BoPi – a distributed machine for experimenting Web Services technologies

Samuele Carpineti
Dept. of Computer Science
University of Bologna, Italy

Cosimo Laneve
Dept. of Computer Science
University of Bologna, Italy

Paolo Milazzo
Dept. of Computer Science
University of Pisa, Italy

Abstract

BoPi *is a programming language with a runtime support that allows the distribution and the execution of programs over the network. The language is a process calculus with* XML *values and datatypes, and with a pattern matching mechanism for deconstructing values. The compiler gives a typesafe bytecode in the form of an* XML *document, that may be deployed on the network. What comes out is a simple, statically typed, and formally defined core* BPEL *language with a basic query mechanism supplied by patterns.*

1 Introduction

The BoPi project (www.cs.unibo.it/BoPi) has two main motivations. The first one is to provide a distributed implementation of the asynchronous π-calculus. In asynchronous π-calculus, a program (or process) has a collection of channels, and it executes through interaction over these channels. A natural distributed setting is to let each channel belong to a single location. For instance, there is one location for the channels x, y, z and another for u, v. A *service* $x(w).P$ goes in the first location; it waits to receive formal parameter w and then continues with P. If a *service request* $\overline{x}[v]$ should arise anywhere else in the system, it knows where to find the matching input resource. This basic scheme has been extensively studied (the $\pi_{1\ell}$ calculus [1], the local pi calculus [2], and is used in previous distributed prototypes (the join calculus [3], Microsoft Biztalk Server [4]).

However, we immediately face the problem of *input capability*, which is the ability in the (asynchronous) π-calculus, to receive a channel name and subsequently accept input on it. Consider the example $x(u).u(v).Q$. This program is located at (the location of) x, but upon reaction with $\overline{x}[w]$ it produces the continuation $w(v).Q\{^w/_u\}$ – and this continuation is still at x, whereas it should actually be at w. Solving the problem of input capability is a key challenge in distributing the π-calculus. The join calculus, the local pi calculus and the $\pi_{1\ell}$ calculus simply disallow input capability, on the grounds that it seems un-implementable. That is, in a term $x(u).P$, the P may not contain any inputs on channel u. Biztalk offers input capability when run over a reliable message service (MSMQ) but not otherwise. Implementation details of Biztalk have not been published; in this respect this paper may be seen as a formal alternative implementation.

We solve the input capability problem in BoPi using the theory of linear forwarders [5]. The solution consists of allowing just a limited atom of input capability – the *linear forwarder*. A linear forwarder $x{\multimap}y$ is a process which allows just one message on x to be turned into a massage on y. (A linear forwarder $x{\multimap}y$ may be safely considered as just the π-calculus process $x(u).\overline{y}[u]$.) To illustrate how linear forwarders encode input capability, consider the term $x(u).u(v).Q$. Then it is encoded as

$$x(u).(u')(u{\multimap}u' \mid u'(v).Q)$$

where the input $u(v)$ has been turned into a local input $u'(v)$ at the same location as x, and where the forwarder allows one output on u to interact with u' instead. The key observation is that the linear forwarder $u{\multimap}u'$ is easy to implement: it is just a packet containing two IP-addresses and directed to the location of u. Using this mechanism, the BoPi machine admits to import a remote (BoPi) service and to perform inputs on it.

The second motivation of the BoPi project is to design a distributed machine running applications that may be exported to the web (web services). For this reason we use the W3C standards WSDL and XML for describing interfaces and values, respectively. BoPi programs may construct XML documents and, by means of a pattern matching mechanism, deconstruct them. The compiler performs a semantic analysis (omitted in this paper, see [6]) guaranteeing that invalid documents can never be produced. The design of the BoPi datatype and pattern languages, as well as most of the algorithms regarding these features, have been strongly influenced by the XDuce [7] and CDuce [8] prototypes – two functional languages with native XML datatypes. We refer to [6] for a detailed discussion.

A major technical difficulty in the BoPi datatype language (with respect to XDuce and CDuce) is that values, as in XML, may contain endpoint references that are channels where values can be sent. An endpoint reference is represented by the Uniform Resource Identifier (URI in the following) of the WSDL interface describing the schema of the values it accepts. The semantics rules, which are omitted in this contribution (we refer the reader to [9]), expose an environment that is partially supplied by local service declarations and partially by the global environment. The maintenance of this environment means that communications also gather information about the schemas of the channels contained in the message. A related problem is found in the algorithm matching a document against a pattern (pattern matching). The algorithm checks if the document conforms the schema specified in the pattern and returns a set of variable bindings. As in XDuce, pattern matching in BoPi is implemented using top-down tree automata, but the presence of URIs inside values increases the complexity of the algorithm. In particular, verifying if a channel matches a pattern, requires to check if the schema of the channel is a subschema of the one specified in the pattern. This, in general, requires exponential time in the size of the tree automata of the pattern [10] and may significantly degrade the run-time efficiency of possible implementations. To alleviate this problem we add some restrictions on schemas that make subschema verification polynomial [9].

