Encoding Abstract Descriptions into Executable Web Services: Towards a Formal Development

Antonella Chirichiello[1] and Gwen Salaün[2]

[1]DIS - Università di Roma “La Sapienza”, Italy
Email: `chirichiello@dis.uniroma1.it`
[2]INRIA Rhône-Alpes, France
Email: `Gwen.Salaun@inrialpes.fr`

Abstract. It is now widely accepted that formal methods are helpful for many issues raised in the web services area. In this report, we advocate the use of process algebra as a first step in the design and development of executable web services. From such formal descriptions, reasoning tools can be used to validate their correct execution. We define some guidelines to encode abstract specifications of services-to-be written using these calculi into executable web services. As a back-end language, we consider the standard orchestration language BPEL. We illustrate our approach through the development of an e-business application.

1 Introduction

Web services (WSs) are network-based software components deployed and then accessed through the internet using standard interface description languages and uniform communication protocols. Each service solves a precise task, and may communicate with other services by exchanging messages. Several XML-based standardized technologies have already been proposed to support WSs development: WSDL interfaces abstractly describe messages to be exchanged, SOAP is a protocol for exchanging structured information, UDDI is a repository to publish and discover WSs, BPEL4WS (BPEL for short) is a notation for describing executable business processes. The definition and deployment of WSs raise many issues which are part of on-going research. An important problem is the way to properly develop new services interacting with available ones.

Formal methods provide an adequate framework (many specification languages and reasoning tools) to describe WSs at an abstract level and then to tackle interesting issues, in particular their (automatic) composition or correct development. Different proposals have emerged recently to abstractly describe WSs and cope with these questions, most of which are grounded on transition system models (Labelled Transition Systems, Mealy automata, Petri nets, etc) [2, 15, 22, 13, 17]. However, very few approaches have been proposed to help the design and then development of WSs, especially from this kind of abstract descriptions (as done in the classical software engineering life cycle to develop software systems).

In this report, we advocate the use of process algebra (PA) [3] (*e.g.* CCS, π-calculus, LOTOS, Promela) as a starting point to develop WSs. As claimed in a previous work [25], PA is a simple, abstract and formally-defined notation to describe the exchange of messages between WSs, and to reason on the specified systems. Compared to the automata-based approaches, its main benefits are its expressiveness, its compositionality (allowing for the definition of more complex behaviours from simple ones) property, and its ability to describe real-size problems thanks to textual notations.

Central to our approach is the definition of a mapping between abstract processes and executable services (implementations and their associated interfaces) as sketched in Figure 1. The use of PA for development purposes may be considered in two ways: (i) *refinement* means specifying abstractly the new service and its interactions with the other participants, and then encoding it using an executable language, (ii) *reverse engineering* means extracting an abstract representation from the service implementation (accordingly the developer implements directly the service using an executable language) to validate its behaviour with respect to its mates. In both situations, we assume that the existing WSs (compared to the one(s) under development) have behavioural interfaces from which abstract descriptions can be obtained (prospective hypothesis particularly with regards to the current WSDL technology, see WS-CDL [29] as an example of proposal in this direction). The restriction of the service visibility to their public interfaces is a common situation in software engineering due to the black-box feature of components. *Orchestration* is a specific case of service development which aims at solving more complex tasks by developing a new service (often called *orchestrator*) using existing services by exchanging messages with them.

This report focuses on the refinement from PA to executable code. In this case, PAs are especially worthy as a first description step because they enable us to analyse the problem at hand, to clarify some points, and to sketch a (first) solution using an abstract language (then dealing only with essential concerns). Therefore, from such a formal description of one or more services-to-be, reasoning tools can be used to validate their correct execution and, if necessary, to verify and ensure relevant temporal properties such as safety and liveness ones.

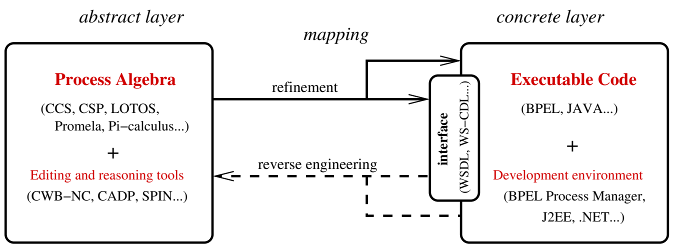

Fig. 1. Overview of our approach

Regarding the WS(s) to be concretely implemented (compared to the other ones which are viewed as behavioural interfaces), we concentrate ourselves, at the concrete level, on WSDL interfaces and BPEL services. We chose these technologies because they are well-known standards of widespread use and because BPEL is process-oriented therefore making the encoding tractable. Depending on the expressiveness of the process algebra used in the initial step, we can obtain either running BPEL code or just skeletons of code to be complemented. We stress that a formal refinement is not achieved yet since BPEL does not have a widely accepted formal semantics.

The organization of this report is as follows. We start in Section 2 with a short introduction of process algebra. For the sake of space, we do not introduce BPEL in this report, and the reader may refer to [1] for such a presentation. Section 3 defines guidelines formalising the encoding of process algebra into BPEL. Section 4 describes an example of e-business application in which all the steps advocated in our approach are successively coped with: analysis, formal description, reasoning, encoding of processes in BPEL services. Section 5 presents related work and compares our contribution to it. We draw up some concluding remarks in Section 6.

2 Introduction to Process Algebra

A PA [3] is a simple and formal language useful to specify dynamic behaviours (sequentiality, concurrency, parallelism, etc). Processes defined using them may communicate together by exchanging messages. Many process calculi (CCS [20], Timed CSP [27], LOTOS [4], π-calculus [23], Promela [14], etc) and accompanying tools exist, which offer a wide panel of expressiveness to deal with valuable issues in the WSs area (*e.g.* abstract description, composition, formal reasoning). In this section, our goal is not to introduce a precise algebra, but to present the common constructs appearing in most of them.

