Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 255 (2009) 3–21
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Model-checking Web Services Orchestrations using BP-calculus

Faisal Abouzaid and John Mullins

CRAC Lab., Computer & Software Eng. Dept., École Polytechnique de Montréal. P.O. Box 6079, Station Centre-ville, Montreal (Quebec), Canada, H3C 3P8. [2]

Abstract

The Business Process Execution Language for Web Services (BPEL) is the standard for implementing orchestrated business processes designed but not limited to, as web services. BPEL is a powerful language but lacks a widely accepted formal semantics, and this makes it difficult to formally validate the correct execution of BPEL implementations. In the other hand, process algebras have proved their efficiency in the specification of web services orchestrations. In this paper we improve the BP-calculus, a π-calculus based formalism designed to ease the automatic generation of verified BPEL code, by defining specific equivalence and logic in order to verify BPEL implementations through their formal specification expressed in this calculus. The formal specification of service-oriented applications allows the checking of functional properties described by means of the new logic, that is shown to be well suited to capture peculiar aspects of services formalized in π-like languages. As an illustrative example, we present the BP-calculus specification and the verification results of a trade market service scenario.

Keywords: Web Services; Orchestration languages; BPEL; Process algebras; π-calculus

1 Introduction

Web services represent a well accepted implementation of service-oriented computing (SOC)and their composition allows for the creation of customized complex applications based on reutilization and composition of existing services. Orchestrations describe the way in which separate Web Services can be brought together in a consistent manner to provide a higher value service. Business Process Execution Language for Web Services (BPEL) [12] is the widely accepted standard that permits to define the business logic between processes interacting in an orchestration. A BPEL process defines how multiple service interactions between partners can be coordinated internally, that is their orchestration, in order to achieve a business goal.

[1] Research partially supported by the author's NSERC grant (Canada)
[2] Email: m.abouzaid,john.mullins@polymtl.ca

doi:10.1016/j.entcs.2009.10.022

Since bad orchestration will result in bad and unprofitable services, it is important to have tools and means to ensure the correctness of such compositions. Current software engineering technologies for SOC, do not support verification tools and lot of researches are devoted to this purpose. However, existing researches tend to provide a formal semantics for BPEL, expressed in terms of various formalisms such as Petri nets, abstract state machines (ASM) or process algebras. But except the work in [17] and [3], none of them provides a way to realize a refinement process or the re-engineering of existing BPEL implementations.

Process algebras (PAs) and associated logics allow a design time verification of the model behavior and strengthen the correctness of service compositions [16] because they are based on solid theoretical concepts. One of the most relevant method is the rich theory of the π-calculus [10] because of its capacity to model mobility, by passing channel names, as data, through channels.

Our objective is to create a system based on a π-like formalism that allows the property checking of real-world business processes and also for the generation of readable and verified BPEL code. Moreover the same approach is used to verify and correct existing BPEL specification by extracting abstract representation from existing implementations.

In order to analyze SOC applications, it is convenient to exploit a logic with modalities indexed by π-calculus actions such as the π-logic [5].

Once the formal specification of the system is verified and validated, the corresponding BPEL code is automatically generated and proved to be correct and complete.

Contributions:

This article provides some theoretical basis for the encoding of BP-processes into readabable BPEL code. For the sake of readability of the generated BPEL code, we need to choose the best suited construct that reflects intentions of the designers. For this purpose, the BP-calculus uses annotations on selected constructs. The novel contribution in this paper is to define an equivalence relation a logic (the BP-logic) that are proved to be adequate. Finally, we illustrate the usability of the encoding by providing examples of non trivial properties of a case study we checked with the HAL-Toolkit [5].

Related works:

Numerous works have been devoted to the formal specification of business process languages, especially BPEL, using different formalisms such as Petri Nets ([17]) or Abstract State Machines ([4]). But the more promising approaches use process algebras and several formalisms based on PA have been proposed: SOCK [7], COWS [8], each one handling particular features of the problem. The framework we present is based on the π-calculus, and differ from the cited approaches since it focuses on a lower level of abstraction and is closer to BPEL.

In [3] authors present a two-way mapping between BPEL and LOTOS that is limited to some basic constructs of BPEL and no formal proof of the correctness of

the mapping is provided, arguing the lack of a semantics for BPEL.

Lucchi and Mazzara [9] provided the first π-calculus based semantics to BPEL by defining a formalism called webπ, tailored to study a simplified version of the scope construct of BPEL. We base our study on this semantics.

Structure of the paper:

This paper is organized as follows. Section 2 introduces some preliminaries e.g. syntax and semantics of the BP-calculus (Section 2.2) and the logics (Section 3.2). In Section 3 we present the behavioural properties of the BP language. Section 4 presents the verification framework that is used in Section 5 to verify the illustrating example and to present the results of the verification. In Section 6 we conclude and provide and some directions for future works.

2 Preliminaries

2.1 BPEL

BPEL [12] is an XML-based specification language for describing business processes orchestrating the interaction of different, existing and possibly dynamically emerging Web Services. As such, it builds on top of the WSDL language for describing the interface of Web Services. This is specified in terms of port types, actions, and messages. BPEL supports the definition of two types of processes: executable and abstract processes. An abstract, (not executable) process is a business protocol, specifying the message exchange behavior between different parties. An executable process, specifies the execution order between a number of activities. However, in this paper we will mainly refer to executable BPEL processes.

Activities describe the precise behavior of the business process. Basic activities include activities such as sending (*invoke*), receiving (*receive*) requests and replies (*reply*), which can specify one or more existing correlation sets they must adhere to, or new correlation sets to be initialized. Among other basic activities, there are variable assignment (*assign*), synchronization of internal concurrent activities through private source and target links (*links*), waiting for a timeout (*wait*), and raising faults (*throw*). Structured activities realize sequential composition (*sequence*), guarded choice (*pick*), parallel composition (*flow*), iteration cycles (*while*, *foreach* and *repeat*), and conditional (*if then else*).

2.2 The BP-calculus

The π-calculus is sufficient to reason on orchestrated services. However, this could be very difficult and confusing. This the reason why we introduce other orchestration primitives in a variant of the π-calculus we call the BP-calculus. We present in this section its syntax and operational semantics.

Terms			
t	::=	x	(variables)
	$\mid$	u	(names)
	$\mid$	$f(t_1, \ldots t_k)$	(tuple)
$\mathcal{C}$	::=	$null \mid \mathcal{C}[\tilde{x} \leftarrow t]$	(correlation set)
Processes :			
P, Q	::=	IG	(input guard)
	$\mid$	$\overline{c}^t\langle M\rangle.P$	(annotated output)
	$\mid$	$\tau.P$	(silent action)
	$\mid$	$P \mid Q$	(parallel composition)
	$\mid$	$P \triangleright_{c(M)} Q$	(sequential composition)
	$\mid$	$A(x_1, \ldots, x_n)$	(service definition)
	$\mid$	$[\mathcal{C} : P]c(\tilde{x}).A(\tilde{y})$	(instance spawn)
	$\mid$	S	(scope)
Guarded choice :			
IG	::=	0	(empty process)
	$\mid$	$c^s(u).P$	(annotated input)
	$\mid$	$IG + IG'$	(guarded choice)
	$\mid$	$if\ M = N\ then\ IG\ else\ IG$	(conditional)
	$\mid$	$[\tilde{x} \leftarrow f(M_1, ..., M_n)]IG$	(function evaluation)
Scopes :			
S	::=	$\{\tilde{x},\ P,\ H\}$	(scope)
H	::=	$\prod_i W_i(P_{i_1}, \cdots, P_{in_i})$	(handlers)

Table 1
BP-calculus Syntax

Syntax of the BP-calculus

Terms: The set of terms $\mathcal{T}$ consists of variables $\mathcal{V}$, names $\mathcal{N}$ and values ($\mathcal{U}$) (integers, booleans, strings, ...). For each term t, $fv(t)$ is the set of variables in t. A message is a closed term (i.e. not containing variables). The set of messages is denoted $\mathcal{M}$.