The result of our motivations – the BoPi machine – is a formally specified distributed machine running programs that are (statically typed π-calculus) processes with XML values, datatypes, and patterns. Channels in the BoPi machine have an associated WSDL document defining their schema and two interfaces: a HTTP interface for not-BoPi clients and a BoPi one for the others. They are first class entities that can be dynamically created and passed to other services that can use them for sending requests. Anticipating some of the syntax of the BoPi language, consider, for instance, the following service:

```
new response:<ok[] + notok[]> in
  s!(book["BoPi"], date["8-10-2004"],
     reply-to[response])
  response?(_ result)
```

Reading every `-!(-)` as a service invocation and every `-?(-)` as a waiting for a value, the example above creates the channel `response` where it receives the result of an invocation. The server accepting the request may either reserve the book and reply to the `response` channel or, as the following code does, pass the channel to another service that sends the notification to the client:

```
s?(book[string title], date[string x],
     reply-to[AnyChan y]) .
// reserve the book
notifier!(notifyTo[y])
```

As it stands, the BoPi machine is a programming technology for defining and experimenting web services. Compared to an emerging standard, the BPEL language [11], the BoPi language has mostly the same operations, except for exception handling and transactions. These operations are definitely relevant for web services, but their formal account and runtime support is outside the scope of this paper. A discussion of this issue is done in the conclusions.

This contribution is structured as follows. Section 2 is an informal introduction to the language constructs through examples. Section 3 specifies the bytecode language and defines the schema of the instructions. Section 4 describes the architecture of the BoPi machine, and in particular the channel manager and the virtual machine. Section 5 describes the loader directives. We conclude in Section 6. The formal details about the BoPi language and its semantics are omitted. The reader is referred to [9].

2 The BoPi programming language

We introduce the basic elements of the BoPi programming language: values and expressions, schemas, patterns, and processes. We also discuss type checking and pattern matching.

2.1 Values and expressions

Values are fragments of XML documents. A value can be an integer, a string, a channel literal – a Uniform Resource Identifier –, or a (possibly empty) sequence of labelled elements. Labelled elements are values tagged with a label. For example:

```
msg["hello"], doc[2]
```

is a document fragment containing a sequence of two elements labelled `msg` and `doc`, and containing the string `"hello"` and the integer 2, respectively. The value `void` represents the empty sequence. Non empty sequences never contain `void`, that is `msg["hello"], void, doc[2]` is not valid.

Channel literals may be channels of BoPi machines or not. In any case these literals have the shape:

```
http://www.BoPi.it/chan1.wsdl
```

namely the URI of the WSDL interface. The WSDL file contains the location where the service is available and the kind of messages it accepts. These information are used by the runtime environment for pattern matching and communications.

Expressions extend values with variables, equality tests, and standard operators for strings and integers. In order to build complex terms, expressions include sequence and labelling operators. For instance

```
a[i - j], b["the",x]
```

is an expression evaluating to a sequence of two labelled elements where the first contains the difference of the value of the integers variables `i`, `j` and the second contains the concatenation of the string `"the"` and the one stored in `x`. If the variable `x` is bound to the value `void`, the concatenation evaluates to the string `"the"`.

In the following, values as `a[void]` are shortened into `a[]`.

2.2 Schemas

Schemas describe collections of values that are structurally similar. For example,

```
a[int], b[string]*
```

describes all the documents with an `a`-labelled element containing an integer, followed by zero or more `b`-labelled elements containing a string. Schemas may be grouped by means of unions:

```
a[int] + b[string]*
```

describes the documents consisting either of a `a`-labelled element containing an integer or of a sequence of `b`-labelled elements containing strings. It is possible to abstract away from the name of the label using `~`. Therefore, the schema

```
~[int]
```

describes the documents consisting of an element with any label and containing an integer. Moreover, the schema:

```
(~\a)[int]
```

describes documents consisting of an element with any label apart from `a` and containing an integer. As for values, schemas as `a[void]` will be shortened into `a[]`.

`BoPi` schemas depart from `XML` DTDs or `XML` Schema for the so-called *channel schema*. A channel schema is written using the channel constructor $< \cdot >$. For example

```
<int>
```

describes the set of channel literals carrying integers.

In `BoPi` schemas are *well-formed* and *determined*. Roughly, a schema is determined if the arguments of the union operator (+) have disjoint set of labels. Therefore schemas like:

```
a[int] + a[string]
```

are forbidden. This constraint is for avoiding expensive runtime checks in the pattern matching algorithm.