Actions and interactions. The basis concept to build dynamic behaviours (or processes) is the so-called *action* (also called event, channel, gate or name in other formalisms). Actions are either *emissions* or *receptions*, denoted respectively $OACT$ (Output ACTions) and $IACT$ (Input ACTions) in the sequel. Two (or more) processes can evolve in parallel and synchronize themselves when they are ready to evolve along the same action name; the basic matching is one sender and one receiver. Different communication models may be considered involving variants such as asynchronous *vs* synchronous communication, or binary *vs* n-ary communication.

Behavioural constructs. First of all, a *termination* END indicates the end of a behaviour. An *hidden* or *internal action* INT may be used to make abstract some possible pieces of behaviours corresponding to internal evolutions. The *hide* HID operator is sometimes used to make explicit the hiding of some actions to the environment of the process. The usual three main constructs are the *sequence* SEQ proposing the execution of an action followed by a behaviour, the *nondeterministic choice* CH between two behaviours which can be fired (sometimes, internal and external choices are used to distinguish the source of the choice), and the *parallel composition* PAR (and all its underlying variants like full synchronization or interleaving) meaning parallel evolution and synchronization among several processes. Many dynamic operators may be used and appear in some other calculi: interruption, sequential composition, or compensation and exception handling for constructs more related to WSs issues [5].

Data descriptions. Data DD are not always described within PA (this is not the case in basic CCS, for instance). Different levels of representation exist, for example in Promela, basic datatypes (*e.g.* integers, boolean) may be handled and one advanced construct (array) is available. A more expressive calculus is LOTOS which allows the representation of expressive data using algebraic specifications [7].

Data terms appear at different locations within dynamic behaviours. First, processes may be parameterized by a (optional) list of formal parameters (local variable declaration VD). Actions may be extended with value passing to exchange values: an emission $OACT$ may carry possible data terms, while a reception $IACT$ may be parameterized with variables. A behaviour may be preceded by a guard GRD, and is therefore executed only if the guard condition is true.

Processes. A process is composed of an identifier ID, a (optional) list of all the actions AD it involves, a (optional) list of its local parameters VD, and its behaviour BHV. The process body is built using the behavioural constructs described above, and in this way more complicated behaviours can be built from basic ones because PAs are compositional languages. The identifier is useful to refer to the behaviour of the process, and particularly to instantiate it and to call it recursively. Note that recursive calls may be used to update local variables.

Semantics and tools. These calculi are formalised either axiomatically with algebraic laws which can be used to verify term equivalences, operationally using a semantics based on Labelled Transition Systems, or denotationally defining the meaning of basic entities (and of their composition) using functions. These formalisms are most of the time tool-equipped, enabling one to simulate possible evolutions of processes, to generate test sequences, to check properties (*e.g.* to ensure that a bad situation never happens), to minimize behaviours, to verify equivalences, etc. CADP[1], CWB-NC[2], SPIN[3] are examples of such verification tools.

In this report, we illustrate the writing of service specification using the LOTOS calculus. To make easier the reading of the forthcoming pieces of specification, we give in Table 1 the correspondence between the PA abstract operators mentioned above and the ones used in LOTOS.

3 From Process Algebra to BPEL

In this section, we define guidelines to enable developers to write out easily advanced skeletons of BPEL code from abstract and validated descriptions of the service(s)-to-be. The presentation of this section follows the structure of the introduction to PA done in Section 2. For each PA element (actions, interactions, dynamic operators, data descriptions, processes), we describe how it can be encoded in BPEL and we illustrate with pieces of XML code. Here, we refer to PA and its underlying constructs in a general way, even though we illustrate with pieces of LOTOS specification. Many simple examples of such encodings (with the comprehensive LOTOS and BPEL code) are available on-line at this URL:

[1] `http://www.inrialpes.fr/vasy/cadp/`

[2] `http://www.cs.sunysb.edu/∼cwb/`

[3] `http://spinroot.com/spin/whatispin.html`

Abstract constructs	LOTOS constructs
$OACT$	act`!`v (* emission of value v *)
$IACT$	act`?`$x : t$ (* reception in var. x of type t *)
END	`exit`
INT	τ
SEQ	act`;`B
CH	B_1`[]`B_2 (* B_1 and B_2 synchronize on *)
PAR	B_1`|[`*sync-actions*`]|`B_2 (* sync. actions *)
HID	`hide` $a_1, ..., a_n$ `in` B
DD	*algebraic specifications*
GRD	`[`*bool-exp*`]` $\rightarrow B$
ID,AD,VD,BHV	P`[`*action-list*`](`*var-list*`):=` B

Table 1. Correspondence between abstract and LOTOS constructs

`http://www.dis.uniroma1.it/∼chiri/DEVofWS`. A more comprehensive example will be introduced in Section 4 to show how to use these guidelines to develop concrete e-business services.

3.1 Actions and interactions

The basis piece of behaviour in PA, the so-called action, is translated in WSDL using messages which are completely characterized by the *message*, *portType*, *operation* and *partnerLinkType* tags. As an example, these elements are set adequately below in order to encode abstract (resp. input and output) actions `request` and `result`.

```
<message name="requestMessage">
   <part name="request" type="xsd:int"/>
</message>
<message name="resultMessage">
   <part name="result" type="xsd:int"/>
</message>
...
<portType name="requestResultPortType">
   <operation name="requestResultOperation">
      <input message="tns:requestMessage"/>
      <output message="tns:resultMessage"/>
   </operation>
 </portType>
 ...
 <plnk:partnerLinkType
        name="requestResultPartnerLinkType">
    <plnk:role name="requestResultAdder">
       <plnk:portType
          name="tns:requestResultPortType"/>
    </plnk:role>
    <plnk:role name="requestResultRequester">
       <plnk:portType
          name="tns:requestResultPortType"/>
       </plnk:role>
 </plnk:partnerLinkType>
```