Functions: Functions model primitives that manipulate messages: $\mathcal{F} \subseteq [\mathcal{M}^k \rightarrow \mathcal{M}^n]$.

Syntax: We let $\tilde{x} = (x_1, ..., x_n)$, (resp. $\tilde{a} = (a_1, ..., a_m)$, $\tilde{u} = (u_1, ..., u_m)$) range over the infinite set of n-tuples of variable (resp. name, value) identifiers. We denote $\tilde{x} \leftarrow \tilde{u}$ the assignment of values $\tilde{u}$ to variables $\tilde{x}$.

Table 1 introduces the syntax of the BP-calculus.

Interpretation

The intended interpretation of the processes is as follows:

- IG is an input guarded process and $IG + IG'$ behaves like a guarded choice and is intended to be translated by a `<pick>`.
- $\overline{a}^t\langle M\rangle$ ($t \in \{invoke, reply, throw\}$) is the usual output which can be an invocation, or a reply to a solicitation, or the throw of a fault, and are translated by a an `<invoke>`, `<reply>` or a `<throw>`, respectively . The annotations on input or output operations are used to ease the translation into BPEL.
- τ is the silent action. This action is useful to modelize communication. Altough BPEL does not provide a silent action, it can be easily specified by means of a

sequence and an aempty process.

- $P|Q$ is the parallel composition of processes P and Q. However the sequential operator imposes that the process $A = P|Q$ terminates when both P and Q terminate.
- $P \triangleright_{c(M)} Q$ expresses a sequential composition from process P passing M to Q (Q can perform actions when P has terminated). M carries binding information between proccesses P and Q. This construct allows to easily mimic the 's element `<sequence>`.
- *if then else* expresses a classical choice based on messages identity is intended to be translated by an `if then else` construct in BPEL 2.0.
- $\mathcal{C}$ is a correlation set, i.e a set of specific valued variables within a scope acting as properties and transported by dedicated parts of a message. Its values, once initiated, can be thought of as an identity of the business process instance. Intuitively, $[\mathcal{C} : P]c_A(\tilde{x}).A(\tilde{y})$ (Instance spawn) represents an orchestration service running a process defined as $c_A(\tilde{x}).A(\tilde{y})$. A reception of a message M over the dedicated channel c_A causes a new service instance (defined as $A(\tilde{y})$) to be spawned. The process P represents the parallel composition of service instances already spawned, $\mathcal{C}$ the correlation set characterizing instances and $\tilde{y}$ the correlation part of M.
- $[x \leftarrow f(M_1, ..., M_n)]P$ assigns the value $f(M_1, ..., M_n)$ to variable x before executing process P. For instance, $[x \leftarrow build(M_1, ..., M_n)]\bar{c}\langle x\rangle$ means that the n-tuple M is built from components $M_1, ..., M_n$ before being sent over the channel c.
- A scope is a wrapper for variables, a primary activity and handlers represented as contexts.

  Let $S ::= \{\tilde{x},\ P,\ H\}$ be a scope, with handlers $H ::= \prod_i W_i(P_{i_1}, \cdots, P_{in_i})$. Then,
  - $\tilde{x}$ are the local variables of the scope, and P its primary activity,
  - H is the scope's execution environment that is modeled as the parallel composition of handlers W_i. Each handler is a wrapper for a tuple of processes $\widehat{P} = (P_1, \ldots, P_n)$ that correspond to the activities the handler has to run when invoked. Not all handlers are mandatory.
  - $W_i(P_{i1}, \cdots, P_{in_i})$ is the process obtained from the multi-hole context $W_i[\cdot]_1 \cdots [\cdot]_{n_i}$ by replacing each occurrence of $[\cdot]_j$ with P_{ij}.
  - Note that the case where the variable x is restricted to a simple process P and that no handler is defined within the scope, corresponds to the usual restriction of the π-calculus that is denoted $(\nu x)P$; that is $(\nu\tilde{u})P \stackrel{def}{=} \{\tilde{u}, P, 0\}$. In this case, $c\langle\nu n\rangle$ where c, n are names will denote a bound output action.

  Due to the lack of place, we refer the reader to [1] for the handlers' syntax.

$$
\begin{array}{rl}
P \mid 0 \equiv P \quad P \mid Q \equiv Q \mid P \quad P \mid (Q \mid R) \equiv (P \mid Q) \mid R & \\
\{\tilde{u}, P, \emptyset\} \equiv (\nu \tilde{u})P \quad \{\tilde{u}, \{\tilde{v}, P, \emptyset\}, \emptyset\} \equiv \{\tilde{v}, \{\tilde{u}, P, \emptyset\}, \emptyset\} & \\
\{\tilde{u}, P, \emptyset\} \mid Q \equiv \{\tilde{u}, P \mid Q, \emptyset\} \ (\forall\ i\ u_i \notin fn(Q)) & \\
\{\tilde{x}, P, H\} \mid \{\tilde{x}, Q, H'\} \equiv \{\tilde{x}, Q, H'\} \mid \{\tilde{x}, P, H\} \quad \{\tilde{x}, P, H\} \mid 0 \equiv \{\tilde{x}, P, H\} & \\
\{\tilde{x}, P, H\} \mid (\{\tilde{x}, Q, H'\} \mid \{\tilde{x}, R, H''\}) \equiv (\{\tilde{x}, P, H\} \mid (\{\tilde{x}, Q, H'\}) \mid \{\tilde{x}, R, H''\} & \\
P \rhd_{c(M)} 0 \equiv P \quad 0 \rhd_{c(M)} P \equiv P & \\
P \rhd_{c(M)} (Q \rhd_{c(M')} R) \equiv (P \rhd_{c(M)} Q) \rhd_{c(M')} R & \\
(IG_1 + IG_2) \rhd_{c(M)} P \equiv IG_1 \rhd_{c(M)} P + IG_2 \rhd_{c(M)} P & \\
[\mathcal{C} : P]c_A(\tilde{x}).A(\tilde{y}) \equiv [null, \mathcal{C} : P]c_A(\tilde{x}).A(\tilde{y}) & \\
[\mathcal{C} : P]c_A(\tilde{x}).A(\tilde{y}) \rhd_{c(M)} [\mathcal{C} : Q]c_B(\tilde{x}).B(\tilde{y}) \equiv [\mathcal{C} : Q]c_B(\tilde{x}).B(\tilde{y}) \rhd_{c(M)} [\mathcal{C} : P]c_A(\tilde{x}).A(\tilde{y}) &
\end{array}
$$

Table 2
Structural Congruence.

