Dipartimento di Informatica e Sistemistica “Antonio Ruberti”
Università degli Studi di Roma “La Sapienza”

Web Services: A Process Algebra Approach

Andrea Ferrara

Technical Report n. 17
2004

I *Technical Reports* del Dipartimento di Informatica e Sistemistica "Antonio Ruberti" svolgono la funzione di divulgare tempestivamente, in forma definitiva o provvisoria, i risultati di ricerche scientifiche originali. Per ciascuna pubblicazione vengono soddisfatti gli obblighi previsti dall'art. 1 del D.L.L. 31.8.1945, n. 660 e successive modifiche.
Copie della presente pubblicazione possono essere richieste alla Redazione.

Dipartimento di Informatica e Sistemistica "Antonio Ruberti"
Università degli Studi di Roma "La Sapienza"
Via Eudossiana, 18 - 00184 Roma
Via Buonarroti, 12 - 00185 Roma
Via Salaria, 113 - 00198 Roma
www.dis.uniroma1.it

www.aracne–editrice.it
info@aracne–editrice.it

Redazione:
00173 Roma
via Raffaele Garofalo, 133 A/B
(06) 72672222 – (06) 93781065
telefax 72672233

ISBN 88–7999–824-2

I edizione: settembre 2004

Finito di stampare nel mese di settembre del 2004
dalla tipografia « Grafica Editrice Romana S.r.l. » di Roma
per conto della « Aracne editrice S.r.l. » di Roma
Printed in Italy

Web Services: A Process Algebra Approach

Andrea Ferrara
DIS - Università di Roma "La Sapienza"
Via Salaria 113, 00198 Roma, Italia
Emails: ferrara@dis.uniroma1.it

abstract

It is now well-admitted that formal methods are helpful for many issues raised in the Web service area. In this paper we present a framework for the design and the verification of WSs using process algebras and their tools. We define a two-way mapping between abstract specifications written using these calculi and executable Web services written in BPEL4WS; the translation includes also compensation, event, and fault handlers. The following choices are available: design and verification in BPEL4WS, using process algebra tools, or design and verification in process algebra and automatically obtaining the corresponding BPEL4WS code. The approaches can be combined. Process algebras are not useful only for temporal logic verification: we remark the use of simulation/bisimulation for verification, for the hierarchical refinement design method, for the service redundancy analysis in a community, and for replacing a service with another one in a composition.

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. A more unusual specificity that distinguishes them from more traditional software components is that they are deployed and then accessed through the internet. Some 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 used to publish and discover WSs, BPEL4WS (BPEL for short) is a notation for describing executable business process behaviors. WSs raise many

theoretical and practical issues which are part of on-going research. Some well-known problems related to 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.

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 (Labelled Transition Systems, Mealy automata, Petri nets, etc.) [5, 14, 23, 12, 19, 11].

With respect to these works, we use process algebras (PAs for short) as abstract representation. Process algebras offer more respect to all these previous approaches: they not only provide temporal logic model checking, but also bisimulation (resp. simulation) analysis, that is we can establish whether two processes have equivalent behaviors (resp. whether one of the two includes the behavior of the other). Bisimulation analysis is useful to establish when a service can substitute another services in a composition [7]; another use of bisimulation is to check the redundancy of service in a community. Because process algebras support simulation analysis, we can apply to WSs a well-know design method, the *hierarchical refinement* [17, 16]: intuitively we start with an abstract description of a process and we refine it iteratively, obtaining at each step a less abstract one. At each stage, using simulation and bisimulation we can verify the correspondence between the current version and the previous (more abstract) one. It can be applied also in the BPEL modelling of WSs, using the two-way mapping. Moreover we argue, with a simple consideration, that the simulation can be part of the problem of automatic composition of services.

In Figure 1 we present a framework, for the design and verification of WSs using process algebras [6] (*e.g.* CCS, π-calculus, LOTOS). In this paper we focus on LOTOS, one of the most expressive process algebra. We provide a two-way mapping between BPEL and LOTOS, and general guidelines for translations between BPEL/WSDL and a process algebra. We choose LOTOS

because it allows us the data handling, and the verification and the modelling of the BPEL handlers.

Respect to the quoted previous works, we study also the direction from a formal language to BPEL. Using the two-way mapping, that allows an automatic translation between the two languages, two choices are available: designing in BPEL and verifying with a process algebra, designing and verifying in a process algebra. These two approaches are not alternative, but they can be combined in the same development.