An abstract reception is expressed in BPEL using the *receive* activity or the *pick* activity with one *onMessage* tag.

```
                    request?x:Nat
```

(PA / LOTOS)

$\downarrow$

(BPEL)

```
<receive partnerLink="Requester"
         portType="tns:requestPortType"
         operation="requestOperation"
         variable="x"/>
```

On the other side, an abstract emission is written in BPEL using the *asynchronous* or *one-way invoke* activity.

```
                    request!x
```

$\downarrow$

```
<invoke partnerLink="Adder"
        portType="tns:requestPortType"
        operation="requestOperation"
        inputVariable="x"/>
```

At the abstract level, an emission followed immediately by a reception may be encoded using the BPEL *synchronous* or *two-way invoke* activity, performing two interactions (sending a request and receiving a response). On the opposite side, the complementary reception/emission is written out using a *receive* activity (or a *pick* activity with one *onMessage* tag) followed by a *reply* one. As an example, we can imagine that such a communication on one side

```
               request!x; result?y:Nat
```

$\downarrow$

```
<invoke partnerLink="Adder"
        portType="tns:requestResultPortType"
        operation="requestResultOperation"
        inputVariable="x"
        outputVariable="y"/>
```

can be composed with a service doing on the other side

```
               request?x:Nat; result!x+z
                           ↓
<receive partnerLink="Requester"
         portType="tns:requestResultPortType"
         operation="requestResultOperation"
         variable="x"/>
         ...
<reply   partnerLink="Requester"
         portType="tns:requestResultPortType"
         operation="requestResultOperation"
         variable="y"/>
```

with the same name of port type and operation. We emphasize that the abstract term `x+z` does not appear in the BPEL code, because it was replaced by the variable `y` denoting the result of the term, and which has to be assigned beforehand to `y` using an *assign* tag (see Section 3.3).

As far as the number of participants to a communication is concerned, let us remark that binary interactions are encoded straightforwardly in BPEL (due to its peer-to-peer interaction foundation) whereas multi-party communications are expressed in BPEL decomposing it in as many needed two-party communications.

3.2 Behavioural constructs

In this part, we explain how the usual constructs belonging to all the existing process algebra (sequence, termination, hidden action, choice, parallel composition) are encoded in BPEL. The sequence SEQ matches the *sequence* activity. The abstract termination END corresponds to the end of the main sequence in BPEL.

```
request!x; result?y:Nat; exit
              ↓
         <sequence>
         <invoke ... />
         <sequence/>
```

An hidden action INT is translated in BPEL as a local evolution of a WS which is not visible from an external point of view. Accordingly, it corresponds to one (or more) *assign* statement. It may also correspond to a communication of the web service at hand with another party but not visible from outside. The HID construct making hidden actions explicit in PA is mapped in BPEL as the hiding of each concerned action (possibly involved in a communication with other partners, then not visible from outside).

The choice CH (possibly multiple) is translated using either the *switch* activity, defining an ordered list of *case* or the *pick* one, defining a set of activities fired by a message reception. Branches of a choice CH involve emissions and receptions, possibly preceded by guards.

In case of a choice among emissions, nondeterminism existing at the abstract level has to be removed. If guards are present, the encoding is straightforward using as many *case* tags as needed, and the *otherwise* tag may be used to translate mutually exclusive conditions. Without guards, determinism may be introduced in two ways: (i) adding a *pick* activity and then defining different messages whose arrival indicates the behaviour to be fired, (ii) using a *switch* activity defining *case* conditions involving for instance values provided beforehand, each condition firing a possible emission.

A choice among receptions without guards is straightforwardly translated with a *pick*: whenever a message comes, it is received on the correct port. In presence of guards, the translation is trickier because a *pick* activity cannot be introduced within a *case* one (a BPEL limit). However, it can be achieved reversing the reception and the guard evaluation. The result in BPEL for each branch is a *pick* with an *onMessage* tag preceding a switch activity used to test the condition and then to execute the behaviour if the guard is true.

A choice involving emissions and receptions (without guards) is translated in BPEL using a *pick* with one *onMessage* tag for each branch appearing at the abstract level. In case of an emission, it should be determinized beforehand (as explained in the case involving only emissions), and in case of reception it corresponds to a message coming from a partner. In presence of guards, we can reuse translation rules mentioned before: using a switch for an emission, using a pick and then a switch for a reception.

We give in Table 2 the correspondence between LOTOS constructs and skeletons of BPEL code, for three of all possible cases (combination of choice, emissions/receptions and guards) at the PA level. Comprehensive BPEL code for these cases and the missing ones can be consulted on line[4]. The proposed encoding makes it possible to obtain running BPEL processes from abstract specifications. Even if tricks and choice

[4] `http://www.dis.uniroma1.it/∼chiri/DEVofWS`

implementations are needed, due to the limits of BPEL as executable language (especially the way interactions among processes are managed, or activities combined), our guidelines make it easier such forthcoming translations.

The parallel composition PAR is used in two different cases: (i) it describes a composition of interacting services; (ii) it may be used internally to a service to represent two pieces of behaviour which has to be carried out in different threads with possible interactions. In both cases, each operand of the parallel composition is encoded as one BPEL WS meaning that for the case (ii), the architecture of the whole is not preserved. However, it is not a problem because WSs are defined in a compositional way, and such auxiliary WSs can be hidden from an external (or user) point of view, or not if one wants to make them reusable.

Finally, we note that many additional behavioural constructs exist among all the existing PA, which would be used and encoded afterwards in BPEL. We have already studied some of them and their translation is not always straightforward, although often tractable. However, it would be too long to comment all these constructs in this report.

3.3 Data descriptions

Three levels of data representation in PA have to be encoded in BPEL: type and operation declarations DD, local variable declarations VD, data management in dynamic behaviours (guards GRD and parameters of actions $IACT$ and $OACT$).