2.3 Operational Semantics

The structural congruence is the smallest equivalence relation closed under the rules in Table 2. The first six rules are standard rules of the π-calculus. All the other rules but the last are about the sequence and scopes and we refer the reader to [1] for detailed comments. The last rule is closely related to the semantics of correlation set update (rule `C-SPF` in Table 3) which guarantees uniqueness of each running instance. Also, the last rule ensures that the correlation sets $\mathcal{C}$ and $null, \mathcal{C}$ will be considered as equal along this recursive process.

The operational semantics of the BP-calculus is a labeled transition system generated by inference rules given in Table 3. Note that the sequential operator implies to introduce a termination predicate denoted ($\checkmark$), e.g P terminates $\Leftrightarrow$ $P\checkmark$. The semantics of this operator is given by: if $s\checkmark$ and $s \xrightarrow{\alpha} s'$ then $s'\checkmark$.

Rules `SCO`, `HAN` and `S-PAR` define the behavior of scopes and handlers. These constructs are defined as multihole contexts. Thus, they can be derived from previous rules since handlers are processes. Rules `IFT-M` and `IFF-M` model the conditional. Rule `EVAL` handles function evaluation.

Rules `C-SP1, C-SPT` and `C-SPF` cope with correlation mechanisms. Actually, the construct $[\mathcal{C} : P]c(\tilde{x}).A(\tilde{y})$ may be viewed as an indexing replication. While rule `C-SP1` allows a spawned service P to execute as standalone service, rule `C-SPT` handles the initial spawning of an instance and the initialization of a correlation set after a reception. Rule `C-SPF` manages the subsequent instance creation. The correlation set $\mathcal{C}$ is updated and an instance of P is created that runs concurrently with existing ones.

The other rules are standard semantic rules of π-calculus ([10] , [14]).

3 Equivalences

Process creation and sequentiality operations need a special attention since they induce nontrivial questions of variable scope.

Open	$\frac{P \xrightarrow{\overline{a}u} P' \quad a \neq u}{(\nu u)P \xrightarrow{\overline{a}\nu u} P'}$	Close	$\frac{P \xrightarrow{a(u)} P' \quad Q \xrightarrow{\overline{a}\nu u} Q' \quad u \notin fn(P)}{P \mid Q \xrightarrow{\tau} (\nu u)P' \mid Q'}$
RES	$\frac{P \xrightarrow{\alpha} P' \quad n \notin fn(\alpha) \cup bn(\alpha)}{(\nu n)P \xrightarrow{\alpha} (\nu n)P'}$	TAU	$\frac{}{\tau.P \xrightarrow{\tau} P}$
OUT	$\frac{}{\overline{c}^t\langle M \rangle.P \xrightarrow{\overline{c}\langle M \rangle} P}$	IN	$\frac{}{c(x).P \xrightarrow{c(M)} P\{M/x\}}$
PAR	$\frac{P \xrightarrow{\alpha} P' \quad bn(\alpha) \cap fn(Q) = \emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$	SYNC	$\frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\overline{\alpha}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$
STRUCT	$\frac{P \equiv P' \quad P' \xrightarrow{\alpha} Q' \quad Q \equiv Q'}{P \xrightarrow{\alpha} Q}$	CHOICE	$\frac{P_i \xrightarrow{\alpha} P_i' \quad i \in \{1,2\}}{P_1 + P_2 \xrightarrow{\alpha} P_i'}$
DEF	$\frac{P\{\tilde{y}/\tilde{x}\} \xrightarrow{\alpha} P' \quad A(\tilde{x}) = P}{A(\tilde{x}) \xrightarrow{\alpha} P'}$		
SCO	$\frac{P \xrightarrow{\alpha} P'}{\{x,P,H\} \xrightarrow{\alpha} \{x,P',H\}}$	HAN	$\frac{H \xrightarrow{\alpha} H'}{\{x,P,H\} \xrightarrow{\alpha} \{x,P,H'\}}$

$$\text{S-PAR} \quad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\overline{\alpha}} Q'}{\{x,P,H_1\} \mid \{x,Q,H_2\} \xrightarrow{\tau} \{x,P',H_1\} \mid \{x,Q',H_2\}}$$

$$\text{IFT-M} \quad \frac{P \xrightarrow{\alpha} P' \quad M = N}{if(M=N)\ then\ P\ else\ Q \xrightarrow{\alpha} P'} \qquad \text{IFF-M} \quad \frac{Q \xrightarrow{\alpha} Q' \quad M \neq N}{if\ (M=N)\ then\ P\ else\ Q \xrightarrow{\alpha} Q'}$$

$$\text{EVAL} \quad \frac{\tilde{M} = f(M_1,\ldots,M_n) \quad P\{\tilde{M}/\tilde{x}\} \xrightarrow{\alpha} P'}{[\tilde{x} \leftarrow f(M_1,\ldots,M_n)]P \xrightarrow{\alpha} P'} \qquad \text{C-SP1} \quad \frac{P \xrightarrow{\alpha} P'}{[\mathcal{C}:P]c_A(\tilde{x}).A(\tilde{y}) \xrightarrow{\alpha} [\mathcal{C}:P']c_A(\tilde{x}).A(\tilde{y})}$$

$$\text{C-SPT} \quad \frac{createInstance(M) = true \quad [\tilde{z} \leftarrow \tilde{u}] = correlationPart(M)}{[null:0]c_A(\tilde{x}).A(\tilde{y}) \xrightarrow{c_A(M)} [[\tilde{z} \leftarrow \tilde{u}]:A(\tilde{u})]c_A(\tilde{x}).A(\tilde{y})}$$

$$\text{C-SPF} \quad \frac{createInstance(M) = true \quad [\tilde{z} \leftarrow \tilde{u}] = correlationPart(M) \quad [\tilde{z} \leftarrow \tilde{u}] \notin \mathcal{C}}{[\mathcal{C}:P]c_A(\tilde{x}).A(\tilde{y}) \xrightarrow{c_A(M)} [\mathcal{C}, [\tilde{z} \leftarrow \tilde{u}]:P \mid A(\tilde{u})]c_A(\tilde{x}).A(\tilde{y})}$$

Table 3
Operational semantics of the BP-calculus.

3.1 Bisimulation and congruence

In this section we develop formal reasoning mechanisms and analytical tools for checking that the BPEL services resulting from an automatic code generation meet desirable correctness properties and do not manifest unexpected behaviors. A standard approach is the use of a bisimulation equivalence ([15], [11]).

Definition 3.1 A binary relation B over a set of BP-processes is a simulation if, whenever P B Q, we have that :

- if $P \xrightarrow{\alpha} P'$ and $fn(P,Q) \cap bn(\alpha) = \emptyset$, then there exists Q' such that $Q \xrightarrow{\alpha} Q'$ and P' B Q'.
- if $P\checkmark$ then $Q\checkmark$

Relation B is a bisimulation if both B and B^{-1} are simulations.

