Negotiation Among Web Services Using LOTOS/CADP

Gwen Salaün, Andrea Ferrara, and Antonella Chirichiello

DIS - Università di Roma "La Sapienza"
Via Salaria 113, 00198 Roma, Italia
Contact salaun@dis.uniroma1.it

Abstract. It is now well-admitted that formal methods are helpful for many issues raised in the web service area. In a previous work, we advocated the use of process algebra to describe, compose and reason on web services at an abstract level. In this paper, we extend this initial proposal, which only dealt with behavioural aspects, to cope with the question of representing data aspects as well. In this context, we show how the expressive process algebra LOTOS (and its toolbox CADP) can be used to tackle this issue. We illustrate the usefulness of our proposal on an important application in e-business: negotiation among web services. The connection between abstract specifications and running web services is made concrete thanks to guidelines enabling one to map LOTOS and the executable language BPEL in both directions.

Keywords: Web Services, Formal Methods, LOTOS, CADP, Negotiation, BPEL.

1 Introduction

Web services (WSs) are distributed and independent pieces of code solving specific tasks which communicate with each other through the exchange of messages. Some issues which are part of on-going research in WSs are to specify them in an adequate, formally defined and expressive enough language, to compose them (automatically), to discover them through the web, to ensure their correctness, etc. Formal methods provide an adequate framework (many specification languages and reasoning tools) to address most of these issues (description, composition, correctness). Different proposals have emerged recently to abstractly describe WSs, most of which are grounded on transition system models [2,11, 18,10,14] and to verify WS description to ensure some properties on them [18,7, 17]. In a previous work [19], we advocated the use of process algebra (PA) [3] for WSs. Being simple, abstract and formally defined, PAs make it easier to specify the message exchange between WSs, and to reason on the specified systems (*e.g.* using bisimulation notions to ensure the correctness of composition).

In this initial proposal [19], we especially experimented the use of the simple process algebra CCS. However, CCS turns out to be only adequate for the specification of (and reasoning on) dynamic behaviours. What was missing in this proposal was to handle *data*. This allows a much finer (less abstract) level of

L.-J. Zhang and M. Jeckle (Eds.): ECOWS 2004, LNCS 3250, pp. 198–212, 2004.

specification, which is clearly needed in some cases. In this paper, we argue that the process algebra LOTOS [12] and its toolbox CADP are useful respectively to describe WSs and to reason on them. We also propose a two-level description of WSs: an abstract one (using LOTOS) and an executable one (using WSDL and BPEL) Following such an approach, we can develop WSs considering the formal and verified specification as a starting point. In the other direction, we can abstract a deployed system to a description in LOTOS. The interest of such an abstract description is that the formality of this language and its readily existing tools enable one to validate and verify specifications through animation and proofs of temporal properties.

To illustrate the interest of such an approach in WSs, we focus on the problem of *negotiation* in which both data and dynamic aspects have to be dealt with. The perspective of intelligent/automated WSs which would be able to automatically perform the necessary negotiation steps to satisfy their user's request in the most satisfactory possible way emerged from artificial intelligence and multi-agent systems. This problem is a typical example of services involving both data (prices, goods, stocks, etc) and behaviours. Negotiation issues appear when several participants (clients and providers) have to interact to reach an agreement that is beneficial to all of them. Our goal is to show how LOTOS/CADP may be used to ensure trustworthy and automated negotiation steps.

The organization of this paper is as follows. First, we introduce in Section 2 the different entities involved in negotiation. Section 3 presents the LOTOS language and its toolbox CADP. They are used in Section 4 to describe negotiating processes at an abstract level, and to reason on them. Section 5 gives some guidelines to map LOTOS specifications and BPEL code in both directions. Related works are introduced in Section 6 and compared with the current proposal. Finally, we draw up some concluding remarks in Section 7. This paper is a shorter version of [20] in which the reader can find much more details.

2 What Does Negotiation Involve?

In this section, we introduce what is involved in negotiation cases. Specification and implementation of such aspects are resp. described in Sections 4 and 5.

Variables. They represent entities on which processes should negotiate, *e.g.* a price. Many variables may be involved in a negotiation at the same time (availability of different products, fees, maximum number of days for a delivery).

Constraints. They represent conditions to respect (called invariants as well) while trying to reach an agreement. Such an invariant is actually a predicate which can be evaluated replacing free variables with actual values. For a requester who is trying to buy by auction a product, such an invariant could be that (s)he is ready to pay €300 at most with a delivery within 10 days, or to accept a possible late delivery if there is a price reduction of 10% at least.

Exchanged information. To reach an agreement, both participants should send values to the other. A simple case is a price, but they can also exchange more advanced constructs (a record of values, a constraint on a value, etc).