As far as DD datatype definitions are concerned, types are described using XML Schema in the WSDL files. Elements in the Schema are either simple (many built-in types are already defined, *e.g.* integer, string, Boolean) or more complex (*e.g.* list, set, etc, built using the *complexType* tag). Considering LOTOS algebraic specifications, correspondences are straightforward because such constructs are shared at both levels. We introduce below a simple datatype defining a product characterized by an identifier and a quantity, first in LOTOS and then in XML Schema.

```
type PRODUCTREQUEST is NATURAL, STRING

  sorts ProductRequest
  opns
    conspr (*! constructor *):
     String, Nat -> ProductRequest
    productId: ProductRequest -> String
    quantity: ProductRequest -> Nat
```

Case	LOTOS	BPEL

emission
+
emission
	`act1!x; ...`
	`[]`
	`act2!y:t; ...`
		`<pick...createInstance="yes">`
		`<onMessage...variable=...>`
		`......`
		`<!-- first emission -->`
		`<invoke...inputVariable=.../>`
		`......`
		`</onMessage>`
		`<onMessage...>`
		`......`
		`<!-- second emission -->`
		`<invoke...inputVariable=.../>`
		`......`
		`</onMessage>`
		`</pick>`
		
		OR
		
		`<receive...createInstance="yes">`
		`<sequence>`
		`<switch>`
		`<case condition=...>`
		`......`
		`<!-- first emission -->`
		`<invoke...inputVariable=.../>`
		`......`
		`</case>`
		`<otherwise>`
		`......`
		`<!-- second emission -->`
		`<invoke...inputVariable=.../>`
		`......`
		`</switch>`
		`</sequence>`

reception
+
reception
	`act1?x:t; ...`
	`[]`
	`act?y:t; ...`
		`<pick...createInstance="yes">`
		`<!-- first reception -->`
		`<onMessage...variable=...>`
		`......`
		`</onMessage>`
		`<!-- second reception -->`
		`<onMessage...>`
		`......`
		`</onMessage>`
		`</pick>`

emission
+
reception
	`act1!x; ...`
	`[]`
	`act2?y:t; ...`
		`<pick...createInstance="yes">`
		`<onMessage...variable=...>`
		`......`
		`<!-- emission -->`
		`<invoke...inputVariable=.../>`
		`......`
		`</onMessage>`
		`<!-- reception -->`
		`<onMessage...>`
		`......`
		`</onMessage>`
		`</pick>`

Table 2. Correspondence between LOTOS and BPEL constructs

```
     ...

endtype
```

$$\downarrow$$

```
 <types>
   <schema ...>
   <element name="ProductRequest">
      <complexType>
        <sequence>
          <element name="productId"
                   type="xsd:string"/>
          <element name="quantity"
                   type="xsd:int"/>
        </sequence>
      </complexType>
   </element>
   ...
   </schema>
 </types>
```

DD make it possible to define types but also operations on them. Such abstract operations may be encoded in BPEL following different ways. First, they can be defined within XPath expressions; several kinds of functions may be called at this level such as core XPath functions, BPEL XPath functions or custom ones. XPath expressions appear in the *assign* activity. They simplify the extraction of information from elements or the storage of a value into a variable. Complex data manipulation may also be written using XSLT, XQuery or JAVA (*e.g.* JAVA *exec* tags enable one to insert Java code into BPEL code). Another way to describe data operations is to use a database and adequate queries to access and manipulate stored information.

With regards to data appearing in behaviours, we gather on one hand $IACT$ and VD because they just involve variables, and on the other hand $OACT$ and GRD because they deal with data terms. First of all, we recall that action parameters in PA for emissions and receptions ($IACT$ and $OACT$) are translated using the *message* tag (in the WSDL file). Each *part* tag matches a parameter of an action.

BPEL variables are used to encode variables appearing in $IACT$ and VD. They are defined using the *variable* tag (global when defined before the activity part) and their scope may be restricted (local declarations) using a *scope* tag. If necessary, local or global variables may be initialized using an assign after their declaration.

```
<variables>
    <variable name="y" type="xsd:int"/>
    <variable name="z" type="xsd:int"/>
  ...
</variables>
```

Now, regarding terms appearing in *OACT* and *GRD*, an emission *send*(y) means that the data expression (*e.g.* `x+z` below) has to be built and assigned to variable y before sending using an *assign* tag, and more precisely the *copy* tag. As far as *GRD* is concerned, guards are defined in BPEL using the *case* tag of the *switch* construct.

```
                       [x<5] -> result!x+z; ...
                                   ↓

<switch>               <!-- &lt; means < in BPEL -->
 <case condition="bpws:getVariableData('x') &lt; 5">
 <assign name="result">
     <copy>
       <from expression="bpws:getVariableData('x')
                        +bpws:getVariableData('z')"/>
       <to variable="y"/>
     </copy>
   </assign>
   <reply ... variable="y"/>
 </case>
</switch>
```

We emphasize that data are not available in every process algebra. Since they are essential in BPEL, it seems judicious to use a process algebra expressive enough to specify data descriptions. Otherwise, they are directly encoded at the concrete level enhancing the BPEL skeletons obtained from the purely dynamic description.