Two agents P and Q are *bisimilar*, written $P \dot{\sim}_{bp} Q$ if $P \mathcal{R} Q$ for some bisimulation $\mathcal{R}$. We call relation $\dot{\sim}_{bp}$ *bisimilarity*.

The side condition $(fn(P,Q) \cap bn(\mu) = \emptyset)$ ensures that there is no free name captured in both processes.

The condition on process termination is added to handle the specific case of the spawn operator. Indeed, two processes are equivalent if they have the same behavior, and in particular if they terminate. The termination predicate ($\checkmark$) induces a new behavior since terminated terms may at the same time still perform actions if they are spawned off as parallel processes as shown in the following example.

Example 3.2 Let $A = \overline{a}\langle\rangle$ the process to be spawned and $[[1] : 0]c_A(M).\overline{a}\langle\rangle$ the first spawned instance. Without the condition on the termination predicates we would have $[[1] : 0]c_A(M).\overline{a}\langle\rangle \dot{\sim} \overline{a}\langle\rangle$, where $\dot{\sim}$ is the standard bisimilarity, but these terms generate different behavior in the context of sequential composition:

$$[[1] : 0]c_A(M).\overline{a}\langle\rangle \rhd \overline{b}\langle\rangle \xrightarrow{\overline{b}\langle\rangle} [[1] : 0]c_A(M).\overline{a}\langle\rangle \rhd 0 \text{ but } \overline{a}\langle\rangle \rhd \overline{b}\langle\rangle \not\xrightarrow{\overline{b}\langle\rangle}.$$

When mobility is involved, e.g. when it is possible to communicate channel names, the bisimulation is not always preserved because of input actions. As a consequence, the bisimulation is not a congruence.

This lack of congruence of prefixing w.r.t. standard bisimilarity $\dot{\sim}$ is well-known from the π-calculus. For instance, $x(z).0 \,|\, \overline{y}\langle z\rangle.0 \;\dot{\sim}\; x(z).\overline{y}\langle z\rangle.0 + \overline{y}\langle z\rangle.x(z).0$ but $y(z).0 \,|\, \overline{y}\langle z\rangle.0 \;\not\dot{\sim}\; y(z).\overline{y}\langle z\rangle.0 + \overline{y}\langle z\rangle.y(z).0$ since both processes are discriminated by a τ-derivation. Thus $w(x).(x(z).0 \,|\, \overline{y}\langle z\rangle.0) \;\not\dot{\sim}\; w(x).(x(z).\overline{y}\langle z\rangle.0 + \overline{y}\langle z\rangle.x(z).0)$ since the name y may be received on w. Hence, there is a context, $w(x).[\cdot]$, not preserving the bisimulation. Consequently, the same remark applies to $\dot{\sim}_{bp}$.

The bisimilarity $\dot{\sim}_{bp}$ not being preserved by input prefixing forces to define the largest congruence $\dot{\simeq}_{bp}$ included in:

Definition 3.3 Two processes P and Q are *congruent* (denoted $P \dot{\simeq}_{bp} Q$) if for any substitution $\sigma = [y_1/x_1, ...y_n/x_n]$ we have: $P_\sigma \dot{\sim}_{bp} Q_\sigma$, where P_σ is P with $y_i = \sigma(x_i)$ substituted to x_i for every $x_i \in fn(P)$.

However, we are interested to check whether crucial properties (such as a variety of safety and liveness properties) hold. We, thus, need to introduce a logic that is adequate (see definition 3.4) w.r.t the congruence.

3.2 The pi-logic

The π-logic permits to formally and unambiguously specify the behavior of a system written in the π-calculus. This logic has been introduced in [5] to express temporal properties of π-processes. It adds the possible future modalities $EF\phi$ and $EF\{\chi\}\phi$ modalities to the modalities for strong next EX and weak next $< \mu >$ modalities defined by Milner [11]. Syntax of the π-formulas is:

$$\phi ::= true \mid \sim\phi \mid \phi \;\&\; \phi' \mid EX\{\mu\}\phi \mid EF\phi \mid EF\{\chi\}\phi$$

where μ is a π-calculus action and χ could be μ, $\sim\mu$, or $\bigvee_{i\in I}\mu_i$ and where I is a finite set.

The semantics of the π-formulae is given below:

- $P \models true$ for any process P;
- $P \models \sim \phi$ iff $P \not\models \phi$;
- $P \models \phi \ \wedge \ \phi'$ iff $P \models \phi$ and $P \models \phi'$;
- $P \models EX\{\mu\}\phi$ iff there exists P' such as $P \xrightarrow{\mu} P'$ and $P' \models \phi$ (**strong next**);
- $P \models EF\phi$ iff there exists $P_0, ..., P_n$ and $\mu_1, ..., \mu_n$, with $n \geq 0$, such as $P = P_0 \xrightarrow{\mu_1} P_1 ... \xrightarrow{\mu_n} P_n$ and $P_n \models \phi$. The meaning of $EF\phi$ is that ϕ must be true sometimes in a possible future.
- $P \models EF\{\chi\}\phi$ if and only if there exists $P_0, ..., P_n$ and $\nu_1, ..., \nu_n$, with $n \geq 0$, such that $P = P_0 \xrightarrow{\nu_1} P_1 ... \xrightarrow{\nu_n} P_n$ and $P_n \models \phi$ with:
  · $\chi = \mu$ for all $1 \leq j \leq n$, $\nu_j = \mu$ *or* $\nu_j = \tau$;
  · $\chi = \sim \mu$ for all $1 \leq j \leq n$, $\nu_j \neq \mu$ *or* $\nu_j = \tau$;
  · $\chi = \bigvee_{i \in I} \mu_i$: for all $1 \leq j \leq n, \nu_j = \mu_i$ for some $i \in I$ *or* $\nu_j = \tau$.
  The meaning of $EF\{\chi\}\phi$ is that the truth of ϕ must be preceded by the occurrence of a sequence of actions χ.

Some useful dual operators are defined as usual: $\phi \ \vee \ \phi$, $AX\{\mu\}\phi$, $< \mu > \phi$ (weak next), $[\mu]\phi$ (Dual of weak next), $AG\phi$ ($AG\{\chi\}$) (always).

π-logic formulae are expressive enough to naturally specify and verify liveness and safety properties and others.

3.3 The BP-logic

Since we are working on BP-calculus specifications we need to adapt the π-logic to this language. For this purpose, we only need to extend the logic to handle the termination predicate, introducing therefore the BP-logic.

The syntax of the BP-logic is:

$$\phi ::= true \mid \checkmark \mid \ \sim \phi \mid \phi \ \& \ \phi' \mid EX\{\mu\}\phi \mid EF\phi \mid EF\{\chi\}\phi$$

where μ is a π-calculus action and χ could be μ, $\sim \mu$, or $\bigvee_{i \in I} \mu_i$ and where I is a finite set.

The interpretation of the logic formulæ defined by the above syntax is the same as the interpretation of the π-formulæ extended with the explicit interpretation of the termination predicate $\checkmark$: $P \models \checkmark$ iff $P\checkmark$.

3.4 Adequacy

The adequacy allows the checking of the bisimulation rather than checking each property separately and thus is useful for the refinement process, since a process may be subtitued to an equivalent one with preservation of desired properties.