Schemas include schema names that are bound by global definitions such as:

```
def t = a[int] + b[string]*
```

Recursion and mutual recursion is admitted in `def U=` S, provided S is *well-formed*. Roughly, S is well formed if schema names occur either within labelled elements or at the end of a sequence [9]. For example

```
def t = doc1[t]
def s = doc2[], s
```

are well-defined, while

```
def t = t, doc1[int]
```

is not legal. This restriction makes schemas correspond to regular tree grammars instead of context free tree grammars [10]. As a consequence, the subschema relation, used for type checking, is decidable in `BoPi`, whilst it is undecidable for context free tree grammars. Recursion can be used to derive the schemas collecting all the values and all the channel values. Let

```
def Empty = Empty
```

be the schema describing an emptyset of values. Then the schema describing every channel value is `<Empty>`, while the schema describing every value is:

```
def _ = (int + string + <Empty> + ~[_])*
```

2.3 Patterns

Patterns allow the deconstruction of values using matching. Patterns are mostly schemas annotated with variables. These variables are replaced at run-time by values, according to a pattern matching algorithm. For example

```
a[int i], b[_ j]
```

is a pattern that matches sequences of a `a`-labelled value containing an integer, and of a `b`-labelled value with any content. When the pattern matching succeeds, `i` is bound to an integer, and `j` to the content of the `b`-labelled value. For instance, the above pattern matches with `a[5], b[c["hello"]]`, binding `i` to `5`, and `j` to `c["hello"]`. Channel patterns specify the type of the channel literals to match. For instance,

```
<a[]> x
```

matches every channel literal carrying values of schema `a[]`. It also matches every channel literal carrying values `a[] + b[]`. The reason for this is that matched channels can be used only for sending values and a channel carrying values of schema `a[] + b[]` can be safely used for sending values of schema `a[]`.

A pattern as

```
(a[<string>] + b[<Empty>]) x
```

matches with all the values having either a label `a` containing a service of type `string` or a label `b` containing any service.

Patterns in `BoPi` are *linear with respect to variables* and *unambiguous*. Linearity means that, in `s x, t y`, variables `x` and `y` do not clash. Therefore, if the pattern matching algorithm succeeds, by linearity it yields exactly one binding for each variable. Unambiguity entails that there is always a unique substitution for unifying a pattern with a matching value (the pattern matching algorithm is deterministic [9]). For instance

```
(a[int] + void) v, (a[int] + void) w
```

is ambiguous because there are two possible substitutions for `v` and `w` when the matching value is `a[5]` (one binding `v` to `a[5]` and `w` to `void` and another binding `v` to `void` and `w` to `a[5]`).

2.4 Processes

Processes are the computing entities of the `BoPi` programming language. Interactions between processes is achieved by asynchronous message passing. The output of a value `V` on a channel `u` is written in `BoPi`:

```
u!(V)
```

The output is non-blocking: the sender does not wait until the receiver really reads the message. Elaborated forms of communication, such as rendez-vous, must be explicitly programmed by using the continuation passing style. When `V` contains channels, a process receiving `V` may use the channels for sending values: using them for receiving messages is disallowed in `BoPi` – only the output capability is transferred. The `BoPi` runtime inspects the channel to grep details about its implementation. Such details let to localize the service and to use the right protocol for outputs.

The operation for receiving messages is

```
u?(F). P
```

where `P` is the scope of the binders in the pattern `F`. `BoPi` allows inputs on a channel only when it is created by the process or it is a parameter of a process constant definition (see below). When a value of an output on `u` matches with the pattern `F`, the continuation $P\sigma$ is triggered, where σ is the substitution yielded by the pattern matching algorithm. The operation is always exhaustive: the pattern `F` matches every possible value carried by `u`. Names like `u` in `u!(V)` and `u?(F)` are called *subjects* in the following.

A process may wait simultaneously on *different* channels through the `select` statement – an operation similar to the `select` in socket programming, to the “pick activity” in `BPEL`, and to the *input-guarded choice* in π-calculus. The `select` statement groups several input-guarded processes and allows the progress of at most one of them. For instance, in the following code, either the `u` or the `v` branch will progress, but not both.

```
select
  { u?(int x).  P }
  { v?(string y).  Q }
```

In this paper, for simplicity, we require inputs underneath select to regard channels on the local machine. (The theory of linear forwarder and our current prototype also support the general form of select.)

The parallel execution of processes is specified by the `spawn` statement. For example:

```
spawn{ P } Q
```

executes `P` and `Q` in parallel. The `spawn` operator is the key to define a useful derived operation: `u!(E). P` means the process `spawn{ u!(E) } P`.

Channels may be created dynamically as in

```
new u:<S> in P
```

where `S` defines the schema of the values that can be sent and received over `u`. The scope of the declaration is restricted to `P`. Creating a channel amounts to define an URI address, as discussed in Section 2.1. In addition to what said before, if the channel must be published in a UDDI registry, the above declaration may be completed with further arguments, such as the name of the registry and the protocol for accessing to the service. On the contrary, if the channel must be used locally (it is never extruded in `P`) then the `WSDL` interface may omit the protocol (concrete) part.