3.4 Processes

As mentioned beforehand, abstract processes are encoded as BPEL services. A global abstract system is described using a main behaviour made up of instantiated processes composed in parallel and synchronizing together (they are not obliged to synchronize on all actions). Let us observe that the main specification ($P_1|P_2|...|P_n$, $|$ denoting a parallel composition) does not match a BPEL process. The correspondence is that each abstract instantiated process (P_i), pertaining to the global system mentioned previously, matches a BPEL WS. However, the architecture of the specification is not always preserved. In very specific cases (parallel behaviours within a process), it might be necessary to define additional services. The set of messages is not defined explicitely in BPEL, but the correspondence between interacting services is made explicit using the *partner link* declaring on which partner link type services interact, and their role in the communication (see below such an interaction definition between an adder and a requester).

```
<partnerLink
  name="Adder"
  partnerLinkType="tns:requestResultPartnerLinkType"
  partnerRole="requestResultAdder"
  myRole="requestResultRequester"/>
```

At a lower level, each abstract process is usually made up of three parts: action declarations AD, process parameters VD, and its behaviour BHV (body of the process). AD are encoded using messages / operations / port types / partner link types in the WSDL file and variables in BPEL. VD correspond to variables in BPEL and BHV is encoded using BPEL activities.

```
            process Adder [request, result] (z: Nat): exit :=
                                       ↓

  <process name="Adder" ...>
     ...
     <variables>
       <variable name="request"
         messageType="tns:requestMessage"/>
       ...
       <variable name="z" type="xsd:int"/>
     </variables>
     <sequence name="main">
     ...
     </sequence>
  </process>
```

Instantiations of processes are encoded in BPEL as an exchange of messages with the new process to be instantiated. In BPEL, instantiations always occur through a reception (a *receive* or a *pick* activity).

```
<receive ... createInstance="yes"/>
```

or

```
<pick createInstance="yes">
<onMessage ...> ...
```

In concrete applications, such instantiations are often customers filling in some forms on-line, and thus requesting some services on the web.

The regular way to encode PA *recursive process calls* is related to the notion of *transaction*. A transaction could be defined as a complete execution of a group of interacting WSs working out a precise task. In this context, only one execution of a process is enough, therefore abstract recursive behaviours are encoded as non recursive services (each transaction corresponds to a new invocation then instantiation of each involved service). Such an encoding is illustrated in Section 4. In the case of an abstract choice among different behaviours, some of them ending with recursive calls and other ones ending with exits, the *while* activity may be used. The condition of the *while* is computed from guards of behaviours ending with exits (conjunction of guard negations).

4 Application: a Stock Management System

Many services involved in e-business applications may be developed following our approach: auction bargaining, on-line sales (CDs, DVDs, flowers, etc), banking systems (own account management for instance), trip organizations, (car) renting, request engines, (hotel, show, etc) reservations, any kind of on-line payment, etc. Such a formal-based development is above all of interest to save time (and money), thereby to respect deadlines, and to favour the correctness of the result.

In this section, our goal is to focus on an example and to show how following our approach (analysis, specification, reasoning, encoding, running code) we can design and develop executable WSs. The comprehensive specification written in LOTOS and all the WSDL and BPEL files encoded for this example are available on-line at this URL:

`http://www.dis.uniroma1.it/∼chiri/DEVofWS`

4.1 Informal requirements and analysis

For space limitation and for the sake of comprehension, we illustrate here with a simple but realistic problem. Let us imagine that a chain of supermarkets (called local stores in the following to be more general) have to be supplied with goods (or products) every day (or as soon as needed, it depends on the local policy). All the needed goods are centralized in a central store whose role is to supply all the local stores when they request something. This central store has to be supplied as well by suppliers particularly dedicated to a specific product (*e.g.* vegetables or cold food).

Our goal herein is to develop the web service corresponding to the central store assuming the existence of services describing local stores and suppliers. It is obvious that many local stores and suppliers can interact with one central store (even if afterwards several instances of central stores could be created).

Local stores and suppliers are viewed through their public interfaces that we assume represented using a simple behavioural description. It is out of scope here to introduce an adequate language for interface descriptions. In this section, we introduce them using transition systems with parameterized actions. A local store (Fig. 2) sends a request (an identifier `id` and a quantity `qt`) to acquire a certain amount of a product and afterwards waits for an acceptance or a refusal. Similarly, a supplier receives a request and replies depending on its ability to satisfy the request.

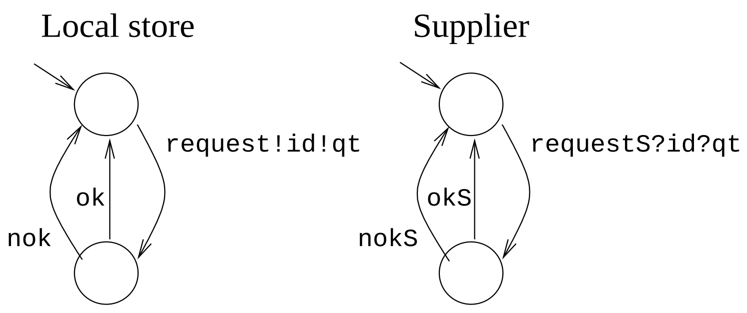

Fig. 2. Behavioural interfaces: local store and supplier

A central store may receive requests from local stores and replies depending on the availability of that product in its stock. It also has to supply its stock and then should request possible missing products to a supplier which answers depending on its ability to satisfy the request. As a simplified version, stocks can be viewed as sets of couples, each couple containing a product identifier and a quantity. A pictorial representation of all the interacting services is represented in Figure 3.

4.2 Specification and reasoning

In this report, LOTOS is appropriate to describe abstractly the system particularly thanks to its expressiveness (data and behaviours) and its

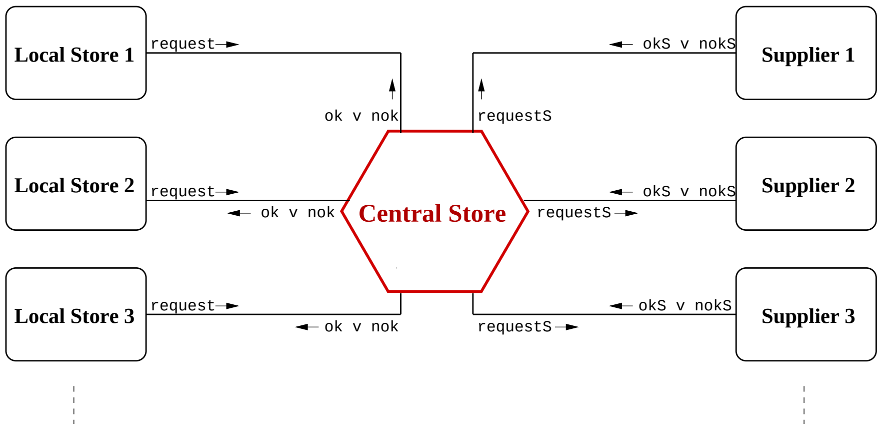

Fig. 3. Overview of the service-to-be and its mates

toolbox CADP is helpful to validate specifications. We start with the description of the data part. We represent a stock as a set of couple *(id,qt)* with two constructors (`empty` and `add`) to build it. Some operations allow to access and update a stock, such as the `increase` operation which increments the quantity of a product.

```
type STOCK is NATURAL, BOOLEAN

  sorts Stock

  opns  empty (*! constructor *) : -> Stock
        add (*! constructor *) :
                   Nat, Nat, Stock -> Stock
        increase : Nat, Nat, Stock -> Stock
        ...
```