Designing in BPEL and verifying with a process algebra. Going from BPEL to a PA allows us the verification step in PA, and the converse allows to see the counterexamples directly in BPEL, hopefully even in the visual interface for designing BPEL services. Obviously one can correct in PA, and the BPEL corrected code is automatically generated. This approach is useful also for reverse engineering issues, and when we want to verify BPEL services developed by others.

Designing and verifying in a process algebra. We point out that using the mapping we can automatically obtain BPEL specifications. To our knowledge this is the first work in this direction. As advocated in a previous work [26], being simple, abstract and formally defined, PAs make it easier to specify the message exchange between WSs, and to reason on the specified systems. They are especially worthy as a first description step because they enable one to analyze the problem at hand, to clarify some points, to sketch a (first) solution using an abstract language (then dealing only with essential concerns), to have at one disposal a formal description of one or more services-to-be, therefore adequate to use existing reasoning tools to verify and ensure some temporal properties (safety, liveness and fairness properties), behavior equivalences (bisimulation), and execution traces. Process algebras design allows the distributed development and software reuse.

In Section 3 we focus on the two-way mapping between LOTOS to BPEL and we give the guidelines formalizing the translation between process algebras and BPEL. In Section 4 we illustrate the features provided by our approach: temporal logic model check-

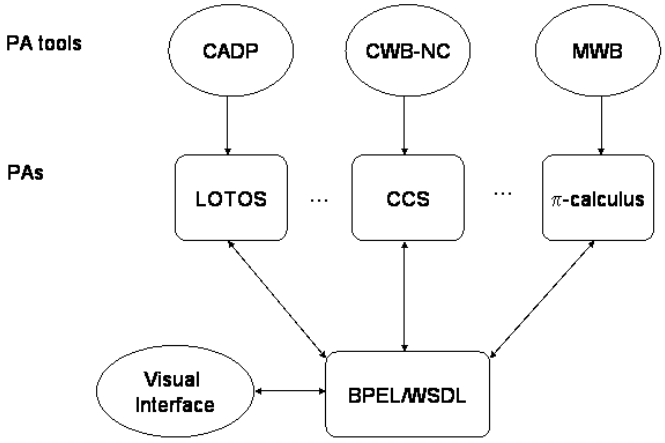

Figure 1: Proposal overview

ing, execution traces, simulation, bisimulation. We discuss the hierarchical refinement and other problems that we can solve using a process algebra representation for WSs and a bisimulation analysis. In Section 5 presents related works and motivates our contribution with respect to them. We draw up concluding remarks in Section 6 and we mention some future works.

2 Preliminaries

2.1 LOTOS in a Nutshell

LOTOS is a specification language for distributed open systems normalized by the ISO [15]. It combines two specification models: one for static aspects (data and operations) which relies on the algebraic specification language ACT ONE [9] and one for dynamic aspects (processes) which draws its inspiration from the CCS [21] and CSP [13] PAs.

2.1.1 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.

2.1.2 Basic LOTOS

This PA authorizes the description of dynamic behaviors evolving in parallel and synchronizing using rendez-vous (all the processes involved in the synchronization should be ready to evolve simultaneously along the same action). A process P denotes a succession of *actions* (also called event, channel or game in other formalisms) which are basic entities representing dynamic evolutions of processes; a process can be recursive. The symbol **stop** denotes an

inactive behavior (it could be viewed as the end of a behavior) and the **exit** one depicts a normal termination. The specific **i** action corresponds to an internal (unobservable) evolution.

Now, we present LOTOS behavioral operators. The prefixing operator G**;**B proposes a rendez-vous on the action G, or an independent firing of this action, and then the behavior B is run. The non deterministic choice between two behaviors 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 behaviors $B1$ and $B2$. It means that $B1$ and $B2$ evolve independently except on the actions G_1, ..., G_n on which they evolve at the same time firing the same action (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 actions), and full synchronization $B_1||B_2$ where the composed processes synchronize on all actions (list containing all the actions used in each process). Moreover, the communication model proposes a multi-way synchronization: n processes may participate to the rendez-vous.

The disabling operator $B_1[>B_2$ model the interruption: the behavior B_1 could be interrupted at any moment by the behavior B_2; when B_1 is interrupted, B_2 is executed (without having interruptions).