Processes may parse the received values and tests whether their schemas are (subschema of) one of a list of schemas. This operation is defined by the `match` operator. Tests are performed in sequence, so that the first matching option will be performed, and the second will be executed only if the first one fails, and so on (first match semantics). The match operation can be used for testing the real schema of a received channel as in the following example:

```
u?(<b[]> x).
match x with {
  | <a[] + b[]> v -> P
  | <b[] + c[]> v -> Q
  | <Empty> v     -> R
}
```

According to the pattern matching rules, `x` may carry messages that include (but are not limited to) those of schema `b[]`. Henceforth, in order to choose the right branch of the `match`, the pattern matching algorithm must verify whether the actual schema of the received channel `x` is a subschema of `<a[] + b[]>` or of `<b[] + c[]>`. This verification is expensive to be performed at runtime (as we said, it has exponential cost). In order to avoid an expensive pattern matching, schemas have been constrained to be well-formed and *determined*, thus supporting a subschema algorithm with polynomial cost [6].

In BoPi it is also possible to define process names:

```
let A(F ; F') = P
```

The scope of the process name A is the entire file whilst the scope of variables in the patterns F and F' is only P. The formal parameters of A are partitioned into two parts by the semicolon. The pattern F is actually a sequence of channels, while F' can be any pattern. Channels in F can be used – in P – with both input and output capabilities (the type system enforces schema equivalence rather than a subschema relation). Channels in F' can be only used with output capability, as in the following example of a forwarding process:

```
let fwdInt(<int> u ; <int> v) =
  u?(int j).  v!(j)
```

Missing the argument u, the process should always forward a message from a given channel, fixed once for all, to another. Thus reducing flexibility.

We conclude with a bit more elaborated process. It defines a permanent photo printing service. Requests carry the photo to be printed and the kind of printing service (either color or black and white):

```
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)
  }
```

2.5 Type checking and the subschema relation

The BoPi compiler prevents a number of runtime type errors mostly regarding communications, invocations, and pattern matching. In particular, the compiler verifies that, for every

u!(E), the schema of E is a subschema of S, where <S> is the schema declared for u.

u?(F). P, the schema of F is a superschema of S, where <S> is the schema declared for u.

A(E ; E'), the schema of E is a sequence of channel schemas which are equivalent to the schema of the channels in the first pattern of the declaration of A; the schema of E' is subschema of the schema of the second pattern of the declaration of A.

match E with $\{(|\mathtt{F}_i \texttt{ -> } \mathtt{P}_i)^{i\in 1..n}\}$, the schema of E is a subschema of the union of the schemas of patterns $\mathtt{F}_i$, $i \in 1..n$ (exhaustivity). The compiler also warns the user if the schema of some pattern $\mathtt{F}_i$ is ambiguous or if it is a subschema of the union of the schema of the previous patterns (irredundancy).

These checks are performed by a subschema relation. The details of this relation and the underlying theory are in [6].

3 The BoPi bytecode

A BoPi program is compiled into a bytecode that is an XML document. As a consequence, bytecodes can be safely transmitted on the network allowing remote execution of processes.

The bytecode, although similar to the BoPi language, has three important differences. First of all, bytecodes use integers instead of variables. Integers represent indexes of the local memory. The second difference is for evaluating expressions. To this aim a store instruction saving partial results is used. The third difference is that bytecodes are flat. Thus scopes and sequentiality are implemented by means of jump and terminate instructions (see below).

A bytecode program is a sequence of BoPi schema declarations, followed by a sequence of process constant declarations. BoPi schema declarations are identified by the element def. This element has an attribute name defining the schema constant and contains the XML representation of the schema. For example, the declaration def U= a[int,string] is compiled into:

```
<def name="U">
 <label name="a">
  <sequence><int/><string/></sequence>
 </label>
</def>
```

Patterns are compiled by using the same elements of schemas plus the element bind. This element has an attribute target defining the index of the variable to bind. The content of bind is the schema to match. For example, the pattern int x is compiled into:

```
<bind target="1"><int/></bind>
```

where 1 is the index of x in the local memory.

Process constant declarations are identified by the process element. This element has two attributes: the name of the process, and the maximum size of the environment. A process contains a sequence of elements. The first two are BoPi patterns, the rest – the body – is a sequence of instructions.

BoPi instructions are defined by the set of XML elements listed below:

<jump ... />: is an empty element with an attribute steps defining the number of instruction to skip.

`<terminate/>` is an empty element which terminates the execution of the process.

`<send ... />`: is an empty element with two attributes: `ch` defines either the subject name or its index in the local memory; `value` defines the index of the argument in the local memory. For example `<send ch="1" value="2"/>` defines an output of the value stored in `2` on the channel stored in `1`.

`<recv ...>...</recv>`: is an element with an attribute `ch` that defines either the subject name or its index in the local memory. The content is a pattern.

`<select> ... </select>`: is an element containing a sequence of `case` elements. The content of each `case` element is a `recv` element followed by a `jump` to the proper continuation.

`<new ...>...</new>`: is an element with an attribute `target` defining the index of the name in the local memory. The content is a `BoPi` schema.

`<match> ... </match>`: is an element with an attribute `value` defining the index of the variable containing the argument. Its content is a sequence of pairs: the first element is the pattern, the second is a `jump` to the proper continuation.