Strategies. *"An agent's negotiation strategy is the specification of the sequence of actions (usually offers or responses) the agent plans to make during the negotiation."* [16]. Strategies may take into account other considerations. For instance, a participant can try to reach an agreement as soon as possible, or to minimize a price. Therefore, strategies are related to minimizing or maximizing objective functions.

In this proposal, we discard lots of possible variants which possibly appear in negotiations such as evolution of the constraints (a requester who is not finding a product less than €300, modifies his/her constraint to pay up to €320 from a certain point in time) or the level of automation (complete automation or intervention of human people). Consequently, all these variations may be combined and may end up to many possible scenarios of negotiation. It is obvious that no negotiation process is better than another. The right process should be selected depending on the bargaining context.

3 LOTOS and CADP in a Nutshell

LOTOS is an ISO specification language [12] which combines two specification models: one for static aspects (data and operations) which relies on the algebraic specification language ACT ONE and one for dynamic aspects (processes) which draws its inspiration from the CCS and CSP process algebras.

Abstract Datatypes. LOTOS allows the representation of data using algebraic abstract types. In ACT ONE, each *sort* (or datatype) defines a set of *operations* with arity and typing (the whole is called *signature*). A subset of these operations, the *constructors*, are sufficient to create all the elements of the sort. *Terms* are obtained from all the correct operation compositions. *Axioms* are first order logic formulas built on terms with variables; they define the meaning of each operation appearing in the signature.

Basic LOTOS. This PA authorizes the description of dynamic behaviours evolving in parallel and synchronizing using rendez-vous (all the processes involved in the synchronization should be ready to evolve simultaneously along the same gate). A process P denotes a succession of actions which are basic entities representing dynamic evolutions of processes. An action in LOTOS is called a *gate* (also called event, channel or name in other formalisms). The symbol **stop** denotes an inactive behaviour (it could be viewed as the end of a behaviour) and the **exit** one depicts a normal termination. The specific i gate corresponds to an internal evolution.

Now, we present the different LOTOS operators. The prefixing operator G**;**B proposes a rendez-vous on the gate G, or an independent firing of this gate, and then the behaviour B is run. The nondeterministic choice between two behaviours is represented using []. LOTOS has at its disposal three *parallel composition* operators. The general case is given by the expression B_1 |[G_1, ..., G_n]| B_2 expressing the parallel execution between behaviours $B1$ and $B2$. It means that $B1$ and $B2$ evolve independently except on the gates G_1, ..., G_n on which they evolve at the same time firing the same gate (they also synchronize

on the termination **exit**). Two other operators are particular cases of the former one to write out interleaving $B_1|||B_2$ which means an independent evolution of composed processes B_1 and B_2 (empty list of gates), and full synchronization $B_1||B_2$ where composed processes synchronize on all actions (list containing all the gates used in each process). Moreover, the communication model proposes a multi-way synchronization: n processes may participate to the rendez-vous.

The operator **hide** G_1, ..., G_n **in** B aims at hiding some internal actions for the environment within a behaviour B. Consequently, the hidden gates cannot be used for the synchronization between B and its environment. The *sequential composition* $B_1 \gg B_2$ denotes the behaviour which executes B_2 when B_1 has successfully terminated (**stop** or **exit**). The *interruption* $B_1 [> B_2$ expresses that the B_1 behaviour can be interrupted at any moment by the behaviour B_2. If B_1 terminates correctly, B_2 is never executed.

Full LOTOS. In this part, we describe the extension of basic LOTOS to manage data expressions, especially to allow value passing synchronizations. A process is parameterized by a (optional) list of formal gates $G_{i\in 1..m}$ and a (optional) list of formal parameters $X_{j\in 1..n}$ of sort $S_{j\in 1..n}$. The full syntax of a process is the following:

$$\textbf{process}\ P\ [G_0, ..., G_m]\ (X_0{:}S_0, ..., X_n{:}S_n) : func := B\ \textbf{endproc}$$

where B is the behaviour of the process P and $func$ corresponds to the functionality of the process: either the process loops endlessly (**noexit**), or it terminates (**exit**) possibly returning results of sort $S_{j\in 1..n}$ (**exit**$(S_0, ..., S_n)$).