2.1.3 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 actions $G_{i\in 1..m}$ and a (optional) list of formal parameters $X_{j\in 1..n}$ of type $T_{j\in 1..n}$. The full syntax of a process is the following:

process P [G_0, ..., G_m] (X_0:T_0, ..., X_n:T_n) : $func := B$
endproc

where B is the behavior of the process P and $func$ corresponds to the functionality of the process: either the process loops end-

lessly (**noexit**), or it terminates (**exit**) possibly returning results of type $T_{j\in 1..n}$ (**exit**(T_0, ..., T_n)).

Action identifiers are possibly enhanced with a set of parameters (offers). An *offer* has either the form G**!**V and corresponds to the emission of a value V, or the form G**?**X:S which means the reception of a value of type S in a variable X.

A behavior may depend on Boolean conditions. Thereby, it is possible that it be preceded by a guard [*Boolean expression*] $\rightarrow B$. The behavior B is executed only if the condition is true. Similarly, the guard can follow an action 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`?`X`:Nat[`X`>3]`). In the sequential composition operator, the left-hand side process can transmit some values (**exit**) to a process B (**accept**):

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

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

2.2 Other Process Algebras

Numerous processes algebras have been proposed: CCS [21], CSP [13], ACP [4] are the basic ones. Extensions are π-calculus [24], Timed CSP [28]. Although syntactically different, all process algebras share a set of basic and dynamic constructs: actions, sequence, parallel composition, synchronizing actions, non deterministic choice, emission, reception, process, local process, recursive process.

2.3 Equivalences between processes

Two process are considered equivalent if their behavior is *indistinguishable* from an external observer interacting with them. In the

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

process algebra community several notions of process equivalence have been proposed. More on the topic can be found in [21]. An approach is *trace-based*: two process are equivalent if they show the same execution traces. A process is contained in another one if the set of its execution traces are included in the set of execution traces of the other. Another approach is *tree-based*: two process are equivalent if they have equivalent execution trees, that is they simulate each other (they bisimulate). A process is simulated by another one if all its behaviors are contained in the behaviors of the other. A group of process running concurrently are simulated by another group of process running concurrently if all their behaviors are contained in the behaviors of the other. It is known that simulation implies containment. As example let us discuss Figure 2.

Figure 2: Processes Equivalences; a is a ticket purchase, b ticket use for the match, c a ticket change.

The left process corresponds in basic LOTOS to $a;(b[]c)$, the right one to $a;b$ [] $a;c$. They have the same traces (ab or ac), and so they are trace-equivalent. They do not bisimulate each other; after doing a the left process will do either b or c, while the right process on doing a, it will either choose to move in a state from which it does b or in a state from which it does$c$; depending on this choice, it cannot do one of the two actions whereas the left process leaves both possibilities open. Let a is a ticket purchase, b ticket use for the match, c a ticket change; the left process always

allows to change the ticket after the purchase, the right one does not.

3 The two-way mapping between LOTOS and BPEL

In this section we show the two-way mapping between LOTOS, a process algebra that allows data handling, and BPEL. Our goal is showing a two-way mapping between the two languages, that allows an automated translation. For lack of space, it is not possible to introduce the basics of BPEL, XMLSchema, and XPath. Accordingly, the reader who is not used with them should refer to [3, 1, 2].

When it is possible, we present together both directions of the mapping. While the translation from BPEL to LOTOS implicitly preserves the BPEL structure, the converse does not: LOTOS allows to use the construct in very flexible manner, BPEL does not. In the LOTOS design we have to be careful, if we want a simple automatic translation, to write behavior structurally similar to BPEL ones. For example in BPEL a service can communicate only with other services, there is no message exchange inside a service. In LOTOS instead, as in all process algebras, there are no constraints about this. In order to obtain a simple automatic translation from a process algebra, we have to follow this simple rule in the design. The details of other similar rules will be given during the explanation. We remark that in our framework, when we design and correct in BPEL, the LOTOS-BPEL direction is free from this problem: we start from LOTOS code, that is BPEL-like structured, because directly obtained by the translation from BPEL.