`<spawn .../>`: is an empty element with an attribute `steps` defining the number of instructions of the spawned process.

`<invoke .../>`: is an empty element with three attributes: `name` defines the process constant; `first` defines first argument of the invocation; `second` defines the second argument of the invocation.

`<store ...>...</store>`: evaluates an expression and writes the result into the variable defined in the attribute `target`. The content is the bytecode of the expression.

The result of the compilation of `PrintPhoto` is the following bytecode program.

```
<process name="PrintPhoto" envSize="4">
 <pattern>
  <bind target="0">
    <chan><type name="OrderT"/></chan>
  </bind>
 </pattern>
 <pattern>
  <bind target="1"><void/></bind>
 </pattern>
 <recv ch="0">
  <sequence>
   <label name="order">
    <bind target="2"><any/></bind>
   </label>
    <bind target="3">
     <choice>
      <label name="color"/><label name="bw"/>
     </choice>
    </bind>
   </sequence>
  </recv>
  <spawn steps="2"/>
  <invoke process="OrderT" first="0" second="1"/>
  <terminate/>
  <match value="3">
   <pattern><label name="color"/></pattern>
   <jump steps="1"/>
   <pattern><label name="bw"/></pattern>
   <jump steps="3">
  </match>
  <send name="cPrinter" value="2"/>
  <terminate/>
  <send name="bwPrinter" value="2"/>
  <terminate/>
 </process>
```

When the bytecode is loaded, it is verified and serialized. The object code that is interpreted by the `BoPi` virtual machine is an array of instructions. In the case of the bytecode of the `PrintPhoto` example we obtain the following array.

```
process Name=PrintPhoto(<OrderT> 0; void 1)
        envSize=4
1:  0.rcv(order[_ 2], (bw[]+color[]) 3)
2:  spawn +2
3:  call PrintPhoto 0 1
4:  terminate
5:  match 3 with color[]->jump+1 bw[]->jump+3
6:  cPrinter.send(2)
7:  terminate
8:  bwPrinter.send(2)
9:  terminate
```

4 The `BoPi` architecture

The `BoPi` architecture is distributed: it is composed by a number of instances of runtime environments running at different locations and interacting by exchanging messages over channels. Each `BoPi` runtime, as illustrated in Figure 1, consists of two main components running as threads: the channel manager and the virtual machine.

The channel manager interfaces the network to the virtual machine. It handles messages sent and received on local channels and delivers messages to remote machines.

The virtual machine executes several threads simultaneously, by interpreting the corresponding bytecode instructions (every thread corresponds either to a loaded process or to a spawned one). It consists of a pool of environments of running threads plus the bytecodes that they are executing. In what follows, we discuss the two components separately.

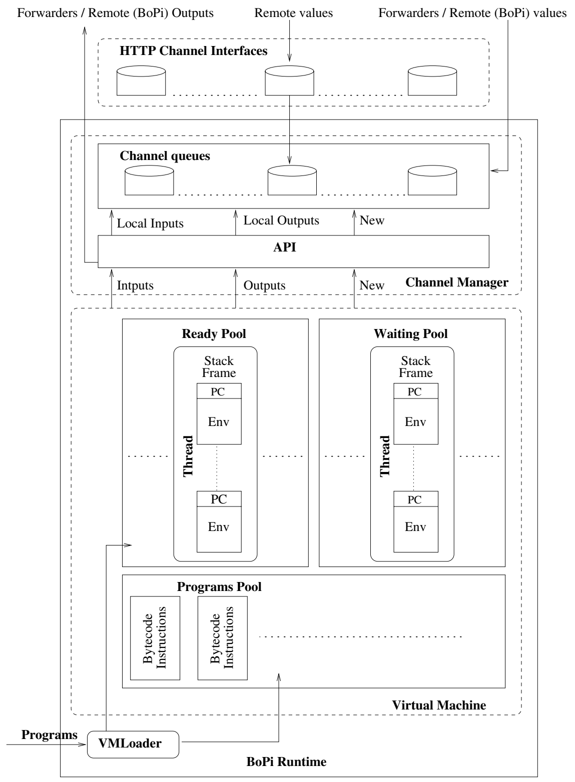

Figure 1: `BoPi`: The runtime

4.1 The virtual machine

The `BoPi` virtual machine includes two main `Java` threads: the first waits for new bytecode fragments coming from the loader, and the second schedules the execution of `BoPi` threads and interprets bytecode instructions. The data structures of the virtual machine are stored in three pools: the programs pool, containing the bytecodes of the running threads, the ready pool, containing threads ready to be interpreted, and a waiting pool containing threads blocked on some receive or select instruction. In order to guarantee safe updates, these pools are protected by mutexes.

Loading a program on a `BoPi` virtual machine amounts to adding the code to the programs pool and to creating and initializing some new threads in the ready pool. The initialization sets the program counter and the environment of each thread. The scheduler selects the first thread of the ready pool (which is a queue) and executes it for a fixed number of instructions or until it blocks. After the execution the thread is enqueued at the end of the ready pool. (This round-robin policy, in combination with the FIFO policy of channel manager queues, guarantees fair executions.)