From the local store and supplier automata describing their public interfaces, LOTOS specifications are easily deduced making the reasoning steps possible. We illustrate with the process `Supplier` in which the previous behaviour is extended to the management of a stock (declared as a parameter of the process) for verification purposes.

```
process Supplier [requestS, okS, nokS]
          (s: Stock): noexit :=

  requestS?id:Nat?q:Nat;
  (
    [isAvailable(id, q, s)] -> okS;
      Supplier[requestS, okS, nokS]
```

```
        (decrease(id, q, s))
    []
    [not(isAvailable(id, q, s))] -> nokS;
      Supplier[requestS, okS, nokS](s)
  )
endproc
```

Now, we focus on the central store process. This process has two possible behaviours: (i) it receives a request from a local store and it replies depending on the availability of the required quantity in the stock, (ii) in case of the existence in the stock of a product whose quantity is less than a threshold (to be defined arbitrarily), a request (the identifier and a quantity, 5 as an example) is sent to a supplier and a reply received. The stock is updated in case of delivery or restocking using adequate operations (`decrease` and `increase`).

```
process CentralStore[request, ok, nok, requestS,
                 okS, nokS] (s: Stock): noexit :=
   (
    request?id:Nat?q:Nat;
    (
      [isAvailable(id, q, s)] -> ok;
        CentralStore[request, ok, nok, requestS,
                 okS, nokS](decrease(id, q, s))
      []
      [not(isAvailable(id, q, s))] -> nok;
          CentralStore[...](s)
    )
    []
    requestS!extract(s)!5;
    (
      okS; CentralStore[...]
            (increase(extract(s),5,s))
      []
      nokS; CentralStore[...](s)
    )
   )
endproc
```

To validate this new service, several verification steps can be performed using CADP to ensure a correct processing of the interacting processes. Accordingly, we show a possible closed system which can be built from the previous definitions. It contains two local stores and two suppliers. Synchronization sets are explicitly declared, *e.g.* the central store interact with one local store along the `request`, `ok` and `nok` gates.

```
let
```

```
 s1: Stock =     (* a product identified by 1 *)
    add(1, 6,      (* and with 6 as quantity *)
     add(2, 5,
      add(3, 8, empty))),
        ...
in
  (
    Supplier[requestS, okS, nokS](s4)
      |||
    Supplier[requestS, okS, nokS](s5)
  )
      |[requestS, okS, nokS]|
    CentralStore[request, ok, nok, requestS,
                 okS, nokS](s1)
      |[request, ok, nok]|
  (
    LocalStore[request, ok, nok](s2)
      |||
    LocalStore[request, ok, nok](s3)
  )
```

Several mistakes in the specification have been found out (in the design, in the interaction flows, in the data description and management). Simulation has helped to clarify misunderstandings in the analysis of the problem at hand. Proofs may be verified using EVALUATOR (an on-the-fly model checker part of CADP) to ensure safety, liveness and fairness properties written in μ-calculus. Due to the simplicity of the problem at hand, we particularly check the absence of deadlock, and liveness properties ensuring for example that the firing of every `request` gate (also checked with `requestS`) is either followed by an acceptance or a refusal.

```
[true* . "request!*"] <("ok" or "nok")> true
```

4.3 Translation into BPEL

In this subsection, our goal is to emphasize how previous guidelines are used to encode abstract processes (written in LOTOS herein) into BPEL. We explain this translation through a sample of the BPEL code implemented for the central store.

Development environment. The Development was carried out using Oracle BPEL Process Manager 2.0, BPEL Console, BPEL Designer[5]. The process manager provides an infrastructure for deploying executing and managing BPEL processes.The console is useful to test the deployed

[5] `http://www.oracle.com/technology/products/ias/bpel /index.html/`

BPEL services. The designer makes it possible to build BPEL code from a developer-friendly visual tool, and to validate and pack BPEL processes. Data have been described using a Microsoft Access relational database (it is obvious that any DBMS could be used at this level). The package *java.sql* was helpful to access the database and the driver JDBC-ODBC to make the connection between Java classes and the database. We use the Java *exec* activity to insert Java code into BPEL code so as to call the Java methods defined to access and update the database.

Following the guidelines. Let us focus on the first half of the central store (the interaction with the local store) we specified previously in LOTOS. We overview how the constructs involved in this behaviour are expressed in BPEL. Firstly, LOTOS emissions and receptions (gates, variables and types) are encoded in the WSDL file. Note that the `ok` and `nok` actions have been encoded in BPEL using the same name of message but both branches are distinguished by the value of its Boolean parameter (*true* in case of availability, *false* otherwise).

Each stock is encoded as a simple table with two fields (identifier, quantity). A Java class was encoded and contains methods to access and update correctly each database (using SQL queries), *e.g.* to test the availability of a product. We highlight that the central store handles one request after the other. Thereby, it possibly updates the database at the end of each transaction and concurrent accesses to the base are then discarded.

