How Process Algebra 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
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

abstract layer

Process Algebra
(CCS, CSP, LOTOS, Promela, Pi-calculus...)

Executable Code
(BPEL, JAVA...)

Concrete layer

plus

refinement

Editing and reasoning tools
(CWB–NC, CADP, SPIN...)

Development environment
(BPEL Process Manager, J2EE, .NET...)

interface
(WSDL, WS-CDL...)

Mapping

reverse engineering
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 constructs:
  - sequence \( a.P \)
  - nondeterministic choice \( P + Q \)
  - parallel composition \( P_1 \mid \ldots \mid P_n \)
  - restriction \( P \setminus \{a_1, \ldots, a_m\} \)
- \( \tau \) for hidden actions, esp. result of a synchronization
- termination using 0 and recursive call \( P \)
- operational semantics: possible evolutions of a process

\[
\begin{align*}
((b.a.0 + c.a.0) \parallel \acute{a}.\acute{c}.0) \setminus \{a\} & \xrightarrow{b} (a.0 \parallel \acute{a}.\acute{c}.0) \setminus \{a\} & \xrightarrow{\tau} (\acute{c}.0 \setminus \{a\}) & \xrightarrow{c} 0
\end{align*}
\]
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 ($\pi$-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

Store

- tau
- nok?
- ok?
- buy!

Supplier

- request?
- refuse!
- accept!
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

- Store
  - Supplier
    - buy
    - refuse
    - accept
  - Supplier
    - emission
    - reception
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. → conc.)
  - For reverse engineering purposes (abst. ← 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 scenarios 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

\[
\begin{align*}
\text{(A)} & \quad \begin{array}{c}
& a \\
\tau & b \\
& c
\end{array} \\[0.5cm]
\text{(B)} & \quad \begin{array}{c}
& a \\
\tau & b \\
& c
\end{array}
\end{align*}
\]

- strong bisimulation too restrictive: strict matching of the \( \tau \) 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
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, $g;B$, $[]$, $B_1[[g_1, \ldots, g_n]]B_2$

```plaintext
c; exit
[]
(
    bb; inter; exit
    |[inter]|
    inter; aa; exit
)
```
LOTOS in a nutshell

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

\[
cc; \text{ exit} \\
[] \\
( \\
\hspace{1em} bb; \text{ inter}\?y:\text{Nat}; ([y>2] \rightarrow cc; \text{ exit}) \\
\hspace{2em} [\text{inter}] \\
\hspace{3em} \text{inter!5; aa; exit}
)\]
LOTOS in a nutshell

- abstract data types: sorts, operations, generators, axioms
- basic LOTOS: gates, exit, $g;B$, $[]$, $B_1|\ldots|B_2$
- full LOTOS: $g!V$, $g?X:S$, $[\text{boolexp}] \rightarrow B$
- 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
  - ... $\sim http://www.inrialpes.fr/vasy/cadp/
Negotiation case: specification

[Diagram of negotiation process]

Client

- request!ref
- reply?b
- refusal
  - commC?p
  - commP?p
- order

Provider

- request?ref
- reply!b
- refusal
  - commC?p
  - commP!p
- order

Client Provider
Negotiation case: specification

```plaintext
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)

<table>
<thead>
<tr>
<th>Participants</th>
<th>States</th>
<th>Trans.</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1c &amp; 1p)</td>
<td>32</td>
<td>47</td>
<td>3.84s</td>
<td>2.15s</td>
<td>2.21s</td>
</tr>
<tr>
<td>(1c &amp; 7p)</td>
<td>17,511</td>
<td>42,848</td>
<td>4.64s</td>
<td>27.70s</td>
<td>27.35s</td>
</tr>
<tr>
<td>(1c &amp; 10p)</td>
<td>145,447</td>
<td>374,882</td>
<td>5.10s</td>
<td>1326.94s</td>
<td>1313.16s</td>
</tr>
<tr>
<td>(2c &amp; 4p)</td>
<td>300,764</td>
<td>944,394</td>
<td>5.31s</td>
<td>117.41s</td>
<td>117.79s</td>
</tr>
<tr>
<td>LOTOS</td>
<td>BPEL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>----------------------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gates + offers</td>
<td>message, portType, operation, partnerLinkType (WSDL), and receive, reply, invoke (BPEL)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>termination ’exit’</td>
<td>end of the main sequence</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sequence ’;’</td>
<td>sequence</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>choice ’[’</td>
<td>pick and switch</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>parallel composition ’[..]’</td>
<td>interacting WSs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>recursive call</td>
<td>new instantiation or while</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>datatypes and operations</td>
<td>XML Schema, DBs, XPath, etc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>guards</td>
<td>case of switch</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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