A thread is executed by interpreting its bytecode. The input and output operations are forwarded to the channel manager. In case of inputs – a receive or a select operation – a thread identifier is passed to the channel manager and the thread is moved to the waiting pool. The thread is moved from the waiting pool to the ready pool directly by the channel manager when a message satisfying the input request is received. The following method is provided by the virtual machine in order to permit the channel manager to unblock the waiting thread:

`void wakeup(Threadid id, XMLnode value, int n)`: takes the `id` of the thread, the received value `value` and an integer `n` identifying the channel that has reacted. It wakes up the thread `id`, stores `value` into its stackframe, and sets the program counter according to `n`. This `n` is `1` if the blocking operation was a receive; otherwise it is the ordinal of one input within a select.

In case of a spawn, a new thread is created, initialized with a reference to the environment of the current thread and with the address of the first instruction of the spawned process. The new thread is added to the ready pool. We remark that the use of the same environment for both the current thread and the new thread is consistent because `BoPi` variables are read-only. In particular, since the values of variables never change, it is not necessary to copy the environment of the current thread to the new thread: variables can be accessed by dereferentiation.

4.2 The channel manager

The channel manager is responsible for the creation and deletion of channels. It manages the input and output operations and performs marshalling and unmarshalling of data. We discuss these operations one by one.

Channel creation. In `BoPi`, channels may be created dynamically; a channel is identified by a globally unique name carrying either the host name or the IP-address of the machine where it has been created and a channel ID. The creation of a new channel amounts to define a queue for storing pending input requests and outputs waiting to be consumed (given the exhaustiveness of the receive statement, every nonempty queue contains either outputs or inputs). A channel may be also created on a remote host (see Section 5). This operation blocks the creator thread until an acknowledgement for the creation is received.

In order to support web services access standards, the channel manager creates a `WSDL` file defining two interfaces (in `WSDL` terminology, two bindings): an HTTP interface and a `BoPi` interface. (Other protocols will be managed in future versions of the prototype.) The HTTP interface, implemented as a CGI script, waits for HTTP requests communicated via POST and carrying a value. When a value arrives, it is passed to the local channel manager. The `BoPi` interface defines the IP address and the port where a `BoPi` channel manager is waiting for network packets. We will specify the format of these packets in the following paragraph “Interfaces of the channel manager”. For example, the `PrintPhoto` service may have the following `WSDL` interface (the schema `Order` is omitted):

```
<!--ABSTRACT PART-->
<message name="OrderMsg">
 <part name="body" element="Order"/>
</message>
<portType name="PrintPhotoPT">
 <operation name="orderPhoto">
  <input message="OrderMsg"/>
 </operation>
</portType>
<!--CONCRETE PART: HTTP SERVICE-->
<binding name="httpB" type="PrintPhotoPT">
 <http:binding verb="POST"/>
 <operation name="orderPhoto">
  <http:operation location="printPhoto"/>
  <input>
   <mime:content type="application/xml"/>
  </input>
 </operation>
</binding>
<service name="PrintPhotoService">
 <port name="PrintPhotoPT" binding="httpB">
  <http:address location=
            "www.example.com/orderPhoto"/>
 </port>
</service>
<!--CONCRETE PART: BOPI SERVICE-->
<binding name="BoPiB" type="printPhotoT">
 <bopi:binding/>
 <operation name="orderPhoto">
  <bopi:operation location="chan1"/>
  <input>
   <mime:content type="application/xml"/>
  </input>
 </operation>
</binding>
<service name="PrintPhotoService">
 <port name="OrderT" binding="BoPiB">
  <bopi:address location=
           "bopi://www.example.com"/>
 </port>
</service>
```

The abstract part of this service defines the supported operations (i.e. the schema of the channel), whilst the concrete part defines the locations and the transport protocols that must be used for communicating with the service.

Channel deletion. The deletion of a channel amounts to delete the corresponding queue. (Deletion is performed for channels which become garbage.)

Channel outputs. When a value for a remote channel is received by the local channel manager, it forwards such value to the corresponding channel manager over the network. When the received value is for a local channel `u`, then the channel manager

1. verifies that the schema of the value conforms with the schema of `u` (only for messages arriving from the network). In case the two schemas are at odd, a (type) error message is emitted in the same communication;
2. checks if there are inputs waiting in the queue of `u`. If there are inputs then the message is forwarded to the stack frame of the receiver and the input is dequeued. Otherwise the message is enqueued.

The type checking in the first step is necessary because the `BoPi` channel manager accepts delivery of values that have not been produced by `BoPi` processes (type errors cannot occur in communications of `BoPi` programs).

Channel inputs. When an input request for a local channel `u` is received, the channel manager checks if there are values in the queue of such a channel. If a value is found, it is sent back to the thread which performed the input operation. Otherwise, the input request is enqueued.