Definition 3.4 [Adequacy] A logic $\mathcal{L}$ is adequate with respect to, a relation ($\mathcal{R}$) defined on a given process language $\mathcal{P}$, if

$$(\forall \phi \in \mathcal{L}, \forall P, Q \in \mathcal{P}, \ P \models \phi \Leftrightarrow Q \models \phi) \ \Leftrightarrow \ P \ \mathcal{R} \ Q$$

That is, $P \ \mathcal{R} \ Q$ if and only if P and Q satisfy the same formulae.

Let $Th(P) = \{\phi : P \models \phi\}$, and the relation be the congruence ($\dot{\simeq}_{bp}$), thus the previous requirement is written: $P \dot{\simeq}_{bp} Q \Leftrightarrow Th(P) = Th(Q)$ that is a *strong requirement*; while *weak adequacy* is defined by : $P \dot{\simeq}_{bp} Q \Rightarrow Th(P) = Th(Q)$

It has been proved [6] that the π-logic is adequate with respect to the strong early bisimulation equivalence [11] of the π-calculus. This means that two π-calculus agents satisfy the same properties that can be expressed in the π-logic provided that they are early bisimilar. We may extend the result to the congruence ($\dot{\simeq}$) since it is included in the bisimulation.

At this stage, we need to prove the weak adequacy of the BP-logic w.r.t $\dot{\simeq}_{bp}$. Since the BP-calculus differs from the π-calculus only by the sequential composition and process spawn operators, we only need to study the effect of these two operators on the adequacy.

Theorem 3.5 *The BP-logic is (weakly) adequate w.r.t congruence ($\dot{\simeq}_{bp}$).*

The proof is based on the result of [6] that states the adequacy of the π-logic w.r.t the strong early bisimulation of the π-calculus. In fact, we only need to examine the sequential operator and the instance spawn. We also need to analyze the effect of the termination predicate on the adequacy.

If P and P' are two BP-processes not containing sequential operator nor instance spawn, one can apply the results of [6]. Now, let's see what happens with these two operators:

- *Sequential operator*:

  Let P and P' such as $P \dot{\simeq}_{bp} P'$ and $P \models \phi$, therefore $P' \models \phi$ (induction hypothesis).

  But $P \dot{\simeq}_{bp} P' \Rightarrow P \triangleright_{c(M)} Q \dot{\simeq}_{bp} P' \triangleright_{c(M)} Q$ due to the definition of the congruence and by applying rule SEQ1 of the operational semantics. We reason by induction on each modality of the BP-logic.

  · By definition of $EX\{\mu\}\phi$, $P \triangleright_{c(M)} Q \models EX\{\mu\}\phi$, means that $\exists \mu$ such as $P \triangleright_{c(M)} Q \xrightarrow{\mu} P_1 \triangleright_{c(M)} Q$ and $P_1 \triangleright_{c(M)} Q \models \phi$. But $P \triangleright_{c(M)} Q \dot{\simeq}_{bp} P' \triangleright_{c(M)} Q$ implies that $P' \triangleright_{c(M)} Q \xrightarrow{\mu} P'_1 \triangleright_{c(M)} Q$ and $P_1 \triangleright_{c(M)} Q \dot{\simeq}_{bp} P'_1 \triangleright_{c(M)} Q$. From the induction hypothesis we deduce that $P'_1 \triangleright_{c(M)} Q \models \phi$. Finally : $P' \triangleright_{c(M)} Q \models EX\{\mu\}\phi$.

  · By definition of $EF\phi$, a path $P \triangleright_{c(M)} Q \xrightarrow{\alpha_1} P_1 \triangleright_{c(M)} Q \xrightarrow{\alpha_2} ... \xrightarrow{\alpha_k} P_k \triangleright_{c(M)} Q$ of length $k \geq 0$ exists such that $P_k \triangleright_{c(M)} Q \models \phi$.

  But $P \triangleright_{c(M)} Q \dot{\simeq}_{bp} P' \triangleright_{c(M)} Q$ implies that $P' \triangleright_{c(M)} Q \xrightarrow{\alpha_1} P'_1 \triangleright_{c(M)} Q \xrightarrow{\alpha_2} ... \xrightarrow{\alpha_k} P'_k \triangleright_{c(M)} Q$ and $P'_k \triangleright_{c(M)} Q \dot{\simeq}_{bp} P' \triangleright_{c(M)} Q$. From the induction hypothesis we deduce that $P'_k \triangleright_{c(M)} Q \models \phi$. Finally : $P' \triangleright_{c(M)} Q \models EF\phi$.

  Finally, $P' \triangleright_{c(M)} Q \models EF\phi$.

  · The same reasonning holds for $P \models EF\{\chi\}\phi$.

  The proof is similar for the second operand: Let Q and Q' such as $Q \dot{\simeq}_{bp} Q'$ and $Q \models \phi$, therefore $Q' \models \phi$ by induction hypothesis.

  We suppose that $P\checkmark$ and thus, does not contain a spawning term (this case is treated in the second half of the proof). We then may apply rule SEQ2 of the

operational semantics: due to the definition of the congruence
$P\checkmark$ and $Q \dot{\simeq}_{bp} Q' \Rightarrow P \triangleright_{c(M)} Q \dot{\simeq}_{bp} P \triangleright_{c(M)} Q'$
Thus, $Q \dot{\simeq}_{bp} Q'$and$P\checkmark$ and $P \triangleright_{c(M)} Q \models \phi \Rightarrow P \triangleright_{c(M)} Q' \models \phi$.
Finally and for the same reasons than in the previous case:

$$P\checkmark \wedge Q \dot{\simeq}_{bp} Q' \wedge P \triangleright_{c(M)} Q \models \phi \Rightarrow P \triangleright_{c(M)} Q' \models \phi$$

- *Instance spawn*:
  Let $P = c_A(\tilde{x}).A(\tilde{y})$ and $P \models \phi$ and $\mathcal{C}$ a correlation set related to P.
  If $\mathcal{C} = \emptyset$, then $[null : 0]c_A(\tilde{x}).A(\tilde{y}) \overset{c_A(M)}{\rightarrow} [\tilde{z} \leftarrow \tilde{u}] : A(\tilde{u})]c_A(\tilde{x}).A(\tilde{y})$ (rule C-SPT of the operational semantics). In this case, the resulting process is P, that obviously satisfies ϕ.
  If $\mathcal{C} \neq \emptyset$ then $[\mathcal{C} : P]c_A(\tilde{x}).A(\tilde{y}) \overset{c_A(M)}{\rightarrow} [\mathcal{C}, [\tilde{z} \leftarrow \tilde{u}] : P|A(\tilde{u})]c_A(\tilde{x}).A(\tilde{y})$ (rule C-SPF of the operational semantics). In this case, the resulting process is $P \mid P \mid ... \mid P$, that satisfies ϕ by induction hypothesis ($P \models \phi \Rightarrow P|P \models \phi$).
  Finally:

$$P \models \phi \wedge \mathcal{C} \text{ is a correlation set } \Rightarrow [\mathcal{C} : P]c(\tilde{x}).A(\tilde{y}) \models \phi$$

The congruence contains a constraint on the termination of involved processes (both congruent processes must terminate). Since this is a restrictive constraint, and since the adequacy holds for all processes, it holds for terminating processes.