In our presentation we refer to Table 1 and to Table 2, where we show sample code of both languages; the correspondence is about both directions of the mapping. Figure 3 gives a very general picture. We show the mapping of basic construct, dynamic behavior, data definition and handling, and fault, compensation, event handlers. Finally we give general guidelines for translations between PAs and BPEL.

Sample BPEL Code	Sample LOTOS Specification

`< ... act1 ... >`
`</act1>`
`<assign ... >`
`<copy>`
`<from expression="5"/>`
`<to var="x"/>`
`<copy>`
`</assign>`
`< ... act2 ... >`
`</act2>`
	..act1..; exit(5) ≫
	accept x:Nat in ..act2..

`<receive ... variable="m">` `</receive`	g?m:Nat;
`<reply ... variable="m">` `</reply>`	g!m:Nat;

`<invoke ... invar="mS"`
`outvar="mR">`
`</invoke>`
	gS!mS:Nat; gR?mR:Nat;

Table 1: The BPEL-LOTOS two-way mapping: examples for basic behaviors.

3.1 General Outline

An external view of interacting WSs shows processes (services) running concurrently. Such a kind of global system in LOTOS is described using LOTOS main behavior (that is the outermost process): it instantiates processes composed in parallel and synchronizing on all actions representing their interactions.

At the basis of our mapping there is the correspondence between LOTOS actions and BPEL interactions. BPEL services and LOTOS processes instantiated in the main process correspond to each other. The direction from BPEL to LOTOS is straightforward: we simply automatically build a main behavior containing the instantiation of all the processes (each of them correspond to a service), in the manner described above. About the other direction, from LOTOS to BPEL, the LOTOS programmer have to respect this rule: he has to write the main behavior simply instantiating all the processes representing services, in the usual manner.

To describe behaviors, in LOTOS we have the process definition, in BPEL the service description. In LOTOS a defined process

Sample BPEL Code	**Sample LOTOS Specification**
<pick ... > <onMessage ... variable="m1"> < ... act1 ...> </onMessage> <onMessage ... variable="m2"> < ... act2 ... > </onMessage> </pick>	(g1?m1:Nat; ..act1..) [] (g2?m2:Nat; ..act2..)
<sequence ...> < ... act1 ... > < ... act2 ... > </sequence>	..act1..; ..act2..
<flow ... > < ... act1 ... > <source linkname="link1" condition="cond1"/> </act1> < ... act2 ... > <target linkname="link1"/> </act2> </flow>	..act1..; ([cond1]->link1 !1; [] [not(cond1)]->link1 !0;) || (link1 ?x:Bool; ([x=1]->..act2.. [] [x=0]->i;))
<switch> <case condition= "bpws:getVariableData(x)>=0"> <..act1..> </..act1..> </case> <otherwise> <..act2..> </..act2..> </otherwise> </switch>	[x>=0] -> ..act1..; [] [x<0] -> ..act2.. ;
<while condition= "bpws:getVariableData(x)>=0"> <..act1..> </..act1..> </while>	proc while1 [..](..) := [x<0]-> i; [] [x>=0]->..act1..; while1[..](..) endproc

Table 2: The BPEL-LOTOS two-way mapping: examples for structured behaviors.

can be instantiated (with action passing, that renames the name of action in the definition, and parameter passing). From LOTOS to BPEL, we use the behaviors specified in the process definition to generate the BPEL service description with the names of partner links, port type, operations, variables. From BPEL to LOTOS, we use the service description to generate, including the names of actions, both the process definition and the process instantiations. We have a process instantiation if the process represents a scope or a while.

We do not consider bindings issues. For the data type definitions in BPEL/WSDL we have XMLSchema, in LOTOS we can define abstract data types. In LOTOS we initialize the data structures defined with the type construct at the beginning of the main process.

To summarize, the main process first initializes data, then instantiates the process/services running concurrently.

3.2 Basic behaviors and interactions

At the core of BPEL process model is the notion of peer-to-peer interaction between partners described in WSDL. All BPEL basic activities perform interactions between WSs. An interaction is characterized by the partner link, the port type, and the operation involved in the two communicating partners (each partner defines these three elements for each interaction). In parallel, LOTOS has at its disposal the notion of action to represent dynamic evolutions and of rendez-vous to describe synchronizations among processes. Consequently, when process/services are instantiated, LOTOS synchronizing actions are equivalent to BPEL interactions. When the process representing a service is defined, an action is simply an emission or a reception. The name of the action stores information (parter link, port type, operation in BPEL, process and action names in LOTOS) on the receiver in the emission case, on the sender in the reception case. This name can contain a description of the interaction (e.g. request, notification, cancellation). When we instantiate, we have to compose the names of the action of both interacting processes/services; we consider two synchronizing action, we concatenate their definition name, and

we give the concatenated name to both.