The LOTOS sequence is directly translated using the *sequence* activity. The reception and the emission in this part of the central store are translated as a *pick* activity with one *onMessage* tag followed by a *reply*. The choice is composed of two branches with guards and emissions (without parameters), consequently a *switch* activity is employed to translate this LOTOS behaviour, and each branch is implemented using a *case* activity with guards corresponding to queries on the database.

Recursive calls are encoded as exits in the BPEL code. In the case at hand, we introduce the transaction notion corresponding to a request posted by a local store and to a reply depending on the availability of the product in the database. Thus, such a transaction is instantiated every time that a new request is received. In the LOTOS specification, recursive calls are accompanied of stock updates. It is done in BPEL modifying judiciously the database before the service completes.

Sample of BPEL. Our goal in this part is to introduce the skeleton of the BPEL code describing interactions between the central store and the local store. The central store service was implemented through a BPEL service, its WSDL interface, a database and a Java class implementing the useful methods to interact with the databases. For experimentation purposes, some other services were encoded (local stores and suppliers) using the same technologies. We remind that the WSDL and BPEL files corresponding to this problem may be consulted on-line and run using the BPEL Process Manager.

```
<sequence name="main">
  <pick createInstance="yes">
  <onMessage
    partnerLink="LocalStore"  portType="tns:CentralStore"
    operation="requestProduct" variable="productRequest">
  <!-- invocation from the LocalStore -->
  <sequence>
   ...
   <!-- verifying the availability of the product -->
    <bpelx:exec language="java" version="1.4">
     <![CDATA[
     ...
     //open the connection with the DB
  DBConnection cscDBC=new DBConnection("CentralStoreStock");
  if(cscDBC.isAvailable((String) productId.getNodeValue()))
      {
        setVariableData("available", new Boolean("true"));
      }
     //close the connection with the DB
     cscDBC.close();
     ]]>
    </bpelx:exec>
   <switch>
     <case condition=
             "bpws:getVariableData('available')=true()">
       <sequence>
        <assign name="isAvailable">
        <copy>
            <from expression="true()">
            </from>
            <to variable="productResponse" part="response"
                query="/tns:ProductResponse/tns:response"/>
        </copy>
        </assign>
        <!-- send the response to the LocalStore -->
        <reply name="response" partnerLink="LocalStore"
           portType="tns:CentralStore"
             operation="requestProduct"
               variable="productResponse"/>
        <!-- if the order is accepted we update the stock -->
        ...
       </sequence>
     </case>
     <otherwise>
       <sequence>
        <!-- state the response to false -->
        ...
        <!-- send the response to the LocalStore -->
        <reply .../>
       </sequence>
     </otherwise>
   </switch>
  </sequence>
  </onMessage>
... </sequence>
```

Implementing WSs in BPEL using all the necessary technologies (Java, databases, BPEL Process Manager) is not so simple. Nevertheless, with a minimum knowledge of such technologies, WSs can be implemented and deployed easily and pretty quickly (depending of course on the size of the application) following our approach.

5 Related Work

We are going to introduce three kinds of related work aiming at: (i) specifying WSs at an abstract level using formal description techniques and reasoning on them, (ii) mapping abstract descriptions and executable languages (mainly BPEL), (iii) developing WSs from abstract specifications.

At this abstract level, lots of proposals originally tended to describe WSs using semi-formal notations, especially workflows [18]. More recently some more formal proposals grounded for most of them on transition system models (LTSs, Mealy automata, Petri nets) have been suggested [15, 22, 13, 2, 17]. With regards to the reasoning issue, works have been dedicated to verifying WS description to ensure some properties of systems [10, 6, 22, 9, 21]. Summarizing these works, they use model checking to verify some properties of cooperating WSs described using XML-based languages (DAML-S, WSFL, BPEL, WSCI). Accordingly, abstract representations are extracted from WS implementations, and some properties may be ensured using *ad-hoc* or well-known tools (*e.g.* SPIN, LTSA).

In comparison to these existing works, the strength of our alternative approach (using PA) is to work out all these issues (description, composition, reasoning) at an abstract level, based on the use of expressive (especially compared to the former proposals) description techniques and equipped with state-of-the-art reasoning tools. Compared to the automata-based approaches, the main benefit of PA is its expressiveness, particularly due to a large number of existing calculi enabling one to choose the most adequate formalism depending on its final use. Additionally, another interest of PA is that their constructs are adequate to specify composition due to their compositionality property (not the case of the automata-based notations for instance). At last, textual notations (even if sometimes criticized to be less readable than transition systems) are more adequate to describe real-size problems, as well as to reason on them.

The second group of related work [10, 24, 9, 22, 28] deals with mappings between abstract and concrete descriptions of WSs. Let us emphasize that in other attempts (especially ours) [25, 26, 8], rough ideas have already been proposed to map process algebra and BPEL. The main improvement in this report is that guidelines are much more precise and everything was experimented, validated and run using the BPEL Process Manager. A relevant related work is [10] in which the authors present an approach to analyse BPEL composite web services communicating through asynchronous messages. They use guarded automata as an intermediate language from which different target languages (and tools) can potentially be employed. They especially illustrate with the formal language Promela and the corresponding model checker SPIN.

Compared to them, our attempt is slightly different while sharing some common points. First, we do not argue for a reverse engineering direction for the mapping but we propose a method to develop WSs. A judicious choice of our abstract language (*e.g.* LOTOS) makes the specification of advanced data descriptions and operations on them possible at the abstract level (more complex than in [10] where operations cannot be modelled as an example) as well as at the executable one. Our method was used to develop and deploy running WSs, consequently it has not remained at a conceptual level as it is most of the time the case.

Finally, we note that, to our knowledge, very few formal approaches have been proposed to develop WSs. The recent proposal of Lau and Mylopoulos [16] argues that TROPOS (an agent-oriented methodology) may be used to design WSs, but this work does not take into account the formal part of the methodology [11]. Mantell [19] advocates a tool to map UML processes into BPEL ones, but the semi-formality of UML is a limit to the validation and verification stage. On the industrial side, platforms like .NET and J2EE make it possible to develop WSs, but they do not propose methods (above all formal-oriented) to achieve this goal.

