

BoPi - a distributed machine for experimenting Web Services technologies

Samuele Carpineti

Phd Student - University of Bologna

June, 9 2005

A joint work with Cosimo Laneve, Paolo Milazzo

Outline

- Introduction to Web Services composition
- Language
 - ▶ types
 - ▶ primitives
- Loading
- Runtime
- Conclusions

Building complex programs 1/2

Complex applications are built composing small components.
In strictly coupled systems we usually assume:

- a common data model
- the common description of the components
- a status of the components

Building complex programs 2/2

In loosely coupled distributed systems like Web services we have:

- a standard data model XML
- a standard description of the components WSDL

WSDL

Abstract Information: schema
Concrete Information: location

- stateless components interacting by exchanging messages

Standard Languages and Composition

C#, Java, ... can be used for composing Web services but:

- the type system cannot express XML documents natively
- they not provide for the right level of abstraction
- they are designed for strictly coupled applications

BPEL4WS

BPEL4WS is the proposed standard language for composition:

- XML as data language
- WSDL as service interface language
- XQuery/XPath as interrogation language

BPEL4WS has: variables, parallel execution of activities, send and receive commands, faults and exception handling, name passing (of untyped endpoint references).

BPEL4WS does not have a clear formal description

The goals of the BoPi project

- a distributed implementation of the asynchronous π -calculus
- a formalized core language for *composing* Web Services
- a platform for *studying* Web Services technologies

... (statically) typed language + runtime

The goals of the BoPi project

- a distributed implementation of the asynchronous π -calculus
 - a formalized core language for *composing* Web Services
 - a platform for *studying* Web Services technologies
- ... (statically) typed language + runtime

The BoPi language

Primitives: asynchronous π -calculus

Data and Types: XML-documents with typed channels (endpoint references)

Expressions: labelling, sequencing, constants (for building complex documents)

Patterns XML pattern matching (for deconstructing documents)

Type systems: Regular Expression Types + Channels

S	$::=$	schema	L	$::=$	labels
	T	base schema		a	(label)
	void	void schema		$L + L$	(union)
	$L[S]$	labelled schema		$L \setminus L$	(difference)
	$L[S], S$	sequence schema		\sim	(any label)
	$S + S$	union schema	F	$::=$	pattern
	$\langle S \rangle$	channel schema		void	empty pattern
	U	constant schema		$S \times$	binder pattern
				$L[F]$	labelled pattern
				F, F	sequence pattern

where we assume $\text{def } U_1 = S_1 \dots \text{def } U_n = S_n$

(1) in order to keep subtyping decidable only tail recursion is allowed

($\text{def } U = a[], U, b[] + \text{void}$ is forbidden)

(2) in order to keep subtyping polynomial unions start with different labels

Type systems: Regular Expression Types + Channels

S	$::=$	schema	L	$::=$	labels
	T	base schema		a	(label)
	void	void schema		$L + L$	(union)
	$L[S]$	labelled schema		$L \setminus L$	(difference)
	$L[S], S$	sequence schema		\sim	(any label)
	$S + S$	union schema	F	$::=$	pattern
	$\langle S \rangle$	channel schema		void	empty pattern
	U	constant schema		$S \times$	binder pattern
				$L[F]$	labelled pattern
				F, F	sequence pattern

where we assume $\text{def } U_1 = S_1 \dots \text{def } U_n = S_n$

(1) in order to keep subtyping decidable only tail recursion is allowed

($\text{def } U = a[], U, b[] + \text{void}$ is forbidden)

(2) in order to keep subtyping polynomial unions start with different labels

Type systems: Regular Expression Types + Channels

S	$::=$	schema	L	$::=$	labels
	T	base schema		a	(label)
	void	void schema		$L + L$	(union)
	$L[S]$	labelled schema		$L \setminus L$	(difference)
	$L[S], S$	sequence schema		\sim	(any label)
	$S + S$	union schema	F	$::=$	pattern
	$\langle S \rangle$	channel schema		void	empty pattern
	U	constant schema		$S \times$	binder pattern
				$L[F]$	labelled pattern
				F, F	sequence pattern

where we assume $\text{def } U_1 = S_1 \dots \text{def } U_n = S_n$

(1) in order to keep subtyping decidable only tail recursion is allowed

($\text{def } U = a[], U, b[] + \text{void}$ is forbidden)

(2) in order to keep subtyping polynomial unions start with different labels

Language primitives

$P ::=$		process
	0	nil
	new $x : \langle S \rangle$ in P	new
	$x!(E)$	send
	$x?(F).P$	receive
	spawn{ P } P	parallel
	match x with{ $ F_1 \rightarrow P_1 \dots F_n \rightarrow P_n$ }	match
	select{ $ x_1?(F_1) \rightarrow P_1 \dots x_n?(F_n) \rightarrow P_n$ }	select

A BoPi program is a list of process definitions: let $A(F; F) = P$;