It is worth to recall that, in the `BoPi` language, a process cannot input on received names. Notwithstanding this restriction, a program may have free inputs on remote channels (see the loader directives `new` or `import` in Section 5). In this case the channel manager creates a linear forwarder on-the-fly [5]. In particular, assuming `u` to be the remote channel, a new local channel `v`, with the same schema as `u` is created, and the linear forwarder process $\mathtt{u} \multimap \mathtt{v}$ is sent to the remote channel `u`.

When an input request is due to a `select` operation, the queues of the channels are inspected one by one. This is possible because inputs underneath a select concern local channels only. If there is a value in one of them, that value is sent back to the receiver. If there is no value, input requests are linked in a list and are added to the corresponding queues. When a value is enqueued in a queue of these linked channels, it is forwarded to the receiver thread and every input in the list is removed.

Interfaces of the channel manager. The channel manager may accepts packets directly from the network (the default port is `2047`). These packets are delivered by other `BoPi` machines and specify the operation that must be executed. In particular, there are three kind of packets:

`"SND";CHAN;LENGTH;DATA`**:** This packet describes a send on `CHAN` of the bytestream specified by `DATA`. The field `LENGTH` defines the size of the bytestream.

`"FWD";CHAN1;CHAN2`**:** This packet describes a forwarder. It informs that a message on `CHAN1` must be forwarded to the remote channel `CHAN2`.

`"NEW";REPLY_CHAN;LENGTH;SCHEMA`**:** This packet describes a remote channel creation. The channel manager creates a new channel whose schema is `SCHEMA` and communicates its name by sending a message on REPLY_CHAN. The field LENGTH defines the size the bytestream SCHEMA.

The interface of the channel manager to the virtual machine is a Java API consisting of (1) a set of objects representing abstractions for channels (Channel), schemas (Schema), and XML documents (XMLNode), and (2) the following set of methods:

Channel newChan(Schema S): returns a channel object representing a new local channel which handles messages of schema S;

void snd(Channel chan, XMLnode data): sends the XML document represented by data over chan. We recall that the behavior of the send depends on the locality of the subject channel;

void rcv(Channel chan, Threadid id): takes a channel chan, and a continuation (the thread identifier) id. It implements the receive operation as described above. We recall that, in case of remote receive, a new channel u is created, a linear forwarder chan$\multimap$u is produced, and the id is enqueued in the queue of u;

void select(Channel[] cvector, Threadid id): takes a nonempty list of channels and the continuation (the thread identifier). It implements the select operation as described before;

Channel getReference(String name): returns a reference to the channel "name" that can be either a BoPi channel or a not BoPi channel. This method is invoked to get the IP address of free channel names;

void delete(Channel chan): deletes the specified channel from the channel manager.

Marshalling and unmarshalling. Two basic operations of the channel manager are marshalling of XML messages from the virtual machine to the network and unmarshalling of bytestream messages arriving from the network. When a value is sent to a remote channel (a remote URI), it is marshalled into a bytestream before transmission. For example, marshalling takes the value a["hello"] and gives the document <a>hello</a>. However marshalling loses type informations of values. For instance both a[5] and a["5"] are marshalled into <a>5</a>. Henceforth, unmarshalling cannot be merely the inverse of the marshalling. For this reason BoPi schemas are restricted to be a subclass of the regular tree languages such that marshalling is an injection. In particular, schemas like a[int] + a[string] in which the two parts differ only on the base types (not in the structure) are forbidden. More formally, the disambiguating machinery is obtained by refining the notion of determined schema in [6] – this issue is overlooked in this contribution.

5 The loader

A BoPi program may be a process or a sequence of process constant declarations. A *loader* loads programs onto one or more BoPi machines, and triggers the corresponding processes or invoke a process constant therein, according to the directives of a *loading file*. A sample loading file is:

```
location SELLER = www.seller.com:2047
location STORE = www.store.com:2047
seller = new <OrderT>@SELLER
store = new <StoreT>@STORE
bank = import
  http://www.bank.it/checkCC.wsdl:CheckT
fedEx = import
  http://www.fedEx.it/shipTo.wsdl:ShipT
load SellerCode.Seller(seller; bank,store,
                            fedEx)@SELLER
load StoreCode.Store(store; void)@STORE
```

The location directives define BoPi machines by giving the URI and the port numbers where they are waiting for bytecode. It is assumed that an instance of the BoPi run-time is running on the hosts addressed by such directives. The new directive delegates the loader to create a channel at a given location (it uses the ability of the channel manager to create new channels, both locally and remotely); this directive has additional parameters for publishing the channel in a UDDI repository. The import directive imports an existing channel – a free name – declaring its schema. The load directive tells the loader to upload a piece of bytecode to the defined locations and to trigger the suitable process constant definition. In the above example the local SellerCode file containing bytecode is uploaded to the location SELLER, and the local StoreCode file is uploaded to location STORE. When the bytecode is uploaded the process constants named Seller and Store are invoked. The parameters written in these invocations are typecheked if compatible with the process constant definition.