Gate identifiers are possibly enhanced with a set of parameters (offers). An *offer* has either the form $G\textbf{!}V$ and corresponds to the emission of a value V, or the form $G\textbf{?}X{:}S$ which means the reception of a value of sort S in a variable X. A single rendez-vous can contain several offers. A behaviour may depend on Boolean conditions. Thereby, it is possible that it be preceded by a guard [*Boolean expression*] $\rightarrow B$. The behaviour B is executed only if the condition is true. Similarly, the guard can follow a gate accompanied with a set of offers. In this case, it expresses that the synchronization is effective only if the Boolean expression is true (*e.g.*, $G\texttt{?}X\texttt{:Nat[}X\texttt{>3]}$). In the sequential composition, the left-hand side process can transmit some values (**exit**) to a process B (**accept**):

$$... \ \textbf{exit}(X_0, ..., X_n) \gg \textbf{accept}\ Y_0{:}S_0, ..., Y_n{:}S_n\ \textbf{in}\ B$$

To end this section, let us say a word about CADP[1] which is a toolbox for protocol engineering. It particularly supports developments based on LOTOS specifications. It proposes a wide panel of functionalities from interactive simulation to formal verification techniques (minimization, bisimulation, proofs of temporal properties, compositional verification, etc).

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

4 Negotiation Using LOTOS/CADP

In this section, we do not argue for a general approach specifying any possible case of negotiation (even though many negotiation variants can be described in LOTOS). Our goal is to illustrate the use and interest of LOTOS/CADP for negotiating services. Consequently, we introduce our approach on a classical case of peer-to-peer sale/purchase negotiation involving one client (it works with more clients but we explain with one) and many providers. The goal of a formal representation and hence of automated reasoning is to prove properties so as to ensure a correct and safe automated negotiation between involved processes.

An assumption in this work is that we have a privileged view of all the participants and their possible behaviours (particularly in case of reverse engineering approach). Consequently, from our point of view, processes are glass boxes. This hypothesis is essential in any situation where we want to reason on interacting processes. An alternative approach would be to consider processes as black boxes and to reason on visible traces. However, it is almost impossible given such inputs, in which little information is available, to ensure critical properties.

Aspects involved in negotiation and itemized in Section 2 may be encoded in LOTOS in different ways. Let us show now an outline of the description we propose (see further for detailed explanations).

• Datatypes are defined using ACT ONE algebraic specifications and are afterwards used to type **variables** locally defined as parameters of processes.

• A specific datatype (`Inv`) is defined to describe **constraints** to be respected by participants while negotiating.

• **Exchanged information** are represented by variables. They can represent simple values (natural numbers) or more complex ones (a constraint on multi-type values). Values are exchanged between processes along gates.

• **Strategies** are much trickier because they are encoded either in the dynamic processes – the way a participant proposes some values and more generally interacts with other participants – and in the local variables managed by processes – initial value, computation of the next ones, constraints to be respected.

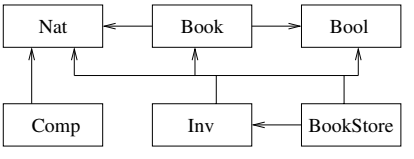
In this section, we introduce the main ideas of the negotiation situation at hand and illustrate them with short pieces of specification borrowed from the comprehensive one[2]. Additionally, this section does not make any assumption regarding the development approach (design approach or reverse engineering).

Case Study. We illustrate our proposal using an on-line auction sale. This case involves a client who interacts with book providers (one after the other) to purchase a book respecting some constraints. In the following, we choose to simplify the case for the sake of readability and comprehension (purchase of a single book, negotiating only the price). However, our approach works for more complex cases and many negotiation variants, as it has been experimented[3]: generalization of data management, handling of bigger sets of books, negotiations on several values and accordingly encoding of more complex invariants.

[2] `http://www.dis.uniroma1.it/~salaun/LOTOS4WS/BASIC`
[3] `http://www.dis.uniroma1.it/~salaun/LOTOS4WS/GEN`

Data Descriptions. First, the ADT abilities of LOTOS are useful to represent the data variables handled during the negotiation as well as the more complex data managed by processes. With regards to our negotiation problem, we need the datatypes gathered on the left-hand side in Figure 1 with their import links (Nat and Bool are already defined in the CADP library). A book is represented using four values: an identifier, a price, the delivery time, the possibility to return the book or not. A bookstore is described as a set of books. Each entry in the bookstore also contains the number of available books and the invariant to be respected, then deciding of the offer acceptance or refusal. Several operations are defined to access, modify and update these datatypes.

Constraints are represented using a generic datatype. They are encoded using the `Inv` sort which defines as many invariants as needed and a `conform` function evaluating invariants with actual values. In the example below (right-hand side in Fig. 1), the invariant means that the maximum price to be paid is €3 (client condition), the corresponding meaning of the `conform` function is written out using the judicious comparison operator on natural numbers. Another example shows how more complex invariant can be expressed (the price has to be less than or equal to three, the delivery within 5 days and the return possible).




```
pMax3 (*! constructor *) : -> Inv
conform : Nat, Inv -> Bool
...
eqns  forall p: Nat, b: Book
ofsort Bool    (* type of the result *)
  conform(p, pMax3) = (p<=3);
  conform(b, myInv) = (price(b)<=3) and
           (days(b)<5) and returnok(b);
```

Fig. 1. Datatype dependence graph (left) and sample of the datatype for constraint descriptions (right)

The computation of the initial value to be negotiated and of the next value to be proposed (*e.g.* adding up one to the price) are also encoded in a generic way using the `Comp` sort. For example, it is done for the price using some operations extending the `Nat` existing datatype.