□

3.5 Verification

Let $\dot{\simeq}_{bp}$ be the congruence we defined in 3.1 and $\dot{\sim}_e$ be the early bisimulation of the π-calculus.

Let P be a BP-process. P's translation to BPEL is denoted $bpel(P)$ and we denote $[\![P]\!]$ the translation of $bpel(P)$ to π-calculus. $[\![P]\!]$ is obtained by means of Lucchi and Mazzara's semantics that we extend for the missing operators (see [2]).

For the purpose of verifying BP-processes using a π-calculus model-checker, we need to define a correspondance between the π-logic and the BP-logic and then to guarantee a soundness property.

Since the only difference between the two logic is the termination modality ($\checkmark$), we can proceed as follows: If a BP-formula does not contain the $\checkmark$ modality it is translated to a π-formula with exactly the same syntax. Otherwise, the formula is translated the same way and the process is checked for termination. This brings us to the following result about soundness.

3.6 Soundness

Since $[\![P]\!]$ is obtained by a translation through BPEL, we only need to show that P and $[\![P]\!]$ satisfy the same set of equivalent properties, to prove that $bpel(P)$ also satisfies the desired properties.

Theorem 3.6

$$\forall \phi,\ [\![P]\!] \models \phi \quad \Rightarrow \quad P \models \phi_{bp}$$

The proof uses the mapping from BP-calculus to BPEL and Lucchi and Mazzara's semantics and is given in Appendix.

As an important consequence, since the π-process is obtained through a mapping to BPEL, one can deduce that if a property holds for the initial BP-process, then the same propery holds for the translation to BPEL. This way, we state the correctness of the mapping.

4 Verification framework

The definition of the BP-calculus presented in Section 2.2 indicates the possible use of functions and equations exactly as it is done in the applied π-calculus. However, since the choice was made to use first the HAL Toolkit [5] associated with the π-logic, this aspect of the BP-calculus is not used here. That means that the specification and the verification stay at a lower level than it could be.

4.1 The verification/refinement process

The HAL formulae checker is used to verify and refine a specification written in BP-calculus. BPEL programs are automatically translated into BP-calculus processes or directly specified in the BP-calculus language. We also specify the desired properties by means of the π-logic. BP-processes are translated to π-processes and the validity of the translation, e.g. the preservation of properties, is asserted by the results of Section 3.6.

We then check if the formulas hold for the defined processes. If they are invalidated by the tool, we iteratively correct the processes and/or the formulas and we repeat the verification process until the system is validated. At this time, a version of the BPEL process is automatically generated.

We can also need to minimize any of the initial or the final formal specification and then we can use the HAL bisimulation checker to verify correctness of this minimization.

The example in the next section shows that the approach is practically feasible. since design languages usually describe few, interesting properties of a system (e.g. its behavior w.r.t. concurrency and communication), while often full verification is impossible due to the size of the implementation

5 Modelling and Verifying the Trading Service

The example presented here is intended to illustrate our approach and is adapted from [13].

5.1 The trading market service

A customer places an order to sell a quantity of shares by contacting a Broker service. The broker invokes other composite services to check the feasability of the transaction and to perform it. This scenario is well-suited to our study because it involves several composite services.

The case study scenario is informally described as follows: The *Customer* contacts the Broker composite service intending to sell the shares. The *Broker* invokes the Analytic service with the request parameters. The trend information is calculated and a trading plan is generated and returned by the *Analytic service* to the *Customer* for confirmation. The *Customer* checks the plan and approves it or rejects it. In case of approval, the *Broker* service submits order according to the plan to the *Exchange service*. Each order that is placed on the *Exchange service* successfully generates a receipt which is returned to the *Customer*. The *Surveillance service* monitors each order and the generated trades to detect possible illegal actions.

5.2 Formal description

We model the scenario by means of the BP-calculus; we do not use the correlation mechanism in this example and thus the processes do not contain the spawn construct. However the example is relevant since it includes handlers.

Let $\tilde{u} = (a, b, decision, o, p, q, r, s, t)$, then the whole BP-process is:

$$CapitalMarket(qty, plan, receipt, ok) := Customer(s, p, qty, a, ok, nok, r) \mid Broker(s, p, plan, q, qty, a, r, receipt) \mid Analytic(q, p, plan) \mid Exchange(o, t, b, ok, r, receipt) \mid Surveillance(b, t, decision)$$

We focus on the Broker process that we define with as scope. This scope is a wrapper for local variables and event and fault handlers. Note that we use the handlers' syntax we developed in [1].

The event handler manages the occurrence of timeout event. Each service has a finite period of time for providing a response to a request. If this time is elapsed, the calling service triggers a timeout event caught by the event handler.

The fault handler manages faults occurring while invoking the Broker service. We consider the three following faults for this service : the Broker is busy (f_{bb}), the Analytic service is down or busy (f_{asd}), the Exchange service is down or busy (f_{esd}),

We now formalize the Broker service and its handlers.

Let $\tilde{u} = (en_{eh}, en_{fh}, dis_{eh}, dis_{fh}, t, timeout, p, receipt, f_{bb}, f_{esd}, f_{asd}, y_{eh}, y_{fh})$ and $\tilde{w} = (a, ok, s, qty, order, q, o, plan, en_{eh}, en_{fh}, dis_{eh}, dis_{fh}, r, x, bb, esd, asd)$

The Broker is defined by: $Broker := \{\tilde{u},\ B,\ H\}$ where B is its main activity and H the set of handlers (event and fault handlers, here):

$$B(\tilde{w}) := s(qty).\Big(\overline{q}^{inv}\langle qty\rangle.a(decision).if(decision = ok)\overline{o}^{inv}\langle order\rangle + \overline{timeout}^{inv}\langle\rangle + Error()\Big)$$

where $Error() := \overline{f_{bb}}^{throw}\langle\rangle \ + \ \overline{f_{esd}}^{throw}\langle\rangle \ + \ \overline{f_{asd}}^{throw}\langle\rangle$

When receiving a plan approval, the Broker may send the order to the Exchange service, or trigger a timeout event; In the latter case, the event handler runs the timeout event handling action ($A_{Timeout}$). Finally, it may trigger a fault; in this case the fault handler is invoked that runs the corresponding action (F_{bb}, F_{asd} or F_{esd}).

$H = \{EH(e_{eh}, y_{eh}, d_{eh}, timeout, t), FH(f_{bb}, f_{esd}, f_{asd}, e_{fh}, y_{fh}, bb, esd, asd, r)\}$

The **Event handler** is as follows:

$EH(e_{eh}, y_{eh}, d_{eh}, timeout, t) := (\nu e_t) \ \ en_{eh}().\Big(\big(timeout().\overline{e_t}^{inv}\langle\rangle$

$+ \ dis_{eh}().\overline{y_{eh}}^{inv}\big) \mid e_t().A_{Timeout}\Big)$ where $A_{Timeout} := \overline{p}^{inv}\langle timeout\rangle$

The **event handler** is enabled using the en_{eh} channel and waits for a unique event (the timeout) on channel e_t. Then, it processes an activity ($A_{Timeout}$) associated with this event. It is disabled using the dis_{eh} channel and y_{eh} signals the disabling of the event handler.