- $x!(-) \equiv$ service invocation
- $x?(-).P \equiv$ service definition
- spawn \equiv flow (parallel) activity
- match \equiv XQuery/XPath
- select \equiv pick activity

new \equiv service creation (\notin BPEL4WS)

Language primitives

$P ::=$		process
	0	nil
	new $x : \langle S \rangle$ in P	new
	$x!(E)$	send
	$x?(F).P$	receive
	spawn{ P } P	parallel
	match x with{ $ F_1 \rightarrow P_1 \dots F_n \rightarrow P_n$ }	match
	select{ $ x_1?(F_1) \rightarrow P_1 \dots x_n?(F_n) \rightarrow P_n$ }	select

A BoPi program is a list of process definitions: let $A(F; F) = P$;

- $x!(-) \equiv$ service invocation
- $x?(-).P \equiv$ service definition
- spawn \equiv flow (parallel) activity
- match \equiv XQuery/XPath
- select \equiv pick activity

new \equiv service creation (\notin BPEL4WS)

Services as first class entities

- services can be dynamically created
- services can be safely sent to other services
 - ▶ services do not need to know each other before the execution
 - ▶ received services can be used only for sending values

`s?(<a[]> + <b[]> x). ...`

- services can be pattern matched:

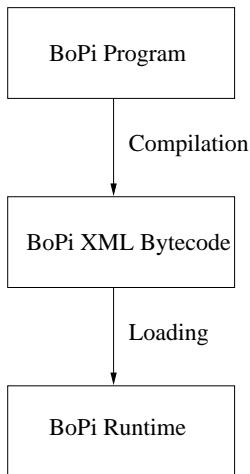
```
match x with {  
  | <a[]> -> P  
  | <b[]> -> Q  
}
```

An example

```
def OrderT=order[photo[_],(color[] + bw[])]

let PrintPhoto(<OrderT> print ; void) =
  print?(order[photo[_ p], (color[] + bw[]) how]
  spawn{ PrintPhoto(print ; void) }
  match how with {
    | color[] -> cPrinter!(p)
    | bw[] -> bwPrinter!(p)
  }
```

Running programs



- 1 the compiler generates typesafe bytecodes represented as XML documents
- 2 the loader loads bytecode fragments into one or more BoPi machines waiting for code to be uploaded

The loading phase

```
location MACHINE1 = www.machine1.com:2047/  
location MACHINE2 = www.machine2.com:2047/  
location MACHINE3 = www.machine3.com:2047/  
cPrinter = import www.printers.com/cp.wsdl  
bwPrinter = import www.printers.com/bw.wsdl  
print = new <OrderT>@MACHINE3  
load PrintPhoto.PrintPhoto(print; void)@MACHINE1  
load PrintPhoto.PrintPhoto(print; void)@MACHINE2
```

print is located at MACHINE3 and is used for receiving values by processes running at different locations.

Here we use linear forwarders (CONCUR03) for solving the input capability problem.

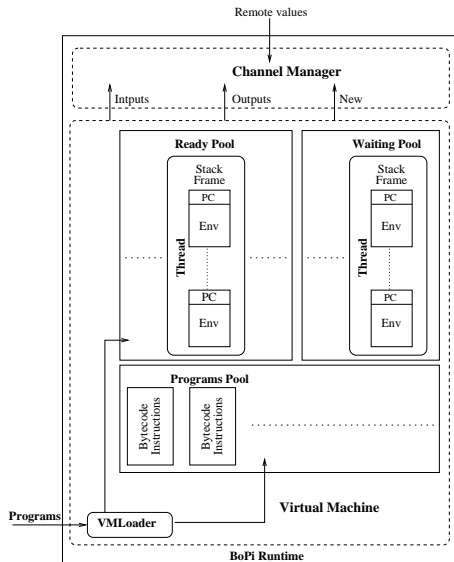
The loading phase

```
location MACHINE1 = www.machine1.com:2047/  
location MACHINE2 = www.machine2.com:2047/  
location MACHINE3 = www.machine3.com:2047/  
cPrinter = import www.printers.com/cp.wsdl  
bwPrinter = import www.printers.com/bw.wsdl  
print = new <OrderT>@MACHINE3  
load PrintPhoto.PrintPhoto(print; void)@MACHINE1  
load PrintPhoto.PrintPhoto(print; void)@MACHINE2
```

print is located at MACHINE3 and is used for receiving values by processes running at different locations.

Here we use linear forwarders (CONCUR03) for solving the input capability problem.