Negotiating Processes. Now, let us define the processes involved in the negotiation steps. In the current negotiation example, we advocate a fully delegated negotiation service (the user does not intervene into search of an agreement and negotiation rounds). In Figure 2, boxes correspond to nested processes and lines to interactions between processes (they hold the gates used by the processes to communicate). Nested processes mean that, at one moment, there is only one control flow: one parent process instantiates another one and waits for the end of its fork before continuing its behaviour. It is compulsory in LOTOS because the use of an intermediate process is the single way to express a looping behaviour inside a bigger one.

The `Controller` process is run by the client with judicious parameters: the identifier of the book to purchase, an initial value to be proposed for the book

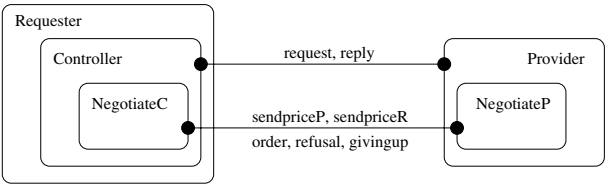


Fig. 2. Processes involved in our negotiation process

while negotiating, an invariant to be respected during the negotiation steps, a function computing the next proposal. Its *exit* statement is accompanied with a Boolean which indicates an order or a failure of the negotiation rounds. On the other hand, each provider has at its disposal a set of books, a function to compute a first value (to be proposed while negotiating) from the actual price as stored in the book set, and a function to compute the price of the book for the next negotiation round. Many parts of our encoding favour reusability, but values involved in the negotiation remain close to the current problem (book identifier, book price, bookstore). Accordingly, adjustments of these parameters are needed to reuse such negotiating processes in another context.

Let us sketch the behaviour of the controller. First, it initiates the interaction with a provider by sending the identifier of the book to be sought, then receives a Boolean answer denoting the availability of the book. A false reply implies a recursive call. A true response induces the instantiation of the negotiation process `NegotiateC` and the waiting for an answer accompanied with a status which is true in case of an order (and then exit alerting the client of the success) and false in case of a failure (recursive call to try another provider). At any moment, the controller may give up and exit alerting the client that every negotiation attempt has failed. Each provider has to be willing to interact with such a controller, and then is made up of the symmetrical behaviour.

```
process Controller
   [request,reply,order,refusal,sendpriceP,sendpriceR,givingup]
   (ref: Nat, pi: Nat, inv: Inv, computfct: Comp): exit(Bool) :=

   request!ref; reply?b:Bool;
   (
     ([b] ->
      NegotiateC[order,refusal,sendpriceP,sendpriceR,givingup]
      (pi, inv, computfct) >> accept status: Bool in
        (
          [not(status)] ->          (* case of a failure *)
            Controller[request,reply,order,refusal,sendpriceP,...]
              (ref, pi, inv, computfct)
          []
          [status] -> exit(true)     (* case of an order *)
        ))
     []
     ([not(b)] ->        (* let us try another bookstore *)
       Controller[request,...]
```

```
        (ref, pi, inv, computfct))
  )
  []
  exit(false)   (* the controller stops the negotiation *)

endproc
```

Below, we show the body of the `NegotiateC` process in charge of the negotiation on behalf of the controller. Negotiation steps are composed of classical information exchanges. One of both participants proposes a price to the other. The other participant accepts whether this price satisfies its local constraints. Otherwise, it refuses and calls recursively itself. The caller updates its local value (price here) for the next proposal using the local function dedicated to such computations. At any moment, the negotiation may be abandoned by one of the participants.

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

   sendpriceP?p:Nat;         (* the provider proposes a value *)
   ( [conform(p,inv)] -> order;exit(true)        (* agreement *)
     []
     [not(conform(p,inv))] -> refusal;
       NegotiateC[order,refusal,sendpriceP,sendpriceR,givingup]
        (curp, inv, computfct) )
   []                  (* proposal of a value to the provider *)
   ( [conform(curp,inv)] -> sendpriceR!curp;
     (
       order; exit(true)        (* agreement *)
       []
       refusal; NegotiateC[order,refusal,sendpriceP,sendpriceR,givingup]
                   (compute(curp,computfct),inv,computfct)
     ) )
   []    (* the client stops because results not satisfactory *)
   givingup; exit(false)

endproc
```

Let us introduce an example of a concrete system in which one requester is seeking a book among three possible providers. First of all, the needed data are described using appropriate algebraic terms. A simple example of a bookstore is the following one containing eight books (we experimented our approach with stores handling tens of books). For illustration purposes, the first book (identified by 0) has three copies still available and has to be sold at least €3.