Let us go forward in more details. Starting the mapping from BPEL, in order to build the name of LOTOS action, we use the information in *partner link*, *port type*, *operation* attributes in the *receive*, *reply*, and *invoke*. Let a partner 1 (resp. 2) have a partner link pl_1 (resp. pl_2), a port type p_1 (resp. p_2), an operation o_1 (resp. o_2), and a variable v_1(resp. v_2) associated with the exchanged message. Let a reservation request the object of the partner 1 message, and an availability response the object of the partner 2 message. Then the process associated with the partner 1 has *in the definition* the action $pl_1\_p_1\_o_1\_resReq$ and the process for the partner 2 the action $pl_2\_p_2\_o_2\_avResp$. When the two processes are instantiated in the main behavior, the name of their synchronized action is $pl_1\_p_1\_o_1\_resReq\_pl_2\_p_2\_o_2\_avResp$, and v_1 (resp. v_2) is the parameter of the action for the partner 1 (resp. 2). Moreover if we have a *message* with N *part* tags, in LOTOS we have an action with N parameters, one for each part of the message.

Starting from LOTOS instead, we extract the port type, operation and message definitions analyzing the names of LOTOS actions in the instantiated processes. For example if we have two actions $pl_1\_p_1\_o_1\_pl_2\_p_2\_o_2$ and $pl_1\_p_1\_o'_1\_pl_2\_p_2\_o_2$, we conclude that we have the service 1 with partner link pl_1, port type p_1 and operations o_1 and o'_1, and a service 2 with partner link pl_2, port type p_2 and operation o_2. Moreover we know that there are two interactions: one between service 1 in partner link pl_1, port type p_1, operation o_1, and service 2 in partner link pl_2, port type p_2, operation o_2; the other interaction is between service 1 in partner link pl_1, port type p_1, operation o'_1, and service 2 in partner link pl_2, port type p_2, operation o_2.

The reception of a message is expressed using the *receive* activity in BPEL and using a action with a reception in all its parameters in LOTOS.

In BPEL, the emission is written with the *reply* or the *asynchronous invoke* activity whereas in LOTOS we use a action with an emission in all its parameters. The BPEL *synchronous invoke*, performing two interactions (sending a request and receiving a response) corresponds in LOTOS to an emission followed imme-

diately by a reception. In LOTOS we have two different actions, because we have two interactions in BPEL; the names of actions share the same partner link, the same port type, the same operation but they differs only by a letter S or R at the end (representing the emission and the reception of the invoke). Using this rule we can distinguish in the LOTOS code when a contiguous emission-reception is an invoke.

3.3 Structured Behaviors

Now we introduce the mapping for LOTOS dynamic constructs and BPEL structured activities.

The *pick* BPEL activity is executed when it receives one message defined in one of its *onMessage* tag or when it is fired by an *onAlarm* event; we cannot model the latter case because basic LOTOS does not have the notion of time. The equivalent construct in LOTOS is obtained using the non deterministic choice, in which the first action of each branch is a reception; it is chosen the branch whose beginning reception is performed first. In the LOTOS modelling, if we use the non deterministic choice with an emission as first action, then an automatic translation to BPEL becomes very difficult. For example the following LOTOS behavior, because the a is an emission, does not have a straightforward translation in BPEL:

```
... a!x:Nat; b?x:Nat; [] c?x:Nat; b?x:Nat;..
```

When we design in a process algebra, we have to think to BPEL code structure, in order to simplify the automatic translation.

The *sequence* activity in BPEL match with the LOTOS prefixing operator ';'.

In BPEL we have the *flow* activity, in LOTOS the full synchronization constructs '||'. Because in BPEL we cannot have interaction inside a service, therefore we do not have synchronizations in a parallel composition inside a process representing a service. The mapping about the *link* tag is more complicated, because LOTOS does not have an explicit construct of dependence relation between concurrent actions. In BPEL we specify with the *source* tag the activity that has to occur first, and with the *target* tag