The **Fault handler** is expressed by :

$FH(f_{bb}, f_{esd}, f_{asd}, e_{fh}, y_{fh}, bb, esd, asd, r) := en_{fh}().\Bigg(\Big(\Big(\big(f_{bb}(p, \tilde{u}).\big(\overline{throw}^{inv}\langle\rangle$

$\mid F_{bb}(p)\big)\Big) \ + \ \Big(f_{esd}(p, \tilde{u}).\big(\overline{throw}^{inv}\langle\rangle \mid F_{esd}(p)\big)\Big) + \ \Big(f_{asd}(p, \tilde{u}).\big(\overline{throw}^{inv}\langle\rangle$

$\mid F_{asd}(p)\big)\Big)\Big) \triangleright \ \Big(\overline{r}^{inv}\langle\rangle \mid \overline{y}_{fh}^{inv}\langle\rangle\Big) + dis_{fh}()\Bigg)$ where each process associated with a fault defined as:

$$F_{bb}(p, brokerbusy) := \overline{p}\langle brokerbusy\rangle \quad \text{(to handle "broker service busy" fault)}$$
$$F_{esd}(p, esd) := \overline{p}\langle esd\rangle \quad \text{(to handle "exchange service down" fault)}$$
$$F_{asd}(p, asd) := \overline{p}\langle asd\rangle \quad \text{(to handle "analytic service down" fault)}$$

The **fault handler** deals with three kinds of faults : $S_f = \{f_{bb}, f_{esd}, f_{asd}\}$ together with their associated activities : $F = \{F_{bb}, F_{esd}, F_{asd}\}$. It is enabled using the e_{fh} channel and then the activity associated with the triggered fault is processed. Finally it signals its termination to the calling scope and the activating throw using the y_{fh} and the channel r. It is finally disabled using the channel dis_{fh}.

The same model is applicable to other composite services which may also contain scopes.

It is worth noting that introducing a scope in the model has involved of a big amount of complexity. The generation of the History Dependent automaton using the HAL Toolkit for the model without scope has been made within one half second and has resulted in a six-state automaton. While the Broker's model with scopes, which is only a part of the whole model, has generated an automaton with more than a 1000 states in more than 600 seconds. This illustrates the fact that the BPEL language is very powerful but its formal verification is not an easy task applied to weighty cases.

Once the system is formally specified, on needs to proceed to the formal verification of desired properties.

5.3 Verification of functional properties

Many properties of interest for services and SOC applications have been defined so far ([16]): Availability, reliability , responsiveness, fairness, or fault-tolerance. Here are some examples of the verification of such properties applied to the case study of Section 5.1.

Responsiveness:

A service is *responsive* if it guarantees a response to every received request in finite time. The property stating that whenever the customer sends a sell order, he will obtain a plan after a finite time, and whenever a customer agrees a selling plan, and the order is approved by the surveillance service, he will receive a receipt, is a responsiveness property. This property can be formalized by the following π-formula: ϕ_1 & ϕ_2 where:

$$\phi_1 = AG([s?qty]EF(< p![plan] > true))$$
$$\phi_2 = AG(([a?(ok)][b?(ok)])EF([r!receipt]true))$$

This property has been validated on the model with the HAL toolkit.

Availability

A service is said available when it is available at any time. The property stating that in every state the broker service may accept a request is an availability property. The π-formula is: $AG([s?qty]true)$.

This property has also been validated on the model with the HAL toolkit.

Reliability

Reliability is the capability to deliver response continuously in time (service reliability) and the capability to correctly deliver messages between two endpoints (message reliability). The property stating that the reception of a plan delivery is guaranteed whenever a sell order has been sent is a reliability property. This can be expressed as a π-formula as follows : $AG([s?qty]EF < p!plan > true)$

This property has also been validated on the model with the HAL toolkit.

Fairness

Fairness stipulates that if a process is continuously enable to communicate on a channel, then it must eventually proceed. The property stating that if a customer sends an infinite number of sell orders, then he will receive an infinite number of plans and receipts, is a fairness property. This can be expressed as a π-formula as follows:

$$\phi_1 \vee AG(\psi_1 \ \& \ \psi_2) \text{ where } \begin{array}{l} \phi_1 = \neg AG(EF < s?* > true) \\ \psi_2 = EF < r!receipt > true \\ \psi_2 = EF < p!plan > true \end{array}$$

This property has also been validated on the model with the HAL toolkit.

Safety

This properties assert that some bad event never happens in the course of a computation. For instance, the property that states that a receipt should never be sent before it has been approved by the surveillance service is a safety property. The π-formula is: $\sim EF(\sim < b!ok > true \ \& \ < r?* > true)$ This property has been invalidated and that is acceptable since a receipt is sent only if the decision is ok.

Liveness

Liveness properties assert that some event does eventually happen. An example of a liveness property relevant to the capital market use case is the following: the system will eventually execute the action $t!order$ (order's checking) whenever it has executed the action $a!ok$ (plan's approval). The π-formula is: $AG(< a!ok > EF < t!order > true)$

This property has been validated on the model with the HAL toolkit.

In this model all the desired properties have been validated (except one safety property that is accepted anyway). In the case where some properties are invalidated, one must modify the specification in such a way that all the properties are accepted. The verification process is re-iterated until all the desired properties are validated. At this time, we can proceed to the automatic generation of the BPEL code.

Since some verifications are time consuming, a formal reasoning might processed upon certain parts of a system, in order to validate these parts of the project requirements. Tis increases the practical feasibility of the approach.

6 Conclusion

In this paper we have presented some theoretical results for the BP-calculus, a π-like calculus that is designed for formal specification of Web Service orchestrations and that allows verification and automatic generation of readable, easy to support and correct BPEL code. We have defined a congruence, that permits to demonstrate the correctness of the mapping from the BP-calculus to BPEL w.r.t the BP-logic. We have also proved the adequacy of this logic w.r.t the congruence.

As an illustration of the applicability of the the calculus, we presented a meaningful case study (the Capital Market process): we first specified the case study, including complex constructs such as handlers, then we used the logic to assert and verify some desirable properties of the system. Using an iterative process, we veri-

fied the correctness of the system and then proceeded to its automatic translation to BPEL.

While some previous works have been done on integrating model-checker toolkits and generating BPEL code that has the same behavior as the model ([17]), our proposal takes into account many significant structured activities, including scopes and handlers and offers integration to a verification/refinement framework for the design.

We are developing a tool integrating the BP-calculus to the the HAL toolkit. Our tool will also be used to generate the correct BPEL code of the business process from the formally verified specification.

References

[1] F. Abouzaid and J. Mullins. A calculus for generation, verification and refinement of bpel specifications. *Electronic Notes in Theoretical Computer Science*, 200(3):43–65, 2008.

[2] F. Abouzaid and J. Mullins. Translating bp-calculus specifications to verified bpel code : A proof of correctness. Technical report, Ecole Polytechnique de Montreal, www.polymtl.ca/crac/abouzaid/rr01-2009.pdf, 2009.