```
bs1: BookStore =
    add2BS(book(0,2,6,true), 3, pMin3,
      ...
        add2BS(book(7,3,2,false), 3, pMin5, emptyBS)...)
```

A lightweight view of the system is now given. This instance (and some variants of it modifying the number of possible providers) is used in the next subsection to assess the reasoning capabilities. The requester is seeking the book identified by `0`, with `2` as initial negotiating value, `pMax3` as price constraint and a computation of the next value obtained adding up one.

```
Requester [request,reply,order,refusal,...] (0,2,pMax3,add1)
  |[request,reply,order,refusal,sendpriceP,sendpriceR,givingup]|
(
   Provider [request,reply,order,refusal,...] (bs1,times2,minus1)
    ||| Provider[...] (bs2,times2,minus1) |||  Provider[...] (...)
)
```

Reasoning. Our purpose here is to validate our negotiation system and to verify some properties on it. All the steps we are going to introduce have been very helpful to ensure a correct processing of the negotiation rounds. Several mistakes in the specification have been found out (in the design, in the interaction flows, in the data description and management). Simulation has been carried out and proofs have been verified using Evaluator (a CADP on-the-fly model checker) and the prototype version Caesar 6.3 [9] building smaller graphs than in the current distribution of CADP. Safety, liveness and fairness properties written in μ-calculus have been verified (examples are resp. introduced below).

A first classical property is the absence of deadlocks (actually there is one deadlock corresponding to the correct termination leading in a sink state). Another one (P1) ensures than an agreement is possible (because it exists one branch labelled by `ORDER` in the global graph). The property P2 expresses that the firing of every `SENDPRICER` gate (also checked with `SENDPRICEP`) is either followed by an order or a refusal. P3 verifies that after the firing of a `REPLY!TRUE` action, either `SENDPRICEP` and `SENDPRICER` are fireable.