The runtime



- the virtual machine loads bytecodes and interprets their instructions
- the channel manager interfaces the virtual machine to the network and handles channels;

The BoPi machine logically

$$M_1$$

\mathbf{x}	\mathbf{y}
V	$y?(F).Q$
$x!(E), y?(F).P$	

- M_1 is the name of the machine
- \mathbf{x}, \mathbf{y} are the channels managed by the local channel manager
- $x!(E), y?(F).P$ are the ready processes
- $y?(F).Q$ is a process waiting for a value on \mathbf{y}

The interpreter

spawn : a new thread is added to the ready pool

match : the pattern matching algorithm is executed and the result (a substitution) is used in the continuation

process invocation : the body of the process is executed with the actual parameters

new/send/receive/select : are forwarded to the channel manager

The channel manager

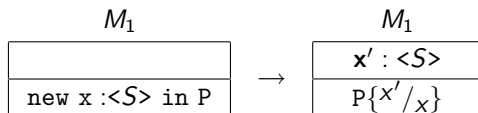
The channel manager is responsible for:

- channels creation (new)
- channels deletion
- outputs (send)
- inputs (receive, select)
- marshalling
- unmarshalling

It keeps a list of queues (one for each managed channel) containing input requests and outputs waiting to be consumed.

New

A local process creates a new channel `new x : <S> in P`

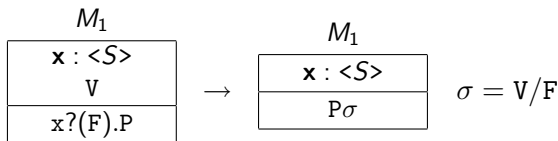
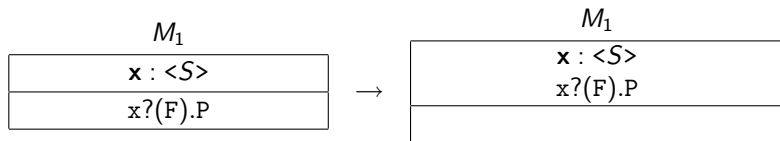


the channel manager creates a new **typed** queue and the WSDL interface of the channel.

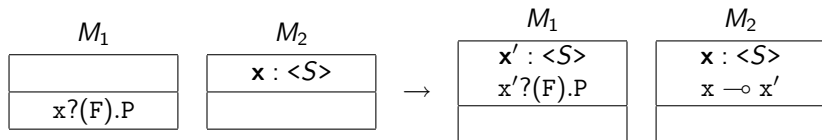
- the runtime representation of a channel is the URI the WSDL
- the channel is typed because data coming from untrusted parties need to be validated before processing

Receive

A local process receives on a local channel:

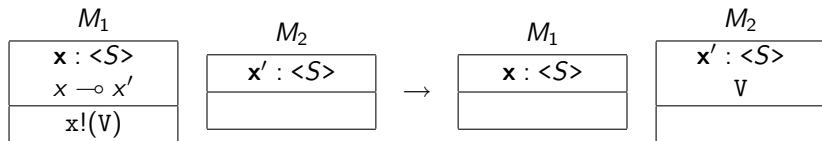
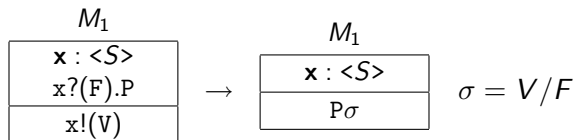
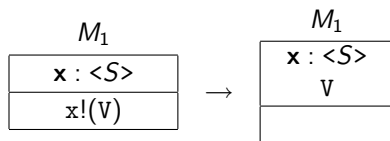


A local process receives on a remote channel:



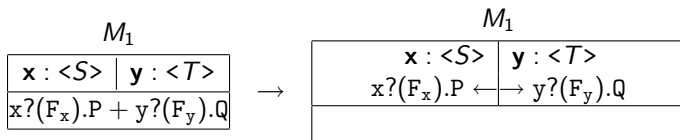
Local Send

A local process sends a documents over a local channel:



Select 1/2

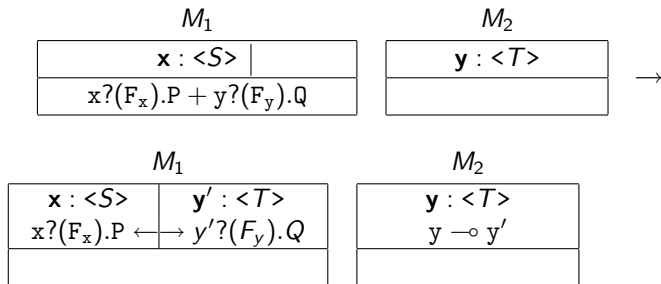
A local process performs an input on a list of local channels:



when a value is received on either x or y inputs linked in the list are deleted.

Select 2/2

A local process performs an input on a list of local/remote channels:



- if a value is received on the channel y' the input on x is deleted
- if a value is received on the local channel x the input of y' is deleted and an undo forwarder $y' \multimap y$ is created

Conclusions and future works

We presented:

- a distributed implementation of the full-asynchronous π -calculus
- a typed language for Web services composition and its runtime
- a simple architecture for experimenting Web services technologies (transactions, error/exception handling, ...)

We plan to:

- compile BPEL4WS into BoPi
- extend the language with transactions and timed-transactions
- study contracts