6 Concluding Remarks

Development of web services distributed on the web is a recent issue and the need of formal approaches to ensure a smooth and validated process is obvious for many e-business applications. In this report, we advocated such an approach starting from simple and abstract descriptions of processes and ending with executable web services implemented in BPEL. The two main contributions of our proposal are first a simplified development thanks to guidelines enabling one to encode abstract processes into executable ones. Secondly, validation steps are made possible through the use of formal reasoning tools, which usually accompany process algebra, thereby enabling one to detect possible errors and flaws at an early stage. Future work aims at developing a prototype accepting as input abstract specifications written using a process algebra (LOTOS in a first attempt), and generating as output the BPEL skeletons corresponding to the input descriptions. It will be implemented using the LOTOS NT compiler construction [12].

Acknowledgments. The authors thank Lucas Bordeaux and Marco Schaerf for judicious comments they made on this work. This work is partially supported by Project ASTRO funded by the Italian Ministry for Research under the FIRB framework (funds for basic research).

References

1. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Specification: Business Process Execution Language for Web Services Version 1.1. 2003.
2. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic Composition of E-services That Export Their Behavior. In *Proc. of ICSOC'03*, volume 2910 of *LNCS*, pages 43–58, Italy, 2003. Springer-Verlag.
3. J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra.* Elsevier, 2001.
4. T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. In P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–73. Elsevier Science Publishers North-Holland, 1989.
5. M. Butler and C. Ferreira. An Operational Semantics for StAC, a Language for Modelling Long-Running Business Transactions. In *Proc. of COORDINATION'04*, volume 2949 of *LNCS*, pages 87–104, Italy, 2004. Springer-Verlag.
6. A. Deutsch, L. Sui, and V. Vianu. Specification and Verification of Data-driven Web Services. In *Proc. of PODS'04*, pages 71–82, Paris, 2004. ACM Press.
7. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, New-York, 1985.
8. A. Ferrara. Web Services: A Process Algebra Approach. In *Proc. of ICSOC'04*, USA, 2004. ACM Press.
9. H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based Verification of Web Service Compositions. In *Proc. of ASE'03*, pages 152–163, Canada, 2003. IEEE Computer Society Press.
10. X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. of WWW'04*, USA, 2004. ACM Press.

11. A. Fuxman, L. Liu, M. Pistore, M. Roveri, and J. Mylopoulos. Specifying and Analyzing Early Requirements: Some Experimental Results. In *Proc. of RE'03*, pages 105–114, USA, 2003. IEEE Computer Society Press.
12. H. Garavel, F. Lang, and R. Mateescu. Compiler Construction Using LOTOS NT. In R. N. Horspool, editor, *Proc. of CC'02*, volume 2304 of *LNCS*, pages 9–13, France, 2002. Springer-Verlag.
13. R. Hamadi and B. Benatallah. A Petri Net-based Model for Web Service Composition. In *Proc. of ADC'03*, volume 17 of *CRPIT*, Australia, 2003. Australian Computer Society.
14. G. Holzmann. *The SPIN Model-Checker: Primer and Reference Manual.* Addison-Wesley, 2003.
15. R. Hull, M. Benedikt, V. Christophides, and J. Su. E-Services: a Look Behind the Curtain. In *Proc. of PODS'03*, pages 1–14, USA, 2003. ACM Press.
16. D. Lau and J. Mylopoulos. Designing Web Services with Tropos. In *Proc. of ICWS'04*, pages 306–314, San Diego, USA, 2004. IEEE Computer Society Press.
17. A. Lazovik, M. Aiello, and M. P. Papazoglou. Planning and Monitoring the Execution of Web Service Requests. In *Proc. of ICSOC'03*, volume 2910 of *LNCS*, pages 335–350, Italy, 2003. Springer-Verlag.
18. F. Leymann. Managing Business Processes via Workflow Technology. *Tutorial at VLDB'01*, Italy, 2001.
19. K. Mantell. From UML to BPEL. IBM developerWorks report, 2003.
20. R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.
21. S. Nakajima. Model-checking Verification for Reliable Web Service. In *Proc. of OOWS'02, satellite event of OOPSLA'02*, USA, 2002.
22. S. Narayanan and S. McIlraith. Analysis and Simulation of Web Services. *Computer Networks*, 42(5):675–693, 2003.
23. J. Parrow. *An Introduction to the π-Calculus*, chapter 8, pages 479–543. Handbook of Process Algebra. Elsevier, 2001.
24. M. Pistore, M. Roveri, and P. Busetta. Requirements-Driven Verification of Web Services. In *Proc. of WS-FM'04*, volume 105 of *ENTCS*, pages 95–108, Italy, 2004.
25. G. Salaün, L. Bordeaux, and M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. In *Proc. of ICWS'04*, pages 43–51, San Diego, USA, 2004. IEEE Computer Society Press.
26. G. Salaün, A. Ferrara, and A. Chirichiello. Negotiation among Web Services using LOTOS/CADP. In *Proc. of ECOWS'04*, volume 3250 of *LNCS*, pages 198–212, Germany, 2004. Springer-Verlag.
27. S. Schneider, J. Davies, D. M. Jackson, G. M. Reed, J. N. Reed, and A. W. Roscoe. Timed CSP: Theory and Practice. In *Proc. of REX Workshop on Real-Time: Theory in Practice*, volume 600 of *LNCS*, pages 640–675, Germany, 1992. Springer.
28. M. Viroli. Towards a Formal Foundation to Orchestration Languages. In *Proc. of WS-FM'04*, volume 105 of *ENTCS*, pages 51–71, Italy, 2004.
29. W3C. *Web Services Choreography Description Language Version 1.0.* Available at `http://www.w3.org/TR/2004/WD-ws-cdl-10-2 0040427/`.