```
< true* . "ORDER" > true                                              (P1)
[true* . "SENDPRICER!*"] <("ORDER" or "REFUSAL")> true                (P2)
[true* . "REPLY!TRUE"] <"SENDPRICEP" and "SENDPRICER"> true           (P3)
```

We end with a glimpse (Tab. 1) of experimentations we carried out on this system to illustrate from a practical point of view the time and space limits of our approach (and of the use of CADP). We especially dealt with one user since the goal is to ensure properties for a given user even if proofs can be achieved with several ones as showed below (however in this case the existence of an agreement P1 is meaningless). We used a Sun Blade 100 equipped with a processor Ultra Sparc II and 1.5 GB of RAM. To summarize, though we can reason on rather realistic applications, our approach is limited by the state explosion problem due especially in our case to the increase of: the number of processes (and underlying new negotiation possibilities), the number of values to be negotiated (if constraints are relaxed, negotiations can be hold on more values), the size of data (above all the number of books managed by each provider).

Table 1. Practical assessment of the approach

	Number of processes	Nb of states	Nb of trans.	P1	P2	P3
Example 1	(1 user & 1 provider)	32	47	3.84s	2.15s	2.21s
Example 2	(1 user & 5 providers)	2,485	5,124	4.88s	3.57s	3.57s
Example 3	(1 user & 7 providers)	17,511	42,848	4.64s	27.70s	27.35s
Example 4	(1 user & 8 providers)	35,479	88,438	5.02s	101.96s	102.27s
Example 5	(1 user & 10 providers)	145,447	374,882	5.10s	1326.94s	1313.16s
Example 6	(2 users & 4 providers)	300,764	944,394	5.31s	117.41s	117.79s

5 Two-Way Mapping Between LOTOS and WSDL/BPEL

First of all, we highlight once again the need of equivalences between behaviours written out in both languages, consequently each one can be obtained from the other. In this section, our goal is not to give a comprehensive mapping between both languages, that is a mapping taking into account all LOTOS and BPEL concepts. Herein, we focus particularly on the notions that we need for the negotiation problem as mentioned in the LOTOS specification introduced in the previous section. For lack of space, it is not possible to introduce BPEL. Accordingly, the reader who is not used with the language should refer to [1].

Two related works are [7,8]. Comparatively, our attempt is more general at least for three reasons: (i) the expressiveness of properties is better in CADP thanks to the use of the μ-calculus (*e.g.* only LTL in [8]), (ii) our abstract language does not deal only with dynamic behaviours; we can specify advanced data descriptions and operations on them at the abstract level (more complex than in [8] where operations cannot be modelled as an example) as well as at the executable one, (iii) we prone a mapping in two ways, useful to develop WSs and also to reason on deployed ones (the latter direction was the single goal of mentioned related works).

LOTOS Gates/Rendez-vous and BPEL Basic Activities. The basic brick of behaviour in LOTOS, the so-called gate, is equivalent to messages described in WSDL which are completely characterized using the *message*, *portType*, *operation* and *partnerLinkType* attributes. Most of the time abstract gates are accompanied of directions (emissions or receptions) because they are involved in interactions. Each action involved in a synchronization maps the four previous elements, and conversely. It is impossible to specify independent evolution of one process in BPEL: everything is interaction. Therefore, synchronizing gates are equivalent to BPEL interactions, especially in our case in which we deal only with binary communication (the core of the BPEL process model being the notion of peer-to-peer interaction between partners).

An abstract action accompanied with a reception (noted with a question mark '?' in LOTOS) may be expressed as a reception of a message using the *receive* activity in BPEL. On the other side, an emission (noted with an exclamation mark '!') is written in BPEL with the *asynchronous invoke* activity. At the abstract level, an emission followed immediately by a reception may be encoded using the BPEL *synchronous invoke*, 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 followed by a *reply* one.

Data Descriptions. There are three levels of data description: type and operation declarations, local variable declarations, data management in dynamic behaviours. We discuss the two first ones in this part and we postpone the latest one in the next subsection.

First, LOTOS datatypes, and operations on them, are specified using algebraic specifications. In BPEL, types are described using XML *schema* in the WSDL files. Elements in the schema can be simple (lots are already defined) or complex types. Data manipulation and computation (equivalent to operations in LOTOS) is defined in BPEL using the *assign* activity, and particularly using adequately XPath (and XPath Query) to extract information from elements. XPath expressions can be used to indicate a value to be stored in a variable. Within XPath expressions, it is possible to call several kinds of functions: core XPath functions, BPEL XPath functions or custom ones. Complex data manipulation may be written using XSLT, XQuery or JAVA (*e.g.* BPELJ makes it possible to insert Java code into BPEL code) to avoid a tricky writing with the previous mentioned means. Another way to describe and manage data is to use a database and corresponding statements to accessing and manipulating stored information. The data correspondence between both languages is not straightforward, although expressiveness of both data description techniques makes it possible to map all specifications from one level to the other.

In LOTOS, variables are either parameters of processes or parameters of a gate. In BPEL, variables can represent both data and messages. 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. A (WSDL) *message* tag corresponds to a set of gate parameters in LOTOS. A *part* tag matches with a parameter of a gate in LOTOS. In LOTOS, only process parameters need to be declared (not necessary for gate variables) whereas in BPEL either global and local variables involved in interactions have to be declared.