It is worth to notice that, according to the BoPi loader syntax, in the above example, only Seller can use the channel seller as subject of an input. However both Seller and Store may use store for receiving data. If the loader file is extended with the lines

```
location SELLER' = www.shops.com:2047
load SellerCode.Seller(seller; bank,store,
                            fedEx)@SELLER'
```

we will have the same service running in two different machines (located at SELLER and at SELLER'), and virtually sharing the same input/output queue of the channel seller. In BoPi this sharing is implemented by means of linear forwarders that avoid distributed consensus by centralizing the inputs of a channel on a single channel manager that is located in the machine of the first declaration (in the above loading file, the one running at SELLER). An alternative declaration is to import a free BoPi channel, therefore allowing several agents to use the channel as subject of inputs. For example, the declaration of the new channel store in the loading file above may be replaced by the following import instruction:

```
store = import bopi
  http://www.BoPi.it/chan1.wsdl:StoreT
```

While it is possible to import a BoPi channel – the bopi attribute of the import directive – and use it with input capability, this is forbidden for generic imported channels because such channels may be handled by not-BoPi machines, thus not using forwarders. Then the line:

```
store = import
  http://www.store.it/buy.wsdl:StoreT
```

is refused by the loader because store is used with input capability but its machine is not a BoPi one (it lacks the attribute bopi attribute).

Declarations in the loading file are typed. This allows type checking at loading time. For instance, it is an error to load a program and invoking a process constant with arguments whose schemas are not subschema of the formal parameters. It is also error if the schema of the imported channels is not equivalent to the schema of the channel as used by the program. This amounts to download the type of every imported channel at loading time. In case of problems a loading type error is manifested.

6 Conclusions

A distributed implementation – the BoPi machine – of the asynchronous π-calculus with XML datatypes and pattern matching has been discussed. The resulting language seems helpful for programming web services, and this motivates our XML idioms, such as XML Schema and WSDL for types and interfaces, respectively.

Although the present BoPi language is expressive enough to model a number of interesting applications, there is place for studying extensions. The first one is to equip the language with primitive error handling and transactional mechanisms. These mechanisms deserve a thorough theoretical analysis and a careful implementation because they use time constraints and allow to coordinate processes located on different machines. We refer to [12] and the references therein for an initial investigation about transactions in the setting of BoPi.

A second extension concerns dynamic XML data, namely those data containing active parts that may be executed on clients' machines. This is obtained by transmitting processes during communications, feature called process migration. The BoPi machine already allows program deployments on the network at loading time. This is mostly due to the fact that bytecodes are XML files. The step towards runtime deployment is quite short: it suffices to introduce a new schema, the bytecode schema, and admit channels carrying messages of such schema. This is for instance the approach taken in [6].

References

[1] R. Amadio, "An asynchronous model of locality, failure, and process mobility," in *COORDINATION 1997*, vol. 1282 of *LNCS*, pp. 374–391, Springer-Verlag, 1997.

[2] M. Merro and D. Sangiorgi, "On asynchrony in name-passing calculi," in *ICALP 1998*, vol. 1443 of *LNCS*, pp. 856–867, Springer-Verlag, 1998.

[3] C. Fournet and G. Gonthier, "The reflexive chemical abstract machine and the join-calculus," in *POPL 1996*, pp. 372–385, ACM, ACM Press, 1996.

[4] Microsoft. Corporation, "Biztalk server." At http://www.microsoft.com/biztalk/.

[5] P. Gardner, C. Laneve, and L. Wischik, "Linear forwarders," in *CONCUR 2003*, vol. 2761 of *LNCS*, pp. 415 – 430, Springer-Verlag, 2002.

[6] A. Brown, C. Laneve, and L. Meredith, "PiDuce: A process calculus with native xml datatypes." At www.cs.unibo.it/BoPi, 2004.

[7] H. Hosoya and B. C. Pierce, "XDuce: A statically typed XML processing language," *ACM Transactions on Internet Technology (TOIT)*, vol. 3, no. 2, pp. 117–148, 2003.

[8] V. Benzaken, G. Castagna, and A. Frisch, "CDuce: an XML-centric general-purpose language," in *8th ACM SIGPLAN ICFP-03*, pp. 51–63, ACM Press, 2003.

[9] S. Carpineti, C. Laneve, and P. Milazzo, "The BoPi machine: a distributed machine for experimenting Web Services technologies." Full version. At www.cs.unibo.it/BoPi, 2005.

[10] H. Comon et al., "Tree automata techniques and applications." At www.grappa.univ-lille3.fr/tata, October, 2002.

[11] F. Curbera et al., "Business process execution language for web services (bpel4ws 1.0)." At www.106.ibm.com/developerworks/webservices/library/ws-bpel/, 2002.

[12] C. Laneve and G. Zavattaro, "Foundations of Web Transactions," in *FOSSACS'05*, vol. 3441 of *LNCS*, pp. 282–298, Springer-Verlag, 2005.