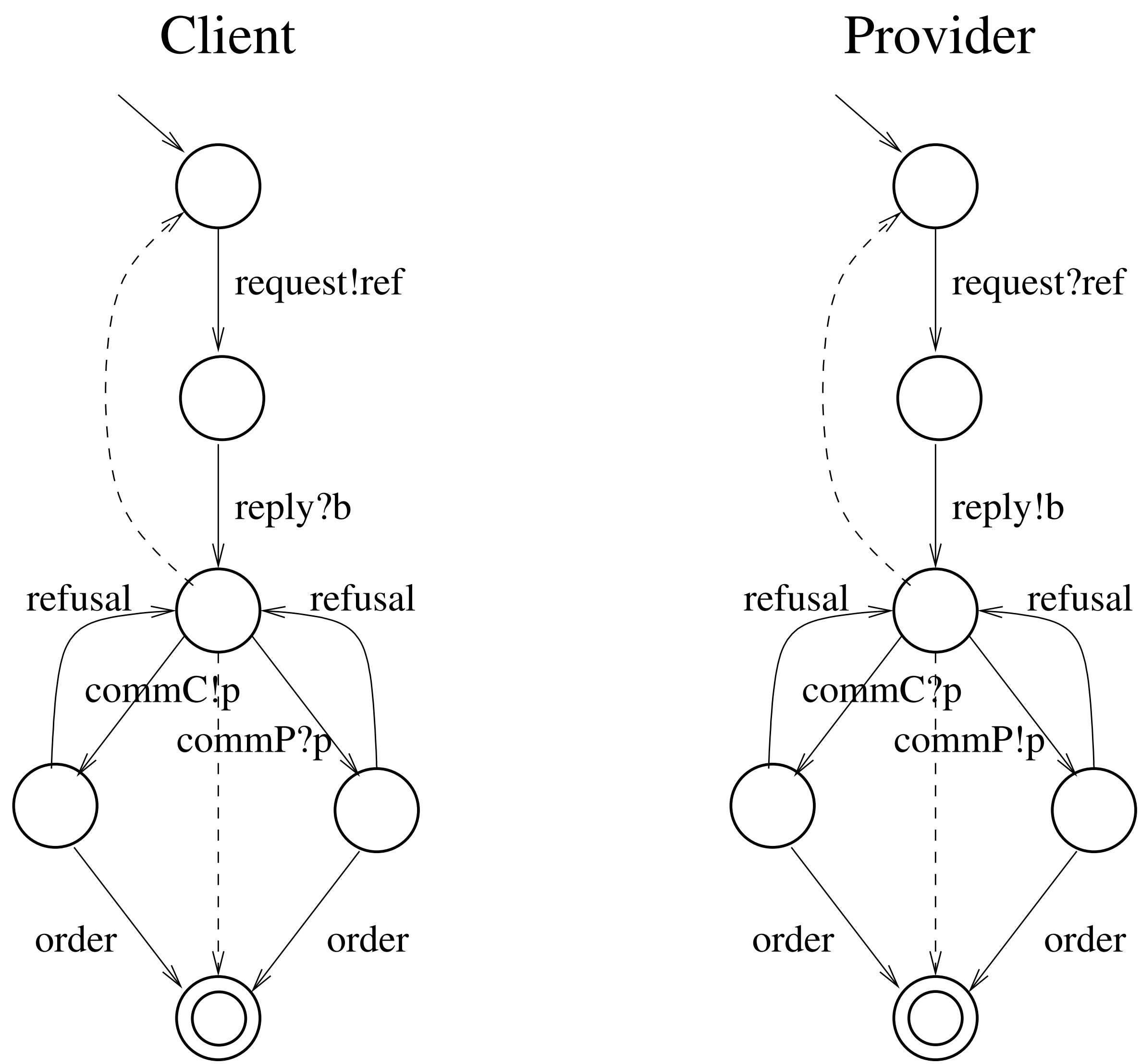
- abstract data types: sorts, operations, generators, axioms
- basic LOTOS: gates, **exit**, *g;B*, **[]**, $B_1|[g_1, ..., g_n]|B_2$
- full LOTOS: *g!V*, *g?X:S*, **[***boolexp***]** $\rightarrow$ *B*

```
cc; exit
[]
(
     bb; inter?y:Nat; ([y>2] -> cc; exit)
          |[inter]|
     inter!5; aa; exit
)
```