LOTOS Dynamic Constructs and BPEL Structured Activities. First of all, the *sequence* activity in BPEL matches with the LOTOS prefixing construct ';'. Intuitively, the LOTOS choice (possibly multiple) corresponds either to the *switch* activity defining an ordered list of *case* (a *case* corresponds to a possible activity which may be executed) or to the *pick* one. Let us clarify the two possible equivalences between LOTOS and BPEL depending on the presence or not of guards: (i) *absence of guards*, a choice is translated with a *pick* activity with *onMessage* tags, (ii) *presence of guards*, the mapping is straightforward using the *switch* operator. The termination used in the LOTOS specification maps with the end of the main sequence in BPEL.

Recursive process calls match with the *while* activity. The condition of the *while* is the *exit* condition of the recursive process. Sometimes, abstract recursive behaviours match BPEL non recursive services. It is the case when dealing with the notion of *transaction* (defined as a complete execution of a group of interacting WSs working out a precise task) because in this context, only one execution

of a process is enough (each transaction corresponds to a new invocation then instantiation of each involved service).

Now, let us focus on the constructs involving dynamic behaviours and data: parameters of messages and processes (in case of recursive calls for the latter), conditions of guards, local definition and modification of data. We have already discussed in the subsection dedicated to the data descriptions how to describe parameters of gates and processes in BPEL. LOTOS guards correspond to BPEL *case* tags of the *switch* construct. The BPEL *assign* tag, and more precisely the *copy* tag matches the four next cases in LOTOS: (i) *let* X_i:T_i=V_i *in* B means the initialization of variables X_i of types T_i with values V_i ($\forall i \in 1..n$) in the behaviour B, (ii) B_1*; exit(*Y_i*)* $\gg$ *accept* X_i:T_i *in* B_2 denotes the modification of variables X_i (replaced by new values Y_i), (iii) *P(*X_i*)* is an instantiation of a process or a recursive call meaning assignments of values X_i to the parameters of the process P, (iv) *send*(X_i) corresponds to an emission of data expressions which have to be built and assigned to variable X_i before sending.

LOTOS and BPEL Processes. BPEL services and LOTOS processes correspond to each other. An external view of such interacting WSs shows processes/services running concurrently. Such a kind of global system in LOTOS is described using a LOTOS main behaviour made up of instantiated processes composed in parallel and synchronizing together. Let us observe that the main LOTOS specification does not match with a BPEL process. The correspondence is that each LOTOS instantiated process, therefore pertaining to the global system mentioned previously, matches a BPEL WS. Accordingly, the architecture of the specification is preserved.

In BPEL, variables, messages/operations/port types/partner links and its main activity match respectively in LOTOS process parameters, action declarations and its behaviour. Due to the possible correspondence between *while* loops and recursive processes, an adequate use of the *scope* activity is needed to map the local variables of such nested processes.

The Negotiation Case in BPEL. The negotiation issue focused on in this paper has been implemented in BPEL and the resulting services work correctly. In this experimentation, we used guidelines in the development stage way, but the opposite direction could be tackled as well using them. Experimentations have been carried out using Oracle BPEL Process Manager 2.0[4] that enables one to design, deploy and manage BPEL processes, and BPEL Designer (an Eclipse plug-in) which is a graphical tool to build BPEL Processes. For lack of space, it is impossible to introduce pieces of BPEL code in the paper. However, all the WSDL and BPEL files implemented for this problem are available on-line[5].

6 Related Work

We split this survey of related works into two separate parts: (i) use of formal methods for WSs, (ii) contribution *wrt* the negotiation issue. We first outline

[4] `http://www.collaxa.com/`

[5] `http://www.dis.uniroma1.it/~salaun/LOTOS4WS/BPEL`

some existing proposals arguing for the use of formal methods as an abstract way to deal with WSs (especially description, orchestration, reasoning). At this abstract level, lots of proposals originally tended to describe WSs using semi-formal notations, especially workflows [15]. More recently some more formal proposals grounded for most of them on transition system models (LTSs, Mealy automata, Petri nets) have been suggested [11,18,10,2,14]. Regarding the reasoning issue, works have been dedicated to verifying WS description to ensure some properties of systems [8,18,7,17]. 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, they abstract their representation and ensure some properties using ad-hoc or existing tools.

In comparison to these existing works, the strenght of our alternative approach 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 adequate tools (respectively LOTOS and CADP), while keeping a two-way connection with the executable layer (BPEL).

A lot of research works about negotiation have been proposed in different research domains and aim at working out different issues. Main issues in negotiation are to describe negotiating systems, to verify the existence of an agreement (and possibly other properties) and to speed up its reaching. Most of the proposals belong to the multi-agent system area. Let us illustrate with former works dedicated to describe and automate negotiations [23,22,6]. As an example, in [22], the authors proposed to use logic-based languages for multi-agent negotiation. Then, they can verify from a given history that an agreement has been reached and for a given protocol that an agreement will be reached.

Other existing works advocate some description frameworks, see [4,21] as examples, to represent all the concepts (profiles, policies, strategies, etc) used in negotiations and to ensure trust negotiations. In [21], the authors advocated a model-driven trust negotiation framework for WSs. Their model is based on state machines extended with security abstractions. Their approach allows dynamic policy evolution as well. Another proposal related to WSs [5] suggested a flexible meta-model based on contracts for e-negotiation services. From this approach, negotiation plans can be derived and implementations can be performed in recent WS platform like .NET. [13] proposed some experiments on automated negotiation systems using WSs. The authors implemented a RFQ-based simple negotiation protocol using the BPEL executable language.

