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

Antonella Chirichiello
Gwen Salaün

Technical Report n. 8
2005

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

Copyright © MMV
ARACNE EDITRICE S.r.l.

www.aracneeditrice.it
info@aracneeditrice.it

Redazione:
00173 Roma
via Raffaele Garofalo, 133 A/B
06 93781065
telex 72678427


I diritti di traduzione, di memorizzazione elettronica, di riproduzione e di adattamento anche parziale, con qualsiasi mezzo, sono riservati per tutti i Paesi.

I edizione: luglio 2005

Finito di stampare nel mese di luglio del 2005
dalla tipografia « Grafica Editrice Romana S.r.l. » di Roma
per conto della « Aracne editrice S.r.l. » di Roma
Printed in Italy
Encoding Abstract Descriptions into Executable Web Services: Towards a Formal Development

Antonella Chirichiello¹ and Gwen Salaün²

¹DIS - Università di Roma “La Sapienza”, Italy
   Email: chirichiello@dis.uniroma1.it
²INRIA 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 vari-
ables. 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
<table>
<thead>
<tr>
<th>Abstract constructs</th>
<th>LOTOS constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>OACT</td>
<td>act!v (* emission of value v *)</td>
</tr>
<tr>
<td>IACT</td>
<td>act?x : t (* reception in var. x of type t *)</td>
</tr>
<tr>
<td>END</td>
<td>exit</td>
</tr>
<tr>
<td>INT</td>
<td>τ</td>
</tr>
<tr>
<td>SEQ</td>
<td>act;B</td>
</tr>
<tr>
<td>CH</td>
<td>B_1 [] B_2 (* B_1 and B_2 synchronize on *)</td>
</tr>
<tr>
<td>PAR</td>
<td>B_1 [sync-actions]</td>
</tr>
<tr>
<td>HID</td>
<td>hide a_1, ..., a_n in B</td>
</tr>
<tr>
<td>DD</td>
<td>algebraic specifications</td>
</tr>
<tr>
<td>GRD</td>
<td>[bool-exp] → B</td>
</tr>
<tr>
<td>ID,AD,V,D,BHV</td>
<td>P<a href="var-list">action-list</a> := B</td>
</tr>
</tbody>
</table>

**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.