[3] A. Chirichiello and Gwen G. Salaün. Encoding process algebraic descriptions of web services into bpel. *Web Intelli. and Agent Sys.*, 5(4):419–434, 2007.

[4] D. Fahland and W. Reisig. ASM-based semantics for BPEL: The negative Control Flow. In D. Beauquier, E. Biger, and A. Slissenko, editors, *Proceedings of the 12th International Workshop on Abstract State Machines (ASM'05)*, pages 131–151. Paris XII, March 2005.

[5] G.L. Ferrari, S. Gnesi, U. Montanari, and M. Pistore. A model-checking verification environment for mobile processes. *ACM Trans. Softw. Eng. Methodol.*, 12(4):440–473, 2003.

[6] S. Gnesi and G. Ristori. A model checking algorithm for pi-calculus agents. In *Applied Logic Series*, volume 16, pages 339–358. Kluwer, 2000.

[7] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. Sock : A calculus for service oriented computing. In Springer Berlin / Heidelberg, editor, *Service-Oriented Computing ICSOC 2006*, volume 4294/2006 of *Lecture Notes in Computer Science*, pages 327–338, 2006.

[8] A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In *Proc. of 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2007.

[9] R. Lucchi and M. Mazzara. A pi-calculus based semantics for ws-bpel. *Journal of Logic and Algebraic Programming*, 2007.

[10] R. Milner. *Communicating and Mobile Systems: The Pi-Calculus.* Cambridge University Press, Cambridge, UK, 1999.

[11] R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 1993.

[12] Oasis. Web service business process execution language version 2.0 specification, oasis standard. *http : //docs.oasis − open.org/wsbpel/2.0/wsbpel − v2.0.pdf*, april 2007.

[13] F. A. Rabhi, F. T. Dabous, Hairong Yu, B. Benatallah, and Y. K. Lee. A case study in developing web services for capital markets. In *Proc. of the 2004 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE 04)*, 2004.

[14] D. Sangiorgi and D. Walker. *The π-calculus: A Theory of Mobile Processes.* Cambridge University Press, 2001.

[15] Davide Sangiorgi and David Walker. On barbed equivalences in pi-calculus. In *CONCUR*, pages 292–304, 2001.

[16] M.H. ter Beek, A. Bucchiarone, and S. Gnesi. Formal methods for service composition. *Annals of Mathematics, Computing & Teleinformatics*, 1(5):1–10, 2007.

[17] W. M. P. van der Aalst and K. B. Lassen. Translating unstructured workflow processes to readable bpel: Theory and implementation. *the International Journal of Information and Software Technology (INFSOF)*, December 2006.

Appendix

A Proof of Theorem 3.6.

A BP-process P is translated to a BPEL process $bpel(P)$ by means of the mapping. $bpel(P)$ is then mapped to a π-calculus process $llbracketPrrbracket$, using the semantics we deduced from Lucchi and Mazzara's one and that we completed by providing a semantics for missing operators. For the need of the proof we introduce the following abstract syntax of BPEL's main constructs:

$$
\begin{aligned}
A := {} & invoke(x, \tilde{i}, \tilde{o}) \quad (synchronous\ invoke) \\
| \ & invoke(x, \tilde{i}) \quad (asynchronous\ invoke) \\
| \ & receive(x, \tilde{i}) \quad (receive) \\
| \ & receive(x, (C), \tilde{i}) \quad (receivewithcorrelation) \\
| \ & reply(x, \tilde{o}) \quad (reply) \\
| \ & sequence(P, Q, M) \quad (sequence) \\
| \ & flow(P, Q) \quad (parallel) \\
| \ & Conditional(cond,\ P,\ Q) \quad (conditional) \\
| \ & scope(\tilde{x}, P, H) \quad (scope) \\
| \ & spawn(\mathcal{C}, P) \quad (instance\ spawn)
\end{aligned}
$$

The proof of the theorem is conducted upon all operators of the language for which we prove that they preserve π-logic properties.

- Let $P = \bar{x}^{inv} \langle \tilde{i} \rangle$ then $bpel(P) = invoke(x, \tilde{i})$ and $[\![P]\!] = \bar{x} \langle \tilde{i} \rangle$. It is obvious that $[\![P]\!] \models \phi \ \Rightarrow \ P \models \phi$.
- the same thing holds for the synchronous output ($P = \bar{x}^{rep}$) and input ($P = x(\tilde{o})$).
- *Parallel operator* : Let $P = P_1|P_2$, then $bpel(P) = flow(P_1, P_2)$ and $[\![P]\!] = [\![P_1]\!]|[\![P_2]\!]$.
  By construction: $[\![P_1]\!] \xrightarrow{\alpha} [\![P_1']\!] \ \Rightarrow \ P_1 \xrightarrow{\alpha} P_1'$.
  Therefore, $[\![P_1]\!]|[\![P_2]\!] \xrightarrow{\alpha} [\![P_1']\!]|[\![P_2]\!]$ and thus $P_1|P_2 \xrightarrow{\alpha} P_1'|P_2$ (by semantics rule `PAR`). We deduce : $[\![P_1]\!]|[\![P_2]\!] \models \phi \ \Rightarrow \ P_1|P_2 \models \phi$.
- *conditional* :
- *Sequential operator*: Let $P = P_1 \triangleright_{c(M)} P_2$, then $bpel(P) = sequential(P_1, P_2, M)$. Suppose that P_1 is of the form $P_1'.\bar{c}\langle M \rangle$ that means that P_1 indicates its termination by performing an output on the private channel c.
  Then $[\![P]\!] = (\nu c)([\![P_1']\!].\bar{c}\langle M \rangle \,|c(M).[\![P_2]\!])$.

  Thus, $P_1' \xrightarrow{\alpha} P_1$" $\Rightarrow$ $[\![P]\!] \xrightarrow{\alpha} (\nu c)([\![P_1\text{"}]\!].\bar{c}\langle M \rangle \,|c(M).[\![P_2]\!])$, that is the same behavior expressed by semantics rule `SEQ1`.

  On the other side, if $P_2 \xrightarrow{\alpha} P_2'$ and $\alpha \neq x(\tilde{i})$ and P_1 terminates, e.g. does not contain a spawn construct, then $[\![P]\!] \xrightarrow{\alpha} [\![P_2']\!]$ that is the same behavior expressed by semantics rule `SEQ2`.

If P_1 does contain a spawn construct and α is an input, then $[\![P]\!] \xrightarrow{\alpha} ([\![P_1']\!]|[\![P_2]\!])$, that is the same behavior expressed by semantics rule `SEQ2'`.

In both cases, the behavior of P and $[\![P]\!]$ is the same and thus, $[\![P]\!] \models \phi \Rightarrow P \models \phi$.

- *spawn operator* : Let $Q = [\mathcal{C} : P]c_A(M).A$ be a BP-process with correlation set (C), then $bpel(P) = receive(c_A, (C), M)$. Finally, $[\![P]\!]$ can be modelled in π-calculus this way:

$$BProc(x) :=!x(y).(\nu z)bary\,\langle z\rangle\,.Instance(z)$$

$$Client(x, y) := barx\,\langle y\rangle\,.y(z).Session(z)$$