Compared to these existing works, we first propose a formal representation of negotiation situations. The formality of our approach implies the ability to reason on it to ensure correctness of interacting processes (either developing concrete WSs or abstracting away from an executable one) particularly to prove the existence of an agreement. We also claim that the use of WSs is adequate to negotiating systems due to many of their characteristics: interoperability, autonomy and automation, expressiveness of description, deployment on the web, etc.

7 Concluding Remarks

The recent advent of WSs raised many promising issues in the web computing area. In this paper, we emphasized that the use of an expressive, formal, tool-equipped PA is convenient to abstractly describe and reason on negotiating WSs. It was reinforced in this paper through the double-mapping we introduced to map abstract and executable layers. On a wider scale, our two-level (abstract and concrete) approach is worthy because it can be used with any kind of interacting WSs and not only for the negotiating ones on which we focused on herein.

When using PA for WSs, during the choice of the description language, an adequate trade-off should be chosen between expressiveness of the calculus and verification abilities of its support tool. For example, LOTOS is adequate to represent negotiation aspects, but is therefore limited at the verification level due to the state explosion problem ensuing the management of data expressions. All the same, we stress that (i) we already tackled and verified realistic applications, (ii) CADP is one of the most efficient tool dealing with automated reasoning on input formats involving mixed descriptions (behaviours and complex data).

Our approach can easily be generalized in many ways. First, during the design stage from PA to WSs, some negotiation patterns can be extracted, simplifying the reusability of our process description. Hence, adjustments should be performed to reuse these processes for other negotiation variants. As an example, we have already experimented multiple issue negotiation (price, time delivery, possible return, etc) implying exchanges of several values and invariants involving several parameters. Secondly, for reverse engineering purposes, guidelines to abstract BPEL code are enough to tackle most of the negotiation situations. Though, they may be complemented to deal with more complex variants.

A first perspective is to propose a methodology formally defining how to use LOTOS/CADP in the context of WSs. A possible application would be the development of certified WSs from scratch to executable BPEL code, with intermediate steps of specification and verification using LOTOS/CADP. A related perspective is to develop a tool automating the two-way mapping to obtain LOTOS or BPEL skeletons. This implementation should be performed from the guidelines (only experimented manually so far) defined in this paper.

Acknowledgments. The authors thank Lucas Bordeaux and Marco Schaerf for judicious comments and fruitful discussions we had on this work, as well as Nicolas Descoubes, Hubert Garavel and Wendelin Serwe for their help on doing experimentations with CADP. 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*.
3. J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra*. Elsevier, 2001.
4. E. Bertino, E. Ferrari, and A. C. Squicciarini. Trust-chi: An XML Framework for Trust Negotiations. In *Proc. of CMS'03*.
5. D. K. W. Chiu, S.-C. Cheung, and P. C. K. Hung. Developing e-Negotiation Process Support by Web Service. In *Proc. of ICWS'03*.
6. S. S. Fatima, M. Wooldridge, and N. R. Jennings. An Agenda-based Framework for Multi-issue Negotiation. *Artificial Intelligence*, 152(1):1–45, 2004.
7. 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.
8. X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. of WWW'04*.
9. H. Garavel and W. Serwe. State Space Reduction for Process Algebra Specifications. In *Proc. of AMAST'04*.
10. R. Hamadi and B. Benatallah. A Petri Net-based Model for Web Service Composition. In *Proc. of ADC'03*.
11. R. Hull, M. Benedikt, V. Christophides, and J. Su. E-Services: a Look Behind the Curtain. In *Proc. of PODS'03*.
12. ISO. LOTOS: a Formal Description Technique based on the Temporal Ordering of Observational Behaviour. Technical Report 8807, International Standards Organisation, 1989.
13. J. B. Kim, A. Segev, and M. G. Cho A. K. Patankar. Web Services and BPEL4WS for Dynamic eBusiness Negotiation Processes. In *Proc. of ICWS'03*.
14. A. Lazovik, M. Aiello, and M. P. Papazoglou. Planning and Monitoring the Execution of Web Service Requests. In *Proc. of ICSOC'03*.
15. F. Leymann. Managing Business Processes via Workflow Technology. Tutorial at VLDB'01, Italy, 2001.
16. A. R. Lomuscio, M. Wooldridge, and N. R. Jennings. A Classification Scheme for Negotiation in Electronic Commerce. *International Journal of Group Decision and Negotiation*, 12(1):31–56, 2003.
17. S. Nakajima. Model-checking Verification for Reliable Web Service. In *Proc. of OOWS'02, satellite event of OOPSLA'02*.
18. S. Narayanan and S. McIlraith. Analysis and Simulation of Web Services. *Computer Networks*, 42(5):675–693, 2003.
19. G. Salaün, L. Bordeaux, and M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. In *Proc. of ICWS'04*.
20. G. Salaün, A. Ferrara, and A. Chirichiello. Negotiation among Web Services using LOTOS/CADP. Technical Report 13.04, DIS - Università di Roma "La Sapienza", 2004. Available on the G. Salaün's webpage.
21. H. Skogsrud, B. Benetallah, and F. Casati. Trust-Serv: Model-Driven Lifecycle Management of Trust Negotiation Policies for Web Services. In *Proc. of WWW'04*.
22. M. Wooldridge and S. Parsons. Languages for Negotiation. In *Proc. of ECAI'00*.
23. G. Zlotkin and J. S. Rosenschein. Mechanisms for Automated Negotiation in State Oriented Domains. *Journal of Artificial Intelligence Research*, 5:163–238, 1996.