LOTOS in a nutshell

- abstract data types: sorts, operations, generators, axioms
- basic LOTOS: gates, **exit**, *g;B*, **[]**, $B_1|[g_1, ..., g_n]|B_2$
- full LOTOS: *g!V*, *g?X:S*, **[***boolexp***]** $\rightarrow$ *B*
- the CADP toolbox:
  - input notations (LOTOS, LTSs)
  - an open environment OPEN/CAESAR, in particular EVALUATOR an on-the-fly model-checker
  - BISIMULATOR: on-the-fly equivalence/preorder checking
  - ... $\rightsquigarrow$ `http://www.inrialpes.fr/vasy/cadp/`

Negotiation case: specification



Negotiation case: specification

```
process NegotiateC [order, refusal, commC, commP]
    (curp: Nat, inv: Inv, computfct: Comp): exit(Bool) :=

    commP?p:Nat;            (* the provider proposes a value *)
    (
      [conform(p, inv)] -> order; exit(true) (* agreement *)
      []
      [not(conform(p, inv))] -> refusal;
        NegotiateC[order, refusal, commC, commP]
         (curp, inv, computfct)
    )
    []                        (* the client proposes a value *)
    ( [conform(curp, inv)] -> commC!curp;
      (
        order; exit(true)                      (* agreement *)
        []
        refusal; NegotiateC[...]
                    (compute(curp, computfct), inv, computfct)
      )
    )
```

Negotiation case: verification

- verification to ensure a correct processing of the negotiation rounds
- simulation, absence of deadlocks, temporal properties (eg. `<true*."ORDER"> true`)

Participants	States	Trans.	P1	P2	P3
(1c & 1p)	32	47	3.84s	2.15s	2.21s
(1c & 7p)	17,511	42,848	4.64s	27.70s	27.35s
(1c & 10p)	145,447	374,882	5.10s	1326.94s	1313.16s
(2c & 4p)	300,764	944,394	5.31s	117.41s	117.79s

Mapping LOTOS <–> BPEL

LOTOS	BPEL
gates + offers	*message*, *portType*, *operation*, *partnerLinkType* (WSDL), and *receive*, *reply*, *invoke* (BPEL)
termination 'exit'	end of the main *sequence*
sequence ';'	*sequence*
choice '[]'	*pick* and *switch*
parallel composition '|[..]|'	interacting WSs
recursive call	new instantiation or *while*
datatypes and operations	XML Schema, DBs, XPath, etc
guards	*case* of *switch*

Outline

- Describing WSs using PA
- Automated reasoning on WSs
- Application: negotiating WSs using LOTOS/CADP
- Concluding remarks

Concluding remarks

- WSs are an emerging and promising area involving important technological challenges
- PAs offer adequate notations and tools to describe, compose and reason on WSs at an abstract level

Perspectives:

- service description: adequate level of description, interface extraction, conformance
- composition of WSs: compatibility, automation, adaptation
- systematic mapping between abstract and concrete description levels

Main references

(1) G. Salaün, L. Bordeaux and M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. *Proc. of ICWS'04*, IEEE CSP, p. 43–51. Extended version to appear in the IJBPIM journal.

(2) G. Salaün, A. Ferrara and A. Chirichiello. Negotiation among Web Services using LOTOS/CADP. *Proc. of ECOWS'04*, LNCS 3250, SV, p. 198–212.

(3) L. Bordeaux, G. Salaün, D. Berardi and M. Mecella. When are two Web Services Compatible? *Proc. of TES'04*, LNCS 3324, SV, p. 15–28.

(4) A. Chirichiello and G. Salaün. Encoding Abstract Descriptions into Executable Web Services: Towards a Formal Development. In *Proc. of WI'05*, IEEE CSP, p